Identifying Functionally Similar Code in Complex Codebases

Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, Simha Sethumadhavan
Columbia University
New York, NY USA
{mikefhsu, jbell, kaiser, simha}@cs.columbia.edu

***Abstract*—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.**

***Index Terms*—I/O behavior; dynamic analysis; code clone detection; data flow analysis; patterns**

I. Introduction

When developing and maintaining code, software engineers are often forced to examine code fragments to judge their functionality. Many studies [1]–[3] 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 [4]–[7]. 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 [5], [8], [9] 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 [8]. 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 [8].

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 [10], 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. HitoshiIO is available under an MIT license on GitHub [1].

II. 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.* [11] 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 [1], [3], [12], AST-based [13], [14] and graph-based [15]–[18]. 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.* [11]. 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* [19] 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 [20], [21], 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 [5]. 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 [6]. 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.

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

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 [22]. 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.* only consider Java's basic types.

Jiang and Su's EQMiner [4] and the comparable system developed by Deissenboeck *et al.* for Java [8] 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 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 [23].
- *How to compare project-specific types of objects between different applications*: We will elaborate the similar issue further in §IV-E: 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.

III. Detecting Clones 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 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,

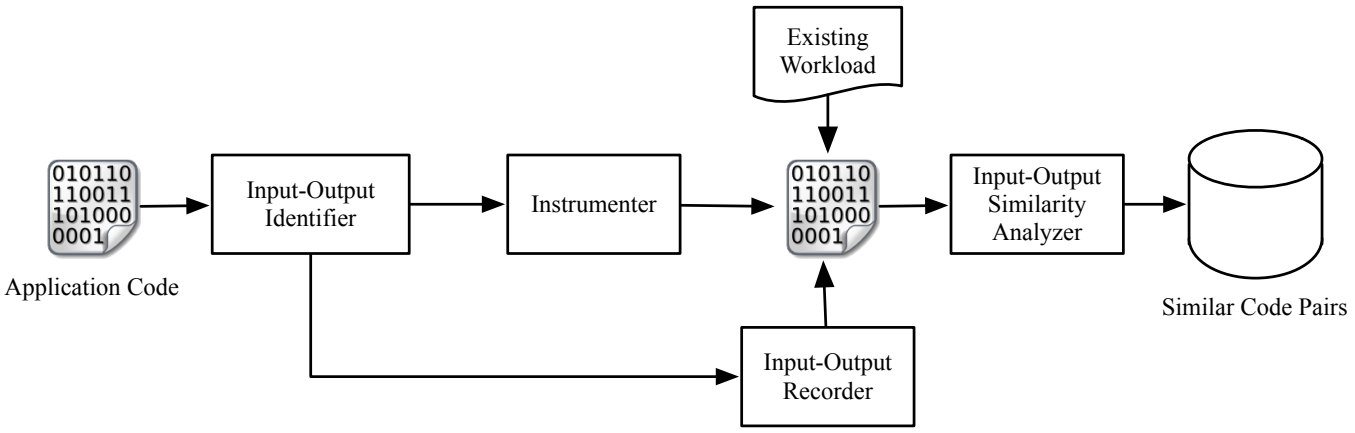

Fig. 1: **High level overview of In-Vivo Clone Detection**. First, individual inputs and outputs of methods are identified, then the application is transformed so that its inputs and outputs can be easily recorded while it is executed under an existing workload. Finally, these recorded inputs and outputs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

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.

A. 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 [24] 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 [25]. 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.

B. 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 1: 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 [4], 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 2: 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 [26] 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* [26]) 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`.

```
1 public class Person {
2   public String name;
3   public int age;
4   public Person[] relatives;
5 }
6
7 public static int addRelative(Person me, //input
8   String rName, int rAge, int pos,
9   double useless) {
10
11   Person newRel = new Person();
12   newRel.name = rName;
13   newRel.age = rAge;
14
15   if (pos > 0) {
16     insert(me, newRel, pos);
17   }
18   int ret = sum(me.relatives);
19
20   double k = useless + 1;
21
22   System.out.println(pos); //output
23   return ret; //output
24 }
25
26 public static void insert(Person me, Person rel, int pos
       ) {
27   me.relatives[pos] = rel;
28 }
29
30 public static int sum(Person[] relatives) {
31   int sum = 0;
32   for (Person p: relatives) {
33     sum += p.age;
34   }
35   return sum;
36 }
```

Fig. 2: A code example with inputs and outputs identified.

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.

C. 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 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 I. 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 I: 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 II: 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`.

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

$$\begin{aligned} &ISrc_c(\texttt{addRelative}) \\ &= \{\texttt{me}, \texttt{rName}, \texttt{rAge}, \texttt{pos}, \texttt{useless}\} \end{aligned} \tag{1}$$

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

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

We then intersect these two sets and conclude the input sources of `addRelative` in Table II. 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, `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* [26]) 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 program execution, many unique I/O records will likely be collected for each method.

```
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];
}
```

Fig. 3: A functional clone detected by HitoshiIO.

D. 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 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 first discuss the concept of *DeepHash* [27] 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 [28] 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 $ISrc$s and $OSink$s as follows.

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

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

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

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

, 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 $IP_{r|i,s|j}$.

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

IV. 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, 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.

A. 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 [29], 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 [29]. 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.

B. 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 §V.

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.

C. 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 III 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.

TABLE III: 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.

D. 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 [8] to adopt the XStream library [30] 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.

E. 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 §III-D) - 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 [8]. To solve this problem, we adopt the *DeepHash* [27] approach, creating a hash of each object. The details of *DeepHash* can refer to §III-D.

The similarity model of HitoshiIO follows §III-D. 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. 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 §V-A.

V. 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 potentially usable by developers?

TABLE IV: 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

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 [31], 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 IV. 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.

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 [32], using a single c4.8xlarge machine, equipped with 36 cores and 60GB of memory.

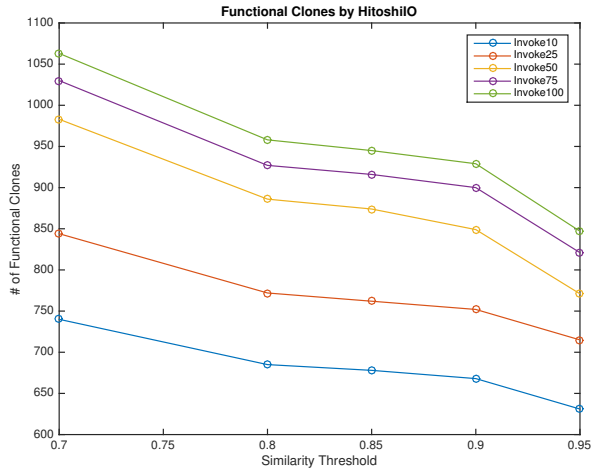

Fig. 4: The number of functional clones detected by HitoshiIO with different parameter settings.

A. 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 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 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 V 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%).

TABLE V: The distributions of clones detected by HitoshiIO cross the problem sets.

Year Pair	Number of Clones			Methods Compared	Analysis Time (mins)
	≤ 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

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 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 [5]. Deissenboeck et al.'s Java system [8], 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.

B. 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

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

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.,* [4], [5], [8]), 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.

C. 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 V, 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 VI. 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 VI: The number of missed functional clones with different invocation thresholds detected by HitoshiIO.

	Clones Detected		Clones Missed	Analysis Time (mins)
$InvT$	Total	Clone Rate		
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

VI. Discussion

A. 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 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.,* [33], [34]) 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 (§IV-B), 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.

B. Future Work

To offer a more advanced mechanism to identify I/Os of programs, we plan to apply a taint tracking system such as [35] 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.

VII. Conclusions

Prior work has underscored the challenges of detecting functionally similar code in object oriented languages. In this paper, 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.

VIII. Acknowledgments

The authors would like to thank Apoorv Prakash Patwardhan and Varun Jagdish Shetty for verifying the experimental results. The authors would also like to thank the anonymous reviewers for their valuable feedback. 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.

References

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *TSE*, Jul. 2002.

[2] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," ser. ESEC/FSE-13, 2005.

[3] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," ser. OSDI'04.

[4] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," ser. ISSTA '09.

[5] R. Elva and G. T. Leavens, "Semantic clone detection using method ioe-behavior," ser. IWSC '12, pp. 80–81.

[6] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: Memory comparison-based clone detector," ser. ICSE '11.

[7] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proc of the 23rd USENIX Conference on Security Symposium*, 2014.

[8] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, "Challenges of the Dynamic Detection of Functionally Similar Code Fragments," ser. CSMR '12.

[9] L. A. Neubauer, "Kamino: Dynamic approach to semantic code clone detection," Department of Computer Science, Columiba University, Tech. Rep. CUCS-022-14.

[10] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," ser. ESEC/FSE 2015.

[11] 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.*, vol. 74, no. 7, pp. 470–495, May 2009.

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

[13] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," ser. ICSM '98.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," ser. ICSE '07.

[15] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," ser. ICSE '08.

[16] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: Detection of software plagiarism by program dependence graph analysis," ser. KDD '06.

[17] J. Krinke, "Identifying similar code with program dependence graphs," ser. WCRE '01.

[18] J. Li and M. D. Ernst, "Cbcd: Cloned buggy code detector," ser. ICSE '12.

[19] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," ser. ASE '01.

[20] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," ser. ICSE '12.

[21] C. S. Collberg, C. Thomborson, and G. M. Townsend, "Dynamic graph-based software fingerprinting," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 6, Oct. 2007.

[22] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," ser. ICSE '15.

[23] C. Murphy, G. Kaiser, I. Vo, and M. Chu, "Quality assurance of software applications using the in vivo testing approach," ser. ICST '09.

[24] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," ser. OOPSLA '07.

[25] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," ser. ESEC/FSE '09.

[26] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, Oct. 1988.

[27] The java-util library. [Online]. Available: https://github.com/jdereg/java-util/

[28] M. Levandowsky and D. Winter, "Distance between sets," *Sci. Comput. Program.*, vol. 234, pp. 34–35, Nov. 1971.

[29] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification*, Java SE 7 ed., Feb 2013. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html

[30] The xstream library. [Online]. Available: http://x-stream.github.io/

[31] Google code jam. [Online]. Available: https://code.google.com/codejam

[32] Amazon ec2. [Online]. Available: https://aws.amazon.com/ec2/

[33] D. E. Krutz and W. Le, "A code clone oracle," ser. MSR '14.

[34] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a Big Data Curated Benchmark of Inter-project Code Clones," ser. ICSME '14.

[35] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," ser. OOPSLA '14.