Identifying Functionally Similar Code in Complex Codebases

Fang-Hsiang Su, Jonathan Bell, Gail Kaiser and Simha Sethumadhavan
Columbia University
500 West 120th St, MC 0401
New York, NY USA
{mikefhsu, jbell, kaiser, simha}@cs.columbia.edu

ABSTRACT

Identifying similar code in software systems can assist many software engineering tasks, including program understanding. While most approaches focus on identifying code that *looks alike*, some researchers propose to detect instead code that *functions alike*, which are known as functional clones. However, previous work has raised the technical challenges to detect these functional clones in object oriented languages such as Java. We propose a novel technique, *In-Vivo Clone Detection*, a language-agnostic technique that detects functional clones in arbitrary programs by observing and mining inputs and outputs. We implemented this technique targeting programs that run on the JVM, creating HitoshiIO (available freely on GitHub), a tool to detect functional code clones. Our experimental results show that it is powerful in detecting these functional clones, finding 185 methods that are functionally similar across a corpus of 118 projects, even when there are only very few inputs available. In a random sample of the detected clones, HitoshiIO achieves 68+% true positive rate, while the false positive rate is only 15%.

CCS Concepts

•**Software and its engineering** → **Dynamic analysis; Object oriented languages; Software maintenance tools;** *Software usability;* Patterns;

Keywords

I/O behavior; dynamic analysis; code clone detection; data flow analysis; patterns

1. INTRODUCTION

When developing and maintaining code, software engineers are often forced to examine code fragments to judge their functionality. Many studies [16, 18, 23] have suggested large portions of modern codebases can be *clones* — for instance, code that is copied-and-pasted from one part of a program to another. These clones can complicate maintenance (for instance, when a bug is copied), but intuitively, may also pose interesting implications for programmer understanding. While traditional code clones refer to syntactic clones (code fragments that look alike), we are interested in *functional clones* (code fragments that have similar function, but may not look alike).

These functional clones could be used to help developers understand what a code fragment does, in the context of other code fragments do (which may look different, and hence be easier to understand). Similarly, once they are identified, we may be interested in extracting functional clones into a common API.

Unfortunately, detecting true functional clones is very tricky: static approaches must be able to fully reason about code's functionality (undecidable in the general case), and dynamic approaches must be able to observe code under sufficient inputs to expose sufficiently diverse program executions to make judgments about general function. Hence, previous approaches towards detecting functional clones [15, 12, 17, 11] have focused on profiling running code, matching different code fragments that show the same outputs for the same inputs. To expose these functional clones, the clone detector executes each target code fragment with a randomly generated input, repeating that same input for different fragments as well, observing when outputs are the same. Hence, previous work towards detecting functional clones has 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, then, is not feasible to scale to complex, object oriented codebases, where it can be incredibly challenging to execute individual methods or code fragments in isolation (and with generated inputs), due to the complexity of generating sufficient drivers for the code to get it to run. Previous works towards detecting functional clones in Java programs [12, 10] have therefore, seen unsatisfactory results: a recent study by Deissenboeck et al. reported that across five Java projects only 28% of the target methods could be executed with this standard input generation approach [10]. 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 could never be declared exactly equivalent [10].

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 candi-

ACMXXX '16 USA

DOI: 10.475/123_4

date clones in isolation, hence allowing it to work in complex, object oriented codebases. Our key insight is that most large and complex codebases include test cases [7], which can be used to feed inputs to the application as a whole. Then, we identify the inputs and outputs of each candidate for code clone matching from these executions, and then cluster similar code based on these profiles. Moreover, our approach allows for a relaxed similarity comparison, enabling efficient detection of code that has very similar inputs or outputs, even when the exact structure of those variables differs. We created HitoshiIO, which implements this approach for the JVM (targeting languages such as Java), and evaluated it on 118 projects, finding 874 functional clones, using only the applications' existing test suites as inputs, even when only using a small subset of those inputs.

HitoshiIO considers every method in a project as a potential functional clone of every other method, recording observed inputs (be they method parameters, or global state variables read by a method) and observed outputs (externally observable writes, including return values and heap variables). The primary contributions of this paper 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 and will be available under an MIT license after the acceptance of the paper on GitHub[1]

2. RELATED WORK

Identifying similar or duplicate code (code clones) can enhance the maintainability of the software system. 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 high level categories: syntactic clone detection, and functional clone detection.

Syntactic clones: Roy et al. [29] conducted a survey regarding the 4 types of code clones and the corresponding techniques to detect them (ranging from those that are exact copy-paste clones to those that are more semantically similar, despite syntactic differences). Most of the approaches to detect syntactic clone involve static analysis, which aims at identifying the code that *looks alike*. In general, these syntactic approaches first parse programs into a type of intermediate representation statically 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 [5, 16, 23], AST-based [6, 14] and graph-based [13, 25, 19, 22]. Among these 3 classes, 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. [29]. Another line of clone detection involves creating fingerprints of code, for instance by tracking API usage [26, 9], to identify clones.

Compared with these approaches that find *look alike* code, HitoshiIO searches for *functionally alike* code. To search for such functionally similar code, HitoshiIO applies the techniques of dynamic analysis.

Functional clones: Our approach is most relevant to previous work in detecting code that is *functionally similar*, despite syntactic differences. For instance, Elva & Leavens proposed detecting functional clones by comparing the outputs of methods that have the exact same outputs and inputs [12]. The MeCC system summarizes the abstract state of a program after each method is executed (relating that state to the method's inputs), allowing for exact matching of outputs [17]. Our approach differs from both of these in that we allow for matching functionally equivalent methods, even when there are minor differences in the formats of inputs and outputs.

Carzaniga et al. studied different measures of redundancy, and how to quantify how functionally redundant two code fragments might be, identifying both the code statements executed and data operations performed [8]. Our notion of functionally similar code is very similar to their notion of redundant code, although we put significantly more weight on comparing inputs and output values, rather than just the sequence of inputs and outputs (we consider *all* values, even those of complex variables, while Carzaniga et al. only consider Java's basic types).

Jiang and Su's EQMiner [15] and Deissenboeck et al.'s comparable system for Java [10] 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 3 of the technical challenges reported by Deissenboeck et al. and our direct solutions:

- *How to appropriately capture I/Os of programs*: Compared with the existing approaches that pre-define which variables can be inputs or outputs of the program, In-Vivo Clone Detection applies data flow analysis to capture which input sources contribute to output sinks at instruction level.

- *How to generate meaningful inputs to drive programs*: Deissenboeck et al. reported between $20\% - 65\%$ of methods that they cannot generate inputs. One 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, which uses real workloads to drive programs (inspired by our prior work in runtime testing [27]).

- *How to compare project-specific types of objects between different applications*: We will elaborate the similar issue further in §4.5: 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.

[1] https://github.com/Programming-Systems-Lab/ioclones

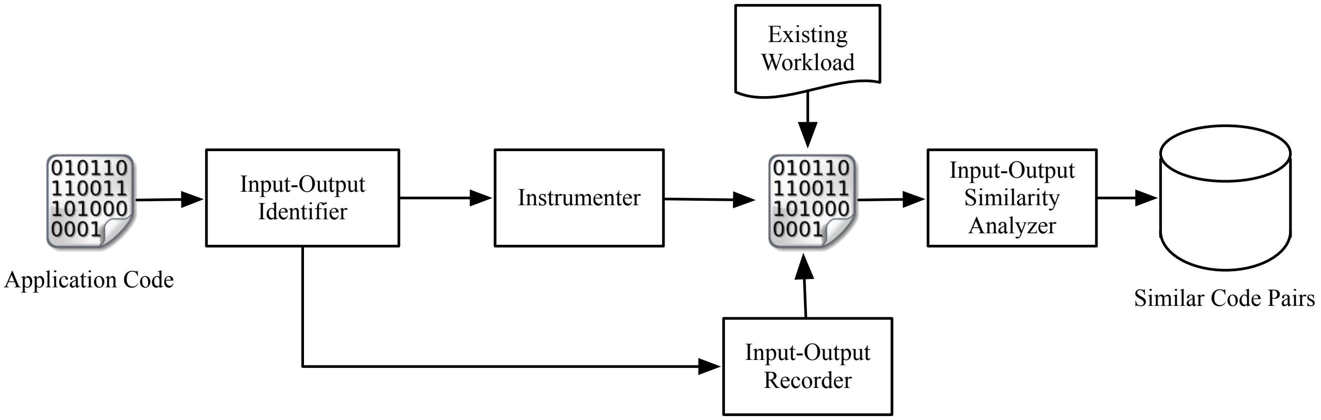

Figure 1: **High level overview of In-Vivo Clone Detection**. First, individual inputs and outputs of methods are identified, then the application is transformed so that its inputs and outputs can be easily recorded while it is executed under an existing workload. Finally, these recorded inputs and outputs are analyzed to detect functionally similar methods.

3. DETECTING CLONES IN-VIVO

At a high level, our approach detects code which appears functionally similar, by observing that for the similar inputs, two different methods produce the similar output (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 tests), rather than relying on targeted input generation techniques. Figure 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, then we instrument the application so that when running it with existing workloads, we can record the individual inputs and outputs to each method, for use in an offline similarity analysis.

3.1 The Input Generation Problem

Previous approaches towards detecting functional clones in programs rely on input generation: where tools generate (randomly, or systematically) inputs to individual methods or code fragments, allowing a tool to observe that on the same generated input, the output is the same. 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 inputs, each of which may have an immense range of potential values. For instance, Randoop [28] 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), guided only by the knowledge of which previous sequences failed to generate a 'valid' state, making it difficult to use in many cases [30]. Many other techniques have been developed to automatically generate inputs for individual methods, but the problem remains unsolved in the general case. For instance, in their 2012 study of input generation for clone detection, Deissenboeck et al. found that input generation and execution failed for approximately 28% of the methods that they examined across five projects.

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 the method and 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.

Lemma 1. *An input source 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.*

A output of a method is the computational result of this method that a developer wants to use. As Jiang and Su stated [15], 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, *OSinks*, for a method in a conservative way:

Lemma 2. *An output sink 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 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 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.

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

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: A simple code example in Java with input source and output sinks identified.

The ownership 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. Take `a.b.c = 5` as the example, where `a` and `b` are objects and `c` is an integer. In this example, the c field owned by the `b` object is written. Because the `a` object owns `b`, our approach will know that the `a` object has been written by the method, even though this write does not happen directly on `a`. This dependency helps identify which values can be from inputs.

3.3 Example

To demonstrate our general approach, we use the method `addRelative` in Figure 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 output sinks to identify the input sources, so we first define the output sinks 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 1. The **Deps.** column records the dependency between two variables, and the **Notes** column explains why these two variables have the dependency. We only show the direct dependencies between variables.

Table 1: The data dependencies in the `addRelative` method.

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

Table 2: The input sources in the `addRelative` method.

Var.	**Notes**
`me`	`me` has the computational dependency to the output sink `ret`.
`rName`	`rName` is written to the output sink `me`.
`rAge`	`rAge` is written to the output sink `me`.
`pos`	`pos` has the computational dependency to the output sink `me`.

Finally, we can define the input sources based on the out sinks and the dependencies between variables. A input source is the one that have direct or transitive dependencies to any of the output sinks. We first define the candidate input sources in `addRelative` as

$$ISource_c(\texttt{addRelative}) = \{\texttt{me}, \texttt{rName}, \texttt{rAge}, \texttt{pos}, \texttt{useless}\} \quad (1)$$

Given 3 output sinks and all dependencies in Table 1, we can infer the parents of these two output sinks as

$$Parents(\{\texttt{ret}, \texttt{me}, \texttt{pos}\}) = \{\texttt{me}, \texttt{rName}, \texttt{rAge}, \texttt{pos}\} \quad (2)$$

We then intersect these two sets and conclude the input sources of `addRelative` in Table 2. We can see that not all input parameters are considered as input sources. The variable `useless` contributes to no output sinks, 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, `pos` serve 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 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 and output variable when a method is called, creating an *I/O record* for each method. Over the course of the program execution, many unique I/O records will likely be collected for each method.

3.4 Mining Functionally Equivalent Methods

```
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;
}
```

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

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

Figure 3: A functional clone detected by HitoshiIO

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, which are cheaply compared, and then use these summaries to perform high-level similarity matching. 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 implementing data structures to represent the same result.

Consider the two code listings shown in Figure 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 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 define the notations here.

- m_i: The i_{th} method in the codebase.
- $inv_{r|m_i}$: The r_{th} invocation of m_i.
- $ISet(inv_{r|m_i})$: the input set of $inv_{r|m_i}$.
- $OSet(inv_{r|m_i})$: the output set of $inv_{r|m_i}$.
- $ISet_{dh}(inv_{r|m_i})$: the deep hash set of $ISet(inv_{r|m_i})$.
- $OSet_{dh}(inv_{r|m_i})$: the deep hash set of $OSet(inv_{r|m_i})$.
- 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 $ISet$s and $OSet$s as the basic components for the functional similarity. The definition for the Jaccard similarity [21] is as follows:

$$J(Set_i, Set_j) = \frac{Set_i \cap Set_j}{Set_i \cup Set_j} \tag{3}$$

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

$$Sim_I(IP_{r|i,s|j}) = J(ISet_{dh}(inv_{r|m_i}), ISet_{dh}(inv_{s|m_j})) \tag{4a}$$

$$Sim_O(IP_{r|i,s|j}) = J(OSet_{dh}(inv_{r|m_i}), OSet_{dh}(inv_{s|m_j})) \tag{4b}$$

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

$$Sim(IP_{r|i,s|j}) = \frac{(1-\beta * e^{Sim_I}) * (1-\beta * e^{Sim_O})}{(1-\beta * e)^2} \tag{5}$$

, where β is a constant. This exponential model punishes the invocation pairs that have either similar $ISet$ or $OSet$, 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 $IP_{r|i,s|j}$.

$$Sim(MP_{ij}) = \max Sim(IP_{r|i,s|j}) \tag{6}$$

4. HitoshiIO

To demonstrate and evaluate in-vivo clone detection, we create HitoshiIO, with a name inspired by the Japanese word for “equivalent”: *hitoshi*. 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, and will be published on GitHub with the acceptance of the paper.

4.1 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 [24], there are two categories of data types: *primitive* and *reference* types. The primitive category includes 8 data types: boolean, byte, character, integer, short, long, float and double. The reference category includes 2 data types: objects and arrays. Objects are instances of classes, which can have fields [24]. 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. HitoshiIO considers all types of variables as inputs and outputs.

4.2 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, then, is identifying the outputs of a given method. For a method m, its output set O consists of all variables that are written by m that are observable outside of m. An output, then, 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 §5.

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 $OSinks$, creating the set of variables $Parents(OSinks)$. Variable v_i in method m is an input if it is $Parents(OSinks)$ 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.

4.3 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 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.

4.4 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 call 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 [10] to adopt the XStream library [4] 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 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`	Cont.	Jump to a branch based on the index on the stack.

4.5 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 behave "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 §3.4) - 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 [10]. To solve this problem, we adopt the *DeepHash* [3] approach, creating a hash of each object. 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 similarity model of HitoshiIO follows §3.4. For 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. After multiple small scale experiments done by us, we set β to 3 for the exponential model of Eq. 6 in HitoshiIO. 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 §5.1.

5. EXPERIMENTS

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:

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

RQ2: Is the false-positive rate of HitoshiIO low enough to be 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 [2], which provides input data, as the codebase of our experiments. The Google Code Jam is the annual online 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 examined 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 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 business logic.

Table 4: A summary of the experimental codebase containing projects from the Google Code Jam competition.

		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

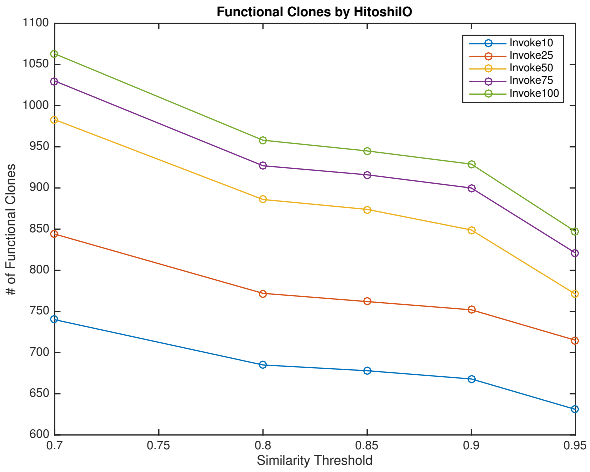

Figure 4: The number of functional clones detected by HitoshiIO with different parameter settings.

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.

We performed all of our similarity computations on Amazon's EC2 infrastructure [1], using a single c4.8xlarge machine, equipped with 36 cores and 60GB of memory.

5.1 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 *ISets* and *OSets*. HitoshiIO stops generating I/O records for a method, when

Table 5: The distributions of clones detected by HitoshiIO cross the problem sets.

Year Pair	Number of Clones			Analysis Time (mins)
	≤ 5 LOC	> 5 LOC	Total	
2011 − 2011	20	14	34	1.17
2012 − 2012	100	32	132	0.9
2013 − 2013	18	144	162	0.8
2014 − 2014	41	65	106	0.9
2011 − 2012	25	26	51	1.9
2011 − 2013	16	24	40	1.8
2011 − 2014	36	40	76	2.2
2012 − 2013	29	30	59	1.7
2012 − 2014	59	61	120	1.5
2013 − 2014	41	53	94	1.6
Total	385	489	874	14.5

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 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 72. Figure 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, therefore, slightly larger on average than most methods in the dataset). Table 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 five 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%).

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 & Leavens' IOE clone detector, were unavailable, despite repeated contacts to the authors [12]. Deissenboeck et al.'s Java system [10], 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.

5.2 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 were given a high level overview of the problem, and were requested for each pair of clones, to report if they truly appeared to be functional clones. 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

$$precision = \frac{\#TP}{\#FP + \#TP} \tag{7}$$

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

Our developer-guided precision evaluation is difficult to compare to previous functional clone works (e.g. [10, 15, 12], 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 used in practice to find functionally similar code.

5.3 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 all of the results, actually identifying 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 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 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.

Table 6: The precision of sampling functional clones detected by HitoshiIO.

	Clones Detected		Clones	Analysis
InvT	Total	Clone Rate	Missed	Time (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

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 used in practice, and (2) it shows that even with very few observed executions (e.g. due to sparse pre-existing workloads).

5.4 System Limitations and Future Work

For the next step of HitoshiIO, we have the following directions. Currently, HitoshiIO can detect the potentially direct or indirect write to the input object from the caller method, even though this write occurs in its callee. However, if the caller method passes a value owned by the input object to its callee, the write to this value in the callee will not be reflected on its owner object in the caller. Enhancing such limitation is our highest priority. We plan to develop a module for HitoshiIO to detect inputs and outputs from external environment. All of these enhancements at the system level can help HitoshiIO identify more input sources and output sinks that are relevant to the computation of the program. In addition to the system level enhancements, we are also interested in adopting Machine Learning techniques to refine and optimize the similarity computation mechanism of HitoshiIO.

6. THREATS TO VALIDITY

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 codebases.

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, crowdsourcing an evaluation of the precision of HitoshiIO might be an interesting way to increase our confidence even further.

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. [20]) 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 output is are not sound (§4.2), and hence, may result in identifying fewer outputs than it ought to. However, these limitations do not effect the validity of our experimental results, as any implementation flaws would hence be reflected in the results.

7. CONCLUSIONS

The difficulty of detecting functionally similar code has been proved by the prior work. How to generate meaningful inputs to driver problems and how to appropriately define I/Os for programs are two major issues. In this paper, we present the HitoshiIO system, which implements our idea of In-Vivo Clone Detection. Instead of fixing the definitions of program I/Os, HitoshiIO applies the technique of data flow analysis to identify potential input sources and output sinks. By using the real workloads to drive programs, HitoshiIO collects I/O values from programs. Based on the similarity model developed by us, HitoshiIO detects 800+ functionally similar clones in the projects of Google Code Jam. The true positive rate of HitoshiIO is more than 68%, while the false positive rate is only 15%. The experimental results have been verified by two student developers who are not authors, in a random sample of the detected functional clones.

8. ACKNOWLEDGMENTS

The authors would like to thank Apoorv Prakash Patwardhan and Varun Jagdish Shetty for verifying the experimental results. Su, Bell and Kaiser are members of the Programming Systems Laboratory. Sethumadhavan is the member of the Computer Architecture and Security Technology Laboratory. This work is funded by NSF CCF-1302269, CCF-1161079 and NSF CNS-0905246.

9. REFERENCES

[1] Amazon ec2. https://aws.amazon.com/ec2/. Accessed: 2016-02-16.

[2] Google code jam. https://code.google.com/codejam. Accessed: 2016-02-15.

[3] The java-util library. https://github.com/jdereg/java-util/. Accessed: 2016-02-14.

[4] The xstream library. http://x-stream.github.io/. Accessed: 2016-02-14.

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

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

[7] J. Bell, G. Kaiser, E. Melski, and M. 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.

[8] A. Carzaniga, A. Mattavelli, and M. Pezzè. Measuring software redundancy. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 156–166, Piscataway, NJ, USA, 2015. IEEE Press.

[9] C. S. Collberg, C. Thomborson, and G. M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29(6), Oct. 2007.

[10] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. Challenges of the Dynamic Detection of Functionally Similar Code Fragments. In *16th European Conference on Software Maintenance and Reengineering*, pages 299–308, 2012.

[11] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 303–317, Berkeley, CA, USA, 2014. USENIX Association.

[12] R. Elva and G. T. Leavens. Semantic clone detection using method ioe-behavior. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 80–81, Piscataway, NJ, USA, 2012. IEEE Press.

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

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

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

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

[17] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[18] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

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

[20] D. E. Krutz and W. Le. A code clone oracle. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 388–391, New York, NY, USA, 2014. ACM.

[21] M. Levandowsky and D. Winter. Distance between sets. *Sci. Comput. Program.*, 234:34–35, Nov. 1971.

[22] J. Li and M. D. Ernst. Cbcd: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 310–320, Piscataway, NJ, USA, 2012. IEEE Press.

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

[24] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification*, Java SE 7 edition, Feb 2013.

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

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

[27] C. Murphy, G. Kaiser, I. Vo, and M. Chu. Quality assurance of software applications using the in vivo testing approach. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 111–120, Washington, DC, USA, 2009. IEEE Computer Society.

[28] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, 2007.

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

[30] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 193–202, 2009.