

International Conference on Information and Communication Technologies (ICICT 2014)

Plagiarism Detection for Java Programs without Source Codes

Anjali V.^{a,*}, Swapna T.R.^a, Bharat Jayaraman^b

^a*Department of Computer Science and Engineering, Amrita Vishwa Vidyapeetham, Coimbatore 641112, India*

^b*Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY 14260, USA*

Abstract

This paper presents a novel dynamic analysis approach to software plagiarism detection. Such an approach is inherently more resilient to code obfuscation techniques such as renaming of program entities, reordering of statements, etc. We develop our technique in the context of a dynamic analysis and visualization system for Java, called JIVE, but the techniques are applicable to other object-oriented languages. Our analyses are based on the execution traces of Java programs (produced by JIVE), and our experimental results confirm that this approach is both efficient and effective in detecting plagiarism of Java programs when their source codes are not available.

© 2015 Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license

(<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

Peer-review under responsibility of organizing committee of the International Conference on Information and Communication Technologies (ICICT 2014)

Keywords: Software plagiarism; dynamic analysis; call trees; key variables

1. Introduction

Software plagiarism, or code theft, is the copying of computer programs without attribution, a phenomenon that has become widespread with the advent of the internet and easy access to and transmission of software. Software plagiarism has been widespread in academia in the US and other countries^{6,10,14} and as a result several tools, such as MOSS⁴, have been developed that compute and report a measure of similarity between two programs. These tools work primarily by comparing the source codes using string-matching algorithms^{21,24}.

* Corresponding author. Tel.: + 919995556264.

E-mail address: anjaliyvylala@gmail.com

In this paper we focus on the detection of software plagiarism in the absence of source codes. In many practical scenarios, the source code of allegedly plagiarized software may not be available, and, even when it is available, the source code is often obfuscated to make it difficult to detect plagiarism. The common methods for obfuscation are renaming identifiers systematically, reordering statements, replacing code segments with syntactically equivalent segments, etc. Our proposed approach focuses on computing a measure of similarity between the *dynamic behaviors* of two programs. Since we do not examine the source code, it is resilient to source-code alteration techniques that do not affect runtime behavior. In this paper, we also consider techniques that address differences in runtime behavior arising due to source code obfuscation.

We develop our plagiarism detection approach for Java programs. Being an object-oriented language, Java programs tend to have smaller methods than traditional procedure-oriented programs and tend to have more frequent interaction between objects. Thus, in analyzing dynamic behavior, we first focus on a comparison of the call-tree structure between two programs. The similarity should be 100% if the trees are identical or the only difference between the two trees is a systematic renaming of method names. The similarity measure should tend towards 0% as the two trees diverge in structure at all levels. The call-tree is resilient to alterations in variable names and intra-method details, including statement reordering and control structure equivalences, as long as the calling structure is not altered. In order to construct the call-tree structure of a Java program, we obtain an execution trace of the program with the aid of a state-of-the-art dynamic analysis and visualization tool for Java, called JIVE (for Java Interactive Visualization Environment)^{8,11,17,20}.

This paper also addresses the case when two call-trees differ due to the presence of ‘dummy’ method calls. We define a dummy method as one that modifies only its locals or dummy field names; and, if it returns a result, it gets bound to a dummy variable. In order to identify dummy entities (variables, statements, calls), it is necessary to have an idea of the *key variables* in the program. These variables are typically the output variables of the program when these outputs are well-defined; otherwise, these have to be determined or specified by the user. By obtaining the dynamic slice²⁵ of the program with respect to the key variables of interest, using JIVE, we can isolate the sub-computation, or sub-tree, that is relevant to producing the output. The JIVE system also provides additional capabilities that aid in the detection of runtime similarity, as explained in Section 3.

Section 2 provides an overview of existing techniques; Section 3 provides an overview of JIVE; Section 4 provides the details of proposed approach; Section 5 discusses the evaluation results; Section 6 presents conclusions and areas for further work.

2. Overview of existing methods

The techniques for source code comparison originated with the string-based algorithms that were used for detecting plagiarism of ordinary English prose. Software systems often contain portions of code that are similar to other systems, and these common portions are referred to as code clones¹⁸. Detecting clones in source code has been recognized as an important issue in software analysis. Most of the existing approaches to detect plagiarism employ counting heuristics or string matching techniques to measure similarity in source code¹. Source code can be represented as graphs. Existing graph theory algorithms can then be applied to measure the similarity between source code graphs¹². Table 1 shows the overview of existing techniques.

Table 1. Overview of existing techniques

Author	Methodology	Remarks
Kustanto and Liem ¹⁹	Token string comparison	Detects copies of similar programs with fewer modifications, Fails to detect structural similarities of two programs
Mudduet al. ²²	Language aware token representation	Resilient to code transformations, Proposed querying and matching technique to detect plagiarism
Cuomoet al. ⁷	Bytecode analysis	Detect syntactically identical fragments except for variations in identifiers, literals and types, Fails to detect copied fragments with slight modifications
Arabyarmohamadyet al. ²	Coding style based detection	Similarity in style and change in regular style of coding shows

Jiet <i>et al.</i> ¹⁶	Token sequence comparison on byte-code	plagiarism. Classify professional programmers and beginners. Small changes in source code results in significant changes in byte code, Effective method for plagiarism detection in commercial java codes.
Narayanan and Simi S ²³	Fingerprinting based on distance measure approach	Language independent, Fingerprint is generated based on the identifier signature and metrics values
Feng <i>et al.</i> ⁹	Abstract syntax tree	Detect renaming of identifiers, reordering of statements, Fails to detect when a control structure is replaced by other
Limet <i>et al.</i> ¹³	Control flow analysis	The birthmark is identified based on the flow paths in the control flow graph.

There are methods based on Program Dependency Graph (PDG) which cannot detect similarities if semantics-preserving transformation is applied on the source code. Birthmarks based on dynamic analysis can also be used to detect plagiarism. Whole Program Path (WPP) birthmarks represent the dynamic control flow of a program are robust to some control flow obfuscation, but vulnerable to semantics-preserving transformations. There are variety of dynamic birthmarks based on system call, sequence of API function call and frequency of API function call. They are also vulnerable to real obfuscation techniques¹⁵. Chanet *et al.*⁵ proposed a birthmark system for JavaScript programs based on the run-time heap. The heap profiler takes multiple snapshots of the JavaScript program during execution. The graph generator generates heap graphs containing objects created during execution as nodes. Plagiarism is detected from the heap graphs of genuine and suspected programs.

3. Java interactive visualization environment

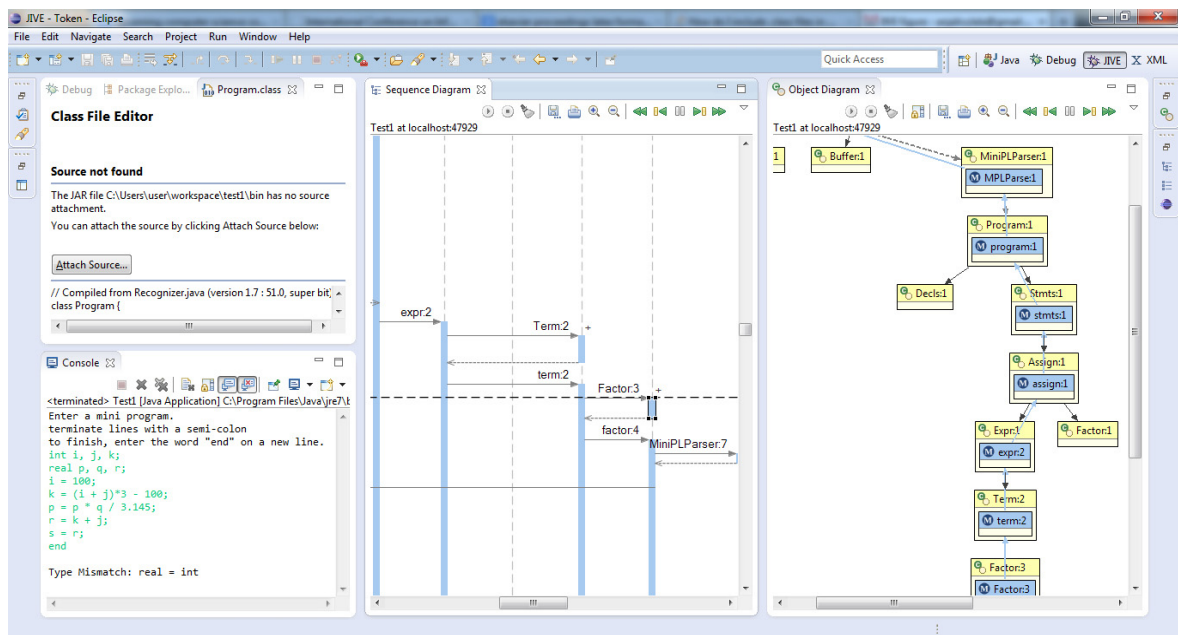


Fig. 1. A Screen-Shot of JIVE running a Java Program for an Object-Oriented Top-down Parser in the absence of Source Codes. The Object Diagram shown here corresponds to the position of the cursor (dashed horizontal line) in the Sequence Diagram. JIVE also shows Method Activations within their Object Contexts for greater clarity. The Object Diagram excludes information about fields and local variables, but these can be seen upon user request.

JIVE is an interactive dynamic analysis and Visualization system for Java. JIVE basically displays the run-time states of a Java program using enhanced object diagrams; it also summarizes the entire execution history in the form a UML-like sequence diagram, and is quite advanced over most debuggers which can at most show the stack trace.

JIVE has the ability to search over the entire execution history and can summarize executions with respect to key variables of interest. In the sequence diagram, the vertical axis is for time and the horizontal axis is for object life-lines. The diagram shows a call from one object to another by a horizontal line and depicts the duration of calls as rectangles on the object life lines (see Fig. 1) shows the object diagram at the place where the cursor is positioned in the sequence diagram.

From the perspective of plagiarism detection, two programs having identical sequence diagrams on a given input strongly indicate that one has been plagiarized from the other. Since a sequence diagram is the result of a program run on a specific input, it is in general necessary to run the programs on different inputs and check the corresponding diagrams. A sequence diagram can be represented as a call-tree, which clarifies the calling structure but abstracts the internal details of the objects on which calls are made. Thus, our first technique for software plagiarism detection is based upon tree comparison. The introduction of dummy method calls is an obfuscation technique that needs to be accommodated during tree comparison, and this detection requires dynamic slicing. The topic of program slicing has been investigated extensively in the literature, with techniques for both static and dynamic slicing as well as forward and backward slicing²⁵. We are interested here in the backward slice of the execution trace with respect to a key variable of interest. JIVE computes the backward slice by a data-flow analysis over the execution trace taking into account inter-procedural calls and object structure.

JIVE also supports the concept of query-based debugging, meaning, that it supports run-time queries through a simple form-based interface using which one ask about variables, methods, objects, exceptions, etc (see Fig. 2). These are temporal queries, for example, a simple query would be 'When did variable x first become negative?'. The result is displayed by highlighting a point on the sequence diagram, from which we can explore the object diagram at that point in execution. One can also ask for all changes to x, all invocations of a method, etc.

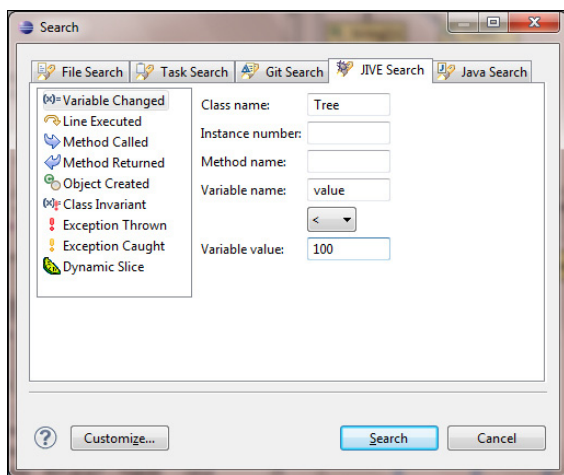


Fig. 2. JIVE Run-time Query Interface: The above query asks for all time points (and corresponding values) when the field 'value' in class 'Tree' was assigned a value less than 100.

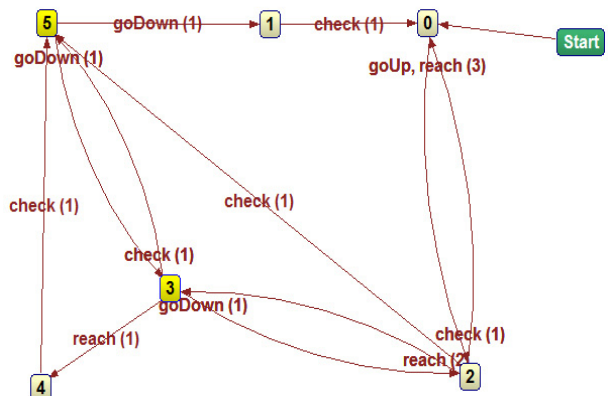


Fig. 3. JIVE State Diagram View: The floors visited by an elevator are depicted by the states in the diagram. Labels on the transitions are the names of the methods.

Run-time queries are also useful in detecting software plagiarism, as noted earlier, since we can determine the entire progression of value changes to a variable. We refer to this as the *value sequence* for a variable. JIVE provides a number of form-based queries to elicit run-time information on variables, methods, objects, exceptions, etc. A sample query form is shown in Fig. 2. Since the answers refer to time points in execution, the result of a query is typically shown by annotating the sequence diagram with the points where the answers are to be found. For the purpose of plagiarism checking, we do not need to use the external interface but we can directly tap into the query API provided by JIVE.

Another useful capability of JIVE is to construct a finite-state machine diagram given a set of key variables of interest. Fig. 3 shows a typical state diagram constructed by JIVE. Essentially, the state changes when one of the key variables changes and the method call causing the change becomes the label of the transition. JIVE has techniques for managing large state diagrams by allowing the user to specify ranges of values of interest for a key variable. This is a generalization of a technique known as *predicate abstraction* in the software model-checking context³.

4. Proposed approach

We are analyzing the dynamic behavior of source codes to capture the similarities among them. Our approach uses method calling structure and the values of key variables in order to carry out different analyses.

main	130	New Object	testing.jav...	object=testing\$1abc4
main	131	Method Call	testing.jav...	caller=testing\$1xyz1#inputxyz1, target=testing\$1abc4#1abc4
main	132	Method Entered	testing.jav...	
main	133	Line Step	testing.jav...	
main	134	Method Exit	testing.jav...	returner=testing\$1abc4#1abc4, value=<void>
main	135	Method Return	testing.jav...	
main	136	Line Step	testing.jav...	
main	137	Variable Write	testing.jav...	context=testing\$1xyz1#inputxyz1, x1=testing\$1abc4
main	138	Line Step	testing.jav...	
main	139	Method Call	testing.jav...	caller=testing\$1xyz1#inputxyz1, target=testing\$1abc4#input3
main	140	Method Entered	testing.jav...	

Fig. 4. Fragment of Execution Trace from JIVE

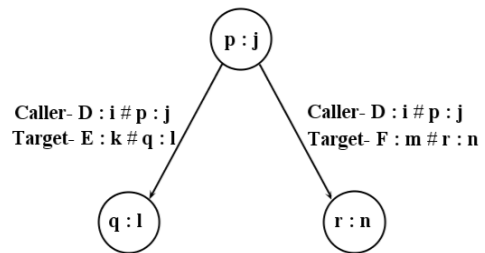


Fig. 5. Fragment of Call-Tree from Execution Trace Entry

4.1. Call-tree comparison

JIVE's execution trace (Fig. 4) collects all the events that occur during execution. The types of events are *system start*, *thread start*, *type load*, *method call*, *method entered*, *new object*, *variable write*, *field write*, *line step*, *method exit* and *method returned*. Events related to methods include details such as caller, target of the method. From the execution trace, we construct the call-tree representation by identifying the caller-callee relationships in an incremental manner. Each node in the call-tree represents a method invocation captured by JIVE. The method name and the object invoking that method are used to uniquely identify the node. Nodes in the tree are connected when a method is invoked from another method. Algorithm 1 can be used to create call-tree from the Execution Trace. Fig. 5 shows a fragment of the call-tree representation: D, E and F represent the class names; i, k and m represent objects of these respective classes; and p: j, q: l and r: n represents the method calls.

Similarities between call-trees can be assessed through aggregate statistics such as the number of method invocations (number of nodes in the tree), number of leaf nodes in the tree, and maximum depth of the tree. To gain a more accurate assessment of the structural similarity between two call-trees, we also take into account the number of common sub-trees in the two call-trees. In order to calculate this count, a level number is assigned to each of the nodes in the call-trees. For example, the root node has level 0; the children of the root node have level 1, and so on. Based on this level number, which is stored in each node, the number of common sub-trees in the call-trees can be identified. Algorithm 2 can be used to count the number of common sub-trees from the call-trees. In order to capture the correspondence between identical sub-trees where there is systematic renaming of method names, we make map each method name to an index in a table where that name is stored. Two method tables are used, one for each program. We carry out inorder traversals of the two trees and generated two sequences of indices. The two index sequences are compared to get the count of matching and non-matching method invocations using algorithm 4. From these counts, the percentage of similarity between the sub-trees can be identified. If these sequences are identical, the two call-trees are identical except for method renaming.

4.2. Detection of dummy method calls

The call-tree comparison provides a measure of similarity between two programs based on the method invocation details collected from the execution trace. As noted in the introduction, this measure is resilient to a number of

changes, such as introduction of dummy variables, control structure reorganization, and intra-procedural details. As long as the method-call structure is preserved, the technique can detect systematic renaming of method names.

Algorithm 1: TREECONSTRUCTION constructs the call-tree from an Execution Trace

Input: Execution Trace of a source code

Output: Call-tree representation of Execution Trace

```

for each method call, M, in the Execution Trace do
    Let M = < C:i # p:j, D:k # q:l >
    /* p:j in object C:I calls q:l in object D:k */
    Let n1 be the tree node for C:i # p:j;
    Create new tree node n2 for D:k # q:l;
    Create a new edge in the tree from n1 to n2
End

```

Algorithm 3: RENAMINGIDENTIFICATION returns the similarity score

Input: Common sub-tree pair, (Tree1, Tree2)

Output: Similarity Score

```

Let L1 = inorder(Tree1); Let L2 = inorder(Tree2);
Let T1, T2 = empty table; Let M1, M2 = empty table;
for each call c in L1 do

```

```

    Let c = p:i;
    If p is not in table T1 then
        Add p to end of T1 at index k;
        Add k to the method index table M1;
    end
    else
        Let k be the index of p in T1;
        Add k to the method index table M1;
    end
end

```

```

for each call c in L2 do
    Let c = p:i;
    If p is not in table T2 then
        Add p to end of T2 at index k;
        Add k to the method index table M2;
    end
    else
        Let k be the index of p in T2;
        Add k to the method index table M2;
    end
end

```

```

score = SIMILARITY(M1, M2);

```

Algorithm 2: COMMONSUBTREECOUNT returns the count of common sub-trees from the two call-trees

Input: Call-trees of two Execution Trace; Ctree1 and Ctree2

Output: Number of common sub-trees in both call-trees

```

Let D1 = Maximum depth of CTree1;
Let D2 = Maximum depth of CTree2;
Let D = minimum(D1, D2);
for each node n1 in CTree1 do
    V1 ← postorder(Tree1) /* Tree1 is the sub-tree rooted
        at n1 and traversal is based on level number
        assigned */

```

```

    n1:label ← V1;

```

```

end

```

```

for each node n2 in CTree2 do

```

```

    V2 ← postorder(Tree2) /* Tree2 is the sub-tree rooted
        at n2 and traversal is based on level number
        assigned */

```

```

    n2:label ← V2;

```

```

end

```

```

for i ← 0 to D do

```

```

    Create a vector LVi;

```

```

    for each node n1 in level i of CTree1 do

```

```

        Add n1.label to LVi /* n1.label represents sub-
        tree rooted at n1 */
    end

```

```

    end

```

```

end

```

```

for i ← 0 to D do

```

```

    for each node n2 in level i of CTree2 do

```

```

        if n2.label is present in LVi then

```

```

            if size of n2.label > 2 then

```

```

                Increment commonSubtreeCount by 1;
            end
        end
    end

```

```

    end

```

```

end

```

```

end

```

```

return commonSubtreeCount;

```

However, as the experimental results in Table 2 shows, when there are many dummy method calls in the plagiarized program, its call-tree structure will diverge significantly from the genuine program. Hence we need to determine such dummy method calls. We define a *dummy method* as one that modifies its local variables and if it modifies any field names, those must necessarily be dummy field names. When a dummy method returns a value, it must necessarily get bound to a dummy variable (local variable of dummy method or a dummy field name). A dummy call, i.e., call to a dummy method, may also make calls to other dummy methods. Thus, there can be a rather elaborate obfuscation making use of dummy field names and dummy methods.

Algorithm 4: SIMILARITY(M1, M2) returns percentage of correspondence identified from M1 and M2
Input: Method index table M1 and M2
Output: Similarity Score
 Let *match* = *notMatch* = 0;
for *i* ← 0 to *M1.size* **do**
 if M1[*i*] == M2[*i*] **then**
match = *match* + 1;
 end
 else
notMatch = *notMatch* + 1;
 end
end
score = *match*/(*match* + *notMatch*);
return *score*

Table 2. Experimental results for tree comparison.

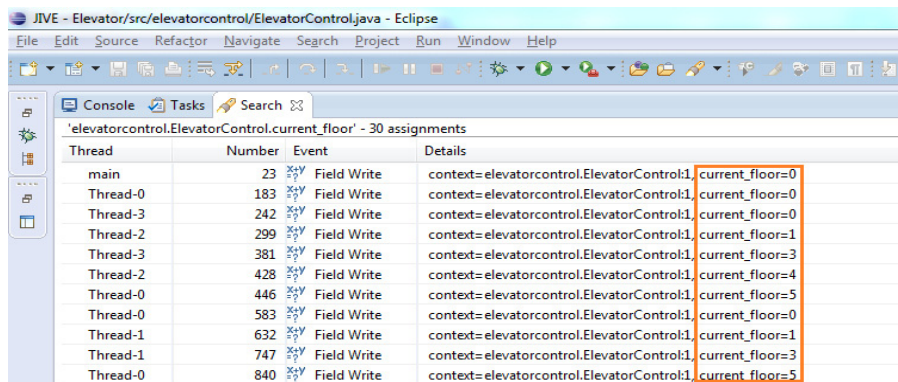
No.	Obfuscation Technique	Matching score in %
1	Rename Methods, Add dummy variables	100
2	1 + Change Control Structures (use equivalent control)	100
3	2 + Few Dummy Calls in Separate Sub-trees	75-90
4	2 + Many Dummy Calls at Higher-Levels of Tree	30-50
5	2 + Many Dummy Calls at Lower-Levels of Tree	0-20
6	2 + Many Dummy Calls at All Levels of Tree	0-5

In order to identify dummy entities (fields, methods), it is necessary to have an idea of the relevant variables in the program. We refer to them as *key variables*. These are typically the output variables of the program when such outputs are well-defined; otherwise, these have to be determined or specified by the user. The basic observation is that the key variables will depend directly or transitively, and possibly through a chain of method calls, only other relevant variables but not on any dummy field names. Therefore, through an inter-procedural backward flow analysis on execution trace starting from some output variable (key variable), we can determine the relevant sub-computation to obtain this output. We refer to this relevant computation as the dynamic slice of the program with respect to the key variable. By taking union of the dynamic slices for all key variables and also for different test cases, we can determine the relevant method calls. The remaining method calls are dummy method calls.

4.3. Value-sequence comparison for key variables

Even though dummy calls are detected, it is possible that plagiarism can be achieved by alterations in the method calling structure that preserve the semantics of the genuine program. For example, a genuine method call (i.e., not a dummy call) might have been in-lined or a genuine method is split into two. In these cases, the call trees would look different. In order to handle such cases, we make use of the key variables in the genuine program. For example, in the Elevator application mentioned earlier, 'current_floor' is a key variable. The value of this variable changes depends on the location of the elevator. Fig. 6 shows the sequence of values obtained from JIVE. If the user knows the key variables in the genuine program, then the corresponding key variables of the suspected program can be identified as follows. We first determine for each of the key variables in the genuine program, the sequence of value changes for that variable. This can be done with JIVE's query feature on variables in a direct way. We then obtain the value sequences for all variables in the suspected program. Through a comparison of these sequences with the

sequences in the genuine program, we determine which the corresponding sets of key variables are in the suspected program. We do not need to examine the entire value sequences in the suspected program if there are large differences in the initial prefixes. Only if the suspected program is a plagiarized program, the comparison of the value sequences will yield a good match. Algorithm 4 can be applied to determine the degree of matching between two value sequences and provides another measure of similarity between two programs based on the key variables identified from those programs.



The screenshot shows the Eclipse IDE with the JIVE plugin. The 'Console' view is open, displaying a table titled "'elevatorcontrol.ElevatorControl.current_floor' - 30 assignments". The table has four columns: Thread, Number, Event, and Details. The 'Details' column contains two parts: a context string and a value for 'current_floor'. The values of 'current_floor' are highlighted in orange in the original image.

Thread	Number	Event	Details
main	23	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=0
Thread-0	183	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=0
Thread-3	242	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=0
Thread-2	299	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=1
Thread-3	381	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=3
Thread-2	428	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=4
Thread-0	446	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=5
Thread-0	583	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=0
Thread-1	632	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=1
Thread-1	747	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=3
Thread-0	840	Field Write	context= elevatorcontrol.ElevatorControl:1, current_floor=5

Fig. 6. Fragment of values of variable current floor from JIVE

4.4. State diagram comparison

The two strategies described earlier can be used to compute similarity measures between genuine and suspected program based on either method invocation or value sequence of key variables. In this section we are proposing a strategy based on JIVE's state diagram view which depicts both method invocations and values of key variables in a single diagram. It provides a more concise view of dynamic behavior program. The states represent values the key variable can take. The state transition from one state to other occurs when a method invocation changes the value of the key variable. Fig. 3 shows an example for state diagram generated by JIVE for the Elevator program. If two programs are having similar method invocation structure and similar value sequence for corresponding key variables, then the state diagrams can be utilized to measure a higher level of similarity. Isomorphic state diagrams of both genuine and suspected program give a similarity measure to conclude whether the suspected program is plagiarized or not.

5. Evaluation

5.1. Dataset

The system was tested on a suite of 60 Java programs. A variety of different programs were represented such as sorting, parsing, and simulations of various devices and scenarios such as an elevator, vending machine, traffic light, etc. Programs with design patterns such as iterator, composite, etc., were also included. Some applications, such as parsing, were implemented by more than one person and hence there were multiple versions of this program. The programs made use of a variety of Java features, including object-oriented features such as inheritance as well as the familiar control structures.

The .class files for these programs were generated by running them through the Java compiler. The execution traces from JIVE for each of the .class files were exported into XML format. Next, we randomly selected 20 projects from the test suite and manually obfuscated the programs; methods were renamed systematically and dummy calls were added. Other obfuscations performed on the programs included inserting dummy variables and assignment statements, replacing code segments with equivalent statements using control-structure transformation, etc. Our subsequent test was carried out on 20 genuine and 20 obfuscated programs.

5.2 Implementation and results

In order to test our approach, we first developed a plagiarism detection system based on the call-tree comparison described in Section 4.1. This approach consists of a *call-tree generation* module, a *metrics calculation* module and a *tree comparison* module. By testing our system against various obfuscation techniques, we see that statically dissimilar source codes showing similarities in their dynamic behavior can be captured using the proposed approach. Since our approach is based on methods and the interactions between them, it is well suited for detecting plagiarism in object oriented framework.

We obtained 100% accurate results when there was obfuscation through systematic renaming of method names. This result holds even with reordering of parameters, introduction of dummy variables and assignment operations in the programs and also replacement of control structures by their equivalents. The call-tree comparison technique is able to detect similarity only in presence of a few dummy method calls occurring near the top of the call-tree, especially as a separate sub-tree included in other sub-trees except the root. However, if the program structure is destroyed by obfuscating with a large number of method calls, the call-tree comparison approach fails to detect similarity. This approach is vulnerable if the method calls are occurring in the intermediate levels of the call-tree. Table 2 summarizes the difference in call trees due to the presence of dummy calls at various levels.

In order to overcome the above limitation, we tested the use of *dynamic slicing* with key variables. We found this gives more accurate result when dummy method calls were present. Dynamic slicing is able to eliminate dummy calls. When the state diagram for the key variables is constructed, it also excludes dummy method calls since, by definition, they do not make any changes in the values of key variables. Hence, we observe better results in these cases. But similarity identification is difficult when dummy method calls that alter the values of key variables. These kinds of dummy calls introduce anonymous states in the *state diagram*, which makes the state diagram of suspected program different from that of the genuine program. Of course, altering the key variables alters the meaning and hence the resulting program is no longer equivalent.

6. Conclusion and future work

We have analyzed plagiarism of Java programs by comparing their execution behaviors. Our approach does not rely on source code directly and works when source code is unavailable or it is obfuscated. We were able to deal with method renaming, introduction of dummy variables, assignments, and methods, as well as reordering of statements and replacement of control structures by their equivalents. We carried our analyses in the context of the JIVE system, making use of its execution trace, dynamic slicing and query capabilities, as well as state diagram construction. We have analyzed only single-threaded Java programs and we plan to extend our work to multi-threaded programs as well. Multi-threaded program execution gives rise to multiple call-trees for a single program, and hence it is necessary to establish correspondence between two sets of call-trees. Also, we need to integrate our proposed techniques into one comprehensive system that can be used in a stand-alone mode and also test more extensively on larger programs.

Acknowledgements

The authors would like to thank their respective universities for their support and infrastructure.

References

1. Ali AMET, Abdulla HMD, Snasel V. Survey of plagiarism detection methods. *Asia Modelling and Symposium* 2011. p.39-42.
2. Arabyarmohamady S, Asadpour M, Moradi H. A coding style based plagiarism detection. *Intl. Conf. on Interactive Mobile and Computer Aided Learning* 2012. p.180-186.
3. Ball T, Majumdar R, Millstein T, Rajamani SK. Automatic predicate abstraction of C programs. *ACM SIGPLAN Not.* 2001. p.203-213.
4. Bowyer KW, Hall LO. Experience using MOSS to detect cheating on programming assignments. *Frontiers in Education Conference* 1999;3. p.18-22.
5. Chan PPF, Hui LCK, Yiu SM. Heap graph based software theft detection. *IEEE Trans. On Information Forensics and Security* 2013;8. p.101-110.
6. Cosma G, Joy M. Towards a definition of source-code plagiarism. *IEEE Trans. on Education* 2008;51. p.195-200.

7. Cuomo A, Santone A, Villano U. A novel approach based on formal methods for clone detection. *IEEE IWSC* 2012. p.8-14.
8. Czyz JK, Jayaraman B. Declarative and visual debugging in Eclipse. In *Proc. OOPSLA Workshop on Eclipse Technology eXchange* 2007. p.31–35.
9. Feng J, Cui B, Xia K. A Code Comparison Algorithm Based on AST for Plagiarism Detection. *Intl.Conf. on Emerging Intelligent Data and Web Technologies* 2013;4. p.393-397.
10. Garcia A, Rodriguez S, Pedraza JL, Rosales F, Mendez R, Nieto MM. Detection of plagiarism in programming assignments. *IEEE Trans. on Education* 2008. p.174–183.
11. Gestwicki PV, Jayaraman B. Methodology and architecture of JIVE. In *ACM Symp. on Software Visualization* 2005. p.95–104.
12. Graves JA. *Source code plagiarism detection using a graph-based approach*. Master's thesis. TennesseeTech. Univ., 2011.
13. Lim H, Park H, Choi S, Han T. A method for detecting the theft of java programs through analysis of the control flow information. *Information and Software Technology* 2009;51. p.1338-1350.
14. Inoue U, Wada S. Detecting plagiarisms in elementary programming courses. *Intl. Conf. on Fuzzy Systems and knowledge Discovery* 2012. p.2308-2312.
15. Jhi YC, Wang X, Jia X, Zhu S, Liu P, Wu D. Value-based program characterization and its application to software plagiarism detection. *Intl. Conf. on Soft. Engg.* 2011. p.756-765.
16. Ji JH, Woo G, Cho HG. A plagiarism detection technique for java program using bytecode analysis. *Intl. Conf. on Convergence and Hybrid Information Technology* 2008;1. p.1092-1098.
17. Jayaraman S, Kishore Kamath D, Jayaraman B. Towards execution summarization of Java programs: From sequence to state diagrams. In *International Conference on Contemporary Computing*. IEEE XPlore, 2014. p.299-305.
18. Koschke R. Frontiers of software clone management. *Frontiers of Software Maintenance* 2008. p.119-128.
19. Kustanto C, Liem I. Automatic source code plagiarism detection. *ACIS Intl. Conf. on Software Engg., Artificial Intelligences, Networking and Parallel/Dist. Computing* 2009;10. p.481-486.
20. Lessa D, Jayaraman B. Explaining the dynamic behavior of java programs (abstract). In *ACM SIGCSE* 2012. p.668-668.
21. Liu C, Chen C, Han J, Yu PS. Gplag:Detection of software plagiarism by program dependence graph analysis. *Knowledge Discovery and Data mining* 2006. p.872-881.
22. Muddu B, Asadullah A, Bhat V. CPDP: A robust technique for plagiarism detection in source code. *IEEE IWSC* 2013. p.39-45.
23. Narayanan S, Simi S. Source code plagiarism detection and performance analysis using finger print based distance measure method. *Intl. Conf. on Computer Science and Education* 2012. p.1065-1068.
24. Lutz Prechelt, Guido Malpohl, Michael Philippsen. Jplag: Finding plagiarism among a set of programs. *Journal of Universal Computer Science* 2002. p.1016-1038.
25. Frank Tip. A survey on program slicing techniques. *Journal of Programming Languages*1995. p.121–189.