Code Relatives: Detecting Similar Software Behavior

Fang-Hsiang Su, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser and Tony Jebara
Columbia University
500 West 120th St, MC 0401
New York, NY USA
{mikefhsu, harvey, simha, kaiser, jebara}@cs.columbia.edu

ABSTRACT

Detecting "similar code" is fundamental to many software engineering tasks. Current tools can help detect code with statically similar syntactic features (code clones). Unfortunately, some code fragments that behave alike without similar syntax may be missed. In this paper, we propose the term "*code relatives*" to refer to code with dynamically similar execution features. Code relatives can be used for such tasks as implementation-agnostic code search and classification of code with similar behavior for human understanding, which code clone detection cannot achieve. To detect code relatives, we present DyCLINK, which constructs an approximate runtime representation of code using a dynamic instruction graph. With our link analysis based subgraph matching algorithm, DyCLINK detects fine-grained code relatives efficiently. In our experiments, DyCLINK analyzed 290+ million prospective subgraph matches. The results show that DyCLINK detects not only code relatives, but also code clones that the state-of-the-art system is unable to identify. In a code classification problem, DyCLINK achieved 96% precision on average compared with the competitor's 61%.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: [Distribution, Maintenance, and Enhancement]; I.1.5 [**Pattern Recognition**]: [Clustering]

General Terms

Algorithms, Experimentation, Design

Keywords

Code relative, runtime behavior, link analysis, subgraph match, code clone

1. INTRODUCTION

Code clones [37], which represent textually or syntactically similar code fragments, have been widely adopted to detect similar software. However, code clone detection systems focus on identifying static patterns in code, so some relevant code fragments that behave similarly at runtime, though with different syntax, are missed. Detecting code fragments that accomplish the same tasks or share similar behavior is pivotal for understanding, and improving the performance of software systems. With such functionality, it is possible to automatically replace an old algorithm in a legacy system with a new one. It also allows quick search and understanding of large codebases, and deobfuscation of code. In general, identifying similar code functionality or behavior is difficult, because it involves understanding the semantics of code fragments [32].

To represent runtime similarity in software, we introduce the concept of *Code Relatives*. Code relatives are continuous or discontinuous code fragments that exhibit similar behavior, but may be expressed in structurally or even conceptually different ways. Code fragments that have a characteristic routine, say a unique linear algebra function, are code relatives due to the fact they execute their functions using similar operations. They are still code relatives, in spite of differences in implementation, data structures, or coding style. By our definition, existing dynamic approaches that detect code fragments with similar behavior [19, 14] at different levels, such as output values and sequence of method calls, are all code relative detection techniques.

In this paper we present our system, DyCLINK, which detects code relatives with fine granularity. DyCLINK traces a program's execution, and constructs a dynamic instruction graph, which encodes denser behavioral information than is found in the program's sequence of method calls or in its functional I/O. To effectively identify code relatives, we apply link analysis on the instruction graph, exposing the program's core behavior. Specifically, we have developed a new algorithm, *LinkSub*, which mitigates the prohibitive time complexity of subgraph matching in program analysis. LinkSub treats the dynamic instruction graph as a network, and ranks the nodes via the PageRank algorithm [28] to identify the most important ones. The important nodes form the *centroids* of the dynamic instruction graphs, which help in selecting candidate nodes for subgraph matching. The use of link analysis not only reduces the cost of traditional graph isomorphism detection, but also produces program representations independent of how the computations are expressed in the code.

We choose Java [22] as the exemplary programming language in this paper, but our methodology applies to most high level languages. By analyzing 7 libraries, for which execution benchmarks are available, and 118 Google Code

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

Jam projects, we find promising code relatives across the codebases efficiently. To the best of our knowledge, no existing subgraph matching algorithm can handle such high order dynamic instruction graphs. The main contributions in this paper are:

- We define *Code Relatives* (code sharing similar dynamic behavior) and their utility.
- We design and implement the DyCLINK system to detect code relatives with fine granularity. DyCLINK uses dynamic instruction graphs for identifying code relatives.
- The key to our scalability and effectiveness is our use of link analysis on the dynamic instruction graphs. We devise the LinkSub algorithm, which efficiently solves the subgraph isomorphism problem for programs with thousands of instructions and dependencies.
- We present a highly-accurate method for classifying programs, by running the K Nearest Neighbors (KNN) [1] algorithm among code relatives.

2. BACKGROUND

Before discussing the details of DyCLINK, we first define the key terms used in this paper and discuss some use cases of code relatives.

2.1 Basic Definitions

- **Code clone:** We quote the definition from Roy *et al.* [37]: "A code fragment CF_2 is a clone of another code fragment CF_1 if they are similar by some given definition of similarity." We express Roy's *et al.* definition as follows. CF_1 and CF_2 are code clones if:

$$Sim(CF_1, CF_2) \geq thresh \quad (1)$$

  , where Sim is a similarity function and $thresh$ is a pre-defined threshold.

- **Code skeleton:** Either a continuous or discontinuous set of code lines.
- **Code relative:** An execution of a code skeleton, CS, generates some behavioral representation, $Exec(CS)$, of that skeleton. Any behavioral representation, such as output values, may be used in detecting code relatives. In DyCLINK, we choose a dynamic instruction graph as the behavioral representation. Given a Sim and a $thresh$, two code skeletons, CS_1 and CS_2, are code relatives if Eq. 2 holds.

$$Sim(Exec(CS_1), Exec(CS_2)) \geq thresh \quad (2)$$

Four types of code clones have been identified [21, 5, 6, 25, 23, 29, 18, 24, 26, 30, 37]. The most advanced one, "Type 4" [37], represents code fragments that are functionally similar. These are close to code relatives, however, as per on the study by Roy *et al.*, Type 4 clone detectors are still static. This implies that they only *approximate* program behavior based on source code. In contrast, code relatives are programs that exhibit similar *real* runtime behavior. This is the reason that we separate code relatives from the four existing types of code clones.

2.2 Motivation

Detecting similar programs is beneficial in supporting several software engineering tasks: helping developers understand and maintain systems [32], identifying code plagiarism [30], and enabling API replacement [26]. Although code clone detection systems can efficiently detect syntactically similar code fragments, they may still miss some cases for optimizing software and/or hardware that require information about runtime behavior [12]. We know that programs which have syntactically similar code fragments usually have similar behavior; however, our hypothesis is that programs can still have similar behavior even if their code is not alike.

Software clustering and *Code search* are two domains that require detecting similarity between programs. Software clustering aims to locate and aggregate programs having similar code or behavior. The clusters support developers understanding code semantics [27, 31], prototyping rapidly [8], and locating bugs [13]. Code search helps developers determine whether their codebase contains programs/APIs befitting their requirements [32]. In general, a code search system takes program specifications as the input, and returns a list of programs, ranked by their relevance to the specification.

Software clustering and code search can be based on static and/or dynamic analysis. Static analysis relies on features such as usage of APIs to *approximate* the behavior of a program. Dynamic analysis identifies traits of executions, such as input/output values and sequences of method calls to represent the *real* behavior. If we can develop a system, which captures more details and represents program behavior more effectively, then we can more precisely detect similar programs in support of both software clustering and code search. Based on the use cases above, instead of identifying static code clones, we have designed a system to detect dynamic *Code Relatives*, which represent similar runtime behavior between programs.

Our approach, DyCLINK, which will be discussed in § 3, detects code relatives with fine granularity at the instruction level. We will answer the following research questions regarding DyCLINK in this paper:

- **RQ1:** Can DyCLINK identify a greater number of similar programs than the state-of-the art code clone detection system?
- **RQ2:** Are the code relatives detected by DyCLINK more precise for classifying relevant programs than are the clones found by the state-of-the-art system?

3. DYNAMIC CODE RELATIVE DETECTION BY LINK ANALYSIS

3.1 System Architecture

The high-level procedure of DyCLINK is shown in Figure 1. DyCLINK has two major components, Graph Construction and Subgraph Crawling. The graph constructor first instruments input methods and inserts an instruction recorder at the beginning of each one. In this paper, we have selected Java [22] as our target language, so the instructions recorded by DyCLINK are Java bytecodes. If the instruction invokes another method, the recorder recursively collects the graph of the invoked method. Immediately before returning from the current method, the graph constructor merges all recorded instructions, dependencies, and recursively-collected

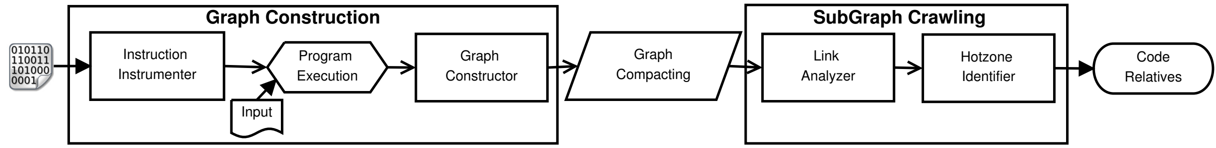


Figure 1: The high-level architecture of DyCLINK including instruction instrumentation, graph construction, graph compacting, link analysis and final hot zone (subgraph) identification.

graphs to construct a full, representative graph of the current method. Once all the representative graphs are generated for a given codebase, the data is passed to the subgraph crawling module.

The search for code segments with high similarity, then, can be modeled as the search for isomorphic subgraphs of their representative instruction graphs (identified by Garey *et al.* as an NP-Complete problem [16]). In this paper, we present a *link analysis-based subgraph isomorphism solver*, **LinkSub**, which can solve this problem in $O(V_{ta} * (\log V_{ta} + V_{te}^2 + E_{te}))$ time, where V_{ta} represents the vertex number in a "target" method (a haystack) and V_{te} and E_{te} represent the vertex and edge numbers in a "testing" method (a collection of needles), respectively. The details of this algorithm are discussed in § 3.5.

3.2 Java Instructions

Before describing the technical details of DyCLINK, we must first discuss the Java instructions (bytecodes) [22], which are fundamental in constructing the representative graph of a program. The source code of a Java program is first compiled into a sequence of Java instructions. The JVM reads each instruction into its stack machine and then performs computations based on the specifications of these instructions. Take the instruction `iadd` in the `mult()` method in Figure 2 as an example. The `iadd` in line 9 takes two integer values, loaded by `iload 3` and `iload 1` onto the JVM's stack, adds them, and puts the sum back onto the stack.

DyCLINK shadows the JVM's stack machine to derive the dependencies between instructions. In the `iadd` example, the `iadd` instruction depends on `iload 3` and `iload 1`. In addition to dependencies based on the Java specification, DyCLINK also considers read-write and control dependencies between instructions. Reader instructions read variable values written by writer instructions, and control instructions decide which instructions are executed after themselves. Table 1 lists the pertinent instructions of these three types.

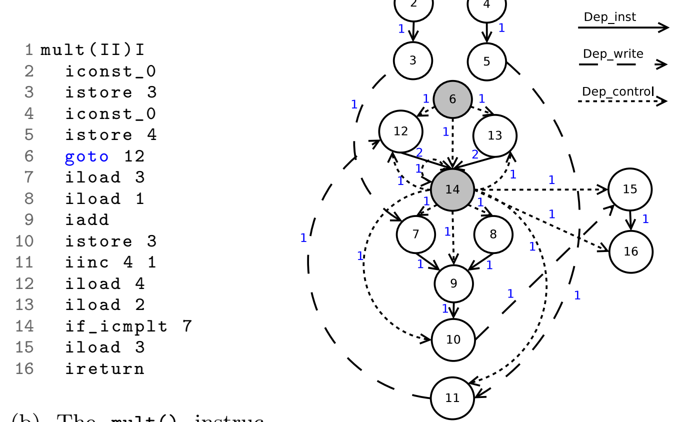
Reader	`iload`, `lload`, `fload`, `dload`, `aload`, `iinc`
Writer	`istore`, `lstore`, `fstore`, `dstore`, `astore`, `iinc`
Control	`if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpge`, `if_icmpgt`, `if_icmple`, `if_acmpeq`, `if_acmpne`, `ifeq`, `ifne`, `iflt`, `ifge`, `ifgt`, `ifle`, `ifnull`, `ifnonnull`, `goto`, `tableswitch`, `lookupswitch`
Method	`invokevirtual`, `invokestatic`, `invokespecial`, `invokeinterface`

Table 1: Reader, writer, control and method instructions (bytecodes) in the Java Virtual Machine.

DyCLINK observes the three types of instructions mentioned above to compute dependencies. However, when a caller method invokes the callee method, DyCLINK merges the callee's graph into the caller's, instead of just recording the instruction dependencies. More technical details about the graph merging between methods is discussed in § 3.4.2. One other method invocation instruction, `invokedynamic`, is rarely used by developers directly and will not be discussed further in this paper, though we mention it here for completeness.

```
1 int mult(int a,int b) {
2   int ret = 0;
3   for(int i = 0;i < b;i++) {
4     ret += a;
5   }
6   return ret;
7 }
```

(a) The `mult()` method



(b) The `mult()` instructions.

(c) The `mult` graph.

Figure 2: The exemplary method `mult()`, its corresponding instructions (bytecodes), and its representative graph.

3.3 Graph definition

A labeled graph, G, is defined [33] as:

$$G = \{V, E, l_V, l_E\} \tag{3}$$

where V represents a set of nodes (vertices) in a graph and $E \subseteq V \times V$. l_V and l_E are two mapping functions, which project a vertex and an edge to a possible vertex label and edge label, respectively. Based on this definition of labeled graphs in general, we define a dynamic instruction graph G_{dig} to be a directed, weighed, labeled graph of the following form:

$$G_{dig} = \{V_{inst}, E_{dep}, l_{V_{inst}}, l_{E_{dep}}\} \tag{4}$$

Each vertex $v \in V_{inst}$ is derived from an instruction in the input program and can be mapped to that instruction's bytecode by the function $l_{V_{inst}}$. Each edge $e_{i,j} \in E_{dep} = (v_i, v_j)$,

where $v_i, v_j \in V_{inst}$ are derived from instructions which have at least one type of dependency between them. The label for such an edge is a tuple consisting of the dependency type(s) and their weighted frequencies over the two nodes. More precisely:

$$l_{E_{dep}}(v_i, v_j) = (dep_{i \to j}, wFreq(dep_{i \to j}, i, j)) \quad (5)$$

where $dep_{i \to j}$ is the set of dependency types between v_i and v_j, and $wFreq()$ maps a set of dependency types to their weighted frequencies over two instructions. In DyCLINK, we define three types of dependencies $\{dep_{\texttt{inst}}, dep_{\texttt{write}}, dep_{\texttt{control}}\}$, each of which has its own individual weight, which is configurable. The definition of the weighted frequency between v_i and v_j is as follows:

$$wFreq(dep_{i \to j}, i, j) = \sum_{dep \in dep_{i \to j}} dep.\texttt{weight} * freq(dep, V_{inst_i}, V_{inst_j})$$

where $freq(dep, V_{inst_i}, V_{inst_j})$ records how many times dep occurs between the instructions corresponding to V_{inst_i} and V_{inst_j} during the execution of their containing method.

3.4 Graph Construction

Graph construction in DyCLINK is similar to that in [39]. Again, we use the `mult()` method in Figure 2 as an example. Each bytecode instruction from Figure 2b has a corresponding vertex in Figure 2c. The details of each edge (dependency) type are:

- $dep_{\texttt{inst}}$: An instructional dependency defined by the JVM Specification [22].
- $dep_{\texttt{write}}$: A data dependency between the writer instruction and the corresponding reader instruction. This type of dependency is computed by DyCLINK. All reader and writer instructions are recorded in Table 1. We do not include read/write instructions for array elements, because they are modeled as $dep_{\texttt{inst}}$.
- $dep_{\texttt{control}}$: A control dependency, as defined by DyCLINK, is different from the traditional definition, which records all instructions that fall under the label pointed to by the control instruction. Instead, DyCLINK computes the transitional probability from a control instruction to each of its successors, so every instruction executed after a control instruction is considered to be one of its dependents, up until the next control instruction appears.

To record executed instructions in a method and generate the corresponding graph representation, we develop a method recorder within DyCLINK. This method recorder also computes each type of dependency between instructions. DyCLINK injects this method recorder at the beginning of each method, which requires Java bytecode instrumentation. We implement our instrumenter upon the ASM framework [3]. DyCLINK's method recorder dumps a representative graph (G_{dig}) of the current method right before the return of a method.

3.4.1 Instruction-to-graph Construction

For demonstrating how DyCLINK constructs the G_{dig} for a method, we keep using the `mult()` method in Figure 2 as an example. We use $\{a = 8, b = 1\}$ as the input arguments to drive the `mult()` method. We will use the line number of each instruction as the ID for it. Take `iload 3` with ID 7 as an example. This instruction will load the integer value of the #3 local variable on the stack. When this instruction is executed, the control instruction is `if_icmplt 7` with ID 14, so the dependency $dep_{\texttt{control}}(14, 7)$ is constructed. Because `iload 3` is a reader instruction that loads the #3 local variable, DyCLINK checks the latest writer instruction of the #3 local variable, which is `istore 3` with ID 3. The dependency $dep_{\texttt{write}}(3, 7)$ is constructed. `iadd` with ID 9 has two $dep_{\texttt{inst}}$ from `iload 3` and `iload 1`, because it consumes two values based on the JVM specification.

Based on this concept, the G_{dig} of the `mult()` method can be constructed as Figure 2c depicts. Each instruction is a vertex in the graph numbered by the instruction ID. Each edge is a dependency between two instructions. The number for each edge is its $wFreq(dep_{i \to j}, i, j)$. In this paper, we set the weighted number to 1 for each dependency type.

3.4.2 Graph Merging: Instruction Set Integration between Caller and Callee Methods

When a method (caller) invokes another method (callee), DyCLINK retrieves the information of the callee graph and store it in the caller. The example in Figure 3 demonstrates the concept of how DyCLINK tracks the information of the callee method. The caller, **`methodA`**, invokes a callee, **`methodB`**. Before the end of a method, the method recorder serializes the representative graph (G_{dig}), and registers the graph ID of the current method in a global register, which tracks every method graph in the execution session. After invoking **`methodB`**, **`methodA`** uses the recorder to retrieve the graph ID of **`methodB`** from the global register and store this information for merging purposes. The final G_{dig} of **`methodA`** contains its own method and the G_{dig} from **`methodB`**.

The goal for merging is to construct the full graph of the method that contains its own instructions and all instructions from its callee methods. Because DyCLINK records the execution frequency of each callee method for the caller, connections between instructions across methods can be stored for crawling inter-method code relatives.

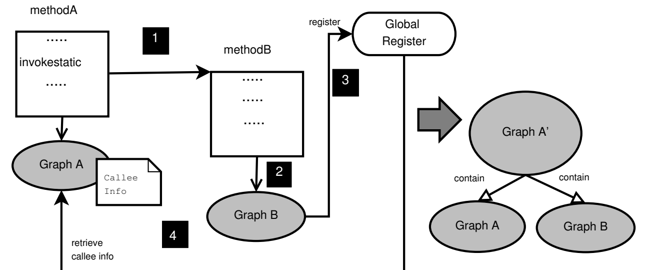


Figure 3: Graph merging of caller and callee methods.

3.4.3 Graph Compacting

Once graph merging is completed for a method, DyCLINK compacts the G_{dig}s. For each caller method, DyCLINK keeps information of each callee method. However, if a callee method is invoked thousands of times, the graph size of the caller graph will be tremendous, because each method execution generates a G_{dig}. Recording each callee G_{dig} hinders the final analysis of graph similarity. Furthermore, what we care about is the execution frequency of the same type of

G_{dig}s from a callee method. How to efficiently classify G_{dig}s from the same method is a problem that we need to tackle.

We propose to group all G_{dig}s derived from the same callee method by their vertex numbers and edge numbers. In a group, all G_{dig}s have the same number of vertices and edges. When DyCLINK tries to record the information of a callee method in the caller, it first checks if the corresponding group already exists in the caller by comparing the vertex and edge numbers of the current callee. If it does not exist, a new group is created and the current callee G_{dig} becomes the representative of this group. If it exists, DyCLINK retrieves the representative of this group and increases the execution frequency of this representative. Our assumption is that if the G_{dig}s from the same callee method have the same numbers of vertex and edge, they share the same features and should be in the same group.

3.5 LinkSub: Link-analysis-based Subgraph Isomorphism Algorithm

After the graph construction is done, DyCLINK starts to detect potential code relatives in the codebase. DyCLINK enumerates every pair of methods in the codebase: given n methods, there are at most $n * (n-1)$ method pairs. Both methods in a method pair play as the target method and the the testing one exactly once. Since the target and testing methods can both be represented as graphs, we can model our code relative detection as a subgraph isomorphism problem:

Problem 1. *Given two graphs,* $G1 = (V, E), G2 = (V', E')$, *does* G_1 *have a subgraph* $G1_s \cong G_2$ *where* $G1_s = (V_s, E_s)$: $V_s \subseteq V, E_s = E \cap V_s \times V_s$*?*

The subgraph isomorphism problem can also be called subgraph matching. There are two types of subgraph matching: exact and inexact [36]. For exact subgraph matching, G_1 needs to have a subgraph that is completely the same as G_2. Several algorithms, such as Ullman [38] and VF2 [35], have been developed to solve the exact subgraph matching problem. DyCLINK attempts to find *similar* subgraphs in the target method, so these algorithms are not suitable.

Inexact subgraph matching is even more complex, because G_1 needs to have similar but not exactly the same subgraph to G_2. Calculating graph similarity efficiently and precisely to support inexact graph matching is an active topic. Some graph kernels [7, 40] attempt to represent graphs by some of their features and then calculate the graph similarity. There are two problems with using these graph kernels:

1. Memory: Most of these graph kernels require generating the adjacency matrix of a graph. However, because the vertex numbers for G_{digs}s can be large ($10K+$), the memory requirement can be huge.

2. Time: The time complexity to solve the inexact graph isomorphism problem for most graph kernels is at least $O(V^4)$. If we want to search for similar subgraphs in G_1, we need to enumerate every possible combination of vertices in G_1 for G_2 to match, which leads to the unacceptable time complexity $O(\binom{V_1}{V_2} * V_2^4)$. V_1 and V_2 denote the vertex numbers in G_1 and G_2, respectively.

To solve our subgraph matching problem in G_{dig} efficiently, we devise a *Link-analysis-based Subgraph Isomorphism Algorithm*, **LinkSub**. The conceptual procedure of LinkSub is depicted in Algorithm 1. Each subroutine of LinkSub will be discussed in this section.

LinkSub models an instruction graph of a method as a network, and utilizes the power of link analysis [9], such as PageRank [28], to rank each vertex in the network. The vertex with the highest rank can be identified as the most important one in a G_{dig}. This vertex is called the *centroid* of a testing graph, G^{te}_{dig}, even though this vertex is not necessarily in the center of a graph. All required information regarding G^{te}_{dig} for subgraph matching, such as the instruction distribution and the centroid, is computed in the `profileGraph` step. We list all instructions of the target graph, G^{ta}_{dig}, in sequence by the feature defined by the developer in the `sequence` step to facilitate locating candidate subgraphs. In this paper, we use the execution time stamp of each instruction as the feature to list instructions in G^{ta}_{dig}. The centroid of G^{te}_{dig} is used to locate candidate subgraphs in G^{ta}_{dig}, in the `locateCandidates` step. The centroid vertex (instruction) of the method can also help identify the behavior of this method, which will be discussed in §4.

Data: The target graph G^{ta}_{dig} and the test graph G^{te}_{dig}
Result: A list of subgraphs in G^{ta}_{dig}, $HotZones$, which are similar to G^{te}_{dig}
$profile_{te}$ = profileGraph(G^{te}_{dig});
seq_{ta} = sequence(G^{ta}_{dig});
$assigned_{ta}$ = locateCandidates(seq_{ta}, $profile_{te}$);
$HotZones = \emptyset$;
for sub *in* $assigned_{ta}$ **do**
  SD = staticDist(SV(sub), $profile_{te}.SV$);
  if $SD > threshold_{stat}$ **then**
    continue ;
  end
  DV^{sub}_{target} = LinkAnalysis(sub);
  $dynSim$ = calSimilarity(DV^{sub}_{target}, $profile_{te}.DV$);
  if $dynSim > threshold_{dyn}$ **then**
    $HotZones \cup sub$;
  end
end
return $HotZones$;

Algorithm 1: Procedure of the LinkSub algorithm

Executing PageRank on every candidate subgraph in G^{ta}_{dig} can affect the performance of DyCLINK, if the candidate number is large. We designed a static filter (`staticDist`) similar to [30], which computes the Euclidean distance between the distribution vectors of instructions from G^{te}_{dig} and a candidate subgraph from the G^{ta}_{dig}. This distribution vector of instructions is represented as $SV(G_{dig})$. If the distance is higher than the static threshold ($threshold_{stat}$) defined by the user, then this pair of subgraph matching is rejected.

If a candidate subgraph from the G^{ta}_{dig} passes the examination of the static filter, DyCLINK applies its Link Analysis to this candidate. DyCLINK flattens and sorts both the G^{te}_{dig} and the current subgraph from the G^{ta}_{dig} to a dynamic vector based on the PageRank of each vertex. This dynamic vector is represented as $DV(G_{dig})$ and its length is always equal to the vertex number of G_{dig}. We use the Jaro-Winkler Distance [10] to measure the similarity of two DVs, which represents the similarity between two G_{dig}s, in the `calSimilarity` step. Jaro-Winkler has better tolerance of element swapping in the array than Edit Distance and is configurable to boost similarity if the first few elements in two strings or arrays are the same. These two features are beneficial for DyCLINK, because the length of $DV(G_{dig})$ is usually long,

which implies frequent instruction swapping, and what we want to detect is the behavior of methods, which are driven by the top ranked instructions in $DV(G_{dig})$. If the similarity between the subgraph from the G^{ta}_{dig} and the G^{te}_{dig} is higher than the dynamic threshold ($threshold_{dyn}$), DyCLINK identifies this subgraph as being isomorphic to the G^{te}_{dig}. We refer to the subgraph similar to the G^{te}_{dig} as a *Code Relative* (Hot Zone) in the G^{ta}_{dig}.

4. EVALUATION

For evaluating the efficacy of DyCLINK, we design two large-scale experiments: *Code relative detection* and *K Nearest Neighbor (KNN) based software classification.* In the first experiment, we compare code relatives detected by DyCLINK with code clones identified by the state-of-the-art system in 7 Java libraries for which execution benchmarks were available. One of several promising cases detected by DyCLINK, which will be discussed in §4.2, show that the programs can have similar behavior, even their source code looks different. To demonstrate the capability of DyCLINK in searching for similar behavior among programs, we collect 171 projects from 4 different problem sets in the Google Code Jam competition [17]. The real label for each program is the problem set it aims to solve. After computing the similarity between each program, we use the labels of the K nearest neighbors of the program to predict its label. If the predicted label is the same with the real label, we mark it as a successful classification. The technical details and the results of the KNN-based experiment will be revealed in §4.3.

To the best of our knowledge, DyCLINK is the first system to detect code relatives at instruction level. However, we decide to use Deckard [18, 15] [1], the state-of-the-art system, which detects code clones in the complex data structure, AST, as our baseline system. After contacting the authors of Deckard, we found that the version of Deckard with PDG analysis [15] has not been released. So we chose the latest version of [18], which has been released on GitHub [11].

4.1 System Settings

Because DyCLINK is instruction-based and Deckard is token-based, we convert both instruction number and token number to the same basis, *Lines Of Code* (LOC). Our estimation is that there are roughly 4.5 instructions per LOC and 9 tokens per LOC. It's hard to set an equivalent similarity setting for two different systems that have different definitions of similarity. Thus, we follow the default similarity thresholds for both systems: 0.82 for DyCLINK and 0.95 for Deckard. We attempt to loose the similarity threshold for Deckard to 0.85, but the false positive rate grows a lot. Because we want to detect larger code relatives that can help developers refactor their software in §4.2, we set the minimum LOC of a code relative/clone as 30. In the KNN-based experiment for classifying software behavior in §4.3, we have two similarity threshold settings for both systems: $\{0.82, 0.9\}$ for DyCLINK and $\{0.9, 0.95\}$ for Deckard.

[1] Deckard uses the grammar of Java 1.4, which was outdated in 2008, for its AST generation. To provide a more current analysis, we extended the Java grammar files of Deckard to be compliant with Java 7 standards. This resulted in a 51.8% reduction in skipped code for the matrix libraries and 100% reduction for the Code Jam projects.

Because DyCLINK is a dynamic approach that needs input generators, we utilize *Java Matrix Benchmark* [20] to generate inputs for programs in the matrix libraries. For the encryption libraries, we used their testing suites as the drivers to generate inputs. We set a three-minute threshold (which is configurable by users) for each test case to execute. For the KNN-based experiment, we use the input files provided by Google to drive each project.

4.2 Code Relative Detection

We applied DyCLINK and Deckard to 7 Java libraries. We then compared the code relatives detected by DyCLINK with the code clones identified by Deckard. Among the 7 libraries we chose, Colt, Jama, Commons Math (CMath), ojAlgo and EJML are for matrix manipulation, while Plexus and Java_codecs (Jcodecs) are for encryption. We permute and analyze every library pair to detect code clones and relatives between libraries. This leads to 21 library comparisons. The number of method graphs (G_{dig}) in these libraries ranges from 23 to 954 (N.B. one method may have multiple graphs depending on our randomly generated inputs). The total number of graph comparisons conducted by DyCLINK among these libraries is $2,759,332$. For each comparison, DyCLINK executes subgraph matching to detect code relatives. The total number of prospective subgraph matches conducted by DyCLINK is $163,962,307$.

With the LinkSub algorithm, DyCLINK completes all library comparisons in an acceptable amount of time, even though the order of most instruction graphs is high. The total time to compare any two libraries ranged up to 18 hours. However, all of the observed comparisons that required long computation times ($8-18$ hours) were from the EJML library. The other comparisons completed within two hours. The experiments conducted by DyCLINK were on c4.8xlarge instances of Amazon EC2 [2].

4.2.1 Analysis of Code Relatives

All of the code relatives and code clones detected by DyCLINK and Deckard, respectively, in our experiment are summarized into two categories "verified" and "dubious". There is no established methodology for automatically determining the validity of detected clones. However, we performed a manual inspection of the results. Among 87 code relatives detected by DyCLINK, 73 were determined to be valid. 12 out of the 14 dubious code relatives were detected between two particular algorithms common to multiple libraries. Deckard detected 37 code clones, where 36 were valid, but half of those were from one particular pair of libraries (ojAlgo and Jama) from which, one of ojAlgo's packages is clearly a direct adaptation of Jama.

The total number of valid code relatives/clones detected by both system was 96. Some methods that are both syntactically and behaviorally similar were detected by both systems. There were 60 code relatives detected only by DyCLINK and 23 code clones detected only by Deckard. However, DyCLINK did not have the chance to evaluate 20 of the Deckard-only clones, because the execution benchmark did not generate input that would cause that code to execute. This is not an algorithmic problem.

We now discuss one of several promising code relatives detected by DyCLINK, which can be seen in Table 2. The table records the information for both of the methods and their important instructions. Because DyCLINK merges

the callee graph into the caller, the important instruction may locate in the caller or any of its callees. The field *Inst. Method* records the location of the important instruction and the field *Inst. type* records the instruction type. *Inst. line no.* records the line number of the instruction. *Inst. rank* contains two values: the ranking of the instruction in the method and the PageRank value of this instruction. *Line Trace* records the line number of each method in the execution trace.

Take `Matrix.solve` of Jama as an example. `Matrix.solve` first initializes an instance of `QRDecomposition` and then calls `QRDecomposition.solve`. These two methods contribute the computation of `Matrix.solve`. The important instruction of `Matrix.solve` is located within itself, but in `QRDecomposition.solve`. The value on the left hand side of → records where the caller method invokes the callee. In this case, `Matrix.solve` invokes `QRDecomposition.solve` on line 816 (line 3 in Figure 4b), and the important instruction is located on line 197 in `QRDecomposition.solve`. There are more promising code relatives detected only by DyCLINK such as Colt's `tred2` method in `EigenvalueDecomposition` and Commons math's `transform` method in `TriDiagonalTransformer`. Because of the space limitation, we do not reveal every code relative.

DyCLINK captures some code relatives where true similarity is dubious. Most dubious cases are between the Singular Value Decomposition (svd) algorithm and the Eigenvalue Decomposition algorithm (eig). These two algorithms are different but related: svd can use eid as the sub-routine. DyCLINK aims to detect code relatives with similar behavior but not necessarily similar overall functionalities. Two reasons cause this code relative to be detected: highly similar distributions of instructions and similar important instructions, which boost the similarity between them.

4.2.2 Recall Comparison

Here we define the terms that will be used in this section. Here, *system_x* can either be DyCLINK or Deckard, and *type* refers to code relatives or code clones, respectively.

- $V(system_x)$: Represents the number of the valid *types* detected by *system_x*.
- Total Case Number, (Total CN): Represents the sum of the valid *types* detected by both systems:

$$TotalCN = |V(\text{DyCLINK}) \cup V(\text{Deckard})| \quad (6)$$

- $R(system_x)$: Represents the recall of valid *types* detected by *system_x*:

$$R(system_x) = V(system_x)/TotalCN \quad (7)$$

We only list the library comparisons for which at least one system can detect a valid *type* in Table 3. Each column, $R(system_x)$, contains two values: $i\%(u\%)$. $i\%$ represents *system_x*'s recall with respect to the intersection of *types* found by both systems. Because DyCLINK is a dynamic approach, it needs input in order to execute any methods. However, some methods in a library may not be covered, if the input generator cannot produce corresponding input. Because the Java Matrix Benchmark only generated symmetric matrices for the eigenvalue decomposers in each library, methods for non-symmetric matrices were not touched by DyCLINK. This is not an algorithmic problem with DyCLINK, but is symptomatic of a larger problem of generating inputs with high coverage of methods in a library. $u\%$ represents *system_x*'s recall with respect to the union of *types* found by both systems. This contain all valid *types* in all the libraries.

Table 3: The recall comparison of the clones detected by DyCLINK and Deckard with the setting $LOC \geq 30$.

Lib1	Lib2	R(DyCLINK)	R(Deckard)	*TotalCN*
Colt	EJML	100%(100%)	0%(0%)	11(11)
Colt	CMath	100%(40%)	0%(60%)	2(5)
Colt	Jama	100%(71%)	60%(71%)	5(7)
Colt	ojAlgo	100%(71%)	60%(71%)	5(7)
CMath	Jama	100%(67%)	0%(33%)	4(6)
CMath	ojAlgo	75%(50%)	25%(50%)	4(6)
CMath	EJML	100%(100%)	0%(0%)	4(4)
Jama	ojAlgo	86%(52%)	64%(78%)	14(23)
Jama	EJML	100%(100%)	0%(0%)	13(13)
ojAlgo	EJML	100%(100%)	0%(0%)	12(12)
Jcodecs	Plexus	100%(100%)	0%(0%)	2(2)

Table 3 shows that DyCLINK outperforms Deckard in recall on almost all library comparisons, even when we choose $u\%$, which is disadvantageous to DyCLINK. If we exclude the problem of input generation, the recall of DyCLINK is 100% for the large majority of library comparisons. Based on our experiment result, we have positive answer for **RQ1** in §2.2: DyCLINK is able to identify more syntactically and/or behaviorally similar programs than the state-of-the-art code clone detector.

4.3 KNN-based Software Classification

In this experiment, we prove that the use of code relatives gives DyCLINK a strong advantage over Deckard in program classification and search tasks. To collect programs with ground-truth classification for our KNN-based experiment, we chose the Google Code Jam competition as our code repository [17]. Google Code Jam is an annual online coding competition hosted by Google. Participants submit their projects' source code online, and Google determines whether they correctly solve a given problem. Since each submission for the same problem attempts to perform the same task, we use the problem name as a ground-truth classification for the submitted projects.

4.3.1 KNN Implementation

The high level procedure of our KNN-based software classification algorithm can be read in Algorithm 2. We first label each program with the name of the problem that it attempts to solve in the `realLabel` step. For example, if a project is submitted for the "Perfect Game" problem set, the real label of that program is "Perfect Game". We then compute the similarity between each program in the `computeSim` step by DyCLINK and by Deckard, respectively.

Next, we apply the *K-Nearest Neighbors (KNN)* classification algorithm to predict the label for each method. For each program, we search for the K other programs that have the greatest similarity to the current one in the `searchKNN` step. Each nearest neighbor program can vote for the current method by its real label in the `vote` step. The label voted by the greatest number of neighbor programs becomes the predicted label of the current program. In the even of a tie, we side with the neighbors with the highest sum of similarity scores.

Table 2: A summary of the code relative between Common maths's `SingularValueDecomposition.<init>` and Jama's `Matrix.solve`. This code relative is only detected by DyCLINK.

Library	Commons math	Jama
Method	`org.apache.commons.math3.linear.` `SingularValueDecomposition.<init>`	`Jama.Matrix.solve`
Inst. Method	`org.apache.commons.math3.linear.` `SingularValueDecomposition.<init>`	`Jama.QRDecomposition.solve`
Inst. type	`dadd`	`dadd`
Inst. line no.	226	$816 \rightarrow 197$
Inst. rank	1(0.017)	1(0.019)
Line trace	`SingularValueDecomposition.<init>`: [178 – 295]	`Matrix.solve`:[815] `QRDecomposition.<init>`:[50 – 85] `QRDecomposition.solve`:[181 – 216]
LOC	57	66

```
1 public SingularValueDecomposition(final RealMatrix
      matrix) {
2   .
3   .
4   // Generate U.
5   for (int j = nct; j < n; j++) {
6     for (int i = 0; i < m; i++) {
7       U[i][j] = 0;
8     }
9     U[j][j] = 1;
10  }
11  for (int k = nct - 1; k >= 0; k--) {
12    if (singularValues[k] != 0) {
13      for (int j = k + 1; j < n; j++) {
14        double t = 0;
15        for (int i = k; i < m; i++) {
16          //Centroid: 226
17          t += U[i][k] * U[i][j];
18        }
19        t = -t / U[k][k];
20        for (int i = k; i < m; i++) {
21          U[i][j] += t * U[i][k];
22        }
23      }
24    .
25    .
26  }
27
28  // Generate V.
29  for (int k = n - 1; k >= 0; k--) {
30    if (k < nrt &&
31      e[k] != 0) {
32      for (int j = k + 1; j < n; j++) {
33        double t = 0;
34        for (int i = k + 1; i < n; i++) {
35          //3rd Inst: 255
36          t += V[i][k] * V[i][j];
37        }
38        t = -t / V[k + 1][k];
39        for (int i = k + 1; i < n; i++) {
40          V[i][j] += t * V[i][k];
41        }
42      }
43    }
44    .
45    .
46  }
47 }
```

(a) Commons maths's `SingularValueDecomposition.<init>`

```
1 public Matrix solve (Matrix B) {
2   return (m == n ? (new LUDecomposition(this)).solve
      (B) :
3     (new QRDecomposition(this)).solve(B));
4 }
5
6 public QRDecomposition (Matrix A) {
7   .
8   .
9   for (int k = 0; k < n; k++) {
10    .
11    .
12    if (nrm != 0.0) {
13      .
14      .
15      for (int j = k+1; j < n; j++) {
16        double s = 0.0;
17        for (int i = k; i < m; i++) {
18          //2nd Inst: 77
19          s += QR[i][k]*QR[i][j];
20        }
21        s = -s/QR[k][k];
22        for (int i = k; i < m; i++) {
23          QR[i][j] += s*QR[i][k];
24        }
25      }
26    }
27    Rdiag[k] = -nrm;
28  }
29 }
30
31 public Matrix solve (Matrix B) {
32   .
33   .
34   for (int k = 0; k < n; k++) {
35     for (int j = 0; j < nx; j++) {
36       double s = 0.0;
37       for (int i = k; i < m; i++) {
38         //Centroid: 197
39         s += QR[i][k]*X[i][j];
40       }
41       s = -s/QR[k][k];
42       for (int i = k; i < m; i++) {
43         X[i][j] += s*QR[i][k];
44       }
45     }
46   }
47   .
48   .
49 }
```

(b) Jama's `Matrix.solve`

Figure 4: A partial comparison of the code for the case in Table 2. The code around the important instructions in `SingularValueDecomposition.<init>` from Commons math and `Matrix.solve` from Jama library shows the similar behavior, which aligns with the detection result of DyCLINK.

Data: The similarity computation algorithm $SimAlg$, the set of subject programs to be classified $Programs$ and the number of the neighbors K
Result: The precision of $SimAlg$
realLabel($Programs$);
$matrix_{sim}$ = computeSim($SimAlg$, $Programs$);
$succ = 0$;
for p *in* $Programs$ **do**
    $neighbors$ = searchKNN(p, $matrix_{sim}$, K);
    $p.predictedLabel$ = vote($neighbors$);
    if $p.predictedLabel = p.realLabel$ **then**
        $succ = succ + 1$;
    end
end
$precision = succ/Programs.size$;
return $precision$;

Algorithm 2: Procedure of the KNN-based software label classification algorithm

Table 4: A summary of the code subjects from the Google Code Jam competition for classifying software.

Year	Problem Set	Abbrev.	Proj.	# of Graphs
2011	Irregular Cake	I	48(30)	762
2012	Perfect Game	P	48(34)	295
2013	Cheaters	C	29(21)	612
2014	Magical Tour	M	46(33)	479

Finally, we compare the predicted label for a program against its real behavioral label. If the predicted label is the same with the real label, we mark the prediction of this method as successful. We define the precision of a similarity computation algorithm ($SimAlg$) as the percentage of program it labels correctly, which can be read in Algorithm 2. We selected 4 problem sets, one per year between 2011 and 2014, which have totally 171 projects. The information of these problems sets and the number of projects can be read in Table 4. The participants of the Google Code Jam can choose to implement their projects to either access the input file provided by Google automatically or read the input from the command line interactively. The Proj. column in Table 4 records two values: the first one for the number of total projects and the second one for the number of the non-interactive projects. We only selected the latter, which facilitate us to execute every project automatically. The total number of the non-interactive projects is 118. We then applied Algorithm 2 with both DyCLINK and Deckard to these methods to calculate the classification precision. The parameter settings for both systems can refer to §4.1. We exclude some utility programs, such as reading a file that are used across different years, which bias the experiment. For these 118 projects, DyCLINK generated 2,148 method graphs at instruction level (G_{dig}). For computing the similarity between each method, DyCLINK conducted 4,509,574 method comparisons with 130,796,923 subgraph matching.

4.3.2 Analysis of Software Behavior Classification

For observing the efficacy of both systems under single and multiple neighbors, we set $K = 1$ and $K = 5$. Also, we wanted to observe the precision of classifying relevant programs under different program sizes, so we had 4 different LOC thresholds, $\{10, 15, 20, 30\}$. Only programs that pass the threshold setting including LOC and similarity were considered as neighbors of the current program.

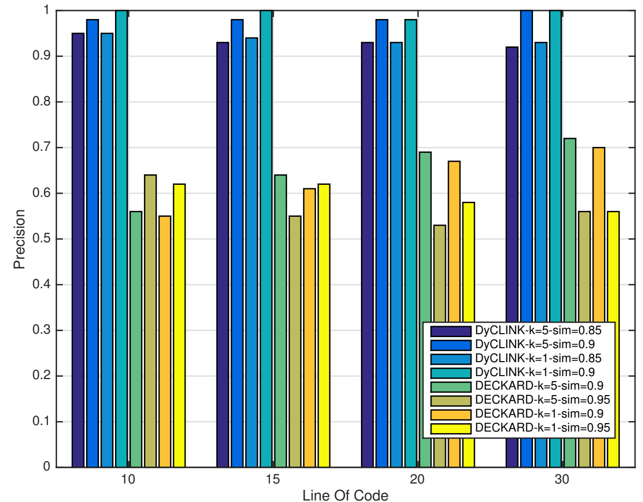


Figure 5: The KNN-based program classification. The similarity between programs is measured by DyCLINK and Deckard.

The advantage of DyCLINK to detect programs with similar behavior can be seen in Figure 5. For each parameter setting with different thresholds of LOC and similarity, DyCLINK achieves 96% precision in average, while Deckard's precision is 61% on average. When the threshold of LOC is high, Deckard may detect clones out of method boundaries, which can affect its classification capability. DyCLINK has better performance with $K = 1$ than with $K = 5$. This reveals that DyCLINK is able to precisely search for and rank programs with the most similar behavior. The first search result is often the most relevant. Moreover, if the thresholds of LOC and similarity are high (LOC= 30 and similarity threshold = 0.9), DyCLINK can even achieve 100% precision. In fact, we also tried $K = 20$ with LOC = 30, but under such high LOC threshold, each system did not report too many neighbors for programs. The result was about the same with $K = 5$. Our software classification result provides a strong support for **RQ2**: DyCLINK is more precise to search for relevant programs than the code clone detector.

Based on programs with similar behavior (code relatives) detected by DyCLINK, we can cluster projects. Figure 6 shows the clustering matrix based on one of our KNN-based classification result with K = 5, LOC = 10 and similarity threshold = 0.9. Each element on both axes of the matrix represents a project indexed by the abbreviation of the problem set it belongs to and the project ID. The abbreviation of each problem set can be read in Table 4. We sort projects by their project indices. Only projects that have at least one code relative with another project are recorded in the matrix. The color of each cell represents the relevance between the i_{th} project and the j_{th} project (the darker, the higher), where i and j represent the row and column in the matrix. The project relevance is the number of code relatives that two projects share. Each block on the matrix forms a *Software Community*, which fits in the problem sets that these projects aim to solve. The result of our KNN-based experiment shows

that DyCLINK is capable to detect programs having similar behavior and then cluster them for further usage such as code search.

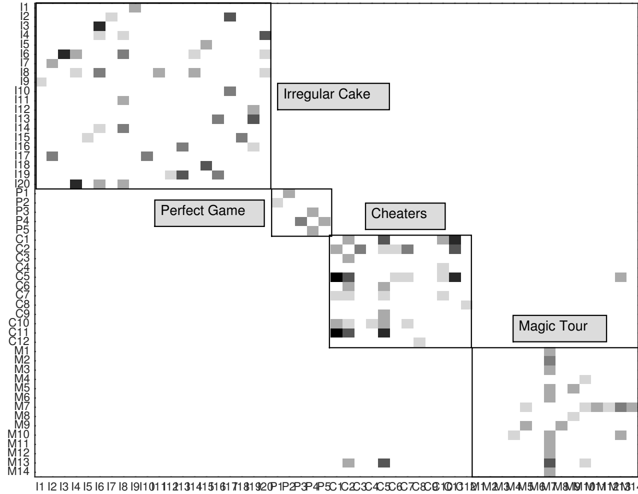


Figure 6: The software community based on code relatives detected by DyCLINK. The darker color in the cell represents higher number of code relatives shared by two projects.

In addition to Deckard, we attempt to follow the functional equivalence approach [19], which clusters C programs based on functional I/O, in Java. However, after conducting preliminary experiments, we find two non-trivial problems due to the nature of object-oriented language that hinder us from executing such functional cluster analysis: 1. If the input parameter of a program is an interface that is not instantiate-able, how to generate valid input instance for it? 2. If output values of programs are instances instantiated from *different* classes, how to effectively compare them? Solving these two problems can be a direction for us to work on in the future.

5. RELATED WORK

We survey relevant publications to code relative as follows.

Code Clone Detection Most code clone detection systems parse a program into an intermediate representation (IR) for computing similarity with other programs. The time complexity of similarity computation will increase if the structure of IR is complex such as a tree and a graph [6, 18, 25, 24, 26, 30], but more structural and semantic information about a code segment can be encoded. Program Dependence Graph (PDG) is a widely used graph for programs. Komondoor and Horwitz [24] generate PDGs for C programs, and then apply program slicing techniques to detect isomorphic subgraphs. The approach designed by Krinke [26] starts to detect isomorphic subgraphs with maximum size k after generating PDGs of programs. The granularity of Krinke's PDG is finer than the traditional one: each vertex roughly maps to a node in an AST. The approach proposed by Gabel *et al.* [15] is a combination of AST and graph. It generates PDG of a method, maps that PDG back to an AST and then uses Deckard to detect clones. GPLAG invented by Liu *et al.* [30] determines when it is worthwhile to invoke the subgraph matching algorithm between two PDGs using two statistic filters. The time limit in their subgraph matching algorithm may miss some larger clones.

Compared with these graph-based approaches that identify *static* code clones, DyCLINK detects the similar *dynamic* behavior of programs (code relatives). Furthermore, because DyCLINK is instruction-based, which has finer granularity than these approaches, DyCLINK can explore more behavior patterns in the codebase. With the LinkSub algorithm, DyCLINK can even process PDGs with large size that most graph-based approaches cannot handle in timely fashion.

Software Behavior Detection In addition to identify code clones, several approaches detect behavior of software statically or dynamically. Demme and Sethumadhavan [12] identifies programs that react similarly to the code optimization. Jiang and Su [19] drive programs by randomly generated input and then observe their output values. The programs having similar outputs are identified as functional equivalence. Egele *et al.* [14] propose to execute functions under different environment settings. The runtime features of these functions, such as system calls, are collected for computing the similarity between functions. McMillan *et al.* [32] computes the similarity between applications based on their API usage. Their approach helps developers search relevant programs to prototype their current projects rapidly. Nguyen and Nguyen develop GraLang [34], which express the API usage in the programs as graphs to suggest APIs to developers. Yang *et al.* [41] abstracts the behavior of the Android app by the usage of security-sensitive APIs. This type of security behavior can be used to detect malicious apps under different context such as time. Avdiienko *et al.* [4] characterize the behavior of the Android app as the usage of the sensitive data. They then apply the data analysis technique to track and detect apps with abnormal behavior.

DyCLINK also aims to detect similar behavior between programs. Most work in this category abstracts the behavior of a program at different levels. Since DyCLINK works at instruction level, it may expose program behavior in more details. Integrating DyCLINK with these systems to support software engineering tasks, such as code search and malware detection, can be our future work.

6. CONCLUSION

In this paper, we presented a novel system, DyCLINK, which can dynamically detect code relatives among methods at the instruction level. A code relative represents a pair of code skeletons having similar runtime behavior with or without the same implementation. DyCLINK converts the execution trace of a method into an instruction dependency graph at runtime. We devised a Link Analysis based subgraph isomorphism algorithm, *LinkSub*, which can detect subgraph matches among thousands of instructions efficiently. In our KNN-based code classification experiment, DyCLINK detected and searched for more neighbor programs having similar behavior precisely than a state-of-the-art code clone detector.

7. ACKNOWLEDGMENTS

The authors thank for Prof. Lingxiao Jiang's suggestions on updating Deckard. Su, Harvey and Kaiser are members of the Programming Systems Laboratory. Sethumadhavan is the member of the Computer Architecture and Security Technology Laboratory. Jebara is the member of the Columbia Machine Learning Laboratory. This work is supported by NSF CCF-1302269.

8. REFERENCES

[1] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[2] Amazon ec2. `http://aws.amazon.com/ec2/instance-types/`. Accessed: 2015-08-17.

[3] Asm framework. `http://asm.ow2.org/index.html`. Accessed: 2015-02-05.

[4] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *2015 International Conference on Software Engineering (ICSE)*, ICSE '15, pages 426–436, 2015.

[5] B. S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, 1992.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, 1998.

[7] K. M. Borgwardt and H.-P. Kriegel. Shortest-path kernels on graphs. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 74–81, 2005.

[8] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 195–205, 2004.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, 1998.

[10] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration*, pages 73–78, 2003.

[11] Deckard source code. `https://github.com/skyhover/Deckard`. Accessed: 2015-03-20.

[12] J. Demme and S. Sethumadhavan. Approximate graph clustering for program characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21, Jan. 2012.

[13] N. DiGiuseppe and J. A. Jones. Software behavior and failure clustering: An empirical study of fault causality. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 191–200, 2012.

[14] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, 2014.

[15] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, 2008.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[17] Google code jam. `https://code.google.com/codejam`. Accessed: 2015-08-18.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, 2007.

[19] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92, 2009.

[20] Java matrix benchmark. `https://code.google.com/p/java-matrix-benchmark/`. Accessed: 2015-03-20.

[21] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 120–126, 1994.

[22] Java virutal machine speicification. `http://docs.oracle.com/javase/specs/jvms/se7/html/`. Accessed: 2015-02-04.

[23] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[24] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, 2001.

[25] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, 2006.

[26] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309, 2001.

[27] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, Mar. 2007.

[28] P. Lawrence, B. Sergey, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[29] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 176–192, 2004.

[30] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, 2006.

[31] J. I. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, ASE '99, pages 251–, 1999.

[32] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 364–374, 2012.

[33] B. T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Trans. Knowl. Data Eng.*, 12(2):307–323, 2000.

[34] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 858–868, 2015.

[35] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, Oct. 2004.

[36] K. Riesen, X. Jiang, and H. Bunke. Exact and inexact graph matching: Methodology and applications. In *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 217–247. Springer, 2010.

[37] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[38] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.

[39] F. Umemori, K. Konda, R. Yokomori, and K. Inoue. Design and implementation of bytecode-based java slicing system. In *SCAM*, pages 108–117. IEEE Computer Society, 2003.

[40] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *J. Mach. Learn. Res.*, 11:1201–1242, Aug. 2010.

[41] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 303–313, 2015.