# On Approximate Matching of Programs for Protecting Libre Software

Arnoldo José Müller Molina and Takeshi Shinohara

Department of Artificial Intelligence, Kyushu Institute of Technology

## Abstract

Libre software licensing schemes are sometimes abused by companies or individuals. In order to encourage open source development it is necessary to build tools that can help in the rapid identification of open source licensing violations. This paper describes an attempt to build such tool. We introduce a framework for approximate matching of programs, and describe an implementation for Java byte-code programs. First, we statically analyze a program to remove dead code, simplify expressions and then extract *slices* which are generated from assignment statements. We then compare programs by matching between sets of slices based on a distance function. We demonstrate the effectiveness of our method by running experiments on programs generated from two compilers and transformed by two commercial grade control flow obfuscators. Our method achieves an acceptable level of precision.

## 1 Introduction

*Libre* software usage is becoming more and more widespread. Studies like [20, 21, 32] show important implications of this software development model. There are companies or individuals who take source code of libre software and use it in their own programs and distribute them without releasing the modified source code. This is called an open source license violation [4]. Most but not all open source licenses have this restriction.

Organizations like the Free Software Foundation [3] track these violations using arguably rudimentary programs like "strings"[5]. The process consists basically of extracting all the strings from binary program and comparing them with the strings embedded in the source code of an open source program. When the binary is encrypted, some additional preprocessing must be performed. This technique can be easily invalidated if those strings are changed. A simple natural language translation or wording modification of the piece of libre software is enough, and occurs frequently enough to invalidate the technique considerably.

The current state of the art in Open Source validation detection and the importance of the open source movement, has motivated us to implement a more robust technique that does not rely on string extraction and one that can withstand binary modifications such as different compiler versions, code generation options and attacks such as obfuscation[25, 14, 15].

The main result we have achieved is that by extracting *slices* from assignments contained in each basic block of an SSA(Static Single

Assignment) graph[16], one can perform retrieval of binary programs with acceptable performance when using different compilers and commercial grade obfuscators in Java. This retrieval result can be the base for additional semantic analysis.

In section 2 we review related research. Section 3 explains the approach we have implemented. Sections 4 and 5 summarize respectively, the overall design and experimental results of a prototype implementation.

# 2 Related Work

The main obstacles present in the field of program matching are the fact that checking the semantic equivalence between two programs is undecidable and the difficulty that 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 the so called obfuscator programs can mangle the binary enough to prevent reverse engineering[15], imposes a challenge. Also, several important static analysis problems are undecidable or computationally hard[24, 28]. However, a result by Vadhan *et al.* [9] 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.

## 2.1 Other Detection Techniques

As mentioned before, the "strings" technique for detecting violations is not robust enough, as obfuscators can encrypt the strings embedded in a program. Baker[8] introduces some techniques based on adapting existing source code similarity analysis tools so that they can handle Java byte-code programs. Sadly, the author recognizes that those techniques do not 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 [27] or dynamic [13] manner). However, because the source code of the applications we want to protect is open and available to anyone, this technique cannot be employed.

A technique called *code cloning* [10, 33] is used to detect duplicate fragments of code in source code in order to reduce maintainability problems in software projects. Automatic tools for measuring *software similarity* [31, 26] have been also presented in order to perform *plagiarism detection*. However, the source code for the application that illegally contains libre software is not available in our problem setting, and both code cloning and plagiarism detection techniques require the source code of the target programs. *Birth-marking* [35, 37, 36, 38] is a technique that extracts unique and "native" characteristics of every class file. The currently available birth-marks for static matchings are[37]:

1. Constant values in fields (initialization values)
2. Sequence of method calls
3. Inheritance structure
4. Used classes

After reviewing the definitions of the birth-marks it is easy to see that two classes with exactly the same birth-marks can have completely different behavior. An obfuscator could easily add fake inheritance relationships and dummy method calls, object references and fields that would distort the matching process. In [39] the same authors acknowledge this and propose a dynamic approach that collects birth-marks on how the application accesses the system's API. However, the approach requires to run the target programs with the same set of inputs in order to obtain similar birth-marks, and there is no guarantee that a pirate program will accept the same type of inputs. Therefore this approach cannot be applied to our problem domain.

Malicious code detection [12] 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 accomplished by finding if the malicious pattern described by the MCA is in the annotated CFG. The

approach is quite powerful but an automatic MCA generator algorithm is not available. As viruses are relatively small, a manual construction approach works well. Building an automatic MCA generator seems much more complicated than the proposal we are presenting here.

Static disassembly of binary files[23] could also be applied. Combined with techniques like *code cloning, plagiarism detection* or more robust techniques like slicing for detecting common code in programs[22, 19] it could accomplish our objective. We do feel that reverse engineering a binary and then to slice the result, represents a waste of computational resources. In [23] Kruegel asserts that tool-specific knowledge is required in order to achieve almost complete disassembling.

The novelty of our approach is first, the usage of simplified slices (section 3). Second, the way we handle constructs that can be modified easily by obfuscation transformations like string values and variable names (section 3.2). Also, we believe that our slice expansion technique is unique (section 3.1).

Our work has been loosely based on translation validation techniques that were introduced by Pnueli, Siegel and Singerman[30] and applied to compiler optimization validation by Necula[29], and Engelen *et al.* [44]. We have focused our attention mostly on the latter work.

Engelen and his group have implemented a transformation validation algorithm for low-level program representations. The approach takes two control flow graph (CFG) structures as an input, and generates *semantic effects* (in this paper, the term is changed to slices) based on the assignment instructions and the entry conditions of each basic block. It then compares the semantic effects that are alive at the exit points of both CFGs, and if they are equal, the CFGs are said to be semantically equivalent. This methodology differs from our research domain in that in the compiler optimization validation field, one can potentially have additional annotations that the compiler can generate to facilitate the semantic equivalence matching process. For example, when comparing the semantic equivalence of two CFGs, a basic block in one CFG can have tags that indicate a correspondence to one or more basic blocks in the other CFG.

In this paper we have not considered code segment encryption obfuscation techniques. We are not considering either in-line method replacement optimizations. We will further explore these transformations in the future.

# 3 Approximate Matching of Programs

An SSA graph is a program representation. Each node contains sequences of variable assignment instructions. The last statement of a node can be a control flow statement (jump, goto, etc.). 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 could be represented in SSA form. Note how the variables *res* and *count* have a subindex that uniquely identifies each assignment. The *Phi* function selects one of the parameters, depending on where the control flow is coming from.

An overall system architecture is shown on figure 2. Initially, a binary program is converted into SSA form[16]. The next step is to take all the assignment expressions within each basic block of the SSA graph and create an order independent structure analogous to Engelen's semantic effects. We call this structure *slice* through all the paper referring to the work initiated by Weiser[45], even though our definition is much more relaxed. Once a set of slices is obtained, they can be stored in a database or can be matched against a database of slices. The *match* procedure receives a database of programs (each program is a multi-set of slices) and one set of slices $P$ and returns a set of pairs $(App_i\ y_i)$ where each $App_i$ represents an application of the database and each $y_i$ represents the similarity of $P$ and $App_i$.

The transformation between a binary program and an SSA representation is outside the scope of this paper. In our current prototype implementation we are using the Soot framework[42, 43] to convert from Java bytecode files to a set of SSA graphs.

The syntax of a slice is defined in figure 3. The `localRef` construct is a reference to an-

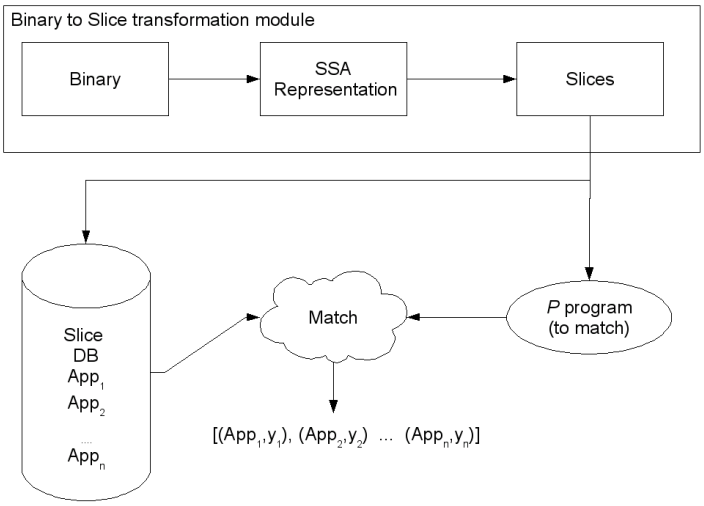


Figure 2: System Overview

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

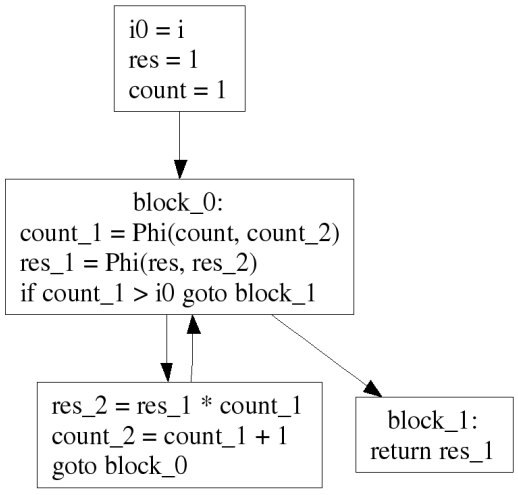


Figure 1: SSA graph generated from f

other variable within the same SSA graph. Engelen's proposal generates similar structures, but he adds conditions at which the slices are applicable. Because control flow obfuscators most likely will change the conditions in which assignments occur[14, 15] we have decided to omit this element from the slices.

Our slices are generated by taking all the right hand side expressions of all the assignments in the SSA graph. Since these expressions tend to be small, we recursively copy the right hand side of the variable defined by `localRef` constructs that are embedded in the original expression until the slice grows to a certain size. Expressions that cannot be expanded "enough" are discarded because the probability of them being in other programs or even methods of the same application is high.

The *FunctionIdentifier* element can contain any function name but an important construct to note is the `phi` construct. It holds the same semantic meaning as in the SSA representation. The "FunctionIdentifier(...)" element denotes a construct whose parameters are ignored. We will use this element with no parameters in sections 3.2 and 3.3.

Following the spirit of Engelen's work we should add a "condition" element next to each

Slice := FunctionIdentifier '(' Slice[,Slice]* ')'
| FunctionIdentifier '(' C ')'
| FunctionIdentifier '(...)'
C := Number | Boolean | String
FunctionIdentifier := 'number' | 'localRef' | 'boolean' | 'phi' ...

Figure 3: Slice Syntax

```
l1=sum(3, 4)
l2=methodInvoke(localRef(o1),
                methodRef(Vector.add),
                string("Hello World"))
l3=phi(0 , localRef(o2))
```

Figure 4: Slice examples

parameter of the `phi` function that indicates under which conditions the parameter should be selected, however, for the same reasons explained above, we have omitted this. Based on the syntax, it is possible to generate expressions such as the ones shown in figure 4.

The first example adds 3 and 4. The second example adds the string "hello world" to the vector o1 and returns a boolean value. The third example returns 0 or o2 depending on which was the previously executed block.

The definition of slice we presented differs from what Weiser [45] defined in his seminal paper. We have deliberately relaxed his definition in order to obtain slices that are more *precise* or, in other words, better tailored to our information retrieval needs. This relaxation is not new, it has been mentioned by Tip [40] and others. Tip [40] surveyed different slicing methods, and he argued they had the following two restrictions:

1. A slice consists of a subset of the statements of the original program, sometimes with the additional constraint that a slice must constitute a syntactically valid program.
2. Slices are computed by tracing data and control dependencies.

He mentions that these restrictions must be removed in order to improve the precision of the slices. We are removing the restriction that our slices are a syntactically valid program (restriction 1), because (1) the slices do not have a condition that indicates when they are executed, and (2) the same hint is not embedded in the `phi` expressions. And as the section 3.1 describes, our slices are not computed by tracing control dependencies (restriction 2).

## 3.1 Slice construction

The right hand side of assignment expressions in an SSA graph tends to be very small. Each assignment is a 3 address assignment like: `i1 = a + b`. We are interested in these right hand sides of assignment statements and also in *expanding* them in order to get features that can be matched later. This expansion consists of recursively finding references to other variables in the right hand side of an assignment instruction and replacing them with the right hand side of the referenced variables.

Algorithm 1 outlines the procedure we implemented to expand slices. The function expandAux prevents us from expanding a slice if it has been previously expanded. This means that if we have a slice such as `sum(localRef(a), localRef(a))`, "a" will be expanded two times only. If while expanding any of the sum's parameters another reference to "a" appears, then this expansion will not be performed because "a" was expanded higher up in the recursion hierarchy.

Symbolic expansion of expressions abstracts how data is modified, independently of execution order, renaming of variables, or temporary variable usage. This has been pointed out already by Necula[29], and Dijkstra[17].

As an example, running expand(a) on an SSA graph that contains the following assignments:

1. `a = phi(localRef(b),2)`
2. `b = phi(localRef(c),3)`
3. `c = sum(localRef(a),3)`

will return:
`phi(phi(sum(localRef(a),3),3),2)`.

### 3.2 Equality of Slices

A slice's FunctionIdentifier element is *special* when its contents are not interesting to us, but only the fact that the FunctionIdentifier element is there. Examples are constructs like `string` or `localRef` that refer to things that can be modified easily by a compilation optimization, obfuscator or a human. Special slice elements are checked for existence, but their contents are ignored. Two slices are equal to each other if they are syntactically equal, and if there are any special constructs within them, their arguments are ignored. In the case of `phi` constructs, the order of the parameters is irrelevant.

For example:

- `sum(number(3),number(2))`$\neq$ `sum(number(2),number(3))`
- `sum(localRef(a),number(1))`= `sum(localRef(b),number(1))` (`localRef` is special)
- `phi(number(3),number(2))`= `phi(number(2),number(3))`
- `f(...)`= `f(...)`

### 3.3 Distance Matching

In order to match slices that are not equal, some distance function must be used. The generally accepted similarity measure for trees is the tree edit distance introduced by Tai[34]. This problem is known to be NP-hard for unordered trees and the algorithms available work on restricted versions of the problem. We implemented a distance heuristic that mimics the behavior of a tree-edit distance function. It supports special constructs. This is just a temporary function and is included for reference only as it was used to perform the experiments.

Let $E$: The set of all slice expressions. Algorithm 2 shows the *toList* procedure. This procedure takes as an input a slice $r$ and returns a list of all the expressions that compose $r$ including $r$ itself. Also for each included element in the result set a copy of the element without parameters is added (we denote this with: $f(\ldots)$). We do this in order to include the case where the construct exists in both slices, but the parameters differ. The structural length $slength : E \rightarrow \mathbb{N}$ of a slice $f(x)$ is defined as $slength(f(x)) = |toList(f(x))|$. Algorithm 3 defines the *dmatch* function. Because the function *toList* always returns an even number of elements, the resulting value is always in $\mathbb{N}$. An example can be found in figure 5. The top element marked as "1" is the original expression to match. The arrows show which elements become part of the multi-set intersection operation.

## 4 Furia: an approximate program matcher

We have tested the techniques presented above by implementing an approximate program matcher for Java byte-code named "Furia". Among the first complications we encountered is the fact that expanded slices can become very large. A function that estimates the final size of a slice after expansion is applied is defined as: $c(n) = 2^{n-1}\theta + 2^{n-1} - 1$

For example, consider a method without `phi` expressions, and maximum number of structures per assignment of 3 structures ($\theta = 3$)[1], if we execute c(20) the result is 2097151. This assumes that there is very high dependency between the slices of a method and that every slice references two other slices. This is an extreme case, but it gives a good idea on how much this expansion can make a slice grow. Experimentally we have seen expanded slices of 30000 or more structures in methods of 270 assignments. Because of this, we have modified the procedure in figure 1 to stop after a certain number of slices has been reached. This threshold is

[1]This is the typical case, for instance i1 = 3 + 4 has 3 structures: sum(number(3),number(4)). This notion of structure size is equivalent to the result returned by *slength* divided by 2.

**Algorithm 1** Slice Expansion

```
// Returns the right hand side of the assignment that defines variableName
rightHand(variableName)
// Replaces all the occurrences of variableName in Slice
// with the right hand side of the assignment that defines variableName
replace(Slice, variableName)
//main function
expand(variableName){ expandAux(variableName, {}); }
expandAux(variableName, visited){
    visited = {variableName} ∪ visited;
    slice = rightHand(variableName);
    for all var in slice{ // get all the localRef(var) in slice
        if(var ∉ visited){ // only expand var if it has not been processed
            toBeReplaced = expandAux(var, visited);
            slice = replace(toBeReplaced,var);
        }
    }
    return slice;
}
```

**Algorithm 2** toList

$toList : E \rightarrow 2^E$ (multi-set)
$toList(f(u_1, \ldots, u_i)) = [f(u_1, \ldots, u_i), f(\ldots)]$ if $u_1 \ldots u_i$ is a string, boolean, or number
$toList(f(s_1, \ldots, s_n)) = [f(s_1, \ldots, s_n), f(\ldots)] \cup toList(s_1) \ldots \cup toList(s_n)$ if $f$ is not special
$toList(f(s_1, \ldots, s_n)) = [f(\ldots), f(s_1, \ldots, s_n)]$ if $f$ is special

**Algorithm 3** dmatch

Using the definition of multi-set intersection given by Blizard [11] The intersection uses the notion of slice equivalence defined in section 3.2

$dmatch : E \times E \rightarrow \mathbb{N}$
$dmatch(e1, e2) = \frac{(slength(e1)+slength(e2))-(2*|toList(e1) \cap toList(e2)|)}{2}$

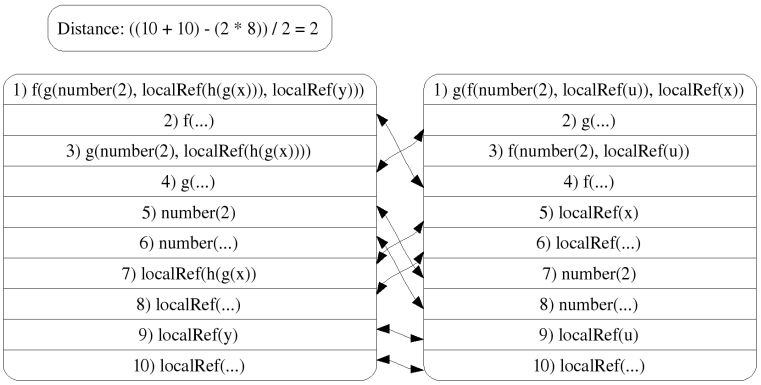


Figure 5: dmatch example

called *slice_cut_threshold.*

We also define a threshold *maximum_acceptable_distance* that indicates the maximum tolerable distance between two slices. So if for two slices $s_1$, $s_2$ $dmatch(s_1, s_2) \leq$*maximum_acceptable_distance* then the slices are said to be equivalent. An additional property is *ignore_slices_lower_than.* All the slices $s$ whose $\frac{slength(s)}{2}$ is lower or equal to this value will be ignored. The reasoning behind this is that, for example, slices of 3 or 4 structures are too common, and have too few features to be relevant.

In order to implement the system described in section 2 it is necessary to have a function that receives two sets of slices and determines how similar the programs are. Our current definition of this function is sketched in algorithm 4. We are aware that this function does not maximize the score between two applications, however, when *maximum_acceptable_distance* is close to 0, the results we are obtaining should not be far from the optimum score.

The current prototype uses the Soot library to transform byte-code files into an SSA representation. In our experiments we make use of Soot's optimization algorithms such as:

- Common subexpression elimination
- Copy propagation
- Constant propagation, folder
- Conditional branch folder
- Dead assignment elimination
- Unreachable code elimination

These analysis and transformations are applied before generating the slices. Algorithms, 2, 3 are implemented with the term rewriting programming language Q[2]. This allows greater flexibility when handling slices as it is very easy to create transformation rules to "massage" the slices into some standardized form.

When a small portion of a large software system is used, all the matching scores will be very low. If the used portion has unique features that are not common to other programs, then few matches greater than zero will be returned. With the current techniques it is impossible to determine when the binary is using a small portion of a program, and when the binary is not related at all and some common slices are being matched. A match function that takes into account the uniqueness of the matched slices could be used to handle this scenario.

# 5 Experiment results

We performed a set of experiments on three different databases. First, in a small database of 18 applications we explain how the different parameters Furia receives affect the matching results. We then proceed to show some results on a database of 269 applications. We finally, show an approximation of what we believe is a real open source violation program, in a database of 363 applications. Each experiment consists of one application $P$ that is matched against a database. The resulting output of the experiment is a list of pairs $(App_i\ y_i)$ where each $App_i$ represents an application of the database and each *score* $y_i$ is calculated by $match(P, App_i)$. The $App_i$ that has the biggest $y_i$ becomes *the candidate* of the match.

When performing the experiments we encountered several difficulties:

1. Soot will take days to generate an SSA graph from an obfuscated binary. Even a non-obfuscated program might be processed for more than an hour. The SSA creation routines have some performance issues.

2. When creating the databases some applications are ignored because Soot will simply abort the parsing of the class files, or the process will time out and we will cancel it. This timeout was set to 30 minutes.

3. Our current expansion technique does not expand methods, therefore some slices are too small. Some applications will not have many features and their matches are meaningless. Applications with less than 200 slices have been arbitrarily removed from the databases.

4. The slice matching techniques explained in section 3.3 are computationally expensive. One application match takes more than 24

**Algorithm 4** match

$match : 2^E \times 2^E \rightarrow \mathbb{R}$
$match(p_1, p_2) = \frac{matchAux(p_1,p_2)}{|p_1|}$

```
// returns the multiplicity of "slice" in the multi−set "p". It uses our notion of slice equivalence.
m(slice, p)
// The matchAux function receives two multi−sets (programs) of slices,
// returns the amount of slices that "dmatched".
matchAux(p1,p2){
  int totalMatches = 0;
  int tdistance = MAX_INT; // distance of the current candidate
  slice candidate = null;
  for each s in p2{
      for each j in p1 { // for each slice in the program p1
          int d = dmatch(s,j);
          if(d ≤ maximum_acceptable_distance ∧ d ≤ tdistance){
              tdistance = d; candidate = j;
          }
      }
      int c = min( m(candidate,p1), m(s, p2));
      totalMatches += c;
      p1 = p1 − {p1 ∗ c}; // remove from p1, c times the matched slice
  }
  return totalMatches
}
```

hours, typically 48 hours, but less than a week.

All these runs are performed in a Suse Linux 10.1 64 bit edition machine with Java 1.5 64 bit. The hardware configuration included two Xeon 64 bit 3.20 GHz processors and 4GB of RAM.

Because of all the previous reasons we have not been able to perform exhaustive tests or precision measures. The matches we have done are those for which Soot finishes within 2 or 3 days. Before optimizing our matching algorithm with techniques like spatial indexing[18] or advanced tree-to-tree correction problem algorithms [46, 41], we wanted to make sure that our ideas had some potential. Our experiments have confirmed this, and future work will focus on improving performance. Problem (3) can be corrected with an inter-procedural slice expansion approach as this would increase the size of the slices. This approach can also protect us from method in-lining obfuscation attacks. We do not include any optimization options experiments because neither JDK 1.5 or Jikes 1.22 support this feature.

Figure 6 shows two examples of preliminary results we have obtained with Furia. We created a database of 18 applications compiling each application with JDK 1.5. In the left table, we performed a matching with an application called jfreechart, compiled with Jikes 1.22 and with *ignore_slices_lower_than* = 4.

The candidate is indeed the match corresponding to "jfreechart". All other candidates have comparatively lower scores. In the right table, the same experiment is performed but this time with *ignore_slices_lower_than* = 15, even though the total score for jfreechart is reduced, the amount of candidates decreases and the matched scores for each of them are very low. Common slices tend to be smaller, and by setting *ignore_slices_lower_than* to 15 many of them get removed.

Figure 7 shows a match with a control flow obfuscated program. The table on the left shows a match for the program "freemind". This application was compiled with JDK 1.5 and control flow obfuscated with Zelix Klass Master (ZKM)[7] version 4.4. Method, class and field names are shortened. Unused classes, methods or fields are automatically removed. This obfuscator encrypts all the strings embedded in the byte-code and has the option of either to embed the string decryption in-

structions or to control obfuscate the string decryption instructions. In this case we selected the first option. Note how even after obfuscating the program, the resulting score is relatively high. However when selecting the second option(in the right table), the total score for "freemind", is substantially reduced. Still the scores of the other candidates are low enough to distinguish the match. Using Soot's static analysis framework, it should be possible to "un-obfuscate" this attack and improve the score for this case. Another way of increasing the score is to employ term rewriting rules to "normalize" the slices. A combination of both approaches might be the easiest and most powerful option to implement. Also, it might be possible to automatically or semi-automatically "learn" and generate these term rewriting rules. In the rest of the paper, programs obfuscated with ZKM are fully obfuscated and the string decryption instructions are control-flow obfuscated.

Figure 8 shows a match when using Smoke Screen obfuscator [6] version 3.43 (demo). The match is quite *clean*.

We performed another series of experiments with a bigger database set. In a real scenario it is not always possible to have all the applications compiled with the same compiler, so we created a database of 269 applications by downloading byte-code versions of different programs. We created a script that downloaded packaged Java programs from sourceforge.net. The script tries to extract all the class files from the compressed file. If there are no class files available in the compressed file, then the script tries to find a .jar file whose name is similar to the Unix-name of the program, or if the Unix-name is not available, then the name of the compressed file is used. This process tries to ensure that no dependency libraries are included within the programs. The left table in figure 9 shows the matching of the application "freesudoku". The matching percentage for "freesudoku" is 0.9. This is because the version of the program or the compiler used to create the program that is in the database is different. Note that even when the database is relatively big, part of the signature of "freesudoku" is preserved. There are no false positives.

slice_cut_threshold=30
ignore_slices_lower_than=15
maximum_acceptable_distance=3
Matching: jacksum
JDK 1.5 + smoke screen (full options)

| **App Name** | **Score** |
|---|---|
| *jacksum* | *0.804* |
| azureus | 0.086 |
| checkstyle | 0.017 |
| jgnash | 0.012 |
| jasperreports | 0.011 |
| findbugs | 0.009 |
| htmlparser | 0.007 |
| ireport | 0.006 |
| pdfbox | 0.006 |
| triplea | 0.005 |
| yale | 0.004 |
| jfreechart | 0.003 |
| schemaspy | 0.003 |
| jmemorize | 0.003 |
| smallexample | 0.003 |
| jmusic | 0.002 |
| freemind | 0.001 |
| freesudoku | 0.000 |

Figure 8: Example with Smoke Screen obfuscator

The right table in Figure 9 shows the matching of a fully obfuscated application ("jmusic") on the same database. As in the smaller database experiment, the score is low, however it is still possible to distinguish the application from the others.

Finally, in a database of 363 programs created from sourceforge.net, freshmeat.net and jpackage repositories, we perform some additional experiments. In Figure 10, we show the effects of modifying the *maximum_acceptable_distance* variable when matching obfuscated programs. Note how changing this variable from 1 to 3 will increase the score for "freesudoku". This happens because obfuscators might change some elements of the slices, but several sub components in the slices should be preserved, and that is why the score is increased substantially. Other matches are not affected noticeably.

Finally, in order to simulate an open source violation, we created a 3000 line application called "trovador". The program analyzes chord patterns in music and "corrects" chords in midi

slice_cut_threshold=30
ignore_slices_lower_than=4
maximum_acceptable_distance=1
Matching: jfreechart (Jikes 1.22)

| App Name | Score |
|---|---|
| *jfreechart* | *0.828* |
| freesudoku | 0.227 |
| htmlparser | 0.188 |
| jgnash | 0.157 |
| checkstyle | 0.115 |
| freemind | 0.109 |
| pdfbox | 0.100 |
| findbugs | 0.084 |
| triplea | 0.079 |
| jmusic | 0.076 |
| jasperreports | 0.076 |
| schemaspy | 0.057 |
| ireport | 0.049 |
| yale | 0.033 |
| azureus | 0.028 |

slice_cut_threshold=30
ignore_slices_lower_than=15
maximum_acceptable_distance=1
Matching: jfreechart (Jikes 1.22)

| App Name | Score |
|---|---|
| *jfreechart* | *0.739* |
| freesudoku | 0.009 |
| jgnash | 0.008 |
| jmusic | 0.001 |
| jasperreports | 0.001 |
| ireport | 0.001 |
| checkstyle | 0.001 |
| findbugs | 0.001 |
| yale | 0.001 |
| azureus | 0.000 |

Figure 6: Effects of changing *ignore_slices_lower_than*

slice_cut_threshold=30
ignore_slices_lower_than=15
maximum_acceptable_distance=1
**Flow obfuscate String decryption: off**
Matching: freemind
JDK 1.5 + ZKM (full)

| App Name | Score |
|---|---|
| *freemind* | *0.518* |
| checkstyle | 0.020 |
| jgnash | 0.012 |
| jfreechart | 0.006 |
| ireport | 0.004 |
| triplea | 0.004 |
| htmlparser | 0.002 |
| jmusic | 0.002 |
| azureus | 0.001 |
| findbugs | 0.001 |
| pdfbox | 0.001 |
| yale | 0.001 |
| jacksum | 0.000 |

slice_cut_threshold=30
ignore_slices_lower_than=15
maximum_acceptable_distance=1
**Flow obfuscate String decryption: on**
Matching: freemind
JDK 1.5 + ZKM (full)

| App Name | Score |
|---|---|
| *freemind* | *0.122* |
| checkstyle | 0.013 |
| jgnash | 0.012 |
| ireport | 0.006 |
| htmlparser | 0.003 |
| pdfbox | 0.003 |
| findbugs | 0.002 |
| azureus | 0.001 |
| jasperreports | 0.001 |
| jmusic | 0.001 |
| yale | 0.001 |
| jacksum | 0.000 |

Figure 7: Matching control flow obfuscated programs

| slice_cut_threshold=30 | | slice_cut_threshold=30 | |
|---|---|---|---|
| ignore_slices_lower_than=15 | | ignore_slices_lower_than=15 | |
| maximum_acceptable_distance=1 | | maximum_acceptable_distance=1 | |
| Matching: freesudoku (JDK 1.5) | | Matching: jmusic (JDK 1.5 + ZKM (full)) | |
| **App Name** | **Score** | **App Name** | **Score** |
| *freesudoku* | *0.900* | *jmusic* | *0.085* |
| JAMonAll_020106 | 0.040 | jquery-2006-Jan-07-dist | 0.030 |
| nachocalendar-0.23 | 0.015 | jreversepro-1.4.1-bin | 0.028 |
| jwebunit-1.2 | 0.013 | coinjema-0.4 | 0.025 |
| jin-2.13.1-unix | 0.009 | mobup_client_0.3.2 | 0.015 |
| ejb3unit-1.0-alpha2 | 0.009 | iHTbot-0.5.1b2 | 0.012 |
| siscweb-bin-0.32 | 0.009 | jmsn-0.9.9b2 | 0.011 |
| matharcade-1.2 | 0.007 | fitdecorator-beta0.2 | 0.009 |
| HTCommunicator_0.1 | 0.005 | jopt_csp_1-0 | 0.008 |
| transform-2.1 | 0.005 | etl-1.0-full | 0.008 |
| polliwog-bin-stable-0.5 | 0.001 | regexSearch-1_2 | 0.007 |
| esper-0.7.0 | 0.001 | jwp_v1.0_beta4_bin | 0.007 |
| Furthur175 | 0.001 | cap4j-0.1.2-beta | 0.005 |
| cayenne-1.2M10 | 0.000 | freemind | 0.005 |

Figure 9: Matching freesudoku and jmusic in a database of 269 applications

| slice_cut_threshold=30 | | slice_cut_threshold=30 | |
|---|---|---|---|
| ignore_slices_lower_than=15 | | ignore_slices_lower_than=15 | |
| maximum_acceptable_distance=1 | | maximum_acceptable_distance=3 | |
| Matching: freesudoku (JDK 1.5 ZKM full) | | Matching: freesudoku (JDK 1.5 ZKM full) | |
| **App Name** | **Score** | **App Name** | **Score** |
| freesudoku | 0.108 | freesudoku | **0.31** |
| DocSearcher-3.88 | 0.018 | DocSearcher-3.88 | 0.020 |
| jnetstream | 0.018 | jnetstream | 0.020 |
| BlinkenApplet0.7 | 0.017 | jgames-0.9.2 | 0.020 |
| jgames-0.9.2 | 0.015 | ocl4javaLib_2.1.7 | 0.010 |

Figure 10: Effects of changing *maximum_acceptable_distance*

slice_cut_threshold=30
ignore_slices_lower_than=15
maximum_acceptable_distance=3
Matching: trovador (JDK 1.5 ZKM full)

| **App Name** | **Score** |
|---|---|
| *jmusic* | *0.202* |
| ChordAssist_0.0.5 | 0.189 |
| pmd-3.3-1jpp.noarch | 0.077 |
| skink | 0.075 |
| dynamicjava-1.1.5-3jpp.noarch | 0.064 |
| catchxsl-1.2.1-3jpp.noarch | 0.059 |
| j80 | 0.057 |
| mockrunner-0.3.6 | 0.040 |

Figure 11: Matching "trovador"

songs that do not belong to a database of "beautiful" chords. The program uses the open source library jmusic[1]. We compiled and obfuscated trovador with ZKM. The results of the match are shown in figure 11. The first match indeed corresponds to "jmusic". The second match is high too. A closer look into the class files of the second application reveals that "ChordAssist" also uses jmusic. It was embedded with the class files because our class file extraction script failed to remove the dependency class files for that application. Because "ChordAssist" not only contains slices of "jmusic" but also slices of its own, the total score has to be lower.

# 6 Conclusions and future work

We have shown how an approximate program matcher can be constructed and we have empirically validated its performance. The most innovative aspect of our work is the way slices are generated and matched by using special constructs and a distance function.

The matching technique we have presented here, is "horizontal" in the sense that it does not use unexpanded slice references in order to make sure that the relationships among slices are preserved. Even though using this information may introduce false negatives, the ability to exhaustively confirm a positive match is desirable in our application domain.

Another important feature we have not implemented yet is slice normalization. We believe that it might be possible to automatically or semi-automatically learn slice normalization rules. Future research should focus on this area. Our short term goal is definitely to increase running performance in order to perform exhaustive experiments.

Our current match function needs to be refined as it does not take into account the amount of slices a program has.

Our current prototype can be fooled easily by method in-lining transformations. It is necessary to implement an inter-procedural slice expansion solution.

# 7 Acknowledgments

This research was partially supported by the Japanese Ministry of Education, Culture, Sports, Science and Technology (grant # 040506).

# 8 About the Authors

Arnoldo Muller is a M.Sc. student at the Department of Artificial Intelligence at Kyushu Institute of Technology. Before joining this department he was working for Intel. His research interests include pattern inference, computer music generation and program transformation. Takeshi Shinohara is a professor at the Department of Artificial Intelligence at Kyushu Institute of Technology. He obtained his B.S. in Mathematics from Kyoto University in 1980, and his Dr. Sci. degree from Kyushu University in 1986. His research interests are computational/algorithmic learning theory, information retrieval and approximate retrieval of multimedia data. The authors can be reached at the address 820-8502 Fukuoka-ken Iizuka-shi, Kawazu 680-4, JAPAN. Their Internet addresses are: arnoldo@daisy.ai.kyutech.ac.jp and shino@ai.kyutech.ac.jp.

# References

[1] http://jmusic.ci.qut.edu.au/.

[2] http://q-lang.sourceforge.net.

[3] http://www.fsf.org/.

[4] http://www.fsf.org/licensing/licenses/ and http://gpl-violations.org/.

[5] http://www.gnu.org/software/binutils/.

[6] http://www.leesw.com/smokescreen/.

[7] http://www.zelix.com.

[8] Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *Proc. of Usenix Annual Technical Conf.*, pages 179–190, 1998.

[9] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology - CRYPTO*, 2001.

[10] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[11] Wayne D. Blizard. Multiset theory. *Notre Dame journal of formal logic*, 30(1):36–66, 1989.

[12] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. *12th USENIX Security Symposium*, pages 169–186, 2003.

[13] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, pages 311–324, 1999.

[14] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

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

[16] R Cytron, J Ferrante, BK Rosen, and MN Wegman. Efficiently computing static single assignment form and the control flow dependence graph. *ACM Symposium on Principles of Programming Languages*, 1989.

[17] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[18] Oliver Gnther. Efficient computation of spatial joins. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 50–59, Washington, DC, USA, 1993. IEEE Computer Society.

[19] Susan 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.

[20] Justin Pappas Johnson. Economics of open source software, 2000.

[21] Asif Khalak. Economic model for impact of open source software, 2000.

[22] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–60, 2001.

[23] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. pages 255–270, 2004.

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

[25] Douglas Low. *Java Control Flow Obfuscation*. PhD thesis, University of Auckland, Auckland, New Zealand, 1998.

[26] Wise M. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.

[27] A. Monden, H. Iida, K. Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *compsac2000, 24th Computer Software and Applications Conference*, 2000.

[28] Eugene M. Myers. A precise interprocedural 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.

[29] George C. Necula. Translation validation for an optimizing compiler. *ACM Sigplan*, pages 83–95, 2000.

[30] A Pnueli, M. Siegel, and E. Singerman. Translation validation. *TACAS'98*, 1998.

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

[32] Eric Steven Raymond. Homesteading the noosphere. In *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, pages 79–135. O'Reilly & Associates, 1999. Originally appeared online in 1998.

[33] Kamiya T, Kusumoto S, and Inoue 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.

[34] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[35] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Detecting the theft of programs using birthmarks. *Information Science Technical Report*, 2003.

[36] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken 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.

[37] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of java programs. *IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 569–575, 2004.

[38] Haruaki TAMADA, Masahide NAKAMURA, Akito MONDEN, and Ken-ichi MATSUMOTO. Java Birthmarks – Detecting the Software Theft–. *IEICE Trans Inf Syst*, E88-D(9):2148–2158, 2005.

[39] Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Dynamic software birthmarks to detect the theft of windows applications. *International Symposium on Future Software Technology 2004 (ISFST 2004)*, 2004.

[40] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

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

[42] Raja Vall-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. *CASCON99*, 1999.

[43] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

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

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

[46] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, pages 754–765, 2005.