JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2016; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Discovering Functionally Similar Code with Taint Analysis

Fang-Hsiang Su*, Jonathan Bell, Gail Kaiser, Simha Sethumadhavan

Columbia University, Computer Science Department, 500 West 120 Street, New York, New York 10027.

SUMMARY

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, in general cases undecidable. We propose a novel technique, *In-Vivo Clone Detection*, which 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. Further, to identify inputs and outputs of programs appropriately, we use the techniques of static and dynamic data flow analysis. These enhancements mitigate the problems in object oriented languages with respect to identifying program I/Os as reported by prior work. We implement such techniques in our system, HitoshiIO, which is open source and freely available. Our experimental results show that HitoshiIO detects ~ 900 and $\sim 2,000$ functional clones by static and dynamic data flow analysis, respectively, across a corpus of 118 projects. In a random sample of the detected clones by the static data flow analysis, HitoshiIO achieves 68+% true positive rate with only 15% false positive rate. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: I/O behavior; dynamic analysis; code clone detection; data flow analysis

1. INTRODUCTION

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 or from an entire 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 may not help find copied bugs, but can bring

*Correspondence to: Columbia University, Computer Science Department, 500 West 120 Street, New York, New York 10027.
†Please ensure that you use the most up to date class file, available from the SMR Home Page at
http://onlinelibrary.wiley.com/journal/10.1002/(ISSN)2047-7481

many other 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 application level 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 observable inputs that can be method parameters or global state variables read by a method, and outputs that are externally observable after the execution of the method including return values and heap variables. Our HitoshiIO (version 1) [11] used static analysis to identify program I/Os. In this paper, HitoshiIO (version 2) adopts Phosphor [12,13], which provides dynamic taint analysis to identify program I/Os.

Our experimental results show that HitoshiIO effectively detects functional clones in complex codebases. Further, we also observe that with dynamic data flow analysis, HitoshiIO detects 2x more functional clones than the static data flow analysis. We evaluated HitoshiIO on 118 projects, finding $\sim$ 900 and $\sim$ 2000 functional clones by static and dynamic data flow analysis, respectively, using only the applications' existing workloads as inputs. Given that no ground truth and/or standard

DOI: 10.1002/smr

is available to judge the validity of these functional clones, how to verify the differences between the functional clones detected by both static and dynamic data flow analysis is a open question. Thus, we open source HitoshiIO under an MIT license on GitHub * for enabling future developments and further analysis in the community.

2. RELATED WORK

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

Static techniques: Roy *et al.* [14] 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, 15], AST-based [16, 17] and graph-based [18–21]. 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.* [14]. 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* [22] 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 [23, 24], 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.

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 [25]. 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

*https://github.com/Programming-Systems-Lab/ioclones

DOI: 10.1002/smr

inputs and outputs. We consider *all* data types, even complex variables, while Carzaniga *et al.* only consider Java's basic types.

Dynamic taint analysis is a technique to detect which data sinks are affected by (dependent on) which data sources. While it is widely used in solving security and privacy problems, such as the leak of private data [26], we use this technique to detect inputs and outputs of methods. We integrate HitoshiIO with a portable taint analysis engine, Phosphor, developed for the Java language [12] [13].

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 data flow analysis to identify which input variables contribute to output variables at instruction level. In this paper, we adopt Phosphor to conduct dynamic data flow analysis.

- *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 [27].

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

3. 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 existing 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 mark the potential inputs as taint sources and the outputs as taint sinks of each method in an application where we consider not just formal parameters, but also relevant application states. We have two techniques to identify these inputs sources and output sinks: static [11] and dynamic.

DOI: 10.1002/smr

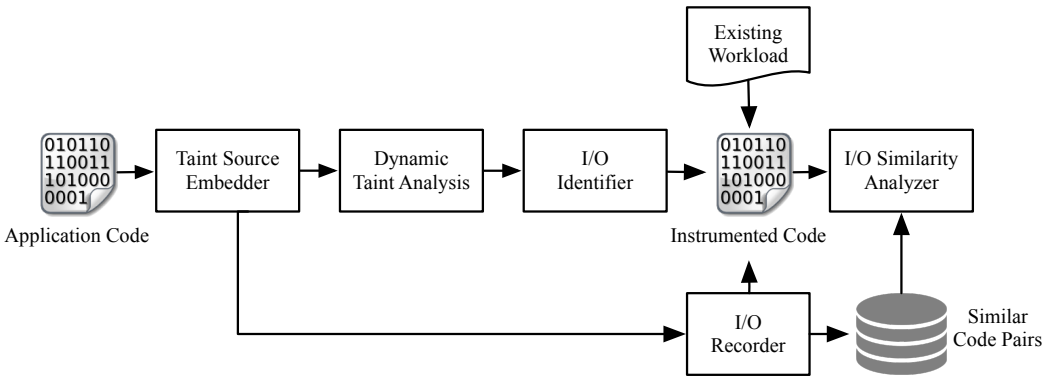


Figure 1. High level overview of In-Vivo Clone Detection. First, taint sources are embedded in methods, then the application is transformed so that its inputs and outputs can be recorded and analyzed based on the taint analysis engine while it is executed under an existing workload. Finally, these recorded inputs and outputs are analyzed to detect functionally similar methods.

In this paper, we will compare the results from both techniques, but our technical discussions will focus more on the dynamic approach. We use a taint analysis engine [12] at runtime to detect which inputs (taint sources) contribute to which outputs (taint sinks). For identifying the potential inputs and outputs in a program, static data flow analysis is faster but may not be as sound as dynamic approaches. Finally 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.

3.1. The Input Generation Problem

Previous approaches towards detecting functional clones in programs randomly or systematically generate inputs to execute individual methods or code fragments first, and then identify code fragments with the identical outputs as functional clones. Especially in the case of object oriented languages like Java, it may be difficult to generate an input to allow an individual method to be executed because each method may have many different input variables, each of which may have an immense range of potential values. Many other techniques have been developed to automatically generate inputs for individual methods, but the problem remains unsolved in the case of detecting functional clones. For instance, Randoop [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. 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 [29]. 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.

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

DOI: 10.1002/smr

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 and outputs at method level, our previous approach [11] conduct a simple static analysis to compute which inputs may affect outputs, while inputs/outputs are instructions that read/store values for programs. To identify inputs given outputs, we follow [30] 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* [30]) helps identify which inputs can affect the computations of outputs.
- Ownership Dependency: This dependency records which values (fields) owned by which objects and/or arrays. Take `int c = a.myInt + b` as the example, where `a` is an object and `myInt` is an integer. In this example, the `myInt` field owned by `a` influences the value of `c`. Because the `a` object owns `myInt`, our approach will know that the `a` object can be an input source, even though this read does not access `a`'s value directly. The ownership dependency helps identify which values can be from inputs. This dependency is transitive, which means that the value owned by an object/array is also owned by the owner of this object/array, if it has any owners.

The inputs are the values that may affect the values of these output sources.

In this paper, we consider both the use of static analysis and dynamic analysis and compare their results. In addition to static analysis, we adopt Phosphor, a taint analysis engine [12], to dynamically compute dependencies between variables. Given a taint sink (output) in a method, its dependencies become the inputs of this method. To conduct taint analysis, we first define the following taint sources and taint sinks at method level.

DOI: 10.1002/smr

- **Taint Source:** Given a method, we define its formal input parameters and its reads from static fields of classes as taint sources. If a potential input is a primitive type, where we follow Java's specification to define primitives as `boolean`, `byte`, `shor`, `int`, `char`, `long`, `float` and `double`, it is a direct taint source. In addition to the 8 primitive types above, we also treat `String` as a primitive in HitoshiIO. If a potential input is an object excluding array, we treat each of its instance fields as taint sources. For an array, we treat each of its element as a taint source.

  The ideal case is that we recursively apply this definition to each potential input. However, this can be computational expensive not only to tainting each field in each object but also to analyze the results in practice, because an object can have other objects as its fields. Take the `Person` object in Figure 2 as the example. The `name` and `age` fields are the taint sources intuitively. However, for the `relatives` field, because it is an array containing `Person` objects, we have to mark all `Person` elements and their fields as taint sources recursively, which may be endless. Thus, we adopt a lazy approach: for each object, only the written fields (updated by `PUTFIELD` instruction) are marked as taint sources. Our assumption is that these written fields have better opportunity to be used by other methods.

- **Taint Sinks:** We treat taint sinks as outputs of a method. Following Definition 2, we mark return values, writes to non-primitive input parameters and writes to static fields in classes in a method as outputs. Given a taint sink (output), we analyze all of its dependencies and extract those from the taint sources marked in the current method as real inputs.

In addition to the data dependencies detected by Phosphor, HitoshiIO considers the values that may change the outputs of the method as the control dependencies. In Figure 2, the length of `relatives` in the `sumAge` method serves as the control dependency, which can decide when the loop should terminate. In our approach, the values from all control dependencies are recorded as inputs.

3.3. Example

To demonstrate our general approach, we use the methods 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 `increAge` method takes a `Person` object and an integer `toAdd` as input parameters. For `toAdd`, as we discussed in Section 3.2, if a formal input parameter is a primitive, it is marked as a taint source. For `me`, because its `age` filed is updated, this field is also marked as a taint source after it is written. Before the exit of a method, HitoshiIO checks if any input object such as `me` is written. In `increAge`, `me` is updated, so HitoshiIO records this object as an output. The values that this object depends on become input(s), which is `toAdd` in `increAge`.

Following the sam concept, we summarize the `updateAndSummary` method, which invokes the `increAge` method and `sumAge` method. We will also use this method to demonstrate how HitoshiIO identifies inputs and outputs in details. Even though this method only has one formal output (return value) `ret`, there are actually 3 taint sinks identified by HitoshiIO: `me`, `COUNTER` and `ret`.

`ret` is the return value, which is a direct taint sink. `COUNTER` is a static field updated in the `updateAndSummary` method, which is identified by HitoshiIO as an output. The only taint source

Prepared using smrauth.cls
DOI: 10.1002/smr

```
1 public class PaperExample {
2
3   public static int COUNTER = 0;
4
5   public static class Person {
6     public String name;
7     public int age;
8     public Person[] relatives;
9   }
10
11  public static int updateAndSummary(Person me, String lastName, int toAdd,
      double useless) {
12    me.name = me.name + lastName;
13
14    increAge(me, toAdd);
15    int ret = sumAge(me.relatives);
16
17    double k = useless + 1;
18
19    COUNTER++;
20    return ret;
21  }
22
23  public static void increAge(Person me, int toAdd) {
24    me.age = me.age + toAdd;
25  }
26
27  public static int sumAge(Person[] relatives) {
28    int sum = 0;
29    for (Person r: relatives) {
30      sum += r.age;
31    }
32    return sum;
33  }
34
35  public static void main(String[] args) {
36    Person jack = new Person();
37    jack.name = "Jack ";
38    jack.age = 25;
39    jack.relatives = new Person[2];
40
41                ...
42                //Prepare data to drive the program
43
44    int totalAge = updateAndSummary(jack, "Foo", 3, 9874);
45    System.out.println("Total age: " + totalAge);
46  }
47
48 }
```

Figure 2. A code example with inputs and outputs identified.

(input) of `COUNTER` is itself. `me` object is update in `updateAndsummary` itself (`me.name`) and one of its callee method `increAge` (`me.age`).

The 4 taint sinks in the `updateAndSummary` method are identified as its outputs:

$$\begin{aligned} &OSink(\texttt{updateAndSummary}) \\ &= \{\texttt{ret}, \texttt{me}, \texttt{COUNTER}\} \end{aligned} \tag{1}$$

The taint sources that affect the values of these taint sinks are identified as the inputs of the `updateAndSummary` method. Before we discuss the taint sources in the `updateAndSummary`

Prepared using smrauth.cls
DOI: 10.1002/smr

Table I. THE DEPENDENCIES IN THE `UPDATEANDSUMMARY` METHOD.

Deps.	Internal	Notes
`relatives[ele]`→`ret`		`ret` is the computational result of `sumAge`, which depends on the `age` fields in every element of `me.relatives`.
`lastName`→`me`	✓	`lastName` is an input parameter that affects the `name` field of `me`.
`toAdd`→`me`		`toAdd` is an input parameter that affects the `age` field of `me`.
`COUNTER`→`COUNTER`	✓	`COUNTER` is a static field, which is only data-dependent on itself.

method, we summarize the dependencies first. We use the variable name to represent the value they contain. And we use $x \rightarrow y$ to represent that y is dependent on x. The dependencies in `updateAndSummary` can be read in Table I. The **Deps.** column records the dependency between two variables, the **Internal** column records if the dependency occurs in `updateAndSummary` and the **Notes** column explains why these two variables have the dependency. To avoid over-computing the dependencies of a value, we conservatively exclude the external dependencies of a taint sink in the current method, such as `relatives[ele]` $\rightarrow$ `ret` that occurs in `sumAge`. However, we keep the external dependencies that affect the fields of the input parameters, such as `toAdd` $\rightarrow$ `me.age` that occurs in `increAge`. Our assumption is that the values written into a field of a object are more likely to be propagated and be used in more methods after the current method.

Finally, we can define the candidate inputs (taint sources) based on the outputs and the dependencies between variables. An input source is the one that have dependencies to any of the outputs. We first define the candidate input sources in `updateAndSummary` as

$$\begin{aligned} &ISrc_c(\texttt{updateAndSummary}) \\ &= \{\texttt{me}, \texttt{lastName}, \texttt{toAdd}, \texttt{useless}\} \end{aligned} \tag{2}$$

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

$$\begin{aligned} &Parents(\{\texttt{ret}, \texttt{me}, \texttt{COUNTER}\}) \\ &= \{\texttt{lastName}, \texttt{toAdd}, \texttt{COUNTER}\} \end{aligned} \tag{3}$$

We then intersect these two sets and conclude the input sources of `updateAndSummary` 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.

Both of the input/output identification and collection in HitoshiIO are during program execution. In the beginning of a method execution, HitoshiIO first marks all potential taint sources/taint sinks and creates an *I/O record* for recording input and output values in the current execution. Then,HitoshiIO executes the method. Over the program execution, HitoshiIO summarizes the taint sources (inputs) that the taint sinks (outputs) depends on, and stores them in the I/O record. Many unique I/O records will likely be collected for each method.

Prepared using smrauth.cls
DOI: 10.1002/smr

Table II. THE INPUT SOURCES IN THE UPDATEANDSUMMARY METHOD.

Var.	Notes
lastName	lastName affects the instance field name of the input parameter me.
toAdd	toAdd affects the instance field age of the input parameter me.
COUNTER	COUNTER affects the write to the static field COUNTER.

```
long sumInRange(long[] arr,
  int a, int b)
{
  long ret = 0;
  for (int i = a; i <= b; i++) {
    ret += arr[i];
  }
  return ret;
}
```

```
public static int L =    ;
public static int R =    ;
public static long sum;

static void rangeSum(long[] arr) {
  long result = 0;
  int i = L;

  while (i <= R) {
    result += arr[i];
    i++;
  }

  sum = result;
}
```

Figure 3. A functional clone detected by HitoshiIO.

3.4. Mining Functionally Similar 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 and two integer as inputs. Further, the first example directly returns the computational result, while the second example stores the result into a static field. We want to consider these functions behaviorally similar, despite these minor differences.

Before detailing our similarity model, we first discuss the concept of *DeepHash* [31] 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.

Prepared using smrauth.cls
DOI: 10.1002/smr

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 [32] is as follows:

$$J(Set_i, Set_j) = \frac{Set_i \cap Set_j}{Set_i \cup Set_j} \tag{4}$$

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.

$$Sim_I(IP_{r|i,s|j}) = J(ISrc_h(inv_{r|m_i}), ISrc_h(inv_{s|m_j})) \tag{5a}$$

$$Sim_O(IP_{r|i,s|j}) = J(OSink_h(inv_{r|m_i}), OSink_h(inv_{s|m_j})) \tag{5b}$$

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

$$Sim(IP_{r|i,s|j}) = \frac{(1 - \beta * e^{Sim_I}) * (1 - \beta * e^{Sim_O})}{(1 - \beta * e)^2} \tag{6}$$

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

$$Sim(MP_{ij}) = \max Sim(IP_{r|i,s|j}) \tag{7}$$

DOI: 10.1002/smr

4. 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. For detecting the dependencies between outputs and potential inputs, HitoshiIO adopts the dynamic taint analysis engine, Phosphor [12] in this paper. HitoshiIO is available on GitHub and released under an MIT license.

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 [33], 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 [33]. 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.

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 is identifying the outputs of a given method. For a method, its output set consists of all variables written by this method that are observable outside of this method. An output could be a variable 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 or other methods (callees) invoked by that method.

The input set of a method consists of variables existing before the execution of this method, which includes formal input parameters (and their fields if they are not primitives) and static fields of classes. It is possible to consider external data sources such as reads from files, databases and networks, as inputs. We leave this development as future work. In addition to these input data sources, HitoshiIO considers the control values that can decide the execution path of the method as inputs.

Based on the definitions of inputs and output, HitoshiIO uses Phosphor to perform a dynamic data flow analysis to compute the dependencies between candidate input sources, $ISrc_c$, and output sinks, $OSink$. HitoshiIO first marks all $ISrc_c$ and $OSink$ in the method before an execution. Before a value flows to a taint sink, which is a real output of the method, HitoshiIO summarizes the dependencies computed by Phosphor to conclude the parents of this taint sink, $Parents(OSink_i)$, where i is the index of this taint sink. If this value flowing to the taint sink

DOI: 10.1002/smr

is primitive type, HitoshiIO summarize all dependencies of this value directly. If this value is an object or array, HitoshiIO summarizes dependencies of the fields and/or elements in this value object. Finally, HitoshiIO intersects these parents with the potential taint sources to identify the real inputs of this execution of the method. To perform such dynamic analysis by marking taints and invoking Phosphor to capture dependencies, HitoshiIO instruments executed methods, which will be discussed in the next section.

The way for our static and dynamic analysis to compute or infer inputs based on an output is similar at high level: which input sources can affect the computational results on output sinks. However, dynamic analysis can offer sounder dependency analysis than static approaches, while static analysis provides better efficiency to infer the dependencies between inputs and outputs.

4.3. Instrumentation

Given the definitions of inputs and outputs, HitoshiIO instrument the application's bytecode to compute and record these values at runtime. For recording the values of inputs and outputs, HitoshiIO creates a I/O recorder in the beginning of the method. For computing the dependencies between potential input sources and output sinks, HitoshiIO inserts instrumentation hints for Phosphor. Each formal input parameter and each read from static fields of classes are marked as taint sources. If the input is a primitive, it is marked as a taint source directly. If the input is an object or array, its fields and/or elements are marked as taint sources recursively with a configurable depth. Before a output value flowing to a taint sink, HitoshiIO queries Phosphor to retrieve the dependencies of this value and summarizes the true input values. These values of inputs and outputs are recorded by the I/O recorder inserted by HitoshiIO in the beginning of the method execution.

Table III describes the various relevant bytecode instructions, their functionality, and the relevant categorization made by HitoshiIO, **Src** instruction that taints values as sources or **Sink** instruction that stores values to an output. HitoshiIO treats the values consumed by the control instructions as inputs. Phosphor will follow the taint sources embedded by HitoshiIO to compute the dependencies between variables.

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 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 [34] 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.

Prepared using smrauth.cls
DOI: 10.1002/smr

Table III. THE POTENTIAL INSTRUCTIONS MARKED AS TAINT SOURCES/SINKS BY HITOSHIIO.

Opcode	Type	Description
`xload`	Src	Load a primitive from a local variable, where `x` is a primitive. If the local variable is a formal input parameter, the loaded value by this instruction is a taint source.
`aload`	Src	Load a reference from a local variable. If the local variable is a formal input parameter, the elements of the loaded array by this instruction are taint sources.
`putstatic`	Src/Sink	Write a value to the field owned by a class. In addition to treat values written to classes as taint sinks, we also marked the values as taint sources for future instructions to read.
`putfield`	Src/Sink	Write a value to the field owned by an object. In addition to treat objects that values written to by this instruction as taint sinks, we also marked the values as taint sources for future instructions to read.
`xreturn`	Sink	Return a primitive value from the method, where `x` is a primitive. The value returned by this instruction is a taint sink.
`areturn`	Sink	Return a reference from the method. The fields/elements in the object or array returned by this instruction are taint sinks.
`ifXXX`	Con.	Represent all `if` instructions. Jump by comparing value(s) on the stack. The value caused these instructions to jump is treated as a direct input of the method.
`tableswitch`	Con.	Jump to a branch based on the index on the stack. The value caused this instruction to jump is treated as a direct input of the method.
`lookupswitch`	Con.	Jump to a branch based on the index on the stack. The value caused this instruction to jump is treated as a direct input of the method.

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 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 §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 [8]. To solve this problem, we

DOI: 10.1002/smr

adopt the *DeepHash* [31] approach, creating a hash of each object. The details of *DeepHash* can refer to §3.4.

The similarity model of HitoshiIO follows §3.4. Optimizing the parameter setting for HitoshiIO's similarity model is extremely expensive. For each different setting, we need to conduct a user study to determine if more or less functional clones can be detected, which is inapplicable. We conduct multiple small scale experiments (*i.e.,* pick a small set of our study subjects) with different βs. Then we manually verify the results to determine the local optimized value for β, which is 3, for the exponential model of Eq. 6 in HitoshiIO. We plan to leverage the power of machine learning to automatically learn the best β for HitoshiIO in the future.

HitoshiIO has two other parameters that control its similarity matching procedure: $InvT$ and $SimT$. We recognize that some hot methods may be invoked millions of times — while others invoked only a handful. $InvT$ provides an upper-bound for the number of individual method input-output profiles that are considered for each method. $SimT$ provides a lower-bound for how similar two methods must be to be reported as a functional clone. We have evaluated various settings for these parameters, and discuss them in greater detail in Section 5.

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 performance of HitoshiIO sufficiently reasonable to use in practice?

RQ3: Is the false-positive rate of HitoshiIO low enough to be potentially usable by developers?

Because HitoshiIO is a dynamic system that requires a workload to drive programs, we selected the Google Code Jam repository [35], 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.

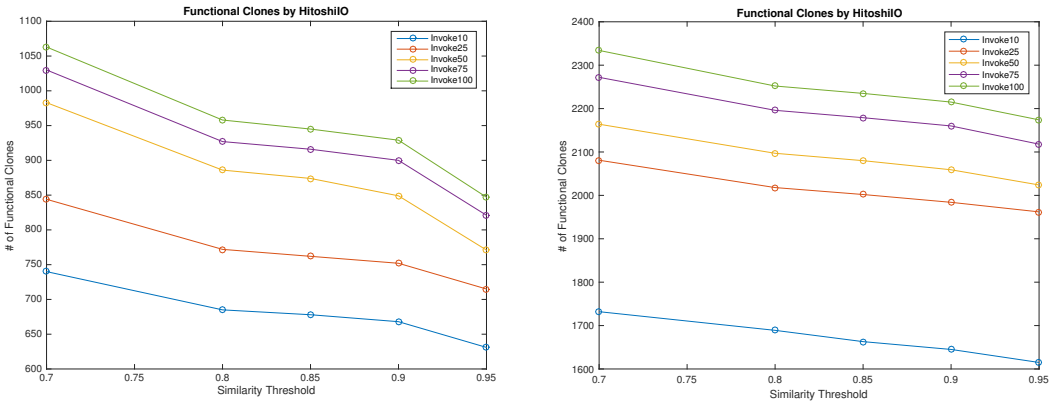
Each annual competition of Google Code Jam usually has several rounds. We examine the projects from four years (2011-2014), and consider the projects that passed the third round of competitions. We only pick the projects that do not require a user to input the data, which can facilitate the automation 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

DOI: 10.1002/smr

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



(a) Static data flow analysis

(b) Dynamic data flow analysis

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

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 [36], 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 $ISrc$s and $OSink$s. HitoshiIO stops recording I/O records for a method, when its invocation threshold is achieved. Intuitively, more functional clones can be detected with a higher invocation

Prepared using smrauth.cls
DOI: 10.1002/smr

Table V. THE DISTRIBUTIONS OF CLONES DETECTED BY HITOSHIIO CROSS THE PROBLEM SETS.

	Number of Clones		Method Compared		Analysis Time (mins)	
Year Pair	Stat.	Dyn.	Stat.	Dyn.	Stat.	Dyn.
$2011-2011$	34	83	11.6M	11.0M	1.2	1.0
$2012-2012$	132	242	11.8M	9.9M	0.9	0.6
$2013-2013$	162	194	8.4M	7.0M	0.8	0.65
$2014-2014$	106	192	9.3M	8.8M	0.9	0.72
$2011-2012$	51	224	24.4M	21.0M	1.9	1.95
$2011-2013$	40	132	20.8M	17.7M	1.8	1.55
$2011-2014$	76	212	21.6M	19.9M	2.2	1.77
$2012-2013$	59	218	21.0M	16.8M	1.7	1.38
$2012-2014$	120	344	21.8M	18.9M	1.5	1.61
$2013-2014$	94	240	18.6M	15.8M	1.6	1.33
Total	874	2080	169.5M	146.7M	14.5	12.6

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). Our parameter tuning is based on the results of static data flow analysis. 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 $2,080$ clones, which contain 247 distinctive methods that average 12 lines of code each. Table V shows the distribution of clones, broken down between the pair of years that each method was found in.We compare the results between static and dynamic data flow analysis, given that the similarity model is the same. While the comparison number of methods and the analysis time is roughly the same, the number and the size of detected functional clones are different. In total, HitoshiIO with dynamic data flow analysis found 288 clones with $LOC \leq 5$ (14%), while 1792 of them are larger than 5 LOC (86%). With static data flow analysis, HitoshiIO found 385 clones with $LOC \leq 5$ (44%), while 489 of them are larger than 5 LOC (56%). We observe that the number of functional clones detected by different techniques to identify I/Os (static or dynamic). The discussion of potential reasons that result in this phenomenon will be discussed in Section 5.4.

About 34% 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), and there are 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).

Prepared using smrauth.cls
DOI: 10.1002/smr

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 $247/825 = 30\%$. 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.

5.2. RQ2: 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. In the static version [11], we developed a simple static analyzer to guide the instrumenter to analyze and instrument the application. The 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). For this version of HitoshiIO, we have two layers of instrumentation: the first layer to insert hints for the second layer, Phosphor, to taint programs and track data flow. Phosphor instruments the whole JVM in addition to our application under study to track data flow, which also raises the performance overhead. We set a 3-minute threshold to execute each project. The overhead for running the application is about 60x compared to executing the application without any profiling.

Compared with the tracing time to collect I/Os of programs, the clone identification time is relatively fast. As shown in Table V, the total analysis time for similarity computation needed to detect these clones was relatively quick though: only 12.6 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, and analysis time by both static and dynamic data flow analysis. While the analysis time is roughly the same, the clone rate of the dynamic data flow analysis is about 8% higher than the static data flow analysis.

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 96% of the clones 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.

DOI: 10.1002/smr

Table VI. THE NUMBER OF MISSED FUNCTIONAL CLONES WITH DIFFERENT INVOCATION THRESHOLDS DETECTED BY HITOSHIIO.

	Clones		Clone Rate		Analysis Time (mins)	
$InvT$	Stat.	Dyn.	Stat.	Dyn.	Stat.	Dyn.
10	678	1663	20.6%	28.1%	0.6	0.5
25	762	2002	21.6%	29.0%	3.8	3.4
50	874	2080	22.4%	30.0%	14.5	12.6
75	916	2179	22.5%	30.5%	32.5	27.3
100	945	2235	22.8%	30.5%	56.6	46

5.3. RQ3: Quality of Functional Clones

Because there is no scientific or rigorous way to evaluate the quality of the detected function clones, we do plan to conduct a large scale user study to verify the validity of the clones detected by HitoshiIO. However, the cost to conduct such analysis can be high: functional clones can look differently but still have similar functionalities. In this section, we refer to the user study conducted in our static version [11].

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

$$precision = \frac{\#TP}{\#FP + \#TP} \tag{8}$$

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

Our student-guided precision evaluation is difficult to compare to previous functional clone works (*e.g.,* [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.

Prepared using smrauth.cls
DOI: 10.1002/smr

5.4. Result Analysis

Compared with our approach [11] with static data flow analysis, HitoshiIO detected about 2x functional clones after adopting the technique of dynamic data flow analysis. We are interested in conducting a large scale user study to analyze if dynamic data flow analysis helps find more true positives or not. Our original assumption is that those programs that function in similar ways should have similar patterns in their dynamic data flow. However, because of the power of dynamic data flow analysis, the I/Os of programs can be represented in details with great resolution. This may result in some programs that actually have different I/Os, but some tiny changes in their I/Os are captured and interpreted by the dynamic data flow analysis in pure primitives or strings. This can cause HitoshiIO detect more false positives given our deep hash based similarity model. With such powerful tool to identify I/Os, one possible approach is to design a new similarity model that can differentiate tiny changes.

Here we raise two open questions and challenges that we plan to answer in the future:

- How to define and verify function clones in a scientific and rigorous way without human involvement?
- Dose better *resolution* to interpret I/Os help identify functional clones?
- Given an I/O identifier, how to design and optimize the similarity computational model?

For most data analysis or mining models, they require feedback and/or metrics to teach them how to optimize and enhance themselves. Unfortunately, now we don't have such metrics, such as recall and precision from an existing benchmark, for HitoshiIO to learn. We look forward to working with the community to solve these fundamental problems to identify functionally similar code.

6. DISCUSSION

6.1. 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.

DOI: 10.1002/smr

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.,* [37,38]) 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, because of the performance overhead, currently we set a time limitation to execute programs and collect their I/Os, which may result in identifying fewer functionally similar code 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.

6.2. Future Work

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. To verify the the validity of the functionally similar programs detected by HitoshiIO, we plan to conduct a large scale user study. However, because these detected programs are *functionally* similar with or without similar appearance, it is likely harder for users to evaluate the results of HitoshiIO than static approaches that focus on look-alike code.

As we mentioned in Section 5.4, the number of detected functional clones by different I/O identifiers are different. We are interested in uncovering the reasons underneath. Based on our assumption, an I/O identifier with great dependency detection capability like dynamic taint analysis should detect functional clones with better *quality*. However, in this paper, even though we showed that HitoshiIO with dynamic flow analysis detects functions clones with better *quantity* than static data flow analysis, we are still searching for a scientific way to prove these clones also have better quality.

At the system level, we have two development plans as follows. Because of the double instrumentations (HitoshiIO instruments programs for embedding taint hints and Phosphor instruments programs to detect dependencies), HitoshiIO has high performance overhead. How to optimize HitoshiIO to achieve an acceptable system performance is a problem we plan to tackle. In addition to the current taint sources we consider, we plan to take more possible input sources into account. As we discussed in Section 4.2, data from the environment external to programs, such as files, databases and networks, can be input sources that HitoshiIO can consider in the future. Further, at method level, it is possible for HitoshiIO to consider the return values from callees of a method as taint sources. These two developments are not independent: while we expand the definitions of input (taint) sources, the performance of HitoshiIO will be affected.

DOI: 10.1002/smr

7. 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, we integrate HitoshiIO with a static data flow analyzer and dynamic taint analysis engine, Phosphor, 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. We observe that based on different types of data flow analysis to identify I/Os of functions, the number of detected functional clones vary: using the dynamic taint analysis detects 2x more functional clones than the static data flow analyzer.

While verifying the validity and the difference between functional clones detected by both I/O identification techniques is an open problem, we look forward to working with the community to define *true* functional clones in a rigorous and scientific way. We have made our system publicly available on GitHub, and are excited by the future investigations and developments in the community.

ACKNOWLEDGEMENT

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. Kamiya T, Kusumoto S, Inoue K. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *TSE* Jul 2002; .
2. Kim M, Sazawal V, Notkin D, Murphy G. An empirical study of code clone genealogies. ESEC/FSE-13, 2005.
3. Li Z, Lu S, Myagmar S, Zhou Y. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. OSDI'04.
4. Jiang L, Su Z. Automatic mining of functionally equivalent code fragments via random testing. ISSTA '09.
5. Elva R, Leavens GT. Semantic clone detection using method ioe-behavior. IWSC '12; 80–81.
6. Kim H, Jung Y, Kim S, Yi K. Mecc: Memory comparison-based clone detector. ICSE '11.
7. Egele M, Woo M, Chapman P, Brumley D. Blanket execution: Dynamic similarity testing for program binaries and components. *Proc of the 23rd USENIX Conference on Security Symposium*, 2014.
8. Deissenboeck F, Heinemann L, Hummel B, Wagner S. Challenges of the Dynamic Detection of Functionally Similar Code Fragments. CSMR '12.
9. Neubauer LA. Kamino: Dynamic approach to semantic code clone detection. *Technical Report CUCS-022-14*, Department of Computer Science, Columiba University.
10. Bell J, Kaiser G, Melski E, Dattatreya M. Efficient dependency detection for safe java test acceleration. ESEC/FSE 2015.
11. Su F, Bell J, Kaiser GE, Sethumadhavan S. Identifying functionally similar code in complex codebases. ICPC'16.
12. Bell J, Kaiser G. Phosphor: Illuminating dynamic data flow in commodity jvms. OOPSLA '14.
13. Phosphor. URL https://github.com/Programming-Systems-Lab/phosphor.
14. Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.* May 2009; **74**(7):470–495.

DOI: 10.1002/smr

15. Baker BS. A program for identifying duplicated code. *Computer Science and Statistics: Proc. Symp. on the Interface*, 1992; 49–57.
16. Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. ICSM '98.
17. Jiang L, Misherghi G, Su Z, Glondu S. Deckard: Scalable and accurate tree-based detection of code clones. ICSE '07.
18. Gabel M, Jiang L, Su Z. Scalable detection of semantic clones. ICSE '08.
19. Liu C, Chen C, Han J, Yu PS. Gplag: Detection of software plagiarism by program dependence graph analysis. KDD '06.
20. Krinke J. Identifying similar code with program dependence graphs. WCRE '01, 2001.
21. Li J, Ernst MD. Cbcd: Cloned buggy code detector. ICSE '12.
22. Marcus A, Maletic JI. Identification of high-level concept clones in source code. ASE '01.
23. McMillan C, Grechanik M, Poshyvanyk D. Detecting similar software applications. ICSE '12.
24. Collberg CS, Thomborson C, Townsend GM. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.* Oct 2007; **29**(6).
25. Carzaniga A, Mattavelli A, Pezzè M. Measuring software redundancy. ICSE '15.
26. Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. OSDI'10.
27. Murphy C, Kaiser G, Vo I, Chu M. Quality assurance of software applications using the in vivo testing approach. ICST '09.
28. Pacheco C, Ernst MD. Randoop: Feedback-directed random testing for java. OOPSLA '07.
29. Thummalapenta S, Xie T, Tillmann N, de Halleux J, Schulte W. Mseqgen: Object-oriented unit-test generation via mining source code. ESEC/FSE '09.
30. Frankl PG, Weyuker EJ. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.* Oct 1988; **14**(10).
31. The java-util library. URL https://github.com/jdereg/java-util/.
32. Levandowsky M, Winter D. Distance between sets. *Sci. Comput. Program.* Nov 1971; **234**:34–35.
33. Lindholm T, Yellin F, Bracha G, Buckley A. *The Java Virtual Machine Specification*. Java SE 7 edn. Feb 2013. URL http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html.
34. The xstream library. URL http://x-stream.github.io/.
35. Google code jam. URL https://code.google.com/codejam.
36. Amazon ec2. URL https://aws.amazon.com/ec2/.
37. Krutz DE, Le W. A code clone oracle. MSR '14.
38. Svajlenko J, Islam JF, Keivanloo I, Roy CK, Mia MM. Towards a Big Data Curated Benchmark of Inter-project Code Clones. ICSME '14.

DOI: 10.1002/smr