Uncovering Features in Behaviorally Similar Programs

Fang-Hsiang Su

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2018

ABSTRACT

Uncovering Features in Behaviorally Similar Programs

Fang-Hsiang Su

The detection of similar code can support many software engineering tasks such as program understanding and program classification. Many excellent approaches have been proposed to detect programs having similar syntactic features. However, these approaches are unable to identify programs dynamically or statistically close to each other, which we call behaviorally similar programs. We believe the detection of behaviorally similar programs can enhance or even automate the tasks relevant to program classification. In this thesis, we will discuss our current approaches to identify programs having similar behavioral features in multiple perspectives.

We first discuss how to detect programs having similar functionality. While the definition of a program's functionality is undecidable, we use inputs and outputs (I/Os) of programs as the proxy of their functionality. We then use I/Os of programs as a behavioral feature to detect which programs are functionally similar: two programs are functionally similar if they share similar inputs and outputs. This approach has been studied and developed in the C language to detect functionally *equivalent* programs having equivalent I/Os. Nevertheless, some natural problems in Object Oriented languages, such as input generation and comparisons between application-specific data types, hinder the development of this approach. We propose a new technique, in-vivo detection, which uses existing and meaningful inputs to drive applications systematically and then applies a novel similarity model considering both inputs and outputs of programs, to detect functionally *similar* programs. We develop the tool, HitoshiIO, based on our in-vivo detection. In the subjects that we study, HitoshiIO correctly detect 68.4% of functionally similar programs, where its false positive rate is only 16.6%.

In addition to functional I/Os of programs, we attempt to discover programs having similar execution behavior. Again, the execution behavior of a program can be undecid-

able, so we use instructions executed at run-time as a behavioral feature of a program. We create DyCLINK, which observes program executions and encodes them in dynamic instruction graphs. A vertex in a dynamic instruction graph is an instruction and an edge is a type of dependency between two instructions. The problem to detect which programs have similar executions can then be reduced to a problem of solving inexact graph isomorphism. We propose a link analysis based algorithm, LinkSub, which vectorizes each dynamic instruction graph by the importance of every instruction, to solve this graph isomorphism problem efficiently. In a K Nearest Neighbor (KNN) based program classification experiment, DyCLINK achieves $90+\%$ precision.

Because HitoshiIO and DyCLINK both rely on dynamic analysis to expose program behavior, they have better capability to locate and search for behaviorally similar programs than traditional static analysis tools. However, they suffer from some common problems of dynamic analysis, such as input generation and run-time overhead. These problems may make our approaches challenging to scale. Thus, we create the system, Macneto, which integrates static analysis with machine topic modeling and deep learning to *approximate* program behaviors from their binaries without truly executing programs. In our deobfuscation experiments considering two commercial obfuscators that alter lexical information and syntax in programs, Macneto achieves $90+\%$ precision, where the groundtruth is that the behavior of a program before and after obfuscation should be the same.

In this thesis, we offer a more extensive view of similar programs than the traditional definitions. While the traditional definitions of similar programs mostly use static features, such as syntax and lexical information, we propose to leverage the power of dynamic analysis and machine learning models to trace/collect behavioral features of programs. These behavioral features of programs can then apply to detect behaviorally similar programs. We believe the techniques we invented in this thesis to detect behaviorally similar programs can improve the development of software engineering and security applications, such as code search and deobfuscation.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisors, Professor Gail Kaiser and Professor Simha Sethumadhavan, for offering me the opportunity to conduct research under their supervision. They have not only taught me how to solve problems but also how to discover new questions and confront challenges. Gail and Simha, I appreciate your guidance and insight throughout my PhD career. I would also like to thank my committee members, Professor Tony Jebara, Professor Baishakhi Ray and Professor Suman Jana, for providing suggestions to enhance my thesis.

I am extremely fortunate to have had numerous opportunities to work and collaborate with many excellent researchers and developers during my PhD career. In no particular order, I would like to acknowledge Chris Murphy, Swapneel Sheth, Nipun Arora, Jonathan Bell, Riley Spahn, Adrian Tang, Hiroshi Sasaki, Teruo Tanimoto, Miguel Arroyo, Anthony Saieva, Kenneth Harvey, Morris Hopkins, John Murphy, Yu Wang, Huimin Sun, Apoorv Patwardhan, Abhaar Gupta, Sriharsha Gundappa, Masudur Rahman, and many others.

Last but not least, I would like to thank my parents and my wife Jing-Yu who have always supported me wholeheartedly to pursue my dreams (and those papers) throughout my PhD career. To my son, Sean – thank you for joining your mom and me! While we have finally completed a PhD, you complete us the first minute you were born.

To my family

Chapter 1

Introduction

Analyzing similar programs can support many software engineering tasks, such as program comprehension [98], API replacement [73], code change prediction [132], bug localization [40, 108] and code reuse [50]. Two programs are similar if they are close enough according to one or multiple features. While many excellent approaches have been proposed to detect code having similar syntax or textual features that are usually referred to "code clone", we propose to detect *behaviorally* similar programs defined as follows:

Definition 1 *Two programs are behaviorally similar if they share one or more features in functionalities, executions or semantics.*

Behaviorally similar programs, we argue, can be extracted by variety of techniques. In this thesis, we propose to leverage the power of dynamic analysis and machine learning models to detect and identify behaviorally similar programs. In this thesis, we use the term "code clone" to represent those syntactically or textually similar programs and use the term "behaviorally similar program" to represent dynamically or statistically similar programs, which differentiates our approaches from those focusing on syntax and text in programs.

The design of the core algorithm or mechanism to detect similar code usually depend on the definition of similarity: which features of code that the approach attempts to capture. We classify similar code into three types as follows.

- Static features: These approaches focus on capturing similar code by using syntactic or textural features of code. Without executing programs, these approaches analyze program abstractions, such as tokens, parse trees and program dependence graphs [60, 89] and extract the information from program bodies, documents or comments [115, 93]. This type of work focuses on detecting *code clones*.

- Dynamic features: In contrast to static features, *dynamic* features represent characteristics of programs during executions. One runtime approach [60] observes *functionalities* of programs defined by inputs and outputs (I/Os) in a dynamic manner. These functionally similar programs are named "simions," which differentiate them from code clones having similar static features. Several technical challenges are identified to implement such a technique in object oriented languages [61]. Our prior work later alleviates these challenges [123], which will be discussed in Chapter 2. In addition to program I/Os, our prior work also proposes to detect programs having similar *executions*, as encoded in fine-grained dynamic dependence graphs [121].
- Hidden features: Instead of using known features to detect which programs are similar, some work proposes using statistical or machine learning techniques to *learn* features or models from "big code," which represents a large volume of code. These features that may be hard to define manually can support further applications such as API recommendation [106] and automatic identifier naming [111].

While most of the previously proposed dynamic approaches care about programs' behaviors in functionalities defined by their inputs and outputs, almost none discusses how to capture programs' behaviors in executions, which is also an important feature of program dynamics. Some behaviorally similar programs may be undiscoverable by the current similarity features. Figure 1.1 provides a trivial but motivating example. The syntax of methods in Figure 1.1a and Figure 1.1b are not the same. Further, their functionalities defined by their I/Os are not the same as well because of the multiplication at line 6 in Figure 1.1b. This makes this pair of methods neither a code clone, nor a simion. In other word, this pair of methods may not be detected as similar by the existing definitions of

```java
1 public double[] initArr1(int len)
      {
2   double[] arr = new double[len];
3   for (int i = 0; i < arr.length;
      i++) {
4     arr[i] = 5;
5   }
6   return arr;
7 }
```

(a) An exemplary method to initialize an array.

```java
1 public double[] initArr2(int size
      ) {
2   double[] ret = new double[size
      ];
3   try {
4     int counter = 0;
5     while (true) {
6       ret[counter++] = -2 * 5;
7     }
8   } catch (Exception ex) {}
9   return ret;
10 }
```

(b) A behavioral similar method with Figure 1.1a.

Figure 1.1: An example of two behaviorally similar methods in executions.

static or dynamic features.

We argue that the example in Figure 1.1 is still behaviorally similar at run-time, because their executed instructions are similar, regardless of the syntax and I/O. In our previous work [121], we name them *code relatives*. We summarize these three types of similar code, *clone*, *simion* and code relative in this thesis. Clones represent code having similar static features, and simions represent behaviorally similar code that functions alike with similar I/Os. Code relatives represent behaviorally similar code that has similar execution, but does not necessarily look or function alike. We will use Figure 1.1 as the example again to explain these three types of similar code. The pair of the programs in Figure 1.1 is a code relative, because they are behaviorally similar in their executions. Without -2 at line 6 in Figure 1.1b, they could be a pair of simion, because they are behaviorally similar in their functionalities (I/Os). Without the `try-catch` block in Figure 1.1b, they can be clones, because their syntax is similar. We argue that the behavioral features *learned* from statistical analysis from codebases could be a fourth type of similar code, which is orthogonal to the other three.

In this thesis, we will discuss our current work for detecting and mining those behaviorally similar programs in their functionalities (Chapter 2) and executions (Chapter 3), which are relevant to code dynamics. In addition, we will delineate our approach to learn a novel hidden feature of programs, topic models embedded in code binary (Chapter 4), for supporting program deobfuscation.

Chapter 2

Functionality as Behavioral Feature

Identifying similar code in software systems can assist many software engineering tasks such as program understanding and software refactoring. While most approaches focus on identifying code that *looks alike*, some techniques aim at detecting code that *functions alike*. Detecting these functional clones — code that functions alike — in object oriented languages remains an open question because of the difficulty in exposing and comparing programs' functionality effectively. We propose a novel technique, *In-Vivo Clone Detection*, that detects functional clones in arbitrary programs by identifying and mining their inputs and outputs. The key insight is to use existing workloads to execute programs and then measure functional similarities between programs based on their inputs and outputs, which mitigates the problems in object oriented languages reported by prior work. We implement such technique in our system, HitoshiIO, which is open source and freely available. Our experimental results show that HitoshiIO detects more than 800 functional clones across a corpus of 118 projects. In a random sample of the detected clones, HitoshiIO achieves 68+% true positive rate with only 15% false positive rate.

2.1 Motivation

When developing and maintaining code, software engineers are often forced to examine code fragments to judge their functionality. Many studies [63, 66, 86] have suggested large portions of modern codebases can be *clones*, which can be code that is copied-and-pasted from one part of a program to another. One problem with these clones is that they can complicate maintenance. For instance, a bug is copied-and-pasted in multiple locations in a software system. While most techniques to detect clones have focused on syntactic ones containing code fragments that look alike, we are interested in *functional clones*: code fragments that exhibit similar functions, but may not look alike.

Identifying functional clones can bring many benefits. For instance, functional clones can help developers understand complex and/or new code fragments by matching them to existing code they already understand. Further, once these functional clones are identified, they can be extracted into a common API.

Unfortunately, detecting true functional clones is very tricky. Static approaches must be able to fully reason about code's functionality without executing it, and dynamic approaches must be able to observe code executing with sufficient inputs to expose diverse and meaningful functions. Currently, the most promising approach to detect functional clones is to execute code fragments with a randomly generated input, apply that same input for different code fragments and observe when outputs are the same [60, 39, 65, 38]. Thus, previous approaches towards detecting functional clones have focused on code fragments that are easily compiled and executed in isolation, allowing for easy control and generation of inputs, and observation of code outputs.

This approach does not scale to complex and object oriented codebases. It is difficult to execute individual methods or code fragments in isolation with randomly generated inputs, due to the complexity of generating sufficient and meaningful inputs for executing the code successfully. Previous work towards detecting functional clones in Java programs [39, 32, 105] have reported unsatisfactory or limited results: a recent study by Deissenboeck et al. showed that across five Java projects only 28% of the target methods could be executed with this randomly input generation approach [32]. Deissenboeck et al. also reported that across these projects, most of the inputs and outputs referred to project-specific data types, meaning that a direct comparison of the inputs and outputs between two programs is hard to be declared equivalent [32].

We present *In-Vivo Clone Detection*, a technique that is language-agnostic, and generally applicable to detect functional clones *without* requiring the ability to execute candidate clones in isolation, and hence allowing it to work on complex and object oriented codebases. Our key insight is that most large and complex codebases include test cases [19], which can supply workloads to drive the application as a whole.

In-Vivo Clone Detection first identifies potential inputs and outputs (I/Os) of each code fragment, and then executes them with existing workloads to collect values from their I/Os. The code fragments with similar values of inputs and outputs during executions are identified as functional clones. Unlike previous approaches that look for code fragments with identical output values, we use a relaxed similarity comparison, enabling efficient detection of code that has very similar inputs and outputs, even when the exact data structures of those variables differ.

We created HitoshiIO, which implements this in-vivo approach for the JVM-based languages such as Java. HitoshiIO considers every method in a project as a potential functional clone of every other method, recording observed inputs that can be method parameters or global state variables read by a method, and observed outputs that are externally observable writes including return values and heap variables. Our experimental results show that HitoshiIO effectively detects functional clones in complex codebases. We evaluated HitoshiIO on 118 projects, finding 874 functional clones, using only the applications' existing workloads as inputs.

The primary contributions of this thesis are:

1. A presentation of our technique, In-Vivo Clone Detection, a language-agnostic technique for detecting functional clones applicable to object-oriented languages
2. Our tool, HitoshiIO for the JVM, which effectively detects functional clones in complex code bases, available under an MIT license on GitHub[55].

2.2 Related Work

Identifying similar or duplicated code (code clones) can enhance the maintainability of software systems. Searching for these code clones also helps developers to find which pieces of code are re-usable. At a high level, work in clone detection can be split into two categories: static clone detection, and dynamic clone detection.

Static techniques: Roy et al. [113] conducted a survey regarding the four types of code clones and the corresponding techniques to detect them ranging from those that are exact copy-paste clones to those that are semantically similar with syntactic differences. In general, these static approaches first parse programs into a type of intermediate representation and then develop corresponding algorithms to identify similar patterns. As the complexity of the intermediate representation grows, the computation cost to identify similar patterns is higher. Based on the types of intermediate representations, the existing approaches can be classified into token-based [14, 63, 86], AST-based [17, 59] and graph-based [49, 89, 73, 83]. Among these general approaches, the graph-based approaches are the most computationally expensive, but they have better capabilities to detect complex clones according to the report of Roy et al. [113]. Compared with these approaches that find *look alike* code, HitoshiIO searches for *functionally alike* code.

Several other techniques make use of general information about code to detect clones rather than strictly relying on syntactic features. Our motivation for detecting function clones that may not be syntactically similar is close to past work that searched for *high level concept clones* [93] with similar semantics. However, our approach is completely different: we use dynamic profiling, while they relies on static features of programs. Another line of clone detection involves creating fingerprints of code, for instance by tracking API usage [98, 29], to identify clones.

Dynamic techniques: Our approach is most relevant to previous work in detecting code that is *functionally similar*, despite syntactic differences by using dynamic profiling. For instance, Elva and Leavens proposed detecting functional clones by identifying methods that have the exact same outputs, inputs and side effects [39]. The MeCC system summarizes the abstract state of a program after each method is executed to relate that state to the method's inputs, allowing for exact matching of outputs [65]. Our approach differs from both of these in that we allow for matching functionally similar methods, even when there are minor differences in the formats of inputs and outputs.

Carzaniga et al. studied different ways to quantify and measure functional redundancy between two code fragments on both of the executed code statements and performed data operations [26]. Our notion of functionally similar code is similar to their notion of redundant code, although we put significantly more weight on comparing input and output values, rather than just the sequence of inputs and outputs. We consider *all* data types, even complex variables, while Carzaniga et al.HitoshiIO only consider Java's basic types.

Jiang and Su's EQMiner [60] and the comparable system developed by Deissenboeck et al. for Java [32] are two highly relevant recent examples of dynamic detection of functional clones. EQMiner first chops code into several chunks and randomly generates input to drive them. By observing output values from these code chunks, the EQMiner system is able to cluster programs with the same output values. The EQMiner system successfully identified clones that are functional equivalent. Deissenboeck et al. follows the similar procedure to re-implement the system in Java. However, they report low detection rate of functional clones in their study subjects. We list three of the technical challenges reported by Deissenboeck et al. and our solutions:

- *How to appropriately capture I/Os of programs*: Compared with the existing approaches that fix the definitions of input and output variables in the program, In-Vivo Clone Detection applies static data flow analysis to identify which input variables potentially contribute to output variables at instruction level.
- *How to generate meaningful inputs to drive programs*: Deissenboeck et al. reported that for $20\% - 65\%$ of methods examined, they could not generate inputs. One

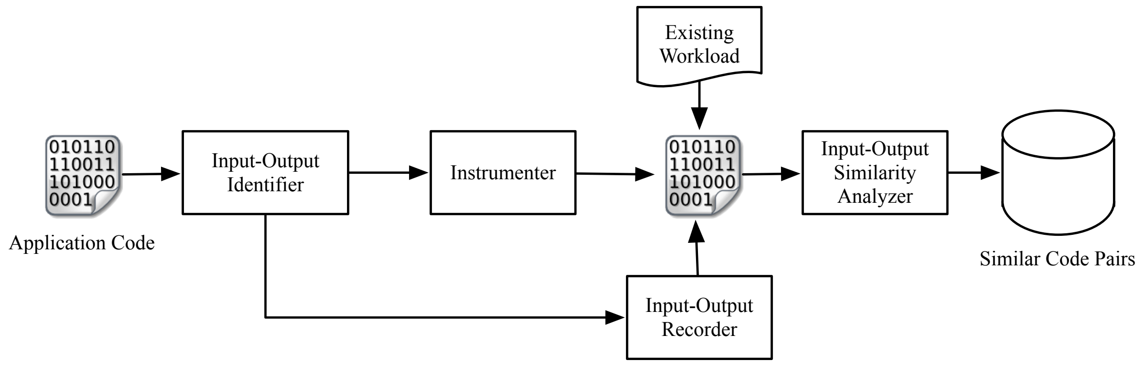


Figure 2.1: High level overview of In-Vivo Clone Detection. First, individual inputs and outputs of methods are identified, then the application is transformed so that its inputs and outputs can be easily recorded while it is executed under an existing workload. Finally, these recorded inputs and outputs are analyzed to detect functionally similar methods.

possible reason is that when the input parameter refers to an interface or abstract class, it is hard to choose the correct implementation to instantiate. Thus, instead of generating random inputs, we invent In-Vivo Clone Detection using real workloads to drive programs, which is inspired by our prior work in runtime testing [102].

- *How to compare project-specific types of objects between different applications*: We will elaborate the similar issue further in Section 2.5.5: different developers can design different classes to represent similar data across different applications/projects. For comparing complex (non-primitive) objects, In-Vivo Clone Detection computes and compares a deep identity check between these objects.

2.3 Approach: Detecting Similar Code In-Vivo

At a high level, our approach detects code which appears functionally similar by observing that for similar inputs, two different methods produce similar outputs (i.e., are functional clones). Our key insight is that we can detect these functional clones *in-vivo* to the context of a full system execution (e.g., as might be exercised by unit or system tests), rather than relying on targeted input generation techniques. Figure 2.1 shows a high level overview of the various phases in our approach. First, we identify the inputs and outputs of each method in an application where we consider not just formal parameters, but also all relevant application states. Then we instrument the application so that when executing it with existing workloads, we can record the individual inputs and outputs to each method, for use in an offline similarity analysis.

2.3.1 The Input Generation Problem

Previous approaches towards detecting functional clones in programs randomly or systematically generate inputs to execute individual methods or code fragments first, and then identify code fragments with the identical outputs as functional clones. Especially in the case of object oriented languages like Java, it may be difficult to generate an input to allow an individual method to be executed because each method may have many different input variables, each of which may have an immense range of potential values. Many other techniques have been developed to automatically generate inputs for individual methods, but the problem remains unsolved in the case of detecting functional clones. For instance, Randoop [107] uses a guided-random approach, in which random sequences of method calls are executed to bring a system to a state to which an individual method can be executed. Randoop is guided only by the knowledge of which previous sequences failed to generate a 'valid' state, making it difficult to use in many cases [128]. In the 2012 study of input generation for clone detection conducted by Deissenboeck et al., they found that input generation and execution failed for approximately 28% of the methods that they examined across five projects. A discussion of how we plan to alleviate the input generation problem in the future can be found in Section 5.5.

2.3.2 Exploiting Existing Inputs

With our In-Vivo approach, it is feasible to detect functional clones even in the cases where automated input generators are unable to generate valid inputs. We observe that in many cases, existing workloads (e.g., test cases) likely exist for applications, at which point we can exploit the individual inputs used by each method. Key to our approach is a simple static analysis to detect variables that are inputs, and those that are outputs for each method in a program. From this static analysis, we can inform a dynamic instrumenter to record these values, and later, compare them across different methods.

The output of a method is any value that is written within a method that is potentially observable from another point in the program: that is, it will remain a live variable even after that method concludes. The input of a method then, is any value that is read within that method and influences any output (either directly through data flow or indirectly through control flow). By this definition, variables that are read within a method, but not computed on, are not considered inputs, reducing the scope of inputs to only those may impact the output behavior of a method.

Definition 2 *An input for a method is the value that exists before execution of this method, is read by this method, and contributes to any outputs of the method.*

An output of a method is the computational result of this method that a developer wants to use. As Jiang and Su stated [60], it is hard to define the output for a method, because we don't know which values derived/computed by the method will be used by the developer. So, we define the outputs for a method in a conservative way:

Definition 3 *An output of a method is the value derived or computed by this method. This computational value still exists in memory after the execution of this method.*

To identify inputs given outputs, we follow [46] to statically identify the following dependencies:

- Computational Dependency: This dependency records which values depends on the computation of which values. Take `int k = i + j` as the example. The value of `k` depends on the values of `i` and `j`. This dependency (*c-use* [46]) helps identify which inputs can affect the computations of outputs.

- Ownership Dependency: This dependency records which values (fields) owned by which objects and/or arrays. Take `int c = a.myInt + b` as the example, where `a` is an object and `myInt` is an integer. In this example, the `myInt` field owned by `a` influences the value of `c`. Because the `a` object owns `myInt`, our approach will know that the `a` object can be an input source, even though this read does not access `a`'s value directly. The ownership dependency helps identify which values can be from inputs. This dependency is transitive, which means that the value owned by an object/array is also owned by the owner of this object/array, if it has any owners.

2.3.3 Example of HitoshiIO

To demonstrate our general approach, we use the method `addRelative` in Figure 2.2. Note that while the code presented is written in Java, our technique is generic, and not tied to any particular language. The `addRelative` method takes a `Person` object, `me`, as the input, and create a new relative, based on the other two input parameters, `rName` and `rAge`. The `insert` method, which is a callee of `addRelative`, inserts `newRel` into the array field owned by `me`. The `sum` method, which is the other callee, computes and return the total age of all relatives owned by `me`.

We use the list of outputs to identify the inputs, so we first define the formal outputs of `addRelative`. `ret` is the return value, which is a natural output. Because `pos` flows to an `OutputStream`, it is recognized as an output. `me` is written in the callee `insert`, so it is also an output.

Before we discuss the input sources, we summarize the data dependencies in `addRelative`. We use the variable name to represent the value they contain. And we use $x \xrightarrow{c.} y$ to represent that y is computational-dependent on x, and $x \xrightarrow{o.} y$ to depict that y is owned by x. The dependencies in `addRelative` can be read in Table 2.1. The **Deps.** column records the dependency between two variables, and the **Notes** col-

```java
public class Person {
  public String name;
  public int age;
  public Person[] relatives;
}

public static int addRelative(Person me, //input
  String rName, int rAge, int pos,
  double useless) {

  Person newRel = new Person();
  newRel.name = rName;
  newRel.age = rAge;

  if (pos > 0) {
    insert(me, newRel, pos);
  }
  int ret = sum(me.relatives);

  double k = useless + 1;

  System.out.println(pos); //output
  return ret; //output
}

public static void insert(Person me, Person rel, int pos) {
  me.relatives[pos] = rel;
}

public static int sum(Person[] relatives) {
  int sum = 0;
  for (Person p: relatives) {
    sum += p.age;
  }
  return sum;
}
```

Figure 2.2: A code example with inputs and outputs identified.

umn explains why these two variables have the dependency. We only show the direct dependencies between variables.

Finally, we can define the input sources based on the outputs and the dependencies between variables. An input source is the one that have direct or transitive dependencies

Table 2.1: The data dependencies in the `addRelative` method.

Deps.	**Notes**
$\texttt{relatives}\overset{c.}{\to}\texttt{ret}$	`ret` is the computational result of `sum`, which depends on `me.relatives`.
$\texttt{me}\overset{o.}{\to}\texttt{relatives}$	`relatvies` is a field of `me`.
$\texttt{newRel}\overset{c.}{\to}\texttt{me}$	`me` is written in the callee `insert`, where newRel is the input.
$\texttt{pos}\overset{c.}{\to}\texttt{me}$	The same reason as the above.
$\texttt{rName}\overset{c.}{\to}\texttt{newRel}$	`newRel` is written by `rName`.
$\texttt{rAge}\overset{c.}{\to}\texttt{newRel}$	`newRel` is written by `rAge`.

to any of the outputs. We first define the candidate input sources in `addRelative` as

$$\begin{aligned} &ISrc_c(\texttt{addRelative}) \\ &= \{\texttt{me}, \texttt{rName}, \texttt{rAge}, \texttt{pos}, \texttt{useless}\} \end{aligned} \tag{2.1}$$

Given 3 outputs and all dependencies in Table 2.1, we can infer the parents of these 3 outputs as

$$Parents(\{\texttt{ret}, \texttt{me}, \texttt{pos}\}) = \{\texttt{me}, \texttt{rName}, \texttt{rAge}, \texttt{pos}\} \tag{2.2}$$

We then intersect these two sets and conclude the input sources of `addRelative` in Table 2.2. We can see that not all input parameters are considered as input sources. The variable `useless` contributes to no outputs, so we do not consider it as an input source.

We consider the values that may change the outputs of the method as the control variables. In Figure 2.2, `pos` serves as the control variables, since they can decide if `newRel` is should be inserted or not. In our approach, the values from all control variables (*p-use* [46]) are recorded as inputs.

After a static analysis determines which variables are inputs and which are outputs, collecting them is simple: during program execution, we record the value of each input

Table 2.2: The input sources in the `addRelative` method.

Var.	Notes
`me`	`me` has the computational dependency to the output `ret`.
`rName`	`rName` is written to `newRel` that contributes to `me`.
`rAge`	`rAge` is written to `newRel` that contributes to `me`.
`pos`	`pos` has the computational dependency to the output `me`.

and output variable when a method is called, creating an *I/O record* for each method. Over the program execution, many unique I/O records will likely be collected for each method.

2.4 Similarity Model: Mining Functionally Equivalent Methods

After collecting all of these I/O records, the final phase in our approach is to evaluate the pairwise similarity between these methods based on their I/O sets. However, there are likely to be many different invocations of each method, and many methods to compare, requiring $O(\binom{m}{2}(n)^2)$ comparisons between m methods and n invocation histories for each method. To simplify this problem, we first create summaries of each method that can be efficiently compared, and then use these summaries to perform high-level similarity comparison. The result may be that two methods have slightly different input and output profiles, but nonetheless are flagged as functional clones. This is a completely intentional result from our approach, based on the insight that in some cases, developers may use different structures to represent the same data.

Consider the two code listings shown in Figure 2.3 — real Java code found to be functional clones by HitoshiIO. Note that at first, the two methods accept different (formal) input parameters: but in reality, both *use* an array as inputs (the second example accesses

```java
long getSum(long[] n, int L, int
    R) {
  long sum = 0;
  if (R >= 0) {
    sum = n[R];
  }
  if (L > 0) {
    sum -= n[L - 1];
  }
  return sum;
}
```

```java
public static long sum(int a, int
    b)
{
  if(a > b)
  {
    return 0;
  }

  return array[b + 1] - array[a];
}
```

Figure 2.3: A functional clone detected by HitoshiIO.

an array that is a static field, while the first accepts an array as a parameter). For the case of $L <= R, a <= b$, the behavior will be very similar in both examples: the result will be the difference between two array elements, one at $b + 1$ (or R), and the other at a (or $L - 1$). We want to consider these functions behaviorally similar, despite these minor differences.

Before detailing our similarity model, we first discuss the concept of *DeepHash* [30] used in our similarity model. The general idea of DeepHash is to recursively compute the hash code for each element and field, and sum them up to represent non-primitive data types. For this purpose, for floating point calculations, we round them to two decimal places, although this functionality is configurable. With the DeepHash function, HitoshiIO can parse a set containing different objects into a representative set of deep hash values, which facilitate our similarity computation.

The strategy of the DeepHash is as follows:

- If there is already a `hashCode` function for the value to be checked, then call it directly to obtain a hash code.
- If there is no existing `hashCode` function for an object, then recursively collect the values of the fields owned by the object and call the DeepHash to compute the hash code for this object.
- For arrays and collections, compute the hash code for each element by DeepHash

and sum them up as the hash code.

- For maps, compute the hash code of each key and values by the DeepHash and sum them up.

The notations we use in the similarity model are as follows.

- m_i: The i_{th} method in the codebase.
- $inv_{r|m_i}$: The r_{th} invocation of m_i.
- $ISrc(inv_{r|m_i})$: the input set of $inv_{r|m_i}$.
- $OSink(inv_{r|m_i})$: the output set of $inv_{r|m_i}$.
- $ISrc_h(inv_{r|m_i})$: the deep hash set of $ISrc(inv_{r|m_i})$.
- $OSink_h(inv_{r|m_i})$: the deep hash set of $OSink(inv_{r|m_i})$.
- MP_{ij}: A method pair contains two methods from the codebase, where $i \neq j$.
- $IP_{r|i,s|j}$: An invocation pair contains $inv_{r|m_i}$ and $inv_{s|m_j}$.

To compare an $IP_{r|i,s|j}$ from two methods, m_i and m_j, we first computes the Jaccard coefficients for $ISrc$s and $OSink$s as the basic components for the functional similarity. The definition for the Jaccard similarity [82] is as follows:

$$J(Set_i, Set_j) = \frac{|Set_i \cap Set_j|}{|Set_i \cup Set_j|} \tag{2.3}$$

If either set is empty, this will compute their coefficient as 0. To simplify the notations, we define the basic similarities between $ISrc$s and $OSink$s as follows.

$$Sim_I(IP_{r|i,s|j}) = J(ISrc_h(inv_{r|m_i}), ISrc_h(inv_{s|m_j})) \tag{2.4a}$$

$$Sim_O(IP_{r|i,s|j}) = J(OSink_h(inv_{r|m_i}), OSink_h(inv_{s|m_j})) \tag{2.4b}$$

The basic similarity represents how similar two $ISrc$s or $OSink$s are. To summarize the I/O functional similarity for a pair of methods, we propose an *exponential* model

$$Sim(IP_{r|i,s|j}) = \frac{(1 - \beta * e^{Sim_I}) * (1 - \beta * e^{Sim_O})}{(1 - \beta * e)^2} \tag{2.5}$$

, where β is a constant. This exponential model punishes the invocation pairs that have either similar $ISrc$ or $OSink$, but not the other. By this similarity model, we can sharply differentiate invocation pairs having similar I/Os from the ones that solely have similar inputs or outputs. We can finally define the similarity for a method pair MP_{ij} as the best similarity of their invocation pairs $IP_{r|i,s|j}$.

$$Sim(MP_{ij}) = \max Sim(IP_{r|i,s|j}) \tag{2.6}$$

In addition to using the best match among invocation pairs, we plan to use *preponderant* matches in the future

$$\underset{\substack{r \in \{inv_{m_i}\} \\ s \in \{inv_{m_j}\}}}{\text{argmax}} \sum_r Sim(inv_{r|m_i}, f : inv_{r|m_i} \rightarrow inv_{s|m_j}) \tag{2.7}$$

, r is an invocation of m_i, s is an invocation of m_j and f is a function to pair the r invocation of m_i with the s invocation of m_j. We plan to use the Hungarian algorithm [78] to solve this assignment problem between invocations of two programs. A potential problem here is that the Hungarian algorithm is computationally expensive with $O(n^3)$ time complexity, where n is the invocation number of a method. A discussion regarding how to use the Hungarian Algorithm to solve preponderant matches is in Section 5.4.

While how to compute the set similarity between program I/Os is an open problem, we believe that there are other potential directions to explore in addition to best and preponderant match. For example, if we can represent inputs and outputs of programs as distributions, we can recruit Bregman divergence [15] to measure distance (similarity) between two programs. An obstacle we can foresee is to devise a generic representation to project I/Os from each program to meaningful numeric values, which we plan to solve in the future.

2.5 Implementation of HitoshiIO

To demonstrate and evaluate in-vivo clone detection, we create HitoshiIO, with a name inspired by the Japanese word for "equivalent": *hitoshii*. HitoshiIO records and compares the inputs and outputs between Java methods, considering every method as a possible clone of every other. In principle, we could extend HitoshiIO to consider code fragments - individual parts of methods, but we leave this implementation to future work. HitoshiIO is implemented using the ASM bytecode rewriting toolkit [11], operating directly on Java bytecode, requiring no access to application or library source code. HitoshiIO is available on GitHub and released under an MIT license.

2.5.1 Java Background

Before describing the various implementation complexities of HitoshiIO, we first provide a brief review of data organization in the JVM. According to the official specification of Java [62], there are two categories of data types: *primitive* and *reference* types. The primitive category includes eight data types: boolean, byte, character, integer, short, long, float and double. The reference category includes two data types: objects and arrays. Objects are instances of classes, which can have fields [62]. A field can be a primitive or a reference data type. An array contains element(s), where an element is also either a primitive or a reference data type.

Primitive types are passed by value, while reference types are passed by reference value. HitoshiIO considers all types of variables as inputs and outputs.

2.5.2 Identifying Method Inputs and Outputs

Our approach relies on first identifying the outputs of a method, and then backtracking to the values that influence those outputs, in order to detect inputs. The first step is identifying the outputs of a given method. For a method m, its output set consists of all variables written by m that are observable outside of m. An output could be a variable passed to another method, returned by the method, written to a global variable (`static` field in the JVM), or written to a field of an object or array passed to that method. By default, HitoshiIO only considers the formal parameters of methods, ignoring the owner object (if the method call is at instance level) in this analysis, although this behavior is configurable.

This approach would, therefore, consider every variable passed from method m_1 to method m_2 to be an output of m_1. As an optimization, we perform a simple intra-procedural analysis to identify methods that do not propagate any of their inputs outside of their own scope (i.e., they do not effect any future computations). For these special cases, HitoshiIO identifies that at call-sites of these special methods, their arguments are not actually outputs, in that they do not propagate through the program execution. To further reduce the scope of potential output variables, we also exclude variables passed as parameters to methods that do not directly write to those variables as inputs. We found that these heuristics work well towards ensuring that HitoshiIO can execute within a reasonable amount of time, and discuss the overall performance of HitoshiIO in §2.6.

Once outputs are identified, HitoshiIO performs a static data and control flow analysis for each method, identifying for each output variable, all variables which influence that output through either control or data dependencies. Variable v_o is dependent on v_i if the value of v_o is derived from v_i (data dependent), or if the statement assigning v_o is controlled by v_i. We recursively apply this analysis to determine the set of variables that influence the output set $OSink$, creating the set of variables $Parents(OSink)$. Variable v_i in method m is an input if it is $Parents(OSink)$ and its definition occurs outside of the scope of m. HitoshiIO then identifies the instructions that load inputs and return outputs, for use in the next step - instrumentation.

2.5.3 Instrumentation

Given the set of instructions that may load an input variable or store an output, HitoshiIO inserts instrumentation hints in the application's bytecode to record these values at runtime. Table 2.3 describes the various relevant bytecode instructions, their functionality, and the relevant categorization made by HitoshiIO (**In**put instruction or **Out**put instruction). HitoshiIO treats the values consumed by the control instructions as inputs. Just after an instruction that loads a value judged to be an input, HitoshiIO inserts instructions to record that value; just before an instruction that stores an output value, HitoshiIO similarly inserts instructions to record that value.

2.5.4 Recording Inputs and Outputs at Runtime

The next phase of HitoshiIO is to record the actual inputs and outputs to each method as we observe the execution of the program. Although the execution of the program is guided by relatively high level inputs (e.g., unit tests, which each likely calls more than one single method), the previous step (input and output identification) allows us to carve out inputs and outputs to individual methods - it is these individual inputs and outputs that we record.

HitoshiIO's runtime recorder serializes all previously identified inputs immediately as they are read by a method, and all outputs immediately before they are written. For Java's primitive types (and Strings), the I/O recorder records the values directly. For objects, including arrays, HitoshiIO follows [32] to adopt the XStream library [130] to serialize these objects in a generic fashion to XML. Once the method completes an execution, this *execution profile* is stored as a single XML file in a local repository for offline analysis in the next step.

Table 2.3: The potential instructions observed by HitoshiIO.

Opcode	Type	Description
`xload`	In.	Load a primitive from a local variable, where `x` is a primitive.
`aload`	In.	Load a reference from a local variable.
`xaload`	In.	Load a primitive from a primitive array, where `x` is a primitive.
`aaload`	In.	Load a reference from a reference array.
`getstatic`	In.	Load a value from a field owned by a class.
`getfield`	In.	Load a value from a field owned by an object.
`arraylength`	In.	Read the length of an array.
`invokeXXX`	Out.	Call a function
`xreturn`	Out.	Return a primitive value from the method, where `x` is a primitive.
`areturn`	Out.	Return a reference from the method.
`putstatic`	Out.	Write a value to the field owned by a class.
`putfield`	Out.	Write a value to the field owned by an object.
`xastore`	Out.	Write a primitive to a primitive array.
`aastore`	Out.	Write a reference to a reference array.
`ifXXX`	Con.	Represent all `if` instructions. Jump by comparing value(s) on the stack.
`tableswitch`, `lookupswitch`	Con.	Jump to a branch based on the index on the stack.

2.5.5 Similarity Computation

Recall that our goal is to find *similarly functioning* methods, not methods that present the exact same output for the exact same input. Hence, our similarity computation mechanism needs to be sufficiently sensitive to identify when two methods function "significantly" differently for the same input, but at the same time ignore trivial differences (e.g., the specific data structure used, order of inputs, additional input parameters that are used). To capture this similarity, we use a Jaccard coefficient (as described in Section 2.4) - a relatively efficient and effective measure of the similarity between two sets. A high Jaccard coefficient indicates a good similarity, and a low coefficient indicates a poor match.

While it is relatively straightforward to compare simple, primitive values (including Strings) in Java directly, comparing complex objects of different structures is non-trivial: one of the key technical roadblocks reported in Deissenbock et al.'s earlier work [32]. To solve this problem, we adopt the *DeepHash* [30] approach, creating a hash of each object. The details of *DeepHash* can refer to Section 2.4.

The similarity model of HitoshiIO follows Section 2.4. Optimizing the parameter setting for HitoshiIO's similarity model is extremely expensive. For each different setting, we need to conduct a user study to determine if more or less functional clones can be detected, which is inapplicable. We conduct multiple small scale experiments (i.e., pick a small set of our study subjects) with different βs. Then we manually verify the results to determine the local optimized value for β, which is 3, for the exponential model of Eq. 2.5 in HitoshiIO. We plan to leverage the power of machine learning to automatically learn the best β for HitoshiIO in the future.

HitoshiIO has two other parameters that control its similarity matching procedure: $InvT$ and $SimT$. We recognize that some hot methods may be invoked millions of times — while others invoked only a handful. $InvT$ provides an upper-bound for the number of individual method input-output profiles that are considered for each method. $SimT$ provides a lower-bound for how similar two methods must be to be reported as a functional clone. We have evaluated various settings for these parameters, and discuss them in greater detail in Section 2.6.

2.6 Experiments of HitoshiIO

To evaluate the efficacy of HitoshiIO, we conduct a large scale experiment in a codebase to examine functional clones detected by HitoshiIO. We set out to answer the following three research questions:

Table 2.4: A summary of the experimental codebase containing projects from the Google Code Jam competitions.

		Total # of		Avg per-method	
Year	Problem Set	Projects	Methods	Invocations	LOC
2011	Irregular Cake	30	201	24	11.2
2012	Perfect Game	34	241	21	6.4
2013	Cheaters	21	163	26	9.2
2014	Magical Tour	33	220	20	8.1
Across all projects		118	825	22	8.6

RQ1: Does HitoshiIO find functional clones, even given limited inputs and invocations?

RQ2: Is the false-positive rate of HitoshiIO low enough to be potentially usable by developers?

RQ3: Is the performance of HitoshiIO sufficiently reasonable to use in practice?

Because HitoshiIO is a dynamic system that requires a workload to drive programs, we selected the Google Code Jam repository [51], which provides input data, as the codebase of our experiments. The Code Jam is the annual programming competition hosted by Google. The participants need to solve the programming problems provided by Google Code Jam and submit their solutions as applications for Google to test. The projects that pass Google's tests are published online.

Each annual competition of Google Code Jam usually has several rounds. We examine the projects from four years (2011-2014), and consider the projects that passed the third round of competitions. We only pick the projects that do not require a user to input the data, which can facilitate the automation of our experiments. Descriptive details for these projects, which form our experimental codebase, can be found in Table 2.4. For measurement purposes, we only consider methods defined in each project — and not those provided by the JVM, but used by the project. We also exclude constructors, static constructors, `toString`, `hashCode` and `equals` methods, since they usually don't provide logic.

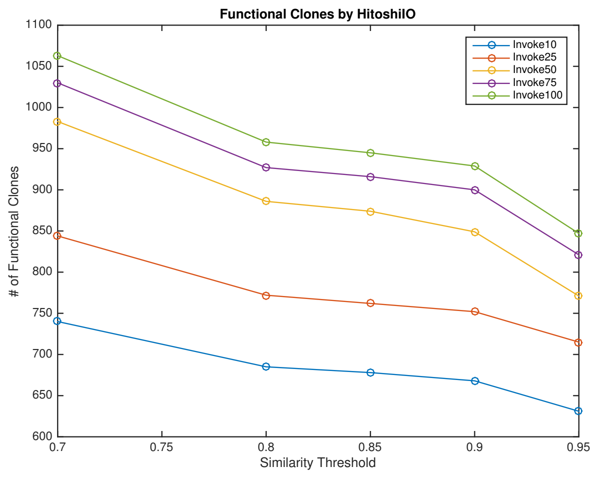


Figure 2.4: The number of functional clones detected by HitoshiIO with different parameter settings.

HitoshiIO observes the execution of each of these methods, exhaustively comparing each pair of methods. In this evaluation, we configured HitoshiIO to ignore comparing methods for similarity that were written by the same developer in the same year. This heuristic simulates the process of a new developer entering the team, and looking for functionally similar code that might look different — reporting functional clone m_2 of m_1 where both m_1 and m_2 were written by the same developer at the same time is unlikely to be particularly helpful or revealing, since we hypothesize that these are likely syntactically similar as well. This suite of projects allows us to draw interesting conclusions about the variety of functional clones detected: are there more functional clones found between multiple implementations of the same overall goal (i.e., between projects in the same year written by different developers), or are there more functional clones found between different kinds of projects overall (i.e., between years)?

We performed all of our similarity computations on Amazon's EC2 infrastructure [5], using a single c4.8xlarge machine, equipped with 36 cores and 60GB of memory.

2.6.1 RQ1: Clone Detection Rate

We manipulate two parameters in HitoshiIO, *invocation threshold, InvT* and *similarity threshold, SimT*, to observe the variation of the number of the detected functional clones. The invocation threshold represents how many *unique* I/O records should be generated from invocations of a method. The way that we define the uniqueness of I/O records is by the hash value derived from their $ISrc$s and $OSink$s. HitoshiIO stops generating I/O records for a method, when its invocation threshold is achieved. Intuitively, more functional clones can be detected with a higher invocation threshold. The similarity threshold sets the lower-bound for how similar two methods must be to be reported as a clone.

Figure 2.4 shows the number of functional clones detected by HitoshiIO while varying the similarity threshold (x-axis) and the invocation threshold (each line). With $InvT \geq 50$, the number of the detected functional clones does not increase too much. However, there is a remarkable increase from $InvT = 25$ to $InvT = 50$. If we fix the $SimT$ to 85%, the difference of detected clones between $InvT = 25$ and $InvT = 50$ is 114, but the difference between $InvT = 50$ and $InvT = 100$ is only 71. Figure 2.4 also shows that the number of clones does not sharply decrease between $SimT = 0.8$ to $SimT = 0.9$. Thus, for the remainder of our analysis, we set $InvT = 50$ and $SimT = 0.85$, and evaluate the quality and number of clones detected with these parameters.

Given this default setting, HitoshiIO detects a total of 874 clones, which contain 185 distinctive methods that average 10.5 lines of code each. The methods found to be clones were slightly larger on average than most methods in the dataset. Table 2.5 shows the distribution of clones, broken down between the pair of years that each method was found in, and the size of each clone (less than or equal to 5 lines of code, or larger). In total, HitoshiIO found 385 clones with $LOC \leq 5$ (44%), while 489 of them are larger than 5 LOC (56%). About half of the clones were found looking between multiple projects in the same year (recall that projects in the same year implement different solutions to the

Table 2.5: The distributions of clones detected by HitoshiIO cross the problem sets.

	Number of Clones			Methods Compared	Analysis Time (mins)
Year Pair	≤ 5 LOC	> 5 LOC	Total		
2011 – 2011	20	14	34	11.6M	1.2
2012 – 2012	100	32	132	11.8M	0.9
2013 – 2013	18	144	162	8.4M	0.8
2014 – 2014	41	65	106	9.3M	0.9
2011 – 2012	25	26	51	24.4M	1.9
2011 – 2013	16	24	40	20.8M	1.8
2011 – 2014	36	40	76	21.6M	2.2
2012 – 2013	29	30	59	21.0M	1.7
2012 – 2014	59	61	120	21.8M	1.5
2013 – 2014	41	53	94	18.6M	1.6
Total	385	489	874	169.5M	14.5

same overall challenge), despite there being fewer potential pairs evaluated ("Methods Compared" column). This interesting result shows that there are many functional clones detected between projects that have the same overall purpose, but there are still plenty of functional clones detected among projects that do *not* share the same general goal (comparing between years).

While we did find many clones, our total clone rate, defined to be the number of methods that were clones over the total number of methods, was $185/825 = 22\%$. It is difficult for us to approximate whether HitoshiIO is detecting all of the functional clones in this corpus, as there is no ground truth available. Other relevant systems, e.g. Elva and Leavens' IOE clone detector, were unavailable, despite contacts to the authors [39]. Deissenboeck et al.'s Java system [32], although not available to us, found far fewer clones with a roughly 1.64% clone rate on a different dataset, largely due to technical issues running their clone detection system. Assuming that the clones we detected truly are functional clones, then we are pleased with the quantity of clones reported by HitoshiIO: there are plenty of reports.

2.6.2 RQ2: Quality of Functional Clones

To evaluate the precision of HitoshiIO, we randomly sampled the 874 clones reported in this study (RQ1), selecting 114 of the clones (approximately 13% of all clones). These 114 functional clones contain 111 distinctive methods with 7.3 LOC in average. For these clones, we recruited two masters students from the Computer Science Department at Columbia University to each examine half (57) of the sampled clones, and determine if they truly were functional clones or not. These students had no prior involvement with the project and were unfamiliar with the exact mechanisms originally used to detect the clones. But they were given a high level overview of the problem, and were requested to report if each pair of clones is functionally similar. The first verifier had 1.5 year of experiences with Java, including constructing research prototypes. The second verifier had 3 years of experiences with Java, including industrial experiences as a Java developer.

We asked the verifiers to mark each clone they analyzed by 3 categories: false positive, true positive, and unknown. To prevent our verifiers from being stopped by some complex clones, we set a (soft) 3-minute threshold for them to analyze each functional clone, at which point they mark the clone as unknown. Both verifiers completed all verifications between 2 to 2.5 hours.

Among these 114 functional clones, $78(68.4\%)$ are marked as true positive, $19(16.6\%)$ are marked as unknown and $17(14.9\%)$ are labeled as false positive. If we only consider the categories of false and true positive, the precision can be defined as

$$precision = \frac{\#TP}{\#FP + \#TP} \tag{2.8}$$

The precision of HitoshiIO over all sampling functional clones is 0.82.

Our student-guided precision evaluation is difficult to compare to previous functional clone works (e.g., [32, 60, 39]), as previous works haven't performed such an evaluation. However, overall we believe that this relatively low false positive rate is indicative that

HitoshiIO can be potentially used in practice to find functionally similar code.

2.6.3 RQ3: Performance

There are several factors that can contribute to the runtime overhead of HitoshiIO: the time needed to analyze and instrument the applications under study, the time to run the applications and collect the individual input and output profiles, and the time to analyze and identify the clone pairs. The most dominant factor for execution time in our experiments was the clone identification time: application analysis was relative quick (order of seconds), and the input-output recorder added only a roughly 10x overhead compared to running the application without any profiling (which was also on the order of seconds). As shown in Table 2.5, the total analysis time for similarity computation needed to detect these 874 clones was relatively quick though: only 14.5 minutes.

The analysis time is very directly tied with the $InvT$ parameter, though: the number of unique input-output profiles considered for each method in the clone identification phase. We varied this parameter, and observed the number of clones detected, as well as the analysis time needed to identify the clones, and show the results in Table 2.6. For each value of $InvT$, we show the number of clones detected, the clone rate, the number of clones that were verified as true positives (in the previous section), but missed, and the total analysis time.

Even considering very few invocations (10) with real workloads, HitoshiIO still detects most of the clones, with very low analysis cost. The time complexity to compute the similarities for all invocations is $O(n^2)$, where n is the number of invocations from all methods. This implies that the processing time under $InvT = 25$ is about 25% of the baseline, but it can detect 95% of the ground truth with real workloads. This result is compelling because: (1) it shows that HitoshiIO's analysis is scalable, and can be potentially used in practice, and (2) it shows that even with very few observed executions (e.g., due to sparse existing workloads), functional clones can still be detected.

Table 2.6: The number of missed functional clones with different invocation thresholds detected by HitoshiIO.

	Clones Detected		Clones	Analysis Time
$InvT$	Total	Clone Rate	Missed	(mins)
10	678	20.6%	10	0.6
25	762	21.6%	4	3.8
50	874	22.4%	0	14.5
75	916	22.5%	0	32.5
100	945	22.8%	0	56.6

2.7 Discussions

In designing our experiments, we attempted to reduce as many potential risks to validity as possible, but we acknowledge that there may nonetheless be several limitations. For instance, we selected 118 projects from the Google Code Jam repository for study, each of which may not necessarily represent the size and complexity of large scale multi-developer projects. However, this choice allowed us to control the variability of the clones: we could look at multiple projects within a year, which would show us method-level functional clones between projects that have the same overall goal, and projects across different years, which would show us those method-level clones between projects that have completely different overall goals. Future evaluations of HitoshiIO will include additional validation that similar results can be obtained on larger, and more complex applications.

For our evaluation of false positives, we recognize the subjective nature of having a human recognize that two code fragments are functionally equivalent. However, we believe that we provided adequate training to well-experienced developers who could therefore, judge whether code was functionally similar or not, especially given the relatively small size of most of the clones examined. Given additional resources, cross-checking the experimental results between users might increase our confidence in evaluating HitoshiIO in the future.

Ideally, we would be able to test HitoshiIO against a benchmark of functional clones: a suite of programs, with inputs, that have been coded by other researchers to provide a ground truth of what functional clones exist. Unfortunately, clone benchmarks (e.g., [76, 124]) are designed for *static* clone detectors, and do not include any workloads to use to drive the applications, making them unsuitable for a dynamic clone detector like HitoshiIO.

There are also several implementation limitations that may be causing the number of clones that HitoshiIO detects to be lower than it should be. For instance, the heuristics that it uses to decide what an I/O is are not sound (Section 2.5.2), which may result in identifying fewer I/Os than it ought to. These limitations do not effect the validity of our experimental results, as any implementation flaws would hence be reflected in the results. To enhance HitoshiIO, we propose to have future developments in the next section.

To offer a more advanced mechanism to identify I/Os of programs, we plan to apply a taint tracking system such as [18] to capture these I/Os. Currently HitoshiIO records inputs and outputs as sets without considering item orders. We expect to develop a new feature that allows users to decide if their data should be stored in sequence or not. Since our target is to explore programs with similar functionalities, code coverage rate is not our main concern. However, we are interested in examining the relation between code coverage rate and detection rate of functional clones in HitoshiIO. For enhancing the similarity computation, given a method pair, we plan to observe the correlation between *all* invocation pairs between them, instead of the current approach to select the one with the highest similarity.

2.8 Conclusions

Prior work has underscored the challenges of detecting functionally similar code in object oriented languages. In this thesis, we presented our approach, In-Vivo Clone Detection, to effectively detect functional clones. We implemented such approach in our system for Java, HitoshiIO. Instead of fixing the definitions of program I/Os, HitoshiIO applies static data flow analysis to identify potential inputs and outputs of individual methods. Then, HitoshiIO uses real workloads to drive the program and profiles each method by their I/O values. Based on our similarity model to compare each I/O profile, HitoshiIO detected $800+$ functional clones in our evaluation with only 15% false positive rate.

With these results, we enable future research that will leverage the information of function clones. For instance, can functional clones help in API refactoring? Can we help developers understand new code by showing them some previous examined code that is functionally similar? We have made our system publicly available on GitHub, and are excited by the future investigations and developments in the community.

Chapter 3

Execution as Behavioral Feature

Detecting "similar code" is useful for many software engineering tasks. Current tools can help detect code with statically similar syntactic and–or semantic features (code clones) and with dynamically similar functional input/output (simions). Unfortunately, some code fragments that behave similarly at the finer granularity of their execution traces may be ignored. In this thesis, we propose the term "*code relatives*" to refer to code with similar execution behavior. We define code relatives and then present DyCLINK, our approach to detecting code relatives within and across codebases. DyCLINK records instruction-level traces from sample executions, organizes the traces into instruction-level dynamic dependence graphs, and employs our specialized subgraph matching algorithm to efficiently compare the executions of candidate code relatives. In our experiments, DyCLINK analyzed 422+ million prospective subgraph matches in only 43 minutes. We compared DyCLINK to one static code clone detector from the community and to our implementation of a dynamic simion detector. The results show that DyCLINK effectively detects code relatives with a reasonable analysis time.

3.1 Introduction

Code clones [113], which represent textually, structurally, or syntactically similar code fragments, have been widely adopted to detect similar pieces of software. However, code clone detection systems typically focus on identifying static patterns in code, so relevant code fragments that behave similarly at runtime, though with different structures, are ignored. Detecting code fragments that accomplish the same tasks or share similar behavior is pivotal for understanding and improving the performance of software systems. For example, with such functionality, it may be possible to automatically replace an old algorithm in a legacy system with a new one or to detect commonly repeated tasks to create APIs (semi)automatically. It may allow quick search and understanding of large codebases, and de-obfuscation of code.

Towards detecting similarly behaving code, previous work observed code fragments that yield the same output for the same input [60, 38] or that share similar identifiers and structural concepts [93, 16, 98]. A significant challenge in detecting *similar* but not equivalent code fragments by comparing input and output pairs, a technique also known as finding *simions* [61], is judging how similar two outputs need to be for the two code fragments to be considered simions. Particularly, with object-oriented languages, this problem may be more complex: the same data can be designed with different project-specific data types between projects [32].

Our key insight is to shift this similarity comparison to study how each code fragment computes its result, rather than simply comparing those output results or comparing what that code looks like. That is, we can gauge how similar two code fragments are without even looking at the respective inputs and outputs. To represent runtime similarity (i.e., how the code fragment computes its result), we introduce the term *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. The key relationship between these code fragments is that they are performing a similar computation regardless of how similar or dissimilar their outputs may be.

Our key contribution is an efficient system for detecting these code relatives that is agnostic to the output format or identifiers used in the code. Our system, DyCLINK, traces each program's execution creating a dynamic dependency graph that captures behavior at the instruction level. These dependency graphs encode rich and dense behavioral information, more than would be found simply observing the outputs of parts of a program or obtained from a static analysis of that program. Code relatives are isomorphic (sub)graphs via fuzzy matching that occur repeatedly between and within these profiled execution graphs. DyCLINK detects code relatives at any granularity: a code relative may be a part of a single method or instead be composed of several methods that are executed in a sequence.

The resulting graphs are large: containing a single node for every instruction, plus edges representing dependencies. Hence, typical approaches for detecting isomorphic (sub)graphs are time prohibitive — requiring expensive comparisons between each potential set of code relatives. In our evaluation, we examined 118 projects for code relatives, containing a total of $1,244$ different dynamic dependence graphs, which represented a total of over 422 million subgraphs that would be compared for similarity. To efficiently identify code relatives in these graphs, we have developed a new algorithm, *LinkSub*, that leverages the PageRank[80] algorithm to compare subgraphs and to reduce the number of pairwise comparisons needed between subgraphs to efficiently detect code relatives (in our evaluation, filtering away over 99% of the comparisons).

We built DyCLINK, targeting Java programs, but our methodology applies to most high level languages. We evaluated DyCLINK on a corpus of Java programs that were known to contain clusters of similar programs. DyCLINK effectively reconstructed the clusters of programs with very high precision (90+%). We compared DyCLINK with one state-of-the-art static clone detector plus one dynamic simion detector (input-output similarity checker), finding it to be more effective at clustering similarly behaving software.

The main contributions in this section are:

- We define *Code Relatives*, code sharing similar fine-grained, instruction-level dynamic behavior, and their utility.
- We design and implement the DyCLINK system to detect code relatives. DyCLINK uses dynamic execution traces for identifying code relatives. We have released Dy-

CLINK on GitHub under an MIT license [37].

- The key to 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 large numbers of instructions and dependencies.
- We present a highly-accurate method for classifying programs by running the K Nearest Neighbors (KNN) [4] algorithm among code relatives.

3.2 Background

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

3.2.1 Basic Definitions

When discussing the notion of *similar* code, it is important to have a clear definition of what *similar* means. For our purpose, two code fragments are similar if they produce similar instruction-level runtime behavior, which is witnessed by execution traces (dynamic dependency graphs) that are roughly equivalent.

- **Code fragment:** Either a continuous or discontinuous set of code lines.
- **Code clone:** "A code fragment CF_2 is a clone of another code fragment CF_1 if they are similar by some given definition of similarity" [113]. We express this as follows. CF_1 and CF_2 are code clones if:

$$f_{sim}(CF_1, CF_2) \geq \theta_{stat} \tag{3.1}$$

where f_{sim} is a similarity function and θ_{stat} is a pre-defined threshold for static code fragments.

- **Code relative:** An execution of a code fragment generates a trace of that execution, $Exec(CF)$. We denote the set of a code fragment's traces as $\{Exec(CF)\}$. Given a similarity function f_{sim} and a threshold θ_{dyn} for code execution, two code fragments, CF_1 and CF_2, are code relatives if:

$$f_{sim}(\{Exec(CS_1)\}, \{Exec(CS_2)\}) \geq \theta_{dyn} \tag{3.2}$$

In this work, we capture execution traces as dynamic program dependency graphs, and we model the similarity between two code fragments as a subgraph isomorphism problem described further in Section 3.4.

Code relatives are distinct from "simions" in that simions are code fragments that show similar outputs given an input, while code relatives show similar behavior, regardless of their outputs [61]. Moreover, code relatives may consider discontinuous code fragments and include cases in which their intermediate results (but not outputs) are similar. Code relatives are not tied to a particular programming abstraction: a code relative may be a portion of a method, or may represent computation that is performed across several methods. All code relatives are behavioral code clones given that the definition of "similarity" is limitless for clones in general. We use the term code relative rather than a variant of code clone or simion to make their distinctions clear and avoid ambiguity.

3.2.2 Motivation

Detecting similar programs is beneficial in supporting several software engineering tasks, such as helping developers understand and maintain systems [98], identifying code plagiarism [89], and enabling API replacement [73]. Although code clone detection systems can efficiently detect structurally similar code fragments, they may still miss some cases for optimizing software and–or hardware that require information about runtime behavior [34]. Programs that have syntactically similar code fragments usually have similar behavior; however programs can still have similar behavior even if their code is not alike [61, 60].

Moreover, programs may have similar behavior even if their outputs for the same or equivalent inputs are not identical. In fact, in many cases, it may be difficult to judge that two outputs are equivalent, or even similar, due to differences in data structures. On detecting functionally equivalent code fragments in Java, Deissenboeck et al. reported that 60-70% of the studied program chunks across five open-source projects referred to project-specific data types[32]. Hence, it is impossible to directly compare inputs and outputs for equivalence across many projects. To get around these dissimilar data types, developers would have to specify adapters to convert from project-specific datatypes to abstract representations that could be compared. By ignoring the outputs of code fragments and observing only their behavior, we can avoid this output equivalence problem.

Consider, for example, the two code examples shown in Figure 3.1, taken from the libraries *Apache Commons Math*[1] and *Jama*[2], both of which perform the same matrix decomposition task. In the case of Figure 3.1a, all computation is done in a single method and the result is stored as instance fields of the object being constructed. In the case of Figure 3.1b, computation is split between several methods: `solve`, which invokes several methods to compute the result, which is returned as a `Matrix` object (a type defined by the library). A simion detector (comparing inputs and outputs) would have difficulty to compare the inputs and outputs when the data structures do not match exactly, and there may not be clearly defined outputs. A typical clone detector using the abstract syntax tree of this code would also find it hard to detect the multi-method clone. It would need to compute callgraph information to consider valid multi-method clones, which again, have many subtle differences in code structure. In fact, clone detection tools may not consider these two code listings to be clones, while we argue that they are code relatives and are

[1]https://commons.apache.org/proper/commons-math/

[2]http://math.nist.gov/javanumerics/jama/

```java
public SingularValueDecomposition(final
    RealMatrix matrix) {
  ....
  // Generate U.
  ....
  for (int k = nct - 1; k >= 0; k--) {
    if (singularValues[k] != 0) {
      for (int j = k + 1; j < n; j++) {
        double t = 0;
        for (int i = k; i < m; i++) {
          t += U[i][k] * U[i][j];
        }
        t = -t / U[k][k];
        for (int i = k; i < m; i++) {
          U[i][j] += t * U[i][k];
        }
      }
    ...
  }
  // Generate V.
  for (int k = n - 1; k >= 0; k--) {
    if (k < nrt &&
      e[k] != 0) {
      for (int j = k + 1; j < n; j++) {
        double t = 0;
        for (int i = k + 1; i < n; i++) {
          t += V[i][k] * V[i][j];
        }
        t = -t / V[k + 1][k];
        for (int i = k + 1; i < n; i++) {
          V[i][j] += t * V[i][k];
        }
      }
    }
    ...
  }
}
```

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

```java
public Matrix solve (Matrix B) {
  return (m == n ? (new LUDecomposition(
    this)).solve(B) : (new
    QRDecomposition(this)).solve(B));
}
public QRDecomposition (Matrix A) {
  ...
  for (int k = 0; k < n; k++) {
    ...
    if (nrm != 0.0) {
      ...
      for (int j = k+1; j < n; j++) {
        double s = 0.0;
        for (int i = k; i < m; i++) {
          s += QR[i][k]*QR[i][j];
        }
        s = -s/QR[k][k];
        for (int i = k; i < m; i++) {
          QR[i][j] += s*QR[i][k];
        }
      }
    }
    Rdiag[k] = -nrm;
  }
}
public Matrix solve (Matrix B) {
  ...
  for (int k = 0; k < n; k++) {
    for (int j = 0; j < nx; j++) {
      double s = 0.0;
      for (int i = k; i < m; i++) {
        s += QR[i][k]*X[i][j];
      }
      s = -s/QR[k][k];
      for (int i = k; i < m; i++) {
        X[i][j] += s*QR[i][k];
      }
    }
  }
  ...
}
```

(b) Jama's `Matrix.solve`

Figure 3.1: A partial comparison of matrix decomposition code from two different libraries. Despite differences in each method, both are code relatives.

indeed detected by DyCLINK.

Software clustering and *Code search* are two domains that rely on similarity detection between programs and could benefit from code relatives. Software clustering locates and aggregates programs having similar code or behavior. The clusters support developers understanding code semantics [77, 91], prototyping rapidly [22], and locating bugs [36]. Code search helps developers determine if their codebases contain programs befitting their requirements [98]. 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 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. A system that captures more details and represents program behavior more effectively (e.g., instead of simions) 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 or dynamic simions, we designed DyCLINK, a system to detect dynamic *Code Relatives*, which represent similar runtime behavior between programs. We have evaluated DyCLINK, finding it to have high precision (90+%) when applied to software clustering, results discussed in Section 3.5.

3.3 Related Work

Code similarity detection tools can be distinguished by the similarity metrics that they use, exact or fuzzy matching, and the intermediate representation used for comparison. Common intermediate representations tend to be token-based [14, 63, 86], AST-based [17, 59], or graph-based [17, 59, 71, 70, 73, 89]. Deckard [59] detects similar but perhaps structurally different code fragments by comparing ASTs. SourcererCC[115] compares code fragments using a static bag-of-tokens approach that is fast, but does not target specifically similarly behaving code with different structures.

Among static approaches, DyCLINK is most similar to those that used program dependence graphs (PDGs) to detect clones. Komondoor and Horwitz [70] generate PDGs for C programs, and then apply program slicing techniques to detect isomorphic subgraphs. The approach designed by Krinke [73] starts to detect isomorphic subgraphs with maximum size k after generating PDGs. The granularity of Krinke's PDGs is finer than the traditional one: each vertex roughly maps to a node in an AST. The approach proposed by Gabel et al. [49] is a combination of AST and graph. It generates the PDG of a method, maps that PDG back to an AST, and then uses Deckard [59] to detect clones. GPLAG [89] determines when to invoke the subgraph matching algorithm between two PDGs using two statistical filters.

Compared with these graph-based approaches that identify *static* code clones, DyCLINK detects the similar *dynamic* behavior of programs (code relatives). This allows DyCLINK to detect code relatives that are dependent upon dynamic behavior, for example splitting across multiple methods.

Other previous works in dynamic code similarity detection focus on observing when code fragments produce the same outputs for the same inputs. Jiang and Su [60] drive programs with randomly generated input and then observe their output values, identifying clones as two methods that provide the same output for the same input. Li et al. detect functional similarity between code fragments using dynamic symbolic execution to generate inputs [85]. Similarly, the MeCC system [65] detects code similarity by observing two methods that result in the same abstract memory state. CCCD, or concolic clone detection [75], takes a similar approach, comparing the concolic outputs of methods to detect function-level input/output similarity. Elva and Leavens propose detecting functionally equivalent code by detecting methods with exactly the same outputs, inputs and side effects [39]. Juergens et al. propose *simions*, two methods that are found to yield similar outputs for the same input, but provide no automated technique for detecting such simions [61, 32]. We implement a simion detector for Java, HitoshiIO [123], which attempts to overcome the problem of different data structures through a fuzzy equivalence matching. HitoshiIO compares the input/output of functions while observing their executions in-vivo.

Code relatives differ from all of these dynamic code similarity detection systems in that similarity is compared between the computations performed, *not* between the resulting outputs. This important distinction allows for similarly behaving code to be detected even when different data structures and output formats are used. Moreover, it allows for arbitrary code fragments to be detected as code relatives: techniques that compare output equivalence tend to work best at a per-function granularity, because that format provides a clear definition of inputs and outputs.

In addition to work on fine-granularity clones, much work has been done in the general field of detecting similarly behaving software. Marcus and Maletic propose the notion of *high level concept clones*, detecting code that addressed the same general problem, but may have significant structural differences, by using information retrieval techniques on code [93]. Similarly, Bauer et al. mine the use of identifiers to detect similar code [16]. In addition to code, several approaches analyze software artifacts such as class diagrams and design documents. This type of analysis helps developers understand similarities/differences between software at system level [72, 25].

Software birthmarking uses some representative components of a program's execution to create an obfuscation-resilient fingerprint to identify theft and reuse [127, 117]. Code relatives are comparable to birthmarks in that both capture information about how a result is calculated. However, code relatives are computed using more information than lightweight birthmarks focusing on the use of APIs [98, 131, 12, 88].

3.4 Approach and Implementation of DyCLINK: Detecting Code Relatives With Link Analysis

The high-level procedure of DyCLINK is shown in Figure 3.2. To begin, the program(s) to be analyzed are instrumented to allow DyCLINK to trace their respective executions. Next, the program(s) are executed given some sample inputs or workloads representative of their typical use cases, and DyCLINK creates graphs to represent executions of each

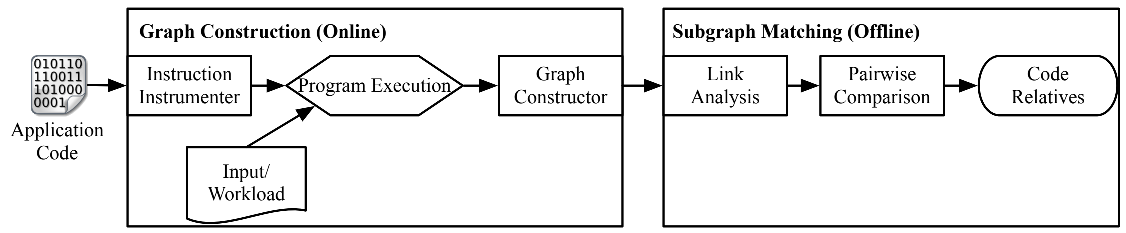


Figure 3.2: The high-level architecture of DyCLINK including instruction instrumentation, graph construction, link analysis and final pairwise subgraph comparison.

program, where each instruction is represented by a vertex, and each data and control dependency is represented by an edge. Then, DyCLINK analyzes these graphs (offline) to detect code relatives. DyCLINK traces program execution, so its results will be dependent upon the inputs given to the program: some methods may not be executed at all, while others may only be executed along some specific paths. One upside to this approach is that it exposes common behavior, allowing code that handles boundary input cases and hence may not be typically executed to be ignored for the purposes of code relative detection. However, it still requires that the inputs to the program are representative of actual and typical workloads. We will discuss this design decision further in Section 3.4.4.

DyCLINK consists of two major components: online graph construction and offline (sub)graph matching. The graph constructor instruments and observes the execution of the code being evaluated to generate these dynamic dependency graphs (Section 3.4.1), while the subgraph matcher analyzes the collected graphs to detect code relatives (Section 3.4.3). We calculate the similarity of the two dynamic dependency graphs by first link-analyzing their important instructions (centroids), linearizing them into vectors, and then calculating the Jaro-Winkler distance between them. This process will be described in detail in the following sections.

We have selected Java [62] as our target language, so the instructions recorded by DyCLINK are Java bytecodes. DyCLINK makes extensive use of the ASM bytecode instrumentation library [11], requiring no modifications to the JVM to find code relatives even without source code present. To implement the graph matcher for other target languages, we could similarly use runtime binary instrumentation to capture execution graphs, an approach examined by Demme et al. [34]. The subgraph matching mechanism, which occurs offline after program execution, is language agnostic.

3.4.1 Constructing Graphs

To construct dependency graphs, DyCLINK follows the JVM's stack machine to derive the dependencies between instructions, recording data and control dependencies. Each execution of each method results in the generation of a new dynamic instruction dependency graph G_{dig}, where each vertex represents an instruction and each edge represents an observed dependency. These graphs contain all instructions executed both by that method, *and* by the methods that method calls. Each edge in the graph is labeled with a weight, representing the frequency of its occurrence. We consider three types of dependencies for our graphs:

- $dep_{\texttt{inst}}$: A data dependency between an instruction and one of its operands.
- $dep_{\texttt{write}}$: A read-after-write dependency on a variable.
- $dep_{\texttt{control}}$: A weighted edge indicating that some instructions are controlled by another (e.g., through a jump), corresponding to the frequency that control will follow that edge based on the observed executions.

While it is possible to set a different weight for each type of dependency, we currently weight each equally.

When one method calls another, DyCLINK stores a pointer from the calling method to its callee, allowing for code relatives to be detected that span method boundaries. This way, when a target method is examined for code relatives, DyCLINK actually considers both the trace of that method and the traces of *all methods that it calls*.

DyCLINK uses two strategies to reduce the total number of graphs recorded. First, DyCLINK stores these graphs in a flattened form — when a method calls another many times (e.g., in a loop), DyCLINK identifies that redundancy by using the number of vertices and edges as a hash value, and simply updates execution counts for each edge in the graph. Second, DyCLINK imposes a configurable quota on the number of times (q_{call}) that a given method will be captured at a given call site, which will be discussed in Section 3.5.

3.4.2 Example of DyCLINK

To showcase how DyCLINK constructs a dependency graph, consider the `mult()` method in Figure 3.3. Figure 3.3a shows the Java source for this method that multiplies two numbers, while Figures 3.3c and 3.3b show the Java compiler's translation of this source code into bytecode. Consider tracing an execution of this code, using $\{a = 8, b = 1\}$ as input arguments. Figure 3.3d shows the graph that may be constructed from such an execution. The label of each numbered vertex is the index of a bytecode in Figure 3.3c, bytecodes in the `add` method (Figure 3.3b) are labeled as A2, A3 and A4; each edge is labeled with a counter indicating the number of times it occurred during the run. Every time that `mult()` is executed during profiling, a new G_{dig} will be generated.

To see how the edges are constructed, consider the `iload 2` instruction on line 7 (`iload x` loads a local variable `x` onto the JVM's stack). When this instruction is executed, the controlling instruction is `if_icmplt 7` at line 14, so the dependency $dep_{\texttt{control}}(14, 7)$ is constructed. Any additional dependencies are captured transitively in the graph. Because `iload 2` is reading the 2_{nd} local variable, DyCLINK detects the last instruction executed that wrote it, which is `istore 2` at line 3, creating the dependency $dep_{\texttt{write}}(3, 7)$. `invokestatic` on line 9 has two $dep_{\texttt{inst}}$ from `iload 2` and `iload 0`, because these instructions are used to invoke the `add` method. When `add` is called, its graph is stored separately, with pointers from the `mult` graph into it (vertices A2, A3 and A4). By including this callee graph (`add`) in its caller graph (`mult`), we can detect code relatives that span multiple methods.

Once the programs are executed with sample inputs, G_{dig}s are then constructed to represent each method execution. We can proceed to the next phase, subgraph matching.

```java
static int mult(int a, int b) {
  int ret = 0;
  for(int i = 0; i < b; i++) {
    ret = add(ret, a);
  }
  return ret;
}

static int add(int a, int b) {
  return a + b;
}
```

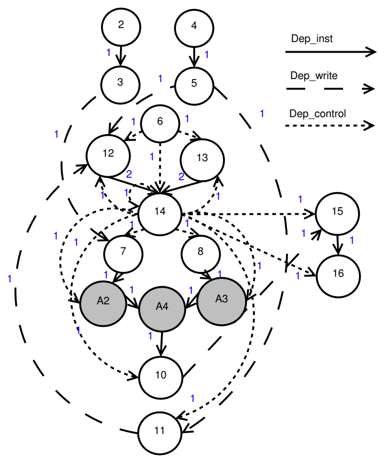
(a) The `mult()` method.

```
1 static add(II)I
2   iload 0
3   iload 1
4   iadd
5   ireturn
```

(b) The `add()` instructions.

```
 1 static mult(II)I
 2   iconst_0
 3   istore 2
 4   iconst_0
 5   istore 3
 6   goto 12
 7   iload 2
 8   iload 0
 9   invokestatic add
10   istore 2
11   iinc 3 1
12   iload 3
13   iload 1
14   if_icmplt 7
15   iload 2
16   ireturn
```

(c) The `mult()` instructions.



(d) The `mult` graph.

Figure 3.3: The `mult()` method in Java (a), translated into bytecode (b), and a dynamic instruction dependency graph (c) generated by running `mult(8,1)`.

3.4.3 LinkSub: Link-analysis-based Subgraph Isomorphism Algorithm

To detect code relatives, DyCLINK first enumerates every pair of G_{dig}s that were constructed: given n G_{dig}s, there are at most $n * (n-1) * sub$ pairs to compare, where sub represents the number of potential subgraphs. Note that because each execution of each method will generate a new G_{dig}, each method will have multiple graphs that represent its executions, meaning that there are more G_{dig}s than methods. Each recorded execution of each method is potentially compared to each of the executions of each other method.

The executions are represented as graphs, so we model code relative detection as a subgraph isomorphism problem. There are two types of subgraph isomorphism (or subgraph matching): exact and inexact [112]. For exact subgraph matching, a test graph needs to to be entirely the same as a (sub)graph of a target graph. Exact subgraph matching would only find cases where all instructions and their dependencies are exact copies between two code fragments; this would be too restrictive to detect code relatives. Because DyCLINK detects *similar* but *not necessarily identical* subgraphs, we are focused on techniques for inexact subgraph matching.

The key to efficiently performing this matching is to filter out pairs of graphs that can never match, reducing the number of comparisons needed to a much smaller set. For example, for each graph, we calculate its centroid, create a simpler representation of each subgraph (simply a sequence of instructions), and then identify candidate graphs to compare it to, filtered to only those that contain that same instruction. Next, we perform a constant-time comparison between each potentially matching subgraph, calculating the euclidean distance between their instruction distributions, to eliminate unlikely matches. For the remaining subgraphs, we apply a link analysis to each subgraph to create a vectorized representation of its instructions, ordered by PageRank. From these ordered vectors, we apply an edit-distance based model to calculate similarity. Hence, we reduce the running time in two ways: we consider only potential subgraph matches that seem *likely* based on some filters, and then we calculate the actual similarity of those subgraphs.

The overall algorithm is shown at a high level in Algorithm 1. The summary of each subroutine of LinkSub is as follows:

- **profileGraph:** Computes statistical information of a G_{dig}, such as ranking of each instruction and instruction distribution to identify its centroid.
- **sequence:** Sort instructions of G_{dig} by the feature defined by the developer to facilitate locating instruction segments. We use the execution order of each instruction to sort a G_{dig}.

- **locateCandidates:** Given the centroid of a G^{te}_{dig}, locate each instance of that centroid instruction in each potential target graph G^{ta}_{dig}.
- **euclidDist:** Compute the euclidean distance between the instruction distributions of two G_{digs}s.
- **LinkAnalysis:** Apply PageRank to a graph, returning a rank-ordered vector of instructions.
- **calcSimilarity:** Calculate the similarity of two PageRank ordered instruction vector using edit distance.

LinkSub models a dynamic instruction dependency graph of a method as a network, and uses link analysis [23], specifically PageRank [80], to rank each vertex in the network. The first phase of the algorithm (`profileGraph`) ranks each vertex in the graph being examined, calculating the highest ranked vertex (*centroid*) of the graph. This step also calculates instruction distribution for subgraph matching. The next phase lists all instructions of the target graph, G^{ta}_{dig}, by execution order in the `sequence` step to facilitate locating candidate subgraphs. In the next step, `locateCandidates`, we select all subgraphs in the target graph that match the centroid of G^{te}_{dig}. If a subgraph in G^{ta}_{dig} contains the centroid instruction of G^{te}_{dig} then it is potentially a code relative, but if it does not contain the centroid instruction, then it can't be. This is effectively the first filter that reduces the largest set of potential subgraphs to compare.

For each of the potential candidate subgraphs, we next apply a simple filter (`euclidDist`) similar to [89], which computes the Euclidean distance between the distributions of instructions in the graph of G^{te}_{dig} and a candidate subgraph from the G^{ta}_{dig}. If the distance is higher than the threshold, θ_{dist}, defined by the user, then this pair of subgraph matching is rejected. We empirically came to a threshold of 0.1 (the lower the better) to include only those subgraphs that were mostly similar.

If a candidate subgraph from the G^{ta}_{dig} passes the euclidean distance filter, DyCLINK applies its link analysis to this candidate. DyCLINK calculates a PageRank dynamic vector,

DV, for the candidate subgraph (`LinkAnalysis`), where the result is a sorted vector of all of the instructions (vertices from the subgraph), ordered by PageRank.

```
Data: The target graph $G^{ta}_{dig}$ and the test graph $G^{te}_{dig}$
Result: A list of subgraphs in $G^{ta}_{dig}$, $CodeRelatives$, which are similar to $G^{te}_{dig}$
//Compute Statistical Information;
$profile_{te}$ = profileGraph($G^{te}_{dig}$);
//Change Representation;
$seq_{ta}$ = sequence($G^{ta}_{dig}$);
//Filter to find possible matches;
$assigned_{ta}$ = locateCandidates($seq_{ta}$, $profile_{te}$);
$CodeRelatives = \emptyset$ ;
for $sub$ in $assigned_{ta}$ do
    //Perform multi-phase comparison;
    $SD$ = euclidDist(SV($sub$), $profile_{te}.SV$);
    if $SD > \theta_{dist}$ then
        continue ;
    end
    $DV^{sub}_{target}$ = LinkAnalysis($sub$);
    $dynSim$ = calcSimilarity($DV^{sub}_{target}$, $profile_{te}.DV$);
    if $dynSim > \theta_{dyn}$ then
        $CodeRelatives \cup sub$ ;
    end
end
return $CodeRelatives$;
```

Algorithm 1: LinkSub.

Finally, in `calcSimilarity`, we use the Jaro-Winkler Distance [27] to measure the similarity of two DVs, which represents the similarity between two G_{dig}s. Jaro-Winkler has better tolerance of element swapping in the instruction vector than conventional edit distance and is configurable to boost similarity if the first few elements in the vectors are the same. These two features are beneficial for DyCLINK, because the length of $DV(G_{dig})$ is usually long, and thus may involve frequent instruction swapping. For representing the behavior of methods, we use the PageRank-sorted instructions from $DV(G_{dig})$. If the similarity between the PageRank vectors from the subgraph of the G^{ta}_{dig} and the G^{te}_{dig} is higher than the dynamic threshold (θ_{dyn}), DyCLINK identifies this subgraph as being inexactly isomorphic to G^{te}_{dig}. We empirically evaluated several values of this threshold, settling on 0.82 as a default in Section 3.5. We refer to the subgraph similar to the G^{te}_{dig} as a *Code Relative* in the G^{ta}_{dig}.

The runtime execution cost of our algorithm will vary greatly with the number of subgraphs that remain after the two filtering stages. While each filtering stage itself is relatively cheap (the PageRank computation requires only $O(V+E)$ for a graph with V nodes and E edges), in the worst case, where we would need to calculate the Jaro-Winkler similarity of every possible pair of (sub)graphs, the overall running time would be dominated by these computations for (sub)graphs. In practice, however, we have found that these two filtering phases tend to dramatically reduce the overall number of comparisons needed, making the running time of *LinkSub* quite reasonable, requiring only 43 minutes on a commodity server to detect candidate code relatives in a codebase with over $7,000$ lines of code. Profiling this code base resulted in $1,244$ G_{dig}s, requiring a total worst-case $422+$ millions of subgraph comparisons. A complete evaluation and discussion of the scalability of our algorithm and system are in Section3.5.1.

3.4.4 Limitations

There are several key limitations inherent to our approach that may result in incorrect detection of code relatives. The main limitation stems from the fact that DyCLINK captures dynamic traces: the observed inputs must exercise sufficiently diverse input cases that are representative of true application behavior. A second limitation comes from our design decision to declare that two code fragments are relatives if there is at least a single input pair that demonstrates the two fragments to be similar. An ideal approach would require profiling the application over large workloads representative of typical usage. If we could guarantee that the inputs observed were truly sufficiently diverse to represent typical application behavior, then it may be reasonable to consider the relative portion of inputs that result in a match compared to those that do not. However, with no guarantee that the inputs that DyCLINK observes are truly representative of the same input distribution observed in practice in a given environment, we decide for now to instead ignore counter examples to two fragments being relatives, declaring them relatives if at least one pair of inputs provide similar behavior.

Consider the following example of a situation where these choices may result in undesirable behavior. The first method will sort an array if the passed array is non-null, returning -1 if the parameter is null. The second method will read a file if the passed file is non-null, returning -1 if the parameter is null. If DyCLINK observes executions of each method with a null parameter, then these two methods will be deemed code relatives, because there is at least one input pair that causes them to exhibit similar behavior. A future version of DyCLINK could instead consider all of the inputs received, and the coverage of each of those inputs towards being representative of overall behavior.

DyCLINK can also fail to detect code that is similar in terms of its input and output if it has different instruction-level behavior. For example, a method can multiply two integers, $\{a, b\}$, in a convoluted way as Figure 3.3 depicts, or it can simply return $a * b$. By our definitions, these are not code relatives, and wouldn't be detected by DyCLINK.

Due to nondeterminism in a running program, DyCLINK may record different execution graphs, causing results to vary slightly between multiple profiling runs. In multi-threaded applications, DyCLINK currently only considers code fragments that execute within the same thread as code relatives - there is no merging of graphs across threads.

3.5 Evaluations of DyCLINK

We evaluate DyCLINK in terms of both runtime performance and precision. We answer two research questions:

- **RQ1:** Given the potentially immense number of subgraph comparisons, is DyCLINK's performance feasible to scale to large applications?

- **RQ2:** Are the code relatives detected by DyCLINK more precise for classifying programs than are the similar code fragments found by previous techniques?

Table 3.1: A summary of the code subjects from the Google Code Jam for classifying software.

		# Proj			**Graph Size**			
Year	**Problem**	**Tot.**	**Aut.**	**Meth.**	#	V_{avg}	V_{max}	E_{avg}
2011	Irregular Cake	48	30	106/154	367	398	6698	958.1
2012	Perfect Game	48	34	122/182	195	138.2	2001	276.6
2013	Cheaters	29	21	95/147	374	283.4	2456	631.7
2014	Magical Tour	46	33	105/159	308	223.6	3709	480.5

Unfortunately, we are limited in our choice of experimental subjects and comparison approaches by what is publicly available. For example, while there are publicly available benchmarks of code clones [124] with a ground truth manually provided, we found many of them did not include sufficient dependencies and build scripts to be compiled and executed dynamically. To focus our evaluation on projects that were build-able and distributed with inputs/test cases, we selected projects from the Google Code Jam repository [51]. 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. Because each submission for the same problem attempts to perform the same task, we assume that each project within the same year will likely share code relatives, while projects between different years solving different requirements will likely not share code relatives or at least fewer.

To compare DyCLINK's code relative detection with static code clone detection, we selected the state-of-the-art clone detector available, SourcererCC[115]. While SourcererCCis highly performant, scaling impressively to "big code", we admittedly do not expect to find many near-miss static code clones in independently written Code Jam entries. In contrast, we would expect to find clusters of dynamic functional I/O simions, since the independently written entries intend to complete the same tasks. Previous simion detec-

Table 3.2: Number of comparisons performed by DyCLINK on the Google Code Jam projects, showing worst case number of comparisons (without any filtering) and actual comparisons performed along with the relative reduction in comparisons achieved by DyCLINK. We also show the total analysis time needed to complete each set of comparisons.

Years	**Subgraphs Compared**			**Analysis Time (sec)**		
Comp.	**Worst Case**	**Actual**	**Reduction**	**DyCLINK**	**HitoshiIO**	**SourcererCC**
2011-2011	49,999,944	258,478	99.48%	836.38	64.00	4.1
2012-2012	5,006,827	7,719	99.85%	14.88	49.00	4.4
2013-2103	35,186,281	280,355	99.2%	392.73	51.00	3.9
2014-2014	19,017,387	123,196	99.35%	230.39	53.00	4.3
2011-2012	38,371,375	12,221	99.97%	49.77	133.00	4.9
2011-2013	93,519,230	45,822	99.95%	193.55	125.00	5.0
2011-2014	70,260,597	10,396	99.99%	70.98	133.00	4.9
2012-2013	30,745,400	32,621	99.89%	68.15	96.00	5.1
2012-2014	21,730,445	31,151	99.86%	63.96	114.00	5.0
2013-2014	58,399,594	460,750	99.21%	653.44	105.00	4.7
Total	422,237,080	1,262,709	99.7%	2574.23	923.00	46.3

tors for object-oriented languages do not address project-specific object data types, due to the technical challenges reported by Deissenboeck et al. [32]. Therefore, we developed a simion detector that we have recently built for Java, HitoshiIO [123], specifically designed to overcome these challenges and enable fair comparison of the similarity models.

The information on the evaluation subjects is shown in Table 3.1. For each competition year, we show the problem name, the number of projects in the repository, the number of automatic projects without human interactions used in this study, the total number of executed methods in those projects and the statistics for the captured G_{dig}s including the number of graphs and the numbers of vertices and edges. For the executed methods, we provide two numbers: $retained/all$. To avoid potentially inflating our results by including matches of trivial methods, we filter out simple methods with little work in them (such as toString and initialization methods). all represents the number of all executed methods, while $retained$ shows the method number after such filtering.

We discuss some parameter settings of DyCLINK for conducting the experiments in this thesis. For constructing G_{dig}s in Section 3.4.1, we empirically set the quota at a given call site, q_{call} as 5. This allows for reasonable performance both in terms of code relative detection and runtime overhead. For conducting the inexact (sub)graph matching, we set θ_{dist} as 0.1 and θ_{dyn} as 0.82 in Algorithm 1, where both parameters range from 0 to 1. The details of each parameter setting can be found in the GitHub page of DyCLINK [37]. While searching for the best parameter setting for DyCLINK is out of the scope of this thesis, we plan to utilize machine learning techniques for optimizing DyCLINK in future.

3.5.1 RQ1: Scalability

To evaluate the scalability of DyCLINK, we measured its performance when running on these 118 projects. The key to DyCLINK's performance is the relative reduction in subgraph comparisons that result from filtering and link analysis steps. If we can greatly reduce the number of candidate subgraphs to be compared, then DyCLINK will scale, even on large graphs. Table 3.2 shows the worst case number of pairwise comparisons that would be needed by a naive subgraph matching algorithm, along with the number of comparisons that were actually necessary to detect the code relatives. We also show the analysis time for each of DyCLINK, HitoshiIO, and SourcererCC.

DyCLINK filtered out over 99% of the potential subgraphs to compare, resulting in a total analysis time of just 43 minutes on an Amazon EC2 "c4.8xlarge" instance. While this analysis time is significantly longer than the static approach, and still more than the simion detector, we believe that the analysis runtime is acceptable given the time complexity to solve the inexact (sub)graph matching problem.

Because DyCLINK is a dynamic profiling approach, there is also a time overhead for collecting the traces and generating the graphs. Our execution tracer implementation is unoptimized and records every single instruction. An optimized version might instead be able to infer and record instructions that expose program behaviors. To trace these applications took a total time of just over 2.5 hours compared to a baseline execution time without instrumentation of approximately 1 minute on an iMac with 8 cores and

Table 3.3: Code Relatives, Simions and Code Clones detected by project-year and by tool for DyCLINK, HitoshiIO and SourcererCC.

Years Compared	**DyCLINK**	**HitoshiIO**	**SourcererCC**
2011-2011	103	21	5
2012-2012	49	59	13
2013-2103	116	181	6
2014-2014	66	43	4
2011-2012	3	19	9
2011-2013	0	9	9
2011-2014	0	19	6
2012-2013	7	6	15
2012-2014	3	25	8
2013-2014	81	24	16
Total	428	406	91

32 GB memory; however the instrumentation overhead can vary significantly with the complexity of the program — one single subject took 114 minutes to execute, while the remaining 117 required only a total of 43 minutes to execute. We are confident that the tracing overhead can be significantly reduced with some optimizations as demonstrated by other Java tracing systems, such as JavaSlicer [54].

3.5.2 RQ2: Code Relative Detection (Best Graph Match)

We first evaluate the quality of the code relatives detected by DyCLINK by looking at the number of code relatives detected in projects across and within each year. For this evaluation, we ran each tool with its default similarity threshold (0.82 for DyCLINK, 0.85 for HitoshiIO and 0.7 for SourcererCC), and a minimum code fragment size of 10 lines of code (45 instructions for DyCLINK). Table 3.3 shows the number of code relatives detected by DyCLINK as well as the number of code clones detected by the other two systems. DyCLINK detected more similar code fragments on average than the other systems did. Those relatives were skewed to be almost entirely among projects within the same year, while the other tools tended to find similar code fragments more evenly distributed among and within the project years (recall that all projects in the same year performed the same task). This result is encouraging, as we expect that there are more code relatives in code that has the same general purpose than in code that is doing different tasks.

Figure 3.4 shows an exemplary pair of similar code fragments detected by DyCLINK in Code Jam projects. The two caller methods, `calcMaxBet` and `maxBet`, exhibit similar functionality to maximize bets, so both of them are detected by DyCLINK and HitoshiIO. However, even though their subroutines, `canDo` and `cost`, have similar behavior to evaluate costs, HitoshiIO cannot detect them as functionally similar by observing their I/Os. The reason is that their output values will be hard to detect as similar: while `canDo` performs a comparison between the cost and budget and returns a boolean, `cost` solely computes the cost and leaves the comparison for its caller `maxBet`. This example shows the difficulty to detect dynamic code similarities by observing functional I/Os of programs.

We did not conduct a user study as part of this experiment other than random sampling performed by the authors to ensure the relatives reported were valid. To judge the system accuracy, we investigated specifically its precision in a software clustering experiment.

Software Community Clustering. To judge the efficacy of DyCLINK in performing software clustering, we applied a KNN-based classification algorithm to the Google Code jam projects. Again, our ground truth is that projects from the same year solving the same problem ought to be in the same cluster.

We apply the *K-Nearest Neighbors (KNN)* classification algorithm to predict the label (project year) for each method and then validate the prediction result by *Leave-One-Out* methodology: each sample (method) plays as a testing subject exactly once, where all the rest of the samples play as the training data. The high-level algorithm is shown in Algorithm 2: for each method, we search for the K other methods that have the greatest similarity to the current one in the `search`KNN step. Each nearest neighbor method can vote for the current method by its real label in the `vote` step. The label voted by the greatest number of neighbor methods becomes the predicted label of the current method.

```java
static long calcMaxBet(long budget,
  long[] x,int winningThings) {
  ...
  if (canDo(budget, x, winningThings, mid)) {
    low = mid;
  } else { high = mid; }
  ...
}

static boolean canDo(long budget,
  long[] x,int winningThings,long lowestBet) {
  long payMoney = 0;
  for (int i = 0; i < x.length; i++) {
    if (x[i] < lowestBet) {
      payMoney += -x[i] + lowestBet;
    }
  }
  return payMoney <= budget;
}
```

(a) The call sequence includes the `canDo` method

```java
long maxBet(long[] a,int count,long b) {
  ...
  if (cost(a, count, mid) <= b) {
    left = mid;
  } else { right = mid; }
  ...
}

long cost(long[] a,int count,long bet) {
  long result = 0;
  for (int i = 0; i < count; i++) {
    result += (bet - a[i]);
  }
  for (int i = count; i < a.length; i++) {
    if (a[i] <= bet) {
      result += (bet + 1 - a[i]);
    }
  }
  return result;
}
```

(b) The call sequence includes the `cost` method

Figure 3.4: An exemplary code relative.

In the event of a tie, we side with the neighbors with the highest sum of similarity scores. Then, we track the precision of the approach as the total number of correctly labeled methods divided by the total number of methods.

```
Data: The similarity computation algorithm $SimAlg$, the set of subject methods to
      be classified $Methods$ and the number of the neighbors $K$
Result: The precision of $SimAlg$
realLabel($Methods$);
$matrix_{sim}$ = computeSim($SimAlg$, $Methods$);
$succ = 0$;
for $m$ in $Methods$ do
    $neighbors$ = searchKNN($m$, $matrix_{sim}$, $K$);
    $m.predictedLabel$ = vote($neighbors$);
    if $m.predictedLabel = m.realLabel$ then
        $succ = succ + 1$;
    end
end
$precision = succ/Methods.size$;
return $precision$;
```
Algorithm 2: Procedure of the KNN-based software label classification algorithm.

For observing the efficacy of the systems under single and multiple neighbors, we set $K = 1$ and $K = 5$. We also vary the line of code thresholds used for each code fragment's minimum size between $\{10, 15, 20, 30\}$. Only programs that pass the threshold setting including LOC and similarity were considered as neighbors of the current program.

The results of this analysis are shown in Table 3.4: DyCLINK showed the highest precision among all three techniques when examining code fragments that consisted of at least ten lines of code. When excluding the smallest fragments (for example, looking only at those with 20 lines of code or more), the simion detector HitoshiIO performed slightly better. The methods being incorrectly categorized by HitoshiIO were mostly less than 20 lines of code. SourcererCCdid not find sufficient clones that were longer than 30 lines of code to allow for clustering at that level, and hence, the precision value is not available. Because we use the project year as the label for each method, it is possible that some syntactically similar code detected by SourcererCCis not identified as a true positive case.

Figure 3.5 shows the clustering matrix based on DyCLINK's KNN-based classification result with K = 1, LOC = 10. Each element on both axes of the matrix represents a project

Table 3.4: Precision results from KNN classification of the Google Code Jam projects using DyCLINK, HitoshiIO and SourcererCC, while varying K and the minimum fragment length considered.

Min.	**K=1**			**K=5**		
Fragm.	**DyCLINK**	**HitoshiIO**	**SourcererCC**	**DyCLINK**	**HitoshiIO**	**SourcererCC**
10	0.94	0.81	0.35	0.91	0.77	0.34
15	0.94	0.86	0.48	0.92	0.86	0.45
20	0.87	0.95	0.55	0.90	0.95	0.45
30	0.92	0.91	N/A	0.91	0.91	N/A

indexed by the abbreviation of the problem set to which it belongs and the project ID. 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. These results show that DyCLINK is capable of detecting programs with similar behavior and then cluster them for further usage such as code search.

3.5.3 Discussions

Through this evaluation, we have shown that DyCLINK is an effective tool for detecting similar code fragments. There are several potential limitations to our experiments, however. Even though we may have manually come to the conclusion that two code fragments are code relatives and assuming that we are internally valid in that conclusion, two developers' definitions of "similarly behaving" code may differ. We believe that we have limited the potential for this bias through our study design: we purposely selected a suite of projects that are known to be likely to contain similarly behaving code, because they were performing the same overall task. Hence, when we conclude that DyCLINK is effective at finding behaviorally similar code, we come to this conclusion both from our

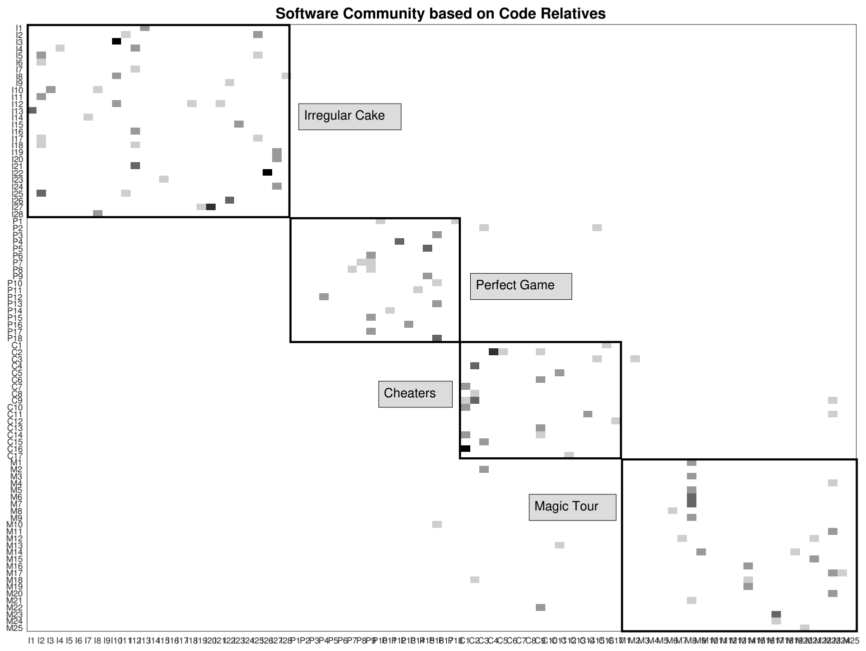


Figure 3.5: The software community based on code relatives detected by DyCLINK. The darker color in a cell represents a higher number of code relatives shared by two projects.

internal review and also from the external construction, that by definition, the code ought to behave similarly (at least on some scale).

However, this selectivity comes at a cost: the projects that we selected might be too homogeneous overall, and not sufficiently representative of software in general. We could bolster our claims by performing a broader study on, for instance, large open-source projects from GitHub. We could construct a user study to help establish a ground truth for what "similar code" really is.

Dynamic analysis and static analysis have their own opportunities and obstacles in detecting different types of similar code. Thus, we plan to distill and integrate the advantages of DyCLINK and SourcererCCto devise a new approach for detecting similar code fragments more effectively with better efficiency.

3.6 Conclusions

Determining when two code fragments are “similar” is a subjective and complex problem. We have distilled the problem of detecting behaviorally similar code fragments into a subgraph isomorphism problem based on dynamic dependency graphs that capture instruction-level behavior. To feasibly analyze the hundreds of millions of potential matching subgraphs, we have devised a novel link-analysis based algorithm, *LinkSub*, which greatly reduces the number of pairwise comparisons needed to detect code relatives, and then efficiently compares them using PageRank. DyCLINK detects behaviorally similar code better than previous approaches, and has reasonable runtime overhead. We have released DyCLINK under an MIT license on GitHub [37]. A tutorial regarding how to use DyCLINK can be found in Section 3.7.

One key limitation of our approach is from its dynamic nature: because it relies on program execution traces to detect code relatives, it is only applicable to situations where the subject code can be executed. In addition to being executable at all, there must be valid inputs that are representative of a program’s normal behavior to expose its typical use cases and generate representative traces. In our previous work [123], we applied applications’ existing test suites for this purpose, but recognize that test suites may not be truly representative of application usage. Alternatively, automated input generation tools [47, 107] could be used to drive the application. We plan to experiment with input generation techniques, allowing us to apply DyCLINK to larger scale systems than studied in this thesis. Furthermore, we plan to construct a benchmark suitable for use for dynamic code similarity detection. This benchmark would contain not only workloads and scripts to compile and run each application, but also a human-judged ground-truth of program similarity, analogous to the static code clone benchmark, BigCloneBench [124].

We also plan to study additional applications of our link-analysis based graph comparison algorithm. For example, we plan to explore the possibility to apply DyCLINK to support software development tasks related to behavior, such as (semi)automatic API generation and code search.

3.7 Artifact Description

We provide a tutorial to replay the result of Table 3.3. A virtual machine (VM) containing DyCLINK and all required software can be accessed from DyCLINK's Github page [37]. Users can first read Section 3.7.7 to check the VM's limitation. We conducted our experiments on an iMac with 8 cores and 32 GB memory to construct graphs (Section 3.7.4) and Amazon ec2 "c4.8xlarge" instances to match graphs (Section 3.7.5).

3.7.1 Required Software Suites

If the user chooses to use our VM, this step can be skipped. The user needs to install JDK 7 [58] to execute our experiments on DyCLINK. DyCLINK is a Maven project [97]. If the user wants to re-compile DyCLINK, the installation of Maven is required. DyCLINK needs a database system and GUI to store/query the detected code relatives. We use MySQL and MySQL Workbench. For downloading and installing them, the user can check MySQL's website [104]. For setting up the database, the user can find more details in `dycl_home/scripts/db_setup`, where `dycl_home` represents the home directory of DyCLINK.

3.7.2 Virtual Machine

We set up the credential with "dyclink" as the username and "Qwerty123" as the password for our VM. The home of DyCLINK is `/home/dyclink/dyclink_fse/dyclink`. For starting MySQL, the user can use the command `sudo service mysql start`. The credential for MySQL is "root" as the username and "qwerty" as the password.

3.7.3 System Configuration

Before using DyCLINK, the user needs to change to the home directory of DyCLINK. The user first uses the command `./scripts/dyclink_setup.sh` to create all required directories for executing DyCLINK. DyCLINK has multiple parameters to specify in the configuration file: `config/mib_config.json`. For reproducing the experimental results, the user can simply use the this configuration file.

3.7.4 Dynamic Instruction Graph Construction

We put our codebases for the experiments under `codebase/bin`. The user will find 4 directories from "R5P1Y11" to "R5P1Y14". These 4 directories contain all Google Code Jam projects we used in the thesis from 2011 to 2014.

Before executing the projects in a single year, the user needs to specify the graph directory for the `graphDir` field in the configuration file. This is to tell DyCLINK where to dump all graphs. For example, the user sets `graphDir` to `graphs/2011` for storing graphs of the projects in 2011. We have created subdirectories for each year under `graphs`.

We prepare a script to automatically execute all projects in a single year: `./scripts/exp_const.sh $yearDir`. For example, the user can execute all projects in 2011 by the command `./scripts/exp_const.sh R5P1Y11`. Most years can be completed between 0.5 to 3 hours on the VM, but 2013 may cost 20+ hours and need more memory.

The `cache` directory records cumulative information for constructing graphs. If users fail any year, they need to first clean the `cache` directory and reset `threadMethodIdxRecord` in the configuration file to be empty, and re-run every year.

3.7.5 (Sub)graph Similarity Computation

Because we compute the similarity between each graph within and between years, there will be totally 10 comparisons. For storing the detected code relatives in the database, the user needs to specify the URL and the username in the configuration file.

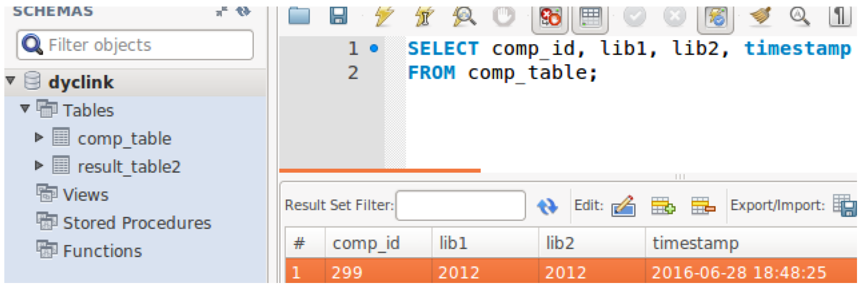
For computing similarities between graphs in the same year, the user can issue `./scripts/dyclink_sim.sh -iginit -target graphs/$year`, where `$year` is between $\{2011, 2014\}$. For different years, the command is `./scripts/dyclink_sim.sh -iginit -target graphs/$y1 -test graphs/$y2`, where
`$y1` and `$y2` are between $\{2011, 2014\}$. DyCLINK will then prompt for user's decision to store the results in the database The user needs to answer "true".

On the VM, we suggest the user to detect code relatives for $2011-2012$, $2011-2014$, $2012-2012$ and $2012-2014$, if we exclude the projects in 2013. The other 6 comparisons may take $20+$ hours to complete on the VM.

3.7.6 Result Analysis

For analyzing code relatives for a comparison, the user needs to retrieve the comparison ID from the `dyclink` database. The user first queries all comparisons by the SQL command as Figure 3.6 shows via MySQL Workbench, and then checks the ID for the comparison. `lib1` and `lib2` show the years (codebases) in a comparison. If the values for `lib1` and `lib2` are different such as $2011-2012$, this comparison contains the code relatives *between* different years. If the values are the same such as $2012-2012$, this comparison is *within* the same year. Figure 3.6 checks the comparison ID (299) for code relatives within 2012 ($2012-2012$).

For computing the number of code relatives, the user can use the command `./scripts/dyclink_query.sh $compId $insts $sim -f` with 4 parameters. The `$compId` represents the comparison ID. The `$insts` represents the



```
SELECT comp_id, lib1, lib2, timestamp
FROM comp_table;
```

#	comp_id	lib1	lib2	timestamp
1	299	2012	2012	2016-06-28 18:48:25

Figure 3.6: The exemplary UI of MySQL Workbench to check the comparison ID.

minimum size of code relatives with 45 as the default value. The `$sim` represents the similarity threshold with 0.82 as the default value. The flag `-f` filters out simple utility methods in our codebases. An exemplary command for the $2012 - 2012$ comparison with `$compId` $= 299$ is `./scripts/dyclink_query.sh 299 45 0.82 -f`.

3.7.7 Potential Problems

The major potential problem is the performance and memory of VM. Some experiments regarding 2013 may cost too much time and need more memory than the VM has. If the `OutOfMemoryError` occurs, the user can increase the memory for the VM and sets `-Xmx` for JVM in the corresponding commands under the `scripts` directory. For completing *all* experiments in our thesis, we suggest to run DyCLINK on a real machine.

Chapter 4

Topics in Program Binary as Behavioral Feature

Android applications are nearly always obfuscated before release, making it difficult to analyze them for malware presence or intellectual property violations. Obfuscators might hide the true intent of code by renaming variables, modifying the control flow of methods, or inserting additional code. Prior approaches toward automated deobfuscation of Android applications have relied on certain structural parts of apps remaining as landmarks, un-touched by obfuscation. For instance, some prior approaches have assumed that the structural relationships between identifiers (e.g. that A represents a class, and B represents a field declared directly in A) are not broken by obfuscators; others have assumed that control flow graphs maintain their structure (e.g. that no new basic blocks are added). Both approaches can be easily defeated by a motivated obfuscator. We present a new approach to deobfuscating Android apps that leverages deep learning and topic modeling on machine code, Macneto. Macneto makes few assumptions about the kinds of modifications that an obfuscator might perform, and we show that it has high precision when applied to two different state-of-the-art obfuscators: ProGuard and Allatori.

4.1 Introduction of MAChiNE TOpic: Macneto

Android apps are typically obfuscated before delivery, in an effort to decrease the size of distributed binaries and reduce disallowed reuse. In some cases, malware authors take advantage of the general expectation that Android code is obfuscated to pass off obfuscated malware as regular code: obfuscation will hide the actual purpose of the malicious code, and the fact that there is obfuscation will not be surprising, as it is already a general practice. Hence, there is great interest in automated *deobfuscators*: tools that can automatically find the original structure of code that has been obfuscated.

Deobfuscators can be used as a part of various automated analyses, for instance, plagiarism detection or detecting precise versions of third party libraries that are embedded in apps, allowing auditors to quickly identify the use of vulnerable libraries. Similarly, deobfuscators can be used to perform code search tasks among obfuscated apps using recovered identifiers and simplified control flow. Deobfuscators can also be used as part of a human-guided analysis, where an engineer inspects applications to determine security risks.

In general, deobfuscators rely on some training set of non-obfuscated code to build a model to apply to obfuscated code. Once trained, deobfuscators can recover the original names of methods and variables, or even the original structure and code of methods that have been obfuscated. For example, some deobfuscation tools rely on the structure of an app's control flow graph. However, they are susceptible to obfuscators that introduce extra basic blocks and jumps to the app's code and can be slow to use, requiring many pairwise comparisons to perform their task [121, 41]. Using another approach, DeGuard [20] is a state-of-the-art deobfuscator that builds a probabilistic model for identifiers based on the co-occurrence of names (e.g., knowing that some identifier a is a field of class b which is used by method c). While this technique can be very fast to apply (after the statistical model is trained), this approach is defeated by obfuscators that change the layout of code (e.g. move methods to new classes or introduce new fields).

We present a novel approach for automated deobfuscation of Android apps: MACNETO, which applies recurrent neural networks and deep learning to the task. MACNETO leverages a key observation about obfuscation: an obfuscator's goal is to transform how a program looks as radically as possible, while still maintaining the original program semantics. MACNETO deobfuscates code by learning the deep semantics of what code does through topic modeling. These topic models are a proxy for program behaviors that are stable despite changes to the layout of code, the structure of its control flow graph, or any metadata about the app (features used by other deobfuscators). These topic models are trained using a relatively simple feature set: a language consisting of roughly 20 terms that represent the different low-level bytecode instructions in Android apps and roughly 200 terms that represent the various Android APIs. MACNETO's topic model is resilient to many forms of obfuscation, including identifier renaming (as employed by ProGuard [109]), method call injection, method splitting, and other control flow modifications (as employed by Allatori [3]).

MACNETO uses deep learning to train a topic classifier on known obfuscated and unobfuscated apps offline. This training process allows MACNETO to be applicable to various obfuscators: supporting a new obfuscator would only require a new data set of obfuscated and deobfuscated apps. Then, these models are saved for fast, online deobfuscation where obfuscated code is classified according to these topics, and matched to its original code (which MACNETO hadn't been trained to recognize). This search-oriented model allows MACNETO to precisely match obfuscated code to its deobfuscated counterpart. This model is very applicable to many malware-related deobfuscation tasks, where a security researcher has various malware samples and is trying to identify if those samples had been hidden in an app. Similarly, it is immediately applicable to plagiarism-related deobfuscation tasks, were an analyst has the deobfuscated version of their code and is searching for obfuscated versions of it.

We evaluated MACNETO by building several deobfuscation models based on over $1,500$ real android apps using two popular, advanced obfuscators: ProGuard [109] and Allatori [3]. Compared to a state-of-the-art statistical approach [20], MACNETO had significantly higher precision at recovering method names obfuscated by ProGuard 96% vs 67%.

On Allatori (which employs significantly more complex obfuscations), MACNETO maintained a good precision (91%), while the state-of-the-art approach could not be applied at all. Moreover, we found that MACNETO performs well even with a relatively small training set. Based on these findings, we believe that MACNETO can be very successful at deobfuscating method names.

The contributions of this thesis are:

- A new approach to deobfuscation leveraging deep learning and machine topics.
- A new approach to automatic classification of programs with similar semantics.
- An evaluation of our tool on two popular obfuscators.

4.2 Background

In general, obfuscators make transformations to code that result in an equivalent execution, despite structural or lexical changes to the code — generating code that looks different, but behaves similarly. Depending on the intended purpose (e.g. hiding a company's intellectual property, disguising malware, or minimizing code size), a developer may choose to use a different kind of obfuscator. At a high level, these obfuscations might include lexical transformations, control transformations, and data transformations [28]. Obfuscators might choose to apply a single sort of transformation, or several.

Lexical transformations are typically employed by "minimizing" obfuscators (those that aim to reduce the total size of code for distribution). Lexical transformations replace identifiers (such as method, class or variable names) with new identifiers. Since obfuscators are applied only to individual apps, they must leave identifiers exposed via public APIs unchanged. Similarly, if some obfuscated class C_1 extends some non-obfuscated class C_2 and overrides method m, then the obfuscator can't change the name of m without breaking inheritance structures.

Control transformations can be significantly more complex, perhaps inlining code from several methods into one, splitting methods into several, reordering statements, adding jumps and other instructions [90, 110]. Control transformations typically leverage the limitations of static analysis: an obfuscator might add additional code to a method, with a jump to cause the new code to be ignored at runtime. However, that jump might be based on some complex heap-stored state which is tricky for a static analysis tool to reason about.

Finally, data transformations might involve encoding data in an app or changing the kind of structure that it's stored in. For instance, an obfuscator might encrypt strings in apps so that they can't be trivially matched, or change data structures (e.g. in Java from an `array` to an `ArrayList`) [90].

In this thesis we define the deobfuscation problem as follows. A developer/security analyst has access to a set of original methods and their corresponding obfuscated versions, and her job is to identify the corresponding original version given obfuscated program. Then the developer/analyst can analyze the original program to identify malware variants which becomes a significantly easier problem. Thus, in our case, deobfuscation essentially becomes a search problem, similar to DeGuard [20].

We assume that obfuscators can make lexical, control, and data transformations to code. We do not base our deobfuscation model on any lexical features, nor do we base it on the control flow structure of or string/numerical constants in the code. When inserting additional instructions and methods, we assume that obfuscators have a limited vocabulary of no-op code segments to insert. That is, we assume that there is some pattern (which need not be pre-defined) that our deep learning approach can detect. MACNETO relies on a training phase that teaches it the rules that the obfuscator follows: if the obfuscator is truly random (with no pattern to the sort of transformations that it makes), then MACNETO would be unable to apply its trained deobfuscation model to other obfuscated apps. However, we imagine that this is a reasonable model: an adversary would have to spend an incredible amount of resources to construct a truly random obfuscator.

Since it relies on static analysis, MACNETO could also be defeated by an obfuscator that inserts many reflective method calls (which are dynamically resolved at runtime). This obfuscator could effectively mask all of the 250 features that Macneto uses to classify methods to topic vectors. In that case, Macneto would be relying only on the bytecode instructions. Macneto could be adapted to better analyze reflection through existing techniques [84].

4.3 Related Work

Although in a programming language identifier names can be arbitrary, real developers usually use meaningful names for program comprehension [87]. Allamanis et al.[1] reported that code reviewers often suggest to modify identifier names before accepting a code patch. Thus, in recent years, naming convention of program identifiers drew significant attention for improving program understanding and maintenance [1, 24, 81, 126, 8, 96]. Among the identifiers, a good method name is particularly helpful because they often summarize the underlying functionalities of the corresponding methods [56, 2]. Using a rule-based technique, Host et al. [56] inferred method names for Java programs using the methods' arguments, control-flow, and return types. In contrast, Allamanis et al. used a neural network model for predicting method names of Java code [2]. Although these two studies can suggest better method names in case of naming bugs, they do not look at the obfuscated methods that can even change the structure of the program.

JSNice [111] and DeGuard [20] apply statistical models to suggest names and identifiers in JavaScript and Java code, respectively. These statistical models work well against so called "minimizers" — obfuscators that replace identifier names with shorter names, without making any other transformations. These approaches can not be applied to obfuscators that modify program structure or control flow.

While Macneto uses topic models as a proxy for application and method behavior, a variety of other systems use input/output behavior [60, 123, 61], call graph similarity

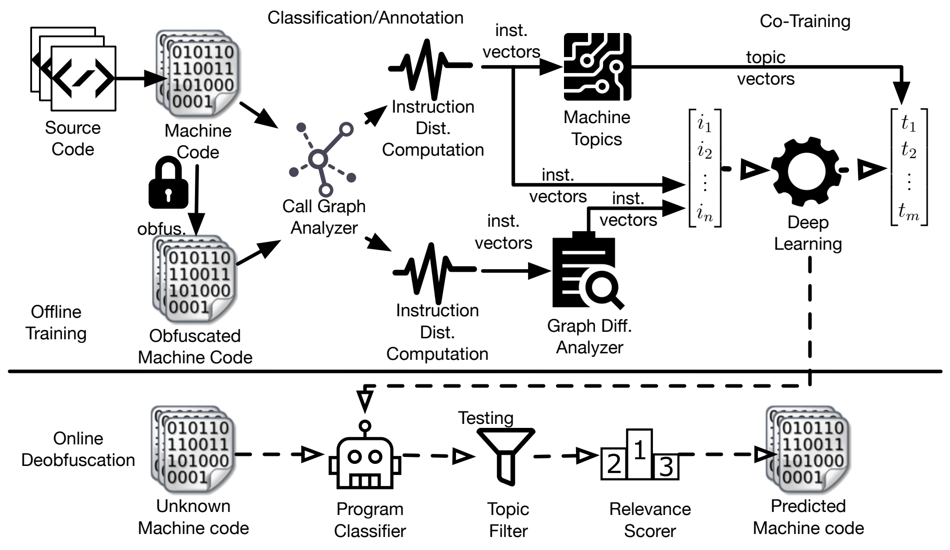


Figure 4.1: The system architecture of Macneto, which consists of four stages: instruction distribution, classification, deep-learning on machine topics and online scoring, to deobfuscate Android apps.

[121, 41], or dynamic symbolic execution [100, 85, 75]. Macneto is perhaps most similar to systems that rely on software birthmarks, which use some representative components of a program's execution (often calls to certain APIs) to create an obfuscation-resilient fingerprint to identify theft and reuse [127, 117, 98, 131, 12, 88]. One concern in birthmarking is determining which APIs should be used to create the birthmark: perhaps some API calls are more identifying than others. Macneto extends the notion of software birthmarking by using deep learning to identify patterns of API calls and instruction mix, allowing it be an effective deobfuscator. Further, Macneto extends the notion of birthmarks by considering not just the code in a single method, but also instructions called by it.

4.4 Macneto Overview

From a set of original and obfuscated methods, Macneto intends to identify the original version of a given obfuscated method. Here we describe an overview of Macneto.

Although obfuscators may perform significant structural and/or naming transformations, the semantics of a program before and after obfuscation remain the same. Macneto leverages such semantic equivalence between an original program executable and its obfuscated version at the granularity of individual methods. The semantics of a method are captured as the hidden topics of its machine code ("machine topics") instead of human texts such as identifier names in methods. By construction, an obfuscated method is semantically equivalent to its original, non-obfuscated method that it is based on. Macneto assumes that the machine topics of an obfuscated method will match those of the original method. In its learning phase, Macneto is provided a training set of methods, which are labeled pairs of obfuscated and non-obfuscated methods. Once training is complete, Macneto can be presented with an arbitrary number of obfuscated and deobfuscated methods, and (assuming that the deobfuscated method exists in the input set) accurately match each obfuscated method with its original version. In the event that the original method body isn't known, Macneto can return a suggested method (that it had been trained on) which was very similar to the original method, and, in our evaluation, typically has the same name.

A high level overview of Macneto's approach for deobfuscation is in Figure 4.1. Macneto utilizes a four stage approach, where the first three stages occur offline and can be pre-trained:

(i) *Computing Instruction Distribution.* For an application binary (original or obfuscated), Macneto parses each method as a distribution of instructions, which is analogous to the term frequency vector for a document. Instead of considering just the instructions of each method, Macneto also recursively considers the instructions of each method's callee(s), which helps Macneto to deeply comprehend behavioral semantics.

(ii) *Machine Topic Modeling.* Identifies machine topics from the instruction distribution of the original method. These machine topics are used as a proxy for method semantics. The same topic model is used later to annotate the corresponding obfuscated method as well.

(iii) *Learning.* Uses a two-layered Neural Network (NN) where the input is the instruction distribution of a method (original and obfuscated), and the output layer is the corresponding machine topic distribution of the original method. MACNETO uses this two-layered NN as a program classifier that maps an original method and its obfuscated version to the same class and represented by machine topic vector . This is the training phase of the NN model. Such model can be pre-trained as well.

(iv) *Deobfuscating.* This is basically the testing phase of the NN model. It operates on a set of original and obfuscated methods that form our testing set. Given an obfuscated method, the above RNN model tries to infer its topic distribution; MACNETO in turn tries to find a set of original methods with similar topic distribution and ranks them as possible deobfuscated methods.

Consider the example `readAndSort` program shown in Figure 4.2, assuming that this is a method belonging to an Android app that we are using to train MACNETO. To compute the instruction distribution of `readAndSort`, MACNETO first calculates the callgraph and identify its two callees `read` and `sort`. The instruction distributions of these two callee methods will be inlined into `readAndSort`. Then MACNETO moves to the next step, applying topic modeling on the instruction distributions of all methods including `readAndSort` in the training app set. The result of this step is a vector containing the probability/membership that a method has toward each topic: a Machine Topic Vector (MTV). MACNETO annotates both the original and obfuscated versions of this method with this same MTV. This annotation process allows our learning phase to predict similar MTVs for a method and its obfuscated version, even when their instruction distributions are different.

4.5 Macneto Approach

This section describes the four stages of Macneto in detail, illustrating our several design decisions. We have designed Macneto to target any JVM-compatible language (such as Java), and evaluate it on Android apps. Macneto works at the level of Java bytecode; in principle, its approach could be applied to other compiled languages as well. In this thesis, executable/binary actually means Java bytecode executable and machine code means Java bytecode.

4.5.1 Computing Instruction Distribution

We use the instruction distribution of a method to begin to approximate its behavior. This involves two main steps:

1. for a target method m (original or obfuscated), Macneto recursively identifies all methods that it invokes (its callees) and then cumulatively computes the instruction distribution of m and its callees using term frequency distribution,
2. Macneto identifies the differences in callees between an original and obfuscated method using a *graph diff* algorithm, and optionally filters out the additional callees from obfuscated methods.

Here, we will explain each of these steps.

Instruction Distribution (ID)

ID is a vector that represents the frequencies of important instructions. For a method M_j, its instruction distribution can be represented as $ID_{M_j} = [freq_j^{I_1}, freq_j^{I_2}, ...freq_j^{I_a}]$, where a represents the index of an instruction and $freq_j^{I_a}$ represents the frequency of the a_{th} instruction in the method M_j. This step is similar to building the term frequency vector for a document.

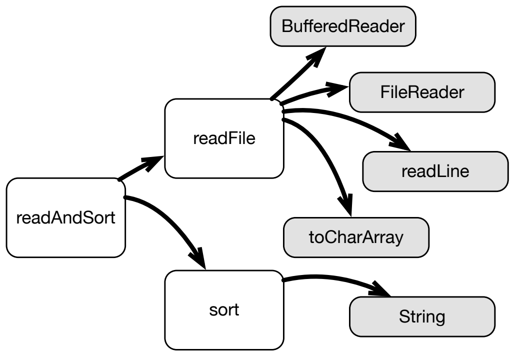
Macneto considers various bytecode operations as individual instructions (e.g. loading or storing variables), as well as a variety of APIs provided by the Android framework. Android API calls provide much higher level functionality than simple bytecode operators, and hence, including them as "instructions" in this step allows Macneto to capture both high and low level semantics. However, including too many different instructions when calculating the distribution could make it difficult to relate near-miss semantic similarity.

To avoid over-specialization, Macneto groups very similar instructions together and represents them with a single word. For instance, we consider all instructions for adding two values to be equivalent by abstracting away the difference between the instruction `fadd` for adding floating point numbers and the instruction `iadd` for adding integers. All instructions for adding different data types are categorized as a single one `xadd`. Table 4.1 lists all of the instructions Macneto considers.

Further, for a target method under analysis, Macneto inlines the instructions from the target's callee method(s) recursively in the instruction distribution to capture the target's context. For that, Macneto constructs callgraphs for applications and libraries using FlowDroid [10], a state-of-the-art tool that uses context-, flow-, field-, and object-sensitive android lifecycle-aware control and data-flow analysis [10]. For example, consider the method `readAndSort` as shown in Figure 4.2. `readAndSort` simply reads the first line of a file as a string and then sorts this string. It delegates its functionality to two subroutines, `readFile` and `sort`. Both `readFile` and `sort` also invoke several methods, such as `toCharArray` and `readLine` APIs included in Android to help them complete their tasks. The corresponding call graph is shown in Figure 4.2a.

Macneto considers following two classes of callee methods: (i) *Android APIs.* These methods are offered by the Android framework directly. Macneto models these APIs as single instructions. `readLine` and `toCharArray` in Figure 4.2 belong to this category. (ii) *Application methods.* These are all the other methods beside Android APIs. These

```java
public String readAndSort(String f) {
  char[] data = readFile(f);
  return sort(data);
}
```



(a) The `readAndSort` method and its callgraph.

```java
public static char[] readFile(String f) {
  try {
    BufferedReader br =
      new BufferedReader(new FileReader(f));
    String first = br.readLine();
    return first.toCharArray();
  } catch (Exception ex) {
  }
  return null;
}
public static String sort(char[] data) {
  for (int i = 1; i < data.length; i++) {
    int j = i;
    while (j > 0 && data[j - 1] > data[j]) {
      char tmp = data[j];
      data[j] = data[j - 1];
      data[j - 1] = tmp;
      j = j - 1;
    }
  }
  return new String(data);
}
```

(b) The callee methods of `readAndSort`.

Figure 4.2: The `readAndSort` method and its callgraph (a). Two callee methods, `readFile` and `sort`, of `readAndSort` (b).

Table 4.1: Macneto's instruction set.

Opcode	Description
`xaload`	Load a primitive or an object from an array, where `x` represents a type of primitive or object.
`xastore`	Store a primitive or an object to an array, where `x` represents a type of primitive or object.
`arraylength`	Retrieve the length of an array.
`xadd`	Add two primitives on the stack, where `x` represents a type of primitive.
`xsub`	Subtract two primitives on the stack, where `x` represents a type of primitive.
`xmul`	Multiply two primitives on the stack, where `x` represents a type of primitive.
`xdiv`	Divide two primitives on the stack, where `x` represents a type of primitive.
`xrem`	Compute the remainder of two primitives on the stack, where `x` represents a type of primitive.
`xneg`	Negate a primitive on the stack, where `x` represents a type of primitive.
`xshift`	Shift a primitive on the stack, where `x` represents integer or long.
`xand`	Bitwise-and two primitives on the stack, where `x` represents integer or long.
`xor`	Bitwise-or two primitives on the stack, where `x` represents integer or long.
`x_xor`	Bitwise-xor two primitives on the stack, where the first `x` represents integer or long.
`iinc`	Increment an integer on the stack.
`xcomp`	Compare two primitives on the stack, where `x` represents a type of primitive.
`ifXXX`	Represent all `if` instructions. Jump by comparing value(s) on the stack.
`xswitch`	Jump to a branch based on the index on the stack, where `x` represents `table` or `lookup`.
`android_apis`	The APIs offered by the Android framework, which usually starts from `android.`, `dalvik.`, `java.`. Macneto records 235 android apis.

are methods from an application or third party libraries used by an application. While Macneto treats Android APIs as individual instructions, all other application method calls are inlined into the calling method, resulting in the instruction distributions of those callee methods being merged directly in the target method. We can then define the instruction distribution of a target method M_j as:

$$ID_{M_j} = ID_{M_j} + \sum_{M_k \in callees(M_j)} ID_{M_k} \qquad (4.1)$$

, where j is the index of the current method and k represents the indices of its callee methods. We use these instruction distributions as the input source for next step to identify topics embedded in programs.

To calculate these instruction distributions, Macneto uses the ASM Java bytecode library [11], and Dex2Jar [35]. This allows Macneto to deobfuscate Android apps (which are distributed as APKs containing Dex bytecode), while only needing to directly support Java bytecode. For collecting Android APIs, we analyze the core libraries from Android API level 7 to 25 [6].

Graph Diff

Some obfuscators may inject additional instructions into the original methods, which might then change instruction distribution and hence the perceived semantics of those methods. Since Macneto inlines application method calls, inserting additional method calls could cause significant distortions to the instruction distribution of a method. Macneto combats this obfuscation by learning (and then filtering out) superfluous method calls that are systematically inserted by obfuscators from the callgraphs before/after obfuscation of the original method. We call this learning module on callgraphs as *graph diff*.

For each method in an Android app, Macneto first analyzes the callee difference of the current method M_j before and after the obfuscation

$$\Delta M_j = callees(M_j^{obfus}) - callees(M_j) \tag{4.2}$$

, where $callees(M_j)$ and $callees(M_j^{obfus})$ represent the callees of M_j before and after the obfuscation, respectively. If ΔM_j is not empty, Macneto records the instruction distributions of the methods in ΔM_j as patterns. In a training app set, Macneto computes the appearances of each pattern (instruction distribution) it learns from the callee differences and reports those patterns having high support. We first define the total callee differences detected by Macneto as

$$Diff_{callee}(T) = \bigcup_k \Delta M_k \tag{4.3}$$

, where T represents a training app set and k is the method index. Then we define the support of a pattern as

$$Support(Pattern_i) = \frac{Count(Pattern_i)}{\sum_j Count(Pattern_j)} \tag{4.4}$$

, where i is pattern index, $Pattern_i \in Diff_{callee}(T)$ and $Count(.)$ returns the appearance number of a pattern.

We again use `readAndSort` example in Figure 4.2 to demonstrate how graph diff works. Let's assume an obfuscator injects a method `decrypt(String)` to method `readAndSort`; the callees of obfuscated `readAndSort` becomes $callee(readAndSort^{obfus}) = \{\texttt{read}, \texttt{sort}, \texttt{decrypt}\}$. Thus, the $\Delta readAndSort = \{\texttt{decrypt}\}$. The instruction distribution of `decrypt`, $ID_{decrypt}$, is recorded by Macneto as a pattern. If the support of such pattern is higher than a threshold (0.03 in this thesis), Macneto reports it as a significant pattern and uses it to filter out superfluous method calls in the testing app set.

Note that the Graph Diff step is optional, but can have a positive impact to recover methods obfuscated by an advanced obfuscator. We evaluated the impact of including the Graph Diff step in detail in Section 4.6.

4.5.2 Machine Topic Modeling

Topic modeling [21] is a generative model that identifies the probabilities/memberships that a document has to (hidden) topics or groups. Topic modeling has been widely used in software engineering literature to understand programs and detect anomalies [116, 52]. Most existing approaches apply topic modeling on human-readable program elements (e.g, method names, comments or texts in source code) or program documents to identify hidden topics. These approaches treat each program as plain text and use topic modeling to identify topics in programs, as labeled by human words. However, these human words may be noisy and inadequate to describe program behavior. Further, an obfuscated or anonymized program may not have a meaningful method name or other identifiers, and would not have any comments left. Existing approaches reliant on human words to cluster or classify programs by topics may fail to process such programs.

In contrast, in this thesis, we propose the concept of *Machine Topics*, where we attempt to identify the probability distribution that a method belongs to multiple topics from *machine code*. We further include machine code of the callee methods to retrieve the contextual semantics as well. Modeling instructions (machine code) as terms and methods as documents, Macneto uses Latent Dirichlet Allocation [21] to extract hidden machine topics in methods. We extract topics from the original method since some noisy instructions such as `nop` might be injected into an obfuscated method by an obfuscator. To the best of our knowledge, Macneto is the first system to identify topics of programs from machine code.

LDA models each document as a sequence of words, where each document can exhibit a mixture of different topics. Each topic is subsequently modeled as a vector of words. Here, we model instruction distribution of a method as a document where each instruction is a word. Thus, following the concept of LDA, a *machine topic* becomes a vector of instructions (machine words) and can be represented as:

$$MT_i = [Pr_i^{I_1}, Pr_i^{I_2} ... Pr_i^{I_a}] \tag{4.5}$$

, where MT_i is the i_{th} machine topic and $Pr_i^{I_a}$ represents the probability of the instruction I_a belonging to the topic in MT_i. Each method can also be modeled in terms of machine topic vector (MTV):

$$MTV(M_j) = [Pr_j^{MT_1}, Pr_j^{MT_2} ... Pr_j^{MT_b}] \tag{4.6}$$

, where M_j represents the j_{th} method and MT_b represents the b_{th} machine topic. $Pr_j^{MT_b}$ represents the probability/membership of the method M_j belonging to the topic MT_b.

In Macneto, we define 35 machine topics ($b = 35$) and have 252 types of instructions ($a = 252$) as we listed in Table 4.1. While optimizing the topic number is out of the scope of this thesis, we observe that 35 machine topics can split all methods in a reasonable way. Using these 35 machine topics, we generate unique machine topic vector (MTV). Note that, the dimension of each MTV is same as the topic number, i.e., 35, although the number of topic vectors can be potentially infinite due to different probability values (see Equation 4.6). Thus, a unique method M_j can have a unique topic vector $MTV(M_j)$ that encodes the probability of the method belonging to each machine topic. $MTV(M_j)$ becomes the semantic representation of both M_j and its obfuscated counter part M_j^{obfus}. We annotate each original and its obfuscated method with the corresponding machine topic vector and use them to train our NN based classifier, which will be discussed in Section 4.5.3. To compute machine topics and topic vector for each method, we use the Mallet library [92].

In the next two steps, Macneto aims to deobfuscate an obfuscated method using a NN based deep learning technique. In the training phase, the NN learns the semantic relationship between a original and obfuscated method through their unique MTV. Next, in the testing (deobfuscating) phase, given a obfuscated method, NN retrieves a set of candidate method having similar MTVs with the obfuscated method. MACNETO then scored these candidate methods and outputs a ranked list of original methods with similar MTVs.

4.5.3 Deep Learning Phase

In this step, MACNETO uses a NN based deep learning technique [119] to project the low-level features (Instruction Distributions) of methods to a relevant distribution of machine topics (Machine Topic Vector). MACNETO treats MTV as a proxy for program semantics, which should be invariant before and after obfuscation. Thus, MTV can serve as a signature (i.e., class) of both original and obfuscated methods. Given a training method set T, MACNETO attempts to project each method $M_j \in T$ and its obfuscated counterpart M_j^{obfus} to the same MTV, i.e., $M_j \rightarrow MTV(M_j) \leftarrow M_j^{obfus}$.

Similar deep learning technique is widely adopted to classify data [119]. However, most of these data comes with pre-annotated classes to facilitate learning. For example, Socher et al. [119] uses deep learning to classify images to relevant wordings. Such work has benchmarked images accompanied with correct descriptions in words to train such classifiers, MACNETO does not have any similar benchmarks. However, MACNETO *does* have available sets of applications, and has access to obfuscators. Hence, MACNETO builds a training set by co-training a classifier on obfuscated and deobfuscated methods (with MACNETO knowing the mapping from each training method to its obfuscated counterpart).

MACNETO characterizes each method M_j and M_j^{obfus} by the same machine topic vector $MTV(M_j)$, allowing it to automatically tag each method for training program classifiers. Given an unknown obfuscated method, MACNETO can first classify it to relevant machine topics (a MTV), which helps quickly search for similar and relevant original method. Only these original methods sharing similar MTVs with the unknown method will be scored and ranked by Macneto, which enhances both system performance and effectiveness of deobfuscation.

To train such projection/mapping, Macneto tries to minimize the following objective function

$$\begin{aligned} J(\Theta) = \sum_{M_j \in T} & \left\| MTV(M_j) - g(\theta^{(2)} \cdot f(\theta^{(1)} \cdot M_j)) \right\|^2 \\ & + \left\| MTV(M_j) - g(\theta^{(2)} \cdot f(\theta^{(1)} \cdot M_j^{obfus})) \right\|^2 \end{aligned} \tag{4.7}$$

, where T is a training method set, $MTV(M_j) \in \mathbb{R}^b$ (because Macneto defines b machine topics), $\Theta = (\theta^{(1)}, \theta^{(2)})$, $\theta^{(1)} \in \mathbb{R}^{h \times a}$ and $\theta^{(2)} \in \mathbb{R}^{b \times h}$. For hidden layers, Macneto uses $tanh$ function ($f(.)$) as the first layer and uses $logistic$ function ($g(.)$) as the second layer. Macneto uses the technique of stochastic gradient descent (SGD) to solve this objective function.

As we discussed in Section 4.5.2, there can be infinite classification (MTV) in Macneto, which may result in the un-convergence of our classifier learning. Thus in this learning phase, we select those methods having high memberships (> 0.67) toward specific machine topics. Our experiment result shows that the classifier built on these high-membership methods actually work on all methods (see Section 4.6).

4.5.4 Deobfuscating

Taking an obfuscated method as a query, Macneto attempts to locate which original method in the codebase have the lowest distance from it. The NN in Macneto can effectively infer the machine topic vector (MTV) of an unknown obfuscated method and then locate a set of original candidates having similar MTVs measured by the cosine similarity.

To further score and rank candidates, we develop a scoring model, which takes both semantic information and structural information of programs [41, 99] into account. For semantic information, we use the instruction distribution as the feature. For structural information, we select the features of methods on callgraphs, which include centrality (PageRank in this thesis), in-degree (how many other methods call this method) and out-degree (how many other methods this method calls) of the method. We then use a linear combination to compute distance between two methods:

$$\begin{aligned} Dist(M_k, M_l, W) = w_{inst} * Dist_{kl}^{inst} + w_c * Dist_{kl}^{c} \\ + w_{in} * Dist_{kl}^{in} + w_{out} * Dist_{kl}^{out} \end{aligned} \tag{4.8}$$

, where k and l are method indices, and $W = \{w_{inst}, w_c, w_{in}, w_{out}\}$. For computing $Dist_{kl}^{inst}$, we apply cosine similarity, while for the other three features, we compute the absolute differences between two methods.

Because our objective is to maximize the precision of the deobfuscation, we can formalize our objective function as

$$\underset{W}{\text{argmax}} \sum_i I(M_i, Deob(M_i^{obfus}, T, Dist(M_k, M_l, W))) \tag{4.9}$$

, where T is a training method set, $Deob(.)$ returns the nearest neighbor method of M_i^{obfus} based on the distance function and $I(.)$ returns 1 if the results from $Deob(.)$ is M_i else return 0. To solve this function, we apply Simulated Annealing [69] to Macneto for optimizing the weighting numbers W.

4.6 Evaluation

We evaluated the performance of Macneto on two popular obfuscators: ProGuard [109] and Allatori [3]. For each obfuscator, we gave Macneto the task of recovering the original version of each obfuscated method. We selected these obfuscators based on a recent survey of Android obfuscators, selecting ProGuard for its widespread adoption and Allatori for its complex control and data-flow modification transformations [13]. We performed our evaluation on the most recent versions of these tools at time of writing: ProGuard 5.3.2 and Allatori 6.0. In particular, we answer the following two research questions:

- **RQ1:** How accurately can MACNETO deobfuscate methods transformed by a lexical obfuscator?
- **RQ2:** How accurately can MACNETO deobfuscate methods that are obfuscated using control and data transformation?

To judge MACNETO's precision for method deobfuscation, we needed a benchmark of plain apps (that is, not obfuscated) from which we could construct training and testing sets. For each app, we applied both ProGuard and Allatori, each of which output a mapping file (to aid in debugging) between the original (not obfuscated) method, and the obfuscated equivalent. Hence, we used the $1,611$ Android apps from the F-Droid open-source repository of Android apps as experimental subjects [42]. In our experiments, we vary the numbers of apps included in training and testing app sets and then randomly select apps into both sets.

We first split these apps into a training set and a testing set and then obfuscate each of them. Both the original and obfuscated training sets are used to train the program classifier using the first three steps outlined above. To evaluate the deobfuscation precision of MACNETO, we use methods in each app in the obfuscated testing set as a query to see if the original versions of these obfuscated methods can be retrieved by MACNETO from the original testing set. In our training and testing phase in this thesis, we filter out the trivial methods having very few instructions (< 30) or very few types of instructions (< 10), because they may not offer sufficient information for MACNETO to deobfuscate. The constructor methods `<init>` and `<clinit>` are also excluded, because their functionality is usually setting up fields in classes/objects without too much logic.

We compare our results directly with the state-of-the-art Android deobfuscator *DeGuard* [20]. While DeGuard can support inferring other obfuscated information, such as field names and data types in programs in addition to method names, we only compare

MACNETO's capability to recover method names. Note that the evaluate suite we used (from F-Droid) matches the same suite used in Bichsel et al.'s evaluation of DeGuard [20]. The size of full app set from F-Droid we use is slightly different with DeGuard: we have $1,611$ apps but DeGuard has $1,784$. This is because about 170 apps cannot be processed by the Allatori obfuscator or use some 3_{rd} party libs that are not included in app, which are detected by MACNETO. DeGuard's approach is not applicable to obfuscators that transform control flow (such as Allatori), and hence, we only include DeGuard results for the ProGuard experiments.

As a baseline, we also compare MACNETO to a naïve approach that simply calculates the distance between two methods using the feature-scoring equation presented in the previous section (equation 4.8). This baseline does not include MACNETO's topic modeling classifier and weighting number optimization for the scoring equation.

4.6.1 Evaluation Metrics

We use two metrics to evaluate MACNETO's performances to deobfuscate programs: precision and Top@K. We first define a testing method set as $\{M_i | i \in \mathbb{R}\}$, and its obfuscated counterpart as $\{M_j^{obfus} | j \in \mathbb{R}\}$, where i and j are the method indices. The definition of precision is

$$precision = \frac{\sum_j I(M_j, Deob(M_j^{obfus}))}{\left|\{M_j^{obfus}\}\right|} \tag{4.10}$$

, where $Deob(.)$ returns the deobfuscation result of M_j^{obfus} by a system and $I(.)$ returns 1 if the result of $Deob(.)$ is the same with the real original version M_j, else returns 0.

As we discussed in Section 4.1, we model the deobfuscation problem as a nearest neighbor search problem. Thus, we also use Top@K, which is widely adopted to measure the performance of search systems, as an evaluation metric. Top@K is a generalized version of precision, if we replace $Deob(.)$ in Eq. 4.11 by a ranking function:

$$Top@K = \frac{\sum_j I(M_j, Rank(M_j^{obfus}, \{M_i\}, K))}{\left|\{M_j^{obfus}\}\right|} \tag{4.11}$$

, where $Rank(.)$ first computes the distance between M_j^{obfus} and each method in the testing method set $\{M_i\}$ by a distance function (Eq. 4.8 in this thesis), and then return K methods having the shortest distances from $\{M_i\}$. $I(.)$ returns 1 if the original version M_j is in the K methods returned by $Rank(.)$, else returns 0. Precision, then, is Top@1. In our experiments, we use $K = \{1, 3, 10\}$ to evaluate the system performance. For DeGuard, because it only predicts the best answer ($K = 1$) and we cannot access their source code after contacting the authors, we only evaluate DeGuard by precision.

4.6.2 Deobfuscating ProGuard

RQ1. How accurately can Macneto deobfuscate methods transformed by a lexical obfuscator?

To compare Macneto with DeGuard, we use the same testing and training data sets as used to evaluate DeGuard [20], which includes 110 Android apps for testing. A direct comparison between the two systems is complicated: Macneto only considers methods that are reachable from any entry point (e.g. those on a callgraph rooted by standard Android entry points), whereas DeGuard considers all methods in an app for deobfuscation (including those that could never be called). Further, for evaluating DeGuard, the original paper [20] attempts to map an obfuscated method to a method in the training apps, so some obfuscated methods having names that never appear in the training apps may not be matched. Because the source code of DeGuard is not freely available online (or from the authors), we were not able to modify DeGuard to fit our evaluation. For completeness, we include the comparison results with this caveat.

We use the rest $1,501$ apps as the training data set to train Macneto and deobfuscate these same 110 apps used in the evaluation of DeGuard. On this task, we found that the precision (i.e. Top@1) of MACNETO was 96.29%. Top@3 and Top@10 of MACNETO are 99.31% and 99.86%, respectively.

While Bichsel et al. report the overall precision (80%) of DeGuard on all program properties including method names, field name, data type, etc., the exact precision for each specific program property is not reported (except graphically) [20]. To determine the precision of DeGuard on method names alone, we used the DeGuard web interface [31] to attempt to deobfuscate these same 110 apps. While we do not find the information about the mapping between obfuscated methods and their deobfuscated counterparts on this web interface, we can only compute the precision by counting the percentage of the deobfuscated method names appearing in the original apps. In this experiment, we found DeGuard's precision on deobfuscating method names to be 66.59% by our evaluation, matching the results in Figure 6 of the original DeGuard paper [20].

MACNETO achieves almost perfect deobfuscation on ProGuard in our evaluation. ProGuard renames identifiers (human words) in programs without further program transformation, such as changing control flow. Thus, the instruction distributions and structural information of each method are similar before and after ProGuard's obfuscation. The program classifier and the scoring function in MACNETO can resolve such identifier renaming, because the information in the machine code mostly stays similar.

We also apply the naïve approach developed by us to deobfuscate these apps. The precision is 53.84%, 74.79% and 90.79% for Top@1, Top@3 and Top@10, respectively. Comparing MACNETO with the naïve approach, we can find that our deep learning based program classifier can help enhance the precision of deobfuscation. The naïve approach also relies on the instruction distribution to deobfucate methods, but this approach seems to be too sensitive. Its precision (Top@1) is only 53.84%, even though its Top@10 is reasonable: $\sim 91\%$ of methods can be ranked in the top 10 positions.

Result 1: MACNETO can deobfuscate lexically obfuscated methods with 96% precision.

Table 4.2: Results of deobfuscating Allatori-obfuscated code.

ID	#Train APKs	#Test APKs	Method Selection	Methods	System	Gdiff.	Worst comp.	Real comp.	Filter	Top@1	Top@3	Top@10	Boost@1
#1	1501	110	0.67	1, 932	Macneto	✓	3.92 M	2.04 M	47.85%	91.10%	96.95%	98.81%	206% ↑
					Naïve					29.76%	33.59%	35.92%	
#2	1501	110	N/A	12, 690	Macneto	✓	174.30 M	11.70 M	93.29%	81.33%	89.55%	92.36%	87% ↑
					Naïve					43.56%	48.79%	58.52%	
#3	1501	110	0.67	2, 148	Macneto		4.81 M	2.77 M	42.35%	75.42%	90.27%	95.58%	231% ↑
					Naïve					22.77%	26.16%	30.30%	
#4	1501	110	N/A	12, 695	Macneto		174.37 M	11.28 M	93.53%	68.96%	82.47%	89.37%	102% ↑
					Naïve					34.21%	40.48%	51.45%	
#5	1111	500	0.67	12, 533	Macneto	✓	213.60 M	46.82 M	78.08%	69.62%	82.37%	92.25%	74% ↑
					Naïve					40.12%	45.83%	51.36%	
#6	1111	500	N/A	97, 578	Macneto	✓	11203.32 M	500.01 M	95.54%	69.86%	81.84%	89.61%	140% ↑
					Naïve					29.16%	33.26%	37.71%	
#7	1111	500	0.67	10, 798	Macneto		161.68 M	38.78 M	76.02%	52.15%	65.91%	78.16%	58% ↑
					Naïve					33.11%	39.78%	46.33%	
#8	1111	500	N/A	97, 567	Macneto		11202.06 M	464.81 M	95.85%	50.30%	63.14%	73.52%	149% ↑
					Naïve					20.19%	23.91%	27.86%	
#9	611	1000	0.67	30, 102	Macneto	✓	1130.81 M	117.40 M	89.62%	63.99%	78.11%	87.09%	129% ↑
					Naïve					27.90%	32.79%	42.52%	
#10	611	1000	N/A	152, 817	Macneto	✓	26991.00 M	984.13 M	96.35%	69.56%	81.37%	88.63%	108% ↑
					Naïve					33.43%	38.01%	43.97%	
#11	611	1000	0.67	25, 064	Macneto		805.63 M	79.81 M	90.09%	35.59%	53.79%	66.57%	115% ↑
					Naïve					16.53%	20.30%	25.85%	
#12	611	1000	N/A	152, 817	Macneto		26991.00 M	1082.85 M	95.99%	48.51%	60.42%	69.56%	105% ↑
					Naïve					23.65%	27.83%	32.25%	

Column Description: ID: Conguration ID; Train APKs and Test APKs: numbers of training and testing APKs, respectively; Method Selection: denotes a method's membership towards a machine topic; Method: total number of testing methods; System: system under evaluation; Gdiff: whether graph diff module is enabled or not; Worst comp.: total number of method comparisons *without* program classifier; Real comp.: total number of method comparisons with the program classifier; Filter: the percentage of unnecessary comparisons that is saved by Macneto; Boost@1: enhancement achieve by Macneto over the naïve approach on precision (Top@1).

It outperforms a naiv̈e approach by 42 percentage points.

4.6.3 Deobfuscating Allatori

RQ2. How accurately can Macneto deobfuscate methods that are obfuscated using control and data transformation?

Compared with ProGuard, which mainly focuses on renaming identifiers in programs, Allatori changes control flow and encrypts/decrypts strings via inserting additional methods into programs. To demonstrate the performance of Macneto against such advanced obfuscations, we trained 6 deobfuscation models (varying several parameters such as the sizes of training and testing app sets of Macneto) and report the precision and Top@K. We consider building and applying each model to two types of testing methods:

1. all methods that have been obfuscated, and
2. only *significant* methods that have been obfuscated, as determined by those that have relatively high memberships (> 0.67) toward any machine topic.

This leads to 12 results (6 models $*$ 2 types of method set) in our evaluation.

The overall results of the 6 models on 2 types of method sets as well as the comparison results between Macneto and the naïve approach can be found in Table 4.2. In Table 4.2, the "ID" column represents the configuration ID, where we have 12 configurations in total. The "Train apps" and "Test apps" columns represent the numbers of training and testing apps, respectively. Note that the size of training apps does not matter to the naïve approach, because it simply relies on instruction distributions and the vanilla weighting numbers to deobfuscate programs. We randomly split all apps from F-Droid into our training and testing app sets. Note that the configuration 4 with $\{Train = 1501, Test = 110\}$ matches the exact same configuration used in the original evaluation of DeGuard. The "Method Selection" column denotes the two types of method selection based on the filtration criteria of methods' membership towards any topic as discussed above, and the "Method" column records the number of testing method. The "System" column shows which system we evaluate, and the "Gdiff" column shows if our graph diff module discussed in Section 4.5.1 is enabled or not.

As discussed in Section 4.5.4, Macneto uses the program classifier to filter out those methods that do not share the similar classification (MTV) of a given obfuscated method. The "Worst comp." column shows the total comparison numbers at the scale of million methods in the deobfuscation stage (see Section 4.5.4) *without* the program classifier that Macneto would have to perform, and the "Real comp." column reports the total comparison number with the program classifier. The "Filter" column represents the percentage of unnecessary comparisons that can be saved by Macneto with the program classifier.

The "Top@K" columns, where $K = \{1, 3, 10\}$, are self-explanatory. The "Boost@1" column shows the enhancement achieve by MACNETO over the naïve approach on precision (Top@1).

There are three key findings we observe in Table 4.2

1. Good deobfuscation performance: In general, MACNETO can achieve $80+\%$ Top@10 under most configurations. As the number of the training apps grows, MACNETO can even achieve $\sim 99\%$ Top@10.
2. Effectiveness of the program classifier trained by deep learning: The filter rate of unnecessary comparisons is usually higher than 80% for most configurations. Such filtering achieved by our program classifier also enhances the system performance of MACNETO to deobfuscate programs, which will be discussed in the following paragraphs.
3. Effectiveness of *graph diff*: Given the same training-testing app set, MACNETO can have up to 20% enhancement on Top@10 by enabling graph diff. Allatori encrypts string literals in programs, so it injects additional methods into programs to decrypt these strings, which changes the program structure and instruction distribution. Our graph diff. module precisely identifies these injected decryption. With the graph diff. module, MACNETO achieves roughly 90% Top@10 when the size of the training data is small.

The Boost column shows the precent increase in precision from the naïve approach to MACNETO —note that this improvement is more than 100% in most models. As the number of testing apps increases, which means that the number of training apps decreases (this only influences MACNETO, since the naïve approach does not have a training phase), the performance of both system drops, even though MACNETO still outperforms the naïve approach. MACNETO offers a relatively stable precision when the size of testing apps ($152K+$ methods in 1000 apps) is large and the size of training apps (611 apps) is small: $\sim 70\%$ Top@10, while the naïve approach drops to about 32%. By enabling the graph diff.

module, MACNETO can achieve $88+\%$ Top@10, which does not even drop significantly due to the increase of the search range (testing set).

The same finding can also be observed when we evaluate both systems on the significant methods having memberships > 0.67 toward specific machine topics in the testing app set. While the performance of the naïve approach does not improve significantly, MACNETO has about 10% improvement on the precision compared with testing on all methods in Table 4.2, when the size of test apps is 110. This results in MACNETO achieves 200% boost over the naïve approach.

In general, the deobfuscation precisions of MACNETO steadily increase as the size of the training/testing set increases/decreases, while the precisions of naïve approach do not enhance significantly as the size of search range (testing set) decreases. For most of the configurations, MACNETO can rank the correct original versions of 80% of the obfuscated methods in the top 10 position. Even though the training app set is small (611 apps), the Top@10 offered by MACNETO is still about $\sim 70\%$.

Result 2: MACNETO can deobfuscate up to 91% precision. It significantly outperforms a naiv̈e approach in all the tested configurations.

4.6.4 Threats to Validity

In our evaluation, we collect $1600+$ apps from an open-source repository. It is possible that such app set is not representative enough. Also, we apply MACNETO to recover programs obfuscated by two obfuscators in our evaluation. Some threat models and transformation techniques adopted by other obfuscators may not be evaluated in this thesis. In the future, we plan to collect more apps and obfuscators to test and enhance the system capability of MACNETO.

When we split the apps into training set and testing set, we use a random approach. It is possible that MACNETO may not perform well on some testing sets, but we only have 3 in our evaluation. To enhance the confidence on MACNETO, we can conduct a K-fold cross validation [7] to make sure the performance of MACNETO is stable.

4.6.5 Limitations

Currently, MACNETO relies on the callgraph of the app to understand the semantic of each method and merge instruction distributions. Thus, only the methods in the callgraph can be deobfuscated by MACNETO. These methods are potentially the ones that will be executed in runtime.

The graph diff. module in MACNETO can identify the methods frequently injected by the obfuscator. By using the instruction distribution as the pattern, MACNETO can compute which patterns appear frequently in the training apps after obfuscation and then filter out the methods having the same instruction pattern in the testing apps. One possible obfuscation that MACNETO may not be able to handle is to randomly generate methods, where each method has different instruction distribution. In this way, MACNETO may not be able to determine which methods are intentionally injected by the obfuscator. The other possibility is that some methods having the similar instruction distributions with the detected patterns can be falsely filtered by MACNETO. Further, because MACNETO recursively merges instructions from callee methods into their caller methods, even if a single method is falsely filtered, a large portions of method can be affected in a negative way. One potential solution for these two limitations is to use data flow analysis [10] in MACNETO to determine which callees may not influence the result of the current method so that they can be safely removed.

4.7 Conclusions

Deobfuscation is an important technique to reverse-engineer an obfuscated program to its original version, which can facilitate developers understand and analyze the program. We present MACNETO, which leverages topic modeling on instructions and deep learning techniques to deobfuscate programs automatically. In two large scale experiments, we apply MACNETO to deobufscate 1600+ Android APKs obfuscated by two well-known obfuscators. Our experiment results show that MACNETO achieves 96 + % precision on recovering obfuscated programs by ProGuard, which renames identifiers in APKs. MACNETO also offers great precision on recovering programs obfuscated by an advanced obfuscator, Allatori, which changes control flow and inserts additional methods into APKs in addition to renaming identifiers. Compared with a naïve approach relying on instruction distribution of the obfuscated method to search for the most similar original version in the codebase, MACNETO has up to 200% boost on the deobfuscation precision.

Chapter 5

Discussions of Similarity Detection

Identifying similar code can support multiple software engineering tasks including enhancing maintenance of software systems, searching for relevant code befitting developers' purposes, clustering/classifying programs etc. In this chapter, we categorize both of our and existing approaches to detect different types of similar programs and discuss their potential applications.

While the categorization of all similar programs/software is an open question, we attempt to classify similar programs into six categories, which can be seen in Table 5.1. The category column describes the name of the category; the description column gives a high level explanation regarding how to detect similar programs by some features of programs for the given category; the analysis column records which types of analysis that the current category usually uses: static, dynamic or both; the existing work column summarizes the papers/publications that we will discuss in this chapter. The *Learning* category can be orthogonal to the other five categories, because the learning techniques can potentially be integrated with these five categories to analyze programs.

5.1 Static Detection of Similar Programs

5.1.1 Revisit of Static Features

Roy et al. [114] conducted a survey regarding the four types of code clones and the corresponding techniques to detect them ranging from those that are exact copy-paste clones to those that are semantically similar with syntactic differences. In general, these static

Table 5.1: A summary of code similarity, which contains six categories: syntactic, functional, executional, learning-based, conceptual and other.

Category	**Description**	**Analysis**	**Existing Work**
Syntactic	Use syntactic information of programs as a feature to detect similar programs. The syntactic information here does not only limit to abstract syntax tree (AST) but also token, program dependence graph or other information that encodes 'structures' of programs.	Static	[14, 63, 115, 17, 59, 73, 89, 67]
Functional	Use inputs and/or outputs of programs as a functional feature to detect similar programs. *While most approaches in this category use dynamic analysis, some approaches use symbolic execution, which is a middle ground of static and dynamic analysis, to approximate functionality of programs.	Dynamic, Static*	[60, 32, 123, 39, 120]
Executional	Use some features of program executions to detect similar programs. *While most approaches in this category also rely on dynamic analysis, some approaches attempt to leverage the power of machine learning to approximate program behavior.	Dynamic, Static*	[121, 34, 33, 38, 43, 65, 26, 9, 44]
Conceptual	Use text information in code, such as comments, documentations and identifiers, as a feature to detect similar programs.	Static	[93, 94, 98, 29]
Other	Use other features of code/machines, such as web traffic and workload, to detect similar code.	Other	[118, 64]
Learning	Use machine learning models to identify hidden features in codebases. These hidden features can support developers observing commonality between code, which is difficult for humans to identify.	Static, Potentially dynamic	[111, 106]

approaches first parse code into a type of intermediate representation (static features) and then develop corresponding algorithms to identify similar patterns. As the complexity of the intermediate representation grows, the computation cost to identify similar patterns is higher. Based on the types of intermediate representations, the existing approaches can be classified into token-based [14, 63, 115], AST-based [17, 59] and graph-based [73, 89]. Among these general approaches, the graph-based approaches are the most computationally expensive, but they have better capabilities to detect complex clones according to the report of Roy et al. [114]. While some approaches in Roy et. al's report attempt to detect similar code with syntactic differences, we still categorize all approaches in this report as syntactically similar, because they still leverage syntactical information in the code.

Several other techniques make use of general information about code to detect counterparts rather than strictly relying on syntactic features. Marcus and Maletic use identifier names and comments in source code to search for *high level concept clones* [93, 94]. Another line of similar code detection involves creating fingerprints of code, for instance by tracking API usage [98, 29], to identify code having similar semantics/concepts.

While the true behavior and/or functionality of a code can only be identified in runtime (dynamically), some approaches leverage the power of symbolic execution [120] or data mining [43] to *approximate* these dynamic features of code. Further use cases based on these approaches will be discussed in Section 5.1.2.

5.1.2 Potential Use Cases of Static Similar Programs

As we discussed in Chapter 2, many software engineering tasks can be supported by similar code (code clone) detection. A common problem caused by these similar code fragments is the maintenance cost [63, 115]. One possibility is that a piece of code is buggy, but it is copied and pasted in multiple locations in a system. The cost to rectify these bugs can be $O(n)$, where n is the time that this buggy code is copied and pasted. Having a tool to identify and locate these similar code fragments can help developers manage and maintain their software systems.

While locating duplicated code to enhance software maintenance usually requires computing *syntactic* similarity between code, some work of static analysis collects other features, such as document, API usage or path constraint, of code to help developers search for code or document which are conceptually or functionally similar. Marcus and Maletic attempt to construct the relationship, tracebility links, between code and its corresponding documents [93, 94]. McMillan et al. [98] computes the similarity between Java software applications by which APIs they use. Such application-based similarity can help developers conduct rapid prototyping and support companies to speed up the drafting of project documents to win the bid of a proposal, i.e., similar application can have similar project documents.

Stolee et al. [120] offers another feature of code to compute similarity: path constraints. Their Satsy system uses I/O examples of methods to generate path constraints of code as features. When a developer gives an I/O example, the Satsy system applies symbolic execution [68] to generate path constraints (specifications) and then uses a SMT solver to check and search for methods that fit into these specifications. Symbolic execution is not a pure static analysis. To be more specific, instead of supplying real input to drive programs, symbolic execution uses *symbolic* input as an abstraction of real input to explore which paths in programs can be visited by which inputs [125].

In addition to software engineering tasks, searching for similar code can support security tasks. Kim et al. [67] develop a high performant clone detection system, Vuddy, which searches for vulnerable functions/methods in a target system. Each function/method in the target system will be first fingerprinted by a hash function and will be compared with vulnerable code recorded in a codebase. Feng et al. [43] devises the system, Genius, to search for potential vulnerable function in Internet Of Things (IoT) devices, given a vulnerable function as a query. Genius fingerprints each function/method as a control flow graph, where each vertex is a basic block of instructions and each edge is the control dependency between two basic blocks. The approach of Genius is similar to our prior work, DyCLINK [121], where the major difference is that DyCLINK encodes each *execution* of a method as a graph but Genius encodes each function/method as a graph. Feng et al. attempts to use graph analysis to approximate malware's behavior as well [44, 45]. However, their approaches invent a new type of graph called inter-component call graph, where a vertex is a component in a programs and an edge encodes the communication including actions and data types between components, to represent the behavior of Android's malware. Mamadroid [95] proposes a different perspective to approximate malware's behavior: computing transitional probability between API calls (Hidden Markov Model) as behavioral signatures of malware, where the transitional probability between two API calls represent the probability that an API is called after another one.

Static features of programs have been studied for years and have been applied in many use cases/applications, which have been discussed in this section. In recent years, researchers attempt to apply other techniques, such as symbolic execution and data mining, to advance the development of static features in programs.

5.2 Dynamic Detection of Similar Code

5.2.1 Revisit of Dynamic Features

Static features are not the only way to identify similar code. Some approaches attempt to detect code having similar dynamic features (behaviorally similar code) despite syntactic differences by using dynamic profiling. For instance, Elva and Leavens [39] proposes detecting functional clones by identifying methods that have the exact same outputs, inputs and side effects. The MeCC system [65] summarizes the abstract state of a program after each method is executed to relate that state to the method's inputs, allowing for exact matching of outputs. Carzaniga et al. [26] studies different ways to quantify and measure functional redundancy between two code fragments on both of the executed code statements and performed data operations.

Jiang and Su's EQMiner [60] and the comparable system developed by Deissenboeck et al. for Java [32] are two recent examples of dynamic detection of functional similar code, which are highly relevant to our HitoshiIO system. EQMiner first chops code into several chunks and randomly generates input to drive them. By observing output values from these code chunks, the EQMiner system is able to cluster programs with the same output values. The EQMiner system successfully identified clones that are functional equivalent. However, to apply such technique on object oriented language, multiple technical challenges have been identified [32]. With data flow analysis and a simple I/O based similarity model developed by us, we are able to detect functionally *similar* (not necessarily equivalent) programs [123].

In addition to detecting behaviorally similar code in functionalities, we propose to identify code having similar runtime executions (instruction graphs) [121], which we call "code relatives". Our prior work [122] addresses the challenges to detect behaviorally similar code such as how to encode program behavior in a computational format.

5.2.2 Potential Use Cases of Dynamic Similar Programs

Compared with using static features to identify similar programs, applying dynamic features is a relatively new area. Thus, we do not have as many applications by using dynamic features as using static ones. However, the experiment results of existing work shows that dynamic features can help developers identify programs having similar dynamic behaviors without similar syntactic features, which can be hard for static analysis to identify.

As we discussed in Section 5.2.1, the EQMiner system proposed by Jiang and Su [60] identifies functionally equivalent code. Their experiment result shows that such dynamic technique can detect more similar code than a static analyzer. A potential use case speculated by them is to create a centralized API for all functionally similar program regardless of their syntactic differences, which again can enhance the maintenance of a software system. Such use case may be hard to be achieved by pure static analysis: a static analyzer needs to understand the functionality of a program without executing it. According to the user study conducted by LaToza et al. [79], 42% of developers agreed that functionally similar code developed by different developers results in problems to maintain software systems, which is the highest among six types of similar code (duplications) reported by LaToza et al. We conjecture that this is because functionally similar code is the only type that is possibly to be syntactically different, while the other five types are syntactically similar at some levels.

In addition, LaToza et al. also pointed out the importance of program understanding/comprehension: 66% of developers thought that code (program) understanding is a serious problem, which is the highest percentage among all proposed problems during software development [79]. An interesting finding in [79] is that program understanding is not positively correlated to software design and development during daily lives of developers. We think that this is because most developers feel understanding programs (implemented by other developers) is difficult, they would rather deign and develop software by themselves without referring to existing useful code. While many static analysis tools have been devised to support developers understanding programs, we speculate that the lack of dynamic information of programs decreases the applicability of these static tools on programs. How to integrate static analysis with dynamic analysis can be a challenging direction for developing the next generation of program understanding tools.

From a hardware perspective, if we can use dynamic analysis to cluster programs based on their dynamic characteristics, we can increase the efficiency of hardware developers to design accelerators for specific programs in the same cluster. Demme and Sethumadhavan [34] propose to encode runtime behaviors of programs into dynamic data flow graphs, where each vertex is a basic block of instruction and each edge is a *real* data dependency between two basic blocks. They then cluster programs based on the similarity of their representative graphs. Based on their observation, the clustering results based on this dynamic graph feature of programs can explore the opportunity to optimize programs. For example, the programs in the same cluster based on their graph similarity can support hardware developers to design an accelerator for these programs, because they share the same dynamic features [33]. Our prior work HitoshiIO [123] and DyCLINK [121] share the similar concept of [60] and [34] to analyze and classify programs based on their dynamic behaviors. Our experiment result show that it is possible that dynamic features of programs can offer better precision to classify programs, which can potentially support some software engineering and security tasks, such as performance optimization and malware detection.

Another line of use cases can be malware detection: a malicious application can have different run-time behavior with what it claims to achieve for its users. For example, a malicious weather forecast application may send the private information of an user through SMS service, but it should only report weather conditions to the user. While some of malware can be detected by dynamic analysis, we speculate that some malicious programs such as logic bomb [48] that will only be triggered if some specific conditions are met can be hard to be identified by dynamic analysis. This is because, again, dynamic analysis requires meaningful workload to drive programs for exposing behaviors that researchers want to observe. It can be difficult to develop an input generator, which can create specific input to visit the malicious code in a program.

5.3 Learn from Programs

In addition to the approaches detecting similar code, we discuss several works relevant to learn program features, such as identifier names and API usage from codebases. Raychev et al. [111] develop a prediction engine, which first constructs a dependency network based on known and unknown properties of the program, and then learns a probability model. Such probability model can infer the unknown properties, such as identifier names or variable types of programs. Nguyen et al. [106] propose to learn the usage of APIs from code changes and then recommend API calls to developers. Gu et al. [53] applies a deep learning algorithm to learn the mappings between method comments and API sequences. The mapping can then be used to retrieve relevant API sequences based on developers' queries.

These approaches learn features of programs by machine learning or natural language processing models for augmenting their program analysis, which is relevant to our MACNETO system. Our MACNETO system integrates the techniques of topic modeling [21] and deep learning [119] with static call graph analysis of code to facilitate developers searching for programs. While we believe dynamic analysis can help developers classify or understand program by their behaviors better, dynamic analysis suffers from high runtime overhead to identify behaviorally similar code. By integrating machine learning models with static analysis, we can *approximate* program behavior in a more economic way. The insight we have for MACNETO is that by treating binaries of programs as documents and using the text mining model such as topic modeling, we are able to unveil and expose program semantics without truly executing them. Further, once we can birthmark each existing program binary via topic modeling, we can leverage the power of deep learning to train a program classifier. Given an unknown program, we speculate that MACNETO can support developers efficiently search for similar and relevant programs in the existing codebases, which facilitate them to understand this unknown program without diving into it.

5.4 A Comparison of Computing Similar Programs

What is the best method to compute similar programs is an open question: there can be various directions to compute program/code similarity. In this chapter, we have already discussed multiple directions to detect and identify similar programs including static analysis, dynamic analysis and machine learning. However, we have to admit that these directions may not be able to cover all similarities in code.

Our conclusion in this thesis is that computing code similarity should be *target-driven*: based on a specific purpose, a researcher can first define his objective to detect similar code, and then devises and applies corresponding algorithm(s). For example, if a research's purpose is to identify and locate which programs that are syntactically similar in a large codebase containing millions lines of code, a static code clone detector can offer great efficiency and effectiveness. On the other hand, if a researcher wants to understand and detect which execution patterns in run-time can be clustered together and can be processed by a specific hardware accelerator, a dynamic analyzer may be a better choice than a static analyzer, since static analyzer can at most approximate the run-time behavior of programs.

While, again, we admit that offering the best direction/strategy to compute similar programs/code is un-deterministic, here we provide a comparison of several important factors to compute similar programs.

- **Best Match vs. Preponderant Match:** It is possible that a program has multiple versions and/or executions, where we define them as *occurrences* of the program. When a similar code detector attempts to compute the similarity between two programs, there can be multiple comparisons because of difference occurrences of a single program. Then a problem can be raised: what will be the best method to select proxy from these multiple comparisons?

  Most of existing approaches, especially dynamic analyzers, [60, 123, 121], choose to use the best match between two programs as the proxy

$$Sim(P_i, P_j) = \max_{\substack{a \in \{occur(P_i)\} \\ b \in \{occur(P_j)\}}} (Sim(P_i^a, P_j^b)) \tag{5.1}$$

, where i and j are indices of programs, $occur()$ is the function that generates a set of occurrences of a program, a and b are indices of occurrences of P_i and P_j, respectively, and $Sim()$ is a similarity function computing the similarity between two occurrences of two programs. With this best match approach, a similar code detector can identify the maximum set of similar programs given a single program: two programs are deemed similar if one of their occurrences are similar. This approach is suitable for researchers who are interested in knowing which programs are *possibly* replaced by a centralized API, because their best similarity is sufficiently high. This approach can be a search engine to locate potential candidates in the codebase with the highest *recall*.

In addition to using the best match among all occurrences between two programs as the proxy, we can also take most occurrences of a pair of program into account to compute similarity, where we name it as preponderant match. Rather than locating every potential candidate in the codebase (good recall), some researchers may prefer to detect similar programs with good *precision*. In other words, they want to realize which programs are frequently similar: preponderant occurrences of these programs are similar. In this thesis, we attempt to reduce this preponderant match between all occurrences between two programs as an *Assignment Problem* [78]

$$\underset{\substack{a \in \{occur(P_i)\} \\ b \in \{occur(P_j)\}}}{\text{argmax}} \sum_a Sim(P_i^a, f : P_i^a \to P_j^b) \tag{5.2}$$

, where f is a bijection function that maps one occurrence of P_i^a of P_i to one occurrence of P_j^b of P_j for maximizing the similarity between all occurrences between P_i and P_j. In the future, we plan to use the Hungarian Algorithm [78] to solve this assignment problem between two programs.

- **Static vs. Dynamic:** In general, the advantage of static analysis is its performance, because it is not required to execute programs. The recent static analyzers [115, 67] can process millions lines of code in just few hours. However, since static analysis does not execute programs, some software engineering tasks that required run-time information of programs, such as clustering programs having similar functionality or behavior, can be difficult to achieve. Some work [43, 95] including our MACNETO system attempt to leverage machine learning/data mining techniques to approximate run-time information of programs. While dynamic analysis suffers from performance overhead because of instrumentation and run-time tracing, it is capable to analyze true run-time functionality and behavior of programs [122, 123, 121].

- **Normal Path vs. Erroneous Path:** Before we start our discussion, we first define *normal paths* in a program as the paths that will not throw exceptions during the program execution, while *erroneous paths* as the paths that will generate exceptions during the program execution. From the perspective of program analysis, in addition to analyzing normal paths in programs, some work focuses on analyzing code in erroneous paths [57, 129]. If a researcher is more interested in how programs handle errors/exceptions in a similar way than how programs actually function, he or she has to analyze the error handling portion instead of the functional portion in the program. This analysis can again be done either statically or dynamically. For static analysis, a researcher can extract code fragments in `catch` blocks or code fragments that result in exceptions to be thrown and then conduct further analysis, such as how programs usually handle different types of exceptions. Some languages, such as C, do not have `try-catch` block to conduct this static analysis, but some work has been proposed to identify error specifications [57, 129] to locate error handling code. For dynamic analysis, a researcher can monitor executions of programs and record erroneous paths once exceptions or runtime errors are thrown.

There is a fundamental problem for dynamic analysis, *Input Generation*: what will be the suitable input to drive the program for exposing paths/functions/behaviors that a researcher is interested? The test suites of a program can be a good resource to explore useful inputs to drive programs. In fact, some tools and systems have been developed to create test cases automatically [47, 107]. However, most of these approaches adopt some search-based methodologies, which attempt to maximize the code coverage of a program, which may not be directly applicable to drive a program, if a research only attempts to analyze normal or erroneous portions of the program. This is the reason that our HitoshiIO [123] and DyCLINK [121] systems exploit the existing workload to expose the true functionalities/behaviors of programs instead of using these test case generation tools. While the input generation problem is out of the scope of this thesis, we will discuss a *seed-based* approach to mitigate such problem in Section 5.5.

5.5 Future Work

In this chapter, we have discussed multiple methodologies to detect different types of similar programs, including syntactically similar, conceptually similar, functionally similar and behaviorally similar programs. While most existing approaches adopt static analysis to detect similar programs, we focus more on leveraging the power of dynamic analysis and machine learning to detect different types of similar programs. Our discussion in Section 5.4 addressed two problems of similar program detection

- Undecidability of similar program detection: It is difficult to design the best algorithm to detect similar programs, because different researchers may have different targets/purposes to identify different types of similar programs. For example, if a researcher attempts to locate all programs having the equivalent syntax, the programs having the same functionality is not necessary to be discovered. Thus, the design of similar program detector should be problem/target-driven.

- Pros. and cons. of different similar program detection: Take static analysis and dynamic analysis as the example. While static analysis is highly performant to process high volume of code, it is not suitable to identify dynamically similar programs. For dynamic analysis, even though it can capture more run-time information to detect dynamically similar programs with syntactic differences, it suffers from high execution overhead, which includes the original execution time and the tracing (instrumentation) time.

As we discussed in Section 5.4, dynamic analysis can suffer from the input generation problem. The dynamic analysis tools, HitoshiIO and DyCLINK, we developed in this thesis are no exceptions. One of the major problems to generate meaningful input or workload to drive problem is that it lacks well-defined interfaces of each program for the input generator to *understand* which values will be meaningful to generate. The rise of microservices architecture [103] can be an opportunity for solving the interface problem. Microservices are "Loosely coupled, service oriented with bounded contexts" according to Adrian Cockroft [103]. An important feature of microservices is that each microservice has well-defined API (interface) so that users and/or other services can invoke it conveniently. If we can take the existing microservices as the seeds and record their I/Os, we can then replay the workload of a microservice on other similar ones to conduct dynamic analysis. While this idea attempts to solve the input generation problem from a different angle with the existing work that generates input randomly [107], we look forward to developing this idea and evaluating its performance with the existing work.

In this chapter, we propose two applications. We attempt to integrate the pros. of different types of similar program detectors to achieve some tasks which may not be achieved by a single detector easily.

Recommendation of Error Handling Code: This idea is inspired by the existing work [57, 129] that attempts to locate bugs in code fragments that are responsible to handle errors in programs. Our target is different: we attempt to recommend or notify developers which types of error handler that they may need based on functions/methods they have already implemented. Before we discuss our approach, we define code in `try` blocks as *normal fragments*, code in `catch` block as *error handlers* and code in `finally` as *state cleaner* (which will clean the state of the current program). First, we can use a syntax analyzer to separate each function/method in each program in the codebase into three types of code fragments we just defined. For the next step we can apply either static or dynamic similar code detector to first cluster programs based on their normal fragments. In other word, we attempt to group functions/methods having similar syntax/functionality/behavior disregarding how they handle errors and clean program states. Then given a program cluster, we then conclude sub-clusters based on their error handlers. For each program cluster, we can then compute a conditional probability

$$P(Sub_{eh}^{j}|Cluster_{nf}^{i}) = \frac{\text{\# programs in the sub cluster j}}{\text{\# all programs in the cluster i}} \tag{5.3}$$

, where Sub_{eh}^{j} represents the j_{th} sub-cluster of error handlers and $Cluster_{nf}^{i}$ represents the i_{th} program cluster based on their normal fragments. Based on Eq. 5.3, we can then compute the conditional probability for each error handling sub-cluster for an unknown method that only have normal fragment without error handlers. We can finally sort each sub-cluster, which represents a type of error handler, and offer the developer a list of error handlers to follow.

Multiphase Similar Code Detection: In this thesis, while we explore multiple approaches/features to detect different types of similar code, we find that each approach/feature has its own pros. and cons. Is it possible the integrate the pros. of different similar code detector for offering developers more insight to understand their software systems or search for code they want? We propose to conduct *Multiphase detection*, which is a generic version of recommending error handling code. For example, if a developer is interested in analyzing the syntactic differences between functionally similar programs, he or she can first cluster programs by their functional similarity, and then construct sub-groups via computing syntactic similarity between programs in the same functional cluster. It is possible that we can then recommend/notify the developer that given the same functionality, there are other versions of code that is shorter (less lines of code) or has succinct syntax. It is also possible that the developer can then create a centralize APIs by first selecting the best version of code (where "best" here can be defined by the developer) in the functional cluster, and then replace all other versions of code in the same functional cluster by this best version. For automatically API replacement, we can consider to apply the existing systematic editing techniques [101, 74]. However, functionally similar programs can have very different syntaxes, but the existing approaches focus on automatically or semi-automatically editing programs that are syntactically similar. How to normalize these functionally similar but syntactically different programs can be one of our future research directions.

Chapter 6

Conclusions

Detecting similar code/program has been studied for years for supporting many software engineering tasks. While most existing work discusses how to detect programs having similar syntax (programs that *look* alike), we discuss another type of feature, program behavior, to cluster or classify programs (programs that *behave* alike) in this thesis. The definition of similarity between programs can be subjective and is nondeterministic. However, we believe that the approaches and features of code in this thesis offer a new perspective other than traditional syntactic similarity for researchers/developers to study and build applications based on behavioral similarity of code.

In this thesis, we propose three types of behavioral features to detect similar programs: functionality, execution and topic distributions in program binary. Given these behavioral features, we develop and open-source the systems/tools to facilitate researchers conducting further study. We first discuss the new technique, in-vivo similar code detection (Chapter 2), which leverages meaningful inputs to expose true functionalities of programs. Our system, HitoshiIO, uses an I/O based similarity model developed by us to detect functionally similar programs. In addition to using functionality as the proxy of program behavior, we propose using program execution as the proxy (Chapter 3). We build the system DyCLINK, which encodes program executions in dynamic instruction graphs, We call the programs with high execution (graph) similarity as code relatives. While HitoshiIO and DyCLINK both use dynamic analysis to expose program behaviors, expensive run-time overhead can be a program to process large volume of code (Big Code). However, static analysis may not be able to capture the true behavior of a program, which requires executing such program. Our insight here is to integrate static analysis with machine learning models to approximate program behaviors. We develop MACNETO (Chapter 4), which identify topic distributions of a program in its executable (machine topics) as its behavior proxy. Based on machine topics, we show that MACNETO can accurately deobfuscate programs after its syntax and/or lexical information are altered.

As we discussed repeatedly in this thesis, the definition of code similarity is subjective. Thus, we summarize existing approaches and our developments in this thesis and envision the future work and potential applications based on code similarity (Chapter 5). Given the extensive definitions of code similarity, I look forward to solving practical problems in software engineering and security.

Bibliography

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[3] Allatori. `http://www.allatori.com`.

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

[5] Amazon ec2. `http://aws.amazon.com/ec2/instance-types/`.

[6] Android build numbers. https://source.android.com/source/build-numbers.

[7] Sylvain Arlot and Alain Celisse. A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4:40–79, 2010.

[8] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2016.

[9] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014*, 2014.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

[11] Asm framework. `http://asm.ow2.org/index.html`.

[12] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *2015 International Conference on Software Engineering (ICSE)*, ICSE '15, pages 426–436, 2015.

[13] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 356–367, 2016.

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

[15] Arindam Banerjee, Srujana Merugu, Inderjit S. Dhillon, and Joydeep Ghosh. Clustering with bregman divergences. *J. Mach. Learn. Res.*, 6:1705–1749, December 2005.

[16] Veronika Bauer, Tobias Völke, and Elmar Jürgens. A novel approach to detect unintentional re-implementations. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 491–495, 2014.

[17] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, 1998.

[18] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, 2014.

[19] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 770–781, 2015.

[20] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical Deobfuscation of Android Applications. In *23rd ACM Conference on Computer and Communications Security*, CCS 2016, pages 343–355, 2016.

[21] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[22] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIG-*

SOFT International Symposium on Software Testing and Analysis, ISSTA '04, pages 195–205, 2004.

[23] Sergey Brin and Lawrence 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.

[24] Simon Butler, Michel Wermelinger, and Yijun Yu. Investigating naming convention adherence in java references. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 41–50, 2015.

[25] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26(1):50–57, 2009.

[26] Antonio Carzaniga, Andrea Mattavelli, and Mauro Pezzè. Measuring software redundancy. ICSE '15.

[27] William W. Cohen, Pradeep Ravikumar, and Stephen 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.

[28] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proffing, and obfuscation: Tools for software protection. *IEEE Trans. Softw. Eng.*, 28(8):735–746, August 2002.

[29] Christian S. Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29(6), October 2007.

[30] The java-util library. `https://github.com/jdereg/java-util/`.

[31] The website of deguard. `http://apk-deguard.com/`.

[32] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. Challenges of the dynamic detection of functionally similar code fragments. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 299–308, 2012.

[33] John Demme. *Overcoming the Intuition Wall: Measurement and Analysis in Computer Architecture.* PhD thesis, New York, NY, USA, 2014. AAI3611781.

[34] John Demme and Simha Sethumadhavan. Approximate graph clustering for program characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21, January 2012.

[35] Dex2jar. https://github.com/pxb1988/dex2jar.

[36] Nicholas DiGiuseppe and James 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.

[37] Dyclink github page. `https://github.com/Programming-Systems-Lab/dyclink`.

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

[39] Rochelle Elva and Gary T. Leavens. Semantic clone detection using method ioe-behavior. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 80–81, 2012.

[40] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, 2001.

[41] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*, February 2016.

[42] The f-droid repository. https://f-droid.org/.

[43] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, 2016.

[44] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, 2014.

[45] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017*, 2017.

[46] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10), October 1988.

[47] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, 2011.

[48] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. *2016 IEEE Symposium on Security and Privacy (SP)*, 00:377–396, 2016.

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

[50] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. Some from here, some from there: Cross-project code reuse in github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 291–301, 2017.

[51] Google code jam. `https://code.google.com/codejam`.

[52] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035, 2014.

[53] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[54] Christian Hammer and Gregor Snelting. An improved slicer for java. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 17–22, 2004.

[55] Hitoshiio github page. `https://github.com/Programming-Systems-Lab/ioclones`.

[56] Einar W Høst and Bjarte M Østvold. Debugging method names. In *European Conference on Object-Oriented Programming*, pages 294–317, 2009.

[57] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 345–362.

[58] Oracle jdk 7. `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`.

[59] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane 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.

[60] Lingxiao Jiang and Zhendong 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.

[61] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. Code similarities beyond copy & paste. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 78–87, 2010.

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

[63] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro 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.

[64] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium, NOMS 2012*, pages 1287–1294, 2012.

[65] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: Memory comparison-based clone detector. ICSE '11.

[66] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. ESEC/FSE-13, 2005.

[67] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *IEEE S&P*, pages 595–614, 2017.

[68] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[69] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[70] Raghavan Komondoor and Susan 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.

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

[72] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Giuliano Antoniol, and Yann-Gael Gueheneuc. Madmatch: Many-to-many approximate diagram matching for design comparison. *IEEE Transactions on Software Engineering*, 39(8):1090–1111, 2013.

[73] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–, 2001.

[74] Giri Panamoottil Krishnan and Nikolaos Tsantalis. Refactoring clones: An optimization problem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 360–363, 2013.

[75] D. E. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 489–490, 2013.

[76] Daniel E. Krutz and Wei Le. A code clone oracle. MSR '14.

[77] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, March 2007.

[78] H. W. Kuhn and Bryn Yaw. The hungarian method for the assignment problem. *Naval Res. Logist. Quart*, pages 83–97, 1955.

[79] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, 2006.

[80] Page Lawrence, Brin Sergey, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[81] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 3–12, 2006.

[82] Michael Levandowsky and David Winter. Distance between sets. *Sci. Comput. Program.*, 234:34–35, November 1971.

[83] Jingyue Li and Michael D. Ernst. Cbcd: Cloned buggy code detector. ICSE '12.

[84] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 318–329, 2016.

[85] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. Measuring code behavioral similarity for programming and software engineering education. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 501–510, 2016.

[86] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan 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.

[87] Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th annual psychology of programming workshop*, 2006.

[88] Mario Linares-Vásquez, Collin Mcmillan, Denys Poshyvanyk, and Mark Grechanik. On using machine learning to automatically classify software applications into domain categories. *Empirical Softw. Engg.*, 19(3):582–618, June 2014.

[89] Chao Liu, Chen Chen, Jiawei Han, and Philip 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.

[90] Douglas Low. Protecting java code via code obfuscation. *Crossroads*, 4(3):21–23, April 1998.

[91] Jonathan I. Maletic and Naveen 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.

[92] Mallet: Machine learning for language toolkit. http://mallet.cs.umass.edu/.

[93] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 107–114, 2001.

[94] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135, 2003.

[95] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017*, 2017.

[96] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[97] Apache maven. `https://maven.apache.org`.

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

[99] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, 2011.

[100] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. Semantic modelling of android malware for effective malware comprehension, detection, and classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 306–317, 2016.

[101] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 392–402, 2015.

[102] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. Quality assurance of software applications using the in vivo testing approach. ICST '09.

[103] State of the art in microservices. https://www.slideshare.net/adriancockcroft/dockercon-state-of-the-art-in-microservices.

[104] Mysql database. `https://www.mysql.com`.

[105] Lindsay Anne Neubauer. Kamino: Dynamic approach to semantic code clone detection. Technical Report CUCS-022-14, Department of Computer Science, Columiba University.

[106] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[107] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, 2007.

[108] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. NEZHA: Efficient Domain-independent Differential Testing. In *Proceedings of the 38th IEEE Symposium on Security & Privacy*, San Jose, CA, May 2017.

[109] Proguard. `http://proguard.sourceforge.net`.

[110] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2014.

[111] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, 2015.

[112] Kaspar Riesen, Xiaoyi Jiang, and Horst 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.

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

[114] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, 115, 2007.

[115] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal Roy, and Cristina Lopes. SourcererCC: Scaling Code Clone Detection to Big Code. ICSE '16.

[116] Trevor Savage, Bogdan Dit, Malcom Gethers, and Denys Poshyvanyk. Topicxp: Exploring topics in source code using latent dirichlet allocation. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–6, 2010.

[117] David Schuler, Valentin Dallmeier, and Christian Lindig. A dynamic birthmark for java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 274–283, 2007.

[118] Similarweb. `https://www.similarweb.com`. Accessed: 2017-08-08.

[119] Richard Socher, Milind Ganjoo, Christopher D Manning, and Andrew Ng. Zero-shot learning through cross-modal transfer. In *Advances in neural information processing systems*, pages 935–943, 2013.

[120] Kathryn T. Stolee, Sebastian Elbaum, and Matthew B. Dwyer. Code search with input/output queries. *J. Syst. Softw.*, 116(C), June 2016.

[121] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[122] Fang-Hsiang Su, Jonathan Bell, and Gail Kaiser. Challenges in behavioral code clone detection. In *Proceedings of the 10th International Workshop on Software Clones*, IWSC 2016, 2016.

[123] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. Identifying functionally similar code in complex codebases. In *Proceedings of the 24th IEEE International Conference on Program Comprehension*, ICPC 2016, 2016.

[124] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a Big Data Curated Benchmark of Inter-project Code Clones. ICSME '14.

[125] Symbolic execution for finding bugs. https://www.cs.umd.edu/ mwh/se-tutorial/symbolic-exec.pdf.

[126] Armstrong A Takang, Penny A Grubb, and Robert D Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[127] Haruaki Tamada, Masahide Nakamura, and Akito Monden. Design and evaluation of birthmarks for detecting theft of java programs. In *Proc. IASTED International Conference on Software Engineering*, pages 569–575, 2004.

[128] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. ESEC/FSE '09.

[129] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 752–762, 2017.

[130] The xstream library. `http://x-stream.github.io/`.

[131] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William 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.

[132] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, 2004.