Supporting Oracle Construction via Static Analysis

Junjie Chen[1,2], Yanwei Bai[1,2], Dan Hao[1,2†], Lingming Zhang[3], Lu Zhang[1,2], Bing Xie[1,2], Hong Mei[1,2]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[2]Institute of Software, EECS, Peking University, Beijing, 100871, China
{chenjunjie,byw,haodan,zhanglucs,xiebing,meihong}@pku.edu.cn
[3]Department of Computer Science, University of Texas at Dallas, 75080, USA
lingming.zhang@utdallas.edu

ABSTRACT

In software testing, the program under test is usually executed with test inputs and checked against a *test oracle*, which is a mechanism to verify whether the program behaves as expected. Selecting the right oracle data to observe is crucial in test oracle construction. In the literature, researchers have proposed two dynamic approaches to oracle data selection by analyzing test execution information (e.g., variables' values or interaction information). However, collecting such information during program execution may incur extra cost. In this paper, we present the first static approach to oracle data selection, SODS (Static Oracle Data Selection). In particular, SODS first identifies the substitution relationships between candidate oracle data by constructing a *probabilistic substitution graph* based on the definition-use chains of the program under test, then estimates the fault-observing capability of each candidate oracle data, and finally selects a subset of oracle data with strong fault-observing capability. For programs with analyzable test code, we further extend SODS via pruning the probabilistic substitution graph based on 0-1-CFA call graph analysis. The experimental study on 11 subject systems written in C or Java demonstrates that our static approach is more effective and much more efficient than state-of-the-art dynamic approaches in most cases.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Test oracle, oracle data selection, static analysis

1. INTRODUCTION

In software testing, a *test oracle* is a mechanism determining whether a program executes as expected for the given test inputs, and it intuitively consists of variables to be observed during testing and their expected values. For the same test inputs, different test oracles may demonstrate different fault-detection capability. Therefore, high-quality test oracles are essential for detecting software faults. In the literature, although researchers proposed various techniques [7, 34, 56, 52, 11, 31] to automatically generate test inputs, the test oracle problem is still recognized as one of the most difficult problems in software testing [5, 12, 71, 73].

Generally speaking, test oracle creation requires the variables for observation and their expected values. As a program usually consists of various internal and output variables, which may be observed at various positions in software testing, the program has many candidate oracle data[1] to be included in the test oracle. The more oracle data a test oracle contains, the more powerful the test oracle is in detecting faults [6, 70]. However, it can be extremely costly to construct a test oracle with all or a large portion of oracle data, because developers need to specify the expected values of these variables. Therefore, the problem of oracle data selection arises, aiming at reducing the number of oracle data in constructing a test oracle [62]. In the literature, two dynamic approaches (i.e., MAODS [62] and DODONA [46]) have been proposed to select oracle data by analyzing the execution information (i.e., variables' values or interactions) of many tests. However, it may incur extra cost to collect the dynamic execution information [48].

To address this issue, we present the first static approach, SODS, to selecting oracle data for observation in software testing. Our approach defines and constructs a *probabilistic substitution graph* based on the definition-use chains for the program under test. The probabilistic substitution graph is a graph that presents to what extent (i.e., in terms of probability) a candidate oracle data may be a substitute for others. Then SODS estimates the capability of each candidate oracle data on observing faults in each statement by considering to what extent substitution relationships transfer (measured by α). Finally, SODS determines the selection order of candidate oracle data based on their fault-observing capability and the impact of selected oracle data (measured by f_p). Furthermore, for any programs with analyzable test code (e.g., JUnit tests), we extend our static approach by tailoring the program under test based on 0-1-CFA call graph analysis of its tests to improve the effectiveness of oracle data selection. For ease of presentation, we call the former as *basic* SODS and the latter extension as *extended* SODS.

To evaluate SODS, we conducted an experimental study on 11 real-world subject systems (including two C subjects

†Corresponding author.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970366

[1]In this paper, the oracle data refer to the variables to be observed in software testing.

and nine Java subjects). Through the study on the impacts of α and f_p, our SODS techniques are more effective when α is set to 0 and f_p is set to be differentiated (explained in Section 3.3.2). Through the study on comparing dynamic approaches and our static approach, our SODS is more effective than the dynamic approaches (including MAODS and DODONA) in most cases, and is also much more efficient than the dynamic approaches, especially MAODS.

In summary, the contributions of this paper are as follows.

- The first static approach to selecting oracle data based on probabilistic substitution graph constructed from definition-use chain analysis.
- A further extension for programs with analyzable test code based on 0-1-CFA call graph analysis.
- An extensive experimental study on 11 real-world subjects demonstrating that our static approach is more effective and much more efficient than the existing dynamic approaches in most cases. Furthermore, for programs with analyzable test code (e.g., JUnit tests), our extended static technique can be even more effective than our basic static technique.

2. AN EXAMPLE

Figure 1 presents an example program, where the method "`factorial`" is to calculate the factorial of a number using the method "`multiply`". To observe faults in this program during software testing, developers need to construct a test oracle, which consists of oracle data including internal variables and output variables. As variables may be defined more than once (e.g., *factorial* is defined at Lines 4 and 10 respectively), we use a tuple between a variable and its definition statement to represent a candidate oracle data. Note that in this paper the concept of "defining a variable" does not refer to variable declaration, but refers to the fact that a variable is assigned a value by some statement. This concept is adopted by the terminology of data-dependency analysis [33, 53]. At Line 3, variable n is defined, which means that Line 3 assigns a value to n and that variable n is viewed as a candidate oracle data (denoted as o_1). Similar, variable i at Line 5 is also a candidate oracle data, denoted as o_3. However, variable n at Line 5 is not viewed as a candidate oracle data in our approach, because Line 5 does not assign a new value to variable n but uses its value, which is assigned at Line 3. That is, in order to decrease the number of candidate oracle data, we consider only the defined variables as candidate oracle data. In inter-procedural analysis, we also regard the variable whose value is assigned through method call or return as a candidate oracle data. For example, $num1$ defined at Line 17 and *factorial* defined at Line 13 (i.e., the first *factorial* at Line 13) are two candidate oracle data, which are o_7 and o_5. By analyzing statements, especially the variables defined by these statements, we construct a set of candidate oracle data.

For the set of candidate oracle data, we analyze their definition-use chains, which are used to measure the fault-observing capability of candidate oracle data. In data-flow analysis, a definition-use chain is the structure that consists of the definition of one variable and the use of the variable, and the former variable reaches the latter use without other intervening definitions [50]. For example, the relation between variable n at Line 3 and variable n at Line 5 is regarded as a definition-use chain. As this paper targets oracle data selection among candidate oracle data, we adapt the definition of a definition-use chain as a structure that consists of two oracle data o_i and o_j satisfying that the definition of the variable in o_i uses the value of the variable defined in o_j without other intervening oracle data. In the remaining of this paper, a definition-use chain refers to such a structure on oracle data. For example, variable i at Line 5 and variable n at Line 3 form a definition-use chain. Moreover, observing the value of variable i at Line 5 may detect faults resulting from the wrong value of variable n at Line 3. Therefore, variable i at Line 5 can be used as a substitute for n at Line 3 in software testing. That is, based on definition-use chains, we can construct substitution relationships between candidate oracle data.

```
1  void factorial() { /* compute factorial of a non−negative
       integer n, i.e., n!*/
2    int n, factorial, i;
3    read("Enter the number:", n); // o1
4    factorial = 1; // o2
5    i = n; // o3
6    if (n < 0)
7      print("wrong input is:", n);
8    else {
9      if (n == 0)
10       factorial = 0;// o4
11     else {
12       while (i > 0) {
13         factorial = multiply(factorial, i);// o5
14         i = i − 1;} }// o6
15     print("The result is:", factorial);} }
16
17 int multiply(int num1, int num2) {// o7,o8
18   int result;
19   return result = num1 * num2;}// o9
```

Figure 1: Example program

Software faults are typically induced via erroneous code, and errors in code can be observed through oracle data. For example, variable i at Line 5 is useful in detecting faults resulting from Line 5 because an error in this statement (e.g., "`i=n+1`") may produce a wrong value for i. Furthermore, this oracle data is also useful in detecting faults resulting from other statements (i.e., Line 3), which define the value for variable n used by Line 5. Therefore, observing the value of i at Line 5 may detect faults resulting from Lines 3 and 5. Based on this insight, for a candidate oracle data, it is feasible to statically identify the statements, whose faults may be observed by this oracle data. Based on this information, we can further estimate the capability of candidate oracle data on observing faults.

As it is costly to construct a test oracle with all the candidate oracle data, it is necessary to determine the selection order of these candidate oracle data so that developers may construct a high-quality test oracle with a small number of oracle data. Intuitively, the candidate oracle data with large capability on observing faults tend to be selected early when constructing a test oracle. However, it may be less effective to select oracle data just based on the descendent order of their fault-observing capability because some candidate oracle data observe the faults resulting from the same statements. For example, observing either the value of the first i at Line 14 or the value of $num2$ at Line 17 may detect the faults resulting from Line 5. Although both of these oracle data may have large capability on observing faults, it is not so necessary to select both of them because their fault-observing capability may overlap with each other.

Therefore, during oracle data selection, it is necessary to identify the set of statements in which each candidate oracle data may detect faults so as to avoid selecting oracle data with significantly overlapping fault-observing capability. That said, oracle data selection should aim to maximize the fault-observing capability of the set of selected oracle data, not to maximize the fault-observing capability of each selected oracle data individually.

In practice, tests are usually not sufficient and test a program partially [29]. Therefore, we can further improve the effectiveness of static oracle data selection by focusing on only the partial program being tested. For example, testers want to test only the method "`multiply`" in Figure 1 using some JUnit test. As the source code in the method "`factorial`" is not tested by this JUnit test, the candidate oracle data in "`factorial`" (i.e., from o_1 to o_6) should be excluded when constructing a test oracle for the JUnit test. Similarly, the definition-use chains occurring on these candidate oracle data should also be excluded. Following this intuition, we extend static oracle data selection for programs with analyzable test code, because the partial program tested by the tests can be statically identified by program analysis.

3. STATIC ORACLE DATA SELECTION

In this section, we first present the details of our basic static technique, including probabilistic substitution graph construction (Section 3.1), fault-observing capability calculation (Section 3.2), and oracle data selection (Section 3.3). Then we present our extended static technique in Section 3.4. Finally, we present the complexity analysis in Section 3.5.

3.1 Constructing the Probabilistic Substitution Graph

For any program under test, our static oracle data selection first takes the definition-use chains of **the whole program** as inputs to construct a probabilistic substitution graph, which presents the probability that a candidate oracle data may be a substitute for others. In particular, the definition-use chains can be automatically generated by using off-the-shelf inter-procedural program analysis tools (e.g., Crystal [57] and WALA [68]). Suppose the set of candidate oracle data is denoted as $\{o_1, o_2, \ldots, o_n\}$, each of which is a definition of a variable. In other words, a candidate oracle data can be viewed as a tuple between a variable and its position, denoting the observation of the variable at the position. For any two oracle data o_i and o_j ($1 \leq i \neq j \leq n$), we represent their definition-use chain by $o_i \Leftarrow o_j$, which shows that the definition of the variable in o_i uses the value of the variable defined in o_j. For Figure 1, the set of candidate oracle data is $\{o_1, o_2, \ldots, o_9\}$, e.g., o_1 refers to observing n after Line 3 and o_9 refers to observing *result* after Line 19. Based on the definition of definition-use chains (Section 2), the example has the following definition-use chains: $o_3 \Leftarrow o_1$, $o_7 \Leftarrow o_2$, $o_6 \Leftarrow o_3$, $o_8 \Leftarrow o_3$, $o_9 \Leftarrow o_7$, $o_9 \Leftarrow o_8$ and $o_5 \Leftarrow o_9$. We regard the process of method call or return as an assignment from one variable to another and thus get inter-procedural definition-use chains like $o_7 \Leftarrow o_2$, $o_8 \Leftarrow o_3$, and $o_5 \Leftarrow o_9$.

Then, we define the probabilistic substitution graph to represent substitution relationships between candidate oracle data. For any program, we define its probabilistic substitution graph (abbreviated as PSG) $G = (V, E, W)$ as follows.

- V is the set of vertices, each of which represents a candidate oracle data. That is, V is denoted as $\{o_1, o_2, \ldots, o_n\}$.
- E is the set of edges, each of which denotes a substitution relationship between two candidate oracle data. For any two candidate oracle data o_i and o_j, if $o_i \Leftarrow o_j$, the former may be a substitute for the latter, which is denoted as $o_i \succ o_j$, and there is an edge from o_j to o_i.
- W is the set of weights on edges. Each weight denotes the probability that each substitution relationship occurs between a pair of candidate oracle data. If $o_i \succ o_j$ with some probability (denoted as $P(o_i \succ o_j)$), there is an edge from o_j to o_i whose weight is $P(o_i \succ o_j)$.

Intuitively, for any edge from o_j to o_i whose weight is $P(o_i \succ o_j)$, faults resulting from the wrong values of the variable in o_j may be caught by o_i with the probability $P(o_i \succ o_j)$. That is, the candidate oracle data o_i is likely to be a substitute for o_j when constructing a test oracle.

For any program, we construct its PSG as follows.

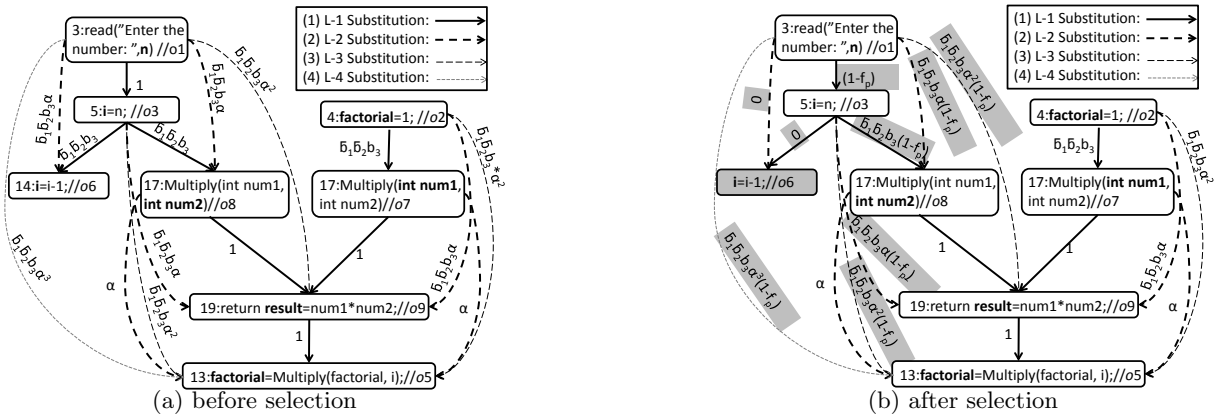
First, our basic static technique constructs vertices and edges of its PSG based on its definition-use chains. Although a variable may be in more than one statement, including its definition statements and use statements, we take only the tuple between the variable and the position immediately after its definition statement as a candidate oracle data. For any two candidate oracle data o_i and o_j, if $o_i \Leftarrow o_j$, we construct an edge from o_j to o_i.

Second, our basic static technique assigns weights on edges to reflect the probability on the substitution relationships. As some substitution relationships occur on some executions, not all the executions, our basic static technique uses the probability on the substitution relationships to estimate how likely the wrong value of the variable in o_j may be observed by the value of the variable in o_i for any edge from o_j to o_i. In particular, if the two statements in o_j and o_i are always executed together (e.g., o_1 and o_3 in Figure 1), $P(o_i \succ o_j)$ is set to 1. Otherwise, $P(o_i \succ o_j)$ is set to the probability that the statement in o_i is executed given that the statement in o_j is executed, i.e., the execution probability of the corresponding branch. Note that since our analysis operates on static control-flow graphs and data-flow graphs, the dynamic loop iteration number cannot be obtained and all the loops will be treated as having one iteration. For example, the while statement "while($i > 0$)" in Line 12 of Figure 1 is treated as "if($i > 0$)" in our basic static technique. Thus, $P(o_6 \succ o_3)$=$\bar{b}_1 * \bar{b}_2 * b_3$, where $\bar{b}_1$, $\bar{b}_2$, and b_3 represent the execution probability of the $false$ branch of Line 6, the execution probability of the $false$ branch of Line 9, and the execution probability of the $true$ branch of Line 12. Although this treatment is a coarse simplification, it avoids producing circles in the PSG for analyzing dependencies between oracle data. To illustrate, Figure 2(a) presents the PSG of the example in Figure 1 with additional dashed edges.

3.2 Estimating Fault-Observing Capability

From the macroscopic perspective, faults are induced by developers' errors in programming, which are demonstrated by erroneous statements. From the microscopic perspective, faults are reflected by variables' values during execution. Therefore, observing oracle data may detect faults resulting from erroneous statements.

For any candidate oracle data o_i, observing it may detect faults resulting from the statement in o_i because any faults



(a) before selection (b) after selection

Figure 2: Example Probabilistic Substitution Graph (PSG) (with transitive substitution)

in this statement may yield a wrong value to the variable in o_i.

For any other candidate oracle data o_j, if $o_i \succ o_j$, the definition of the variable in o_i uses the value of the variable in o_j directly. Observing o_i can potentially also detect faults resulting from the statement in o_j because a fault in this statement may produce an incorrect value which in turn may propagate to the definition of the variable in o_i.

For any other candidate oracle data o_{k1} and o_{k2} ($i \neq k1 \wedge i \neq k2$), if $o_i \succ o_{k1}$ and $o_{k1} \succ o_{k2}$ (i.e., the definition of the variable in o_i uses the variable in o_{k1} and the definition of the latter uses the variable in o_{k2}), observing o_i may also detect faults resulting from the statement in o_{k2} because a fault in this statement may produce an incorrect value that may propagate to the definition of the variable in o_i through the definition of the variable in o_{k1}. If $o_i \succ o_{k1} \succ \ldots \succ o_{ks}$ ($i \neq k1$, ..., $i \neq ks$), it is likely to detect faults resulting from the statements in o_{k1}, ..., o_{ks} via observing o_i. Note that due to our simplification on loop statements, the preceding analysis on transitive substitution relationships does not produce circles like $o_i \succ o_{k1} \succ \ldots \succ o_i$.

For any candidate oracle data o_i, we traverse the PSG to find the set of statements whose faults are likely to be detected via observing o_i. We denote this set as $W(o_i)$. Obviously, $W(o_i)$ includes the statement in o_i. For any o_j belonging to $W(o_i)$, it is likely to detect faults resulting from the statement in o_j via observing o_i. That is, o_i has the capability on observing faults resulting from the statement in o_j. We denote this capability as $FOC(i, j)$.

In particular, given two candidate oracle data o_i and o_j, we estimate the value of $FOC(i, j)$ as follows. If o_i and o_j are the same oracle data (i.e., i=j), $FOC(i, i) = 1$ because o_i is the most suitable oracle data to detect faults resulting from the statement in o_i itself. If there is an edge from o_j to o_i in the PSG, we use the weight on this edge (i.e., $P(o_i \succ o_j)$) to measure $FOC(i, j)$ because $P(o_i \succ o_j)$ shows how likely faults resulting from the wrong values of the variable in o_j may be caught by o_i. In this circumstance, $FOC(i, j) = P(o_i \succ o_j)$. Supposing $o_i \succ o_{k1} \succ \ldots \succ o_{ks} = o_j$ is a path (denoted as $path$) from o_j to o_i in the PSG, we use Formula 1 to estimate the FOC value resulting from this path.

$$FOC(path) = \alpha^{s-1} * P(o_i \succ o_{k1}) * ... * P(o_{ks-1} \succ o_{ks}) \quad (1)$$

Here α (which may be a constant or variable) is a parameter on measuring to what extent the substitution relationship transfers between two adjacent edges in paths with length $\geq$2. Note that for any path of length 1 (i.e., $s = 1$), the FOC value for the path would be $P(o_i \succ o_{k1})$. We will empirically investigate the impact of α in Section 4. The dashed edges in Figure 2(a) also illustrate the transitive substitution relationship along paths between oracle data, e.g., the *L-i Substitution* edge denotes the transitive substitution relationship along paths with length i. In particular, *L-1 Substitution*, represented by solid edges in the figure, is actually a substitution relationship without transition. Since there may be more than one path from o_j to o_i in the PSG, we sum up the estimated FOC values resulting from all the paths to calculate $FOC(i, j)$. In particular, supposing there are l paths (denoted as $path_1$, $path_2$, ..., $path_l$) from o_j to o_i in the PSG, Formula 2 calculates the value of $FOC(i, j)$ considering all paths from o_j to o_i. In particular, as these paths share the same starting vertex (i.e., o_j) and ending vertex (i.e., o_i), the value of $FOC(i, j)$ cannot be larger than 1. Furthermore, we are always able to consider all paths from o_j to o_i because PSG is acyclic.

$$FOC(i, j) = \sum_{t=1}^{l} FOC(path_t) \quad (2)$$

We adopt a variant of the Floyd algorithm [21] by enumerating new paths from o_j to o_i and updating $FOC(i, j)$ when a vertex in V is added to the set of intermediate vertices along paths. As a PSG may be represented by a sparse matrix by discarding zero elements, we plan to refine our basic technique via further optimizations, e.g., orthogonal list representation [35] and Johnson's algorithm [44], which target storing and operating sparse matrices.

3.3 Selecting Oracle Data

Using the values of $FOC(i, j)$, we can calculate the total capability of o_i on observing faults in statements of the program under test (denoted as $FOC(i)$) via Formula 3.

$$FOC(i) = \sum_{o_j \in W(o_i)} FOC(i, j) \quad (3)$$

Values of $FOC(i)$ provide a basic guideline for selecting oracle data. However, given two oracle data o_i and o_j, if $FOC(i, k) > 0$ and $FOC(j, k) > 0$, both o_i and o_j can help detect faults resulting from the statement in o_k. Therefore, if o_i is already selected for detecting faults resulting from the statement in o_k, it may be less necessary to select o_j

for the same purpose. We use f_p to measure the impact of selected oracle data on unselected oracle data. Furthermore, different numbers of oracle data may be used in different circumstances. Therefore, we propose several heuristics for oracle data selection in the following sub-sections:

Algorithm 1 Oracle data selection

```
1: for each i (1 ≤ i ≤ n) do
2:    Selected[i] ← false
3:    Foc[i] ← 0
4: end for
5: for each i (1 ≤ i ≤ n) do
6:    for each j (1 ≤ j ≤ n) do
7:       if FOC[i, j] > 0 then
8:          Foc[i] ← Foc[i] + FOC[i, j]
9:       end if
10:   end for
11: end for
12: for each j (1 ≤ j ≤ m) do
13:   current ← 0
14:   k ← 0
15:   for each i (1 ≤ i ≤ n) do
16:      if not Selected[i] then
17:         if Foc[i] > current then
18:            current ← Foc[i]
19:            k ← i
20:         end if
21:      end if
22:   end for
23:   Selected[k] ← true
24:   Foc[k] ← 0
25:   for each i (1 ≤ i ≤ n) do
26:      if FOC[k, i] > 0 then
27:         for each s (1 ≤ s ≤ n) do
28:            if k ≠ s ∧ FOC[s, i] >0 ∧ not Selected[s] then
29:               Foc[s] ← Foc[s] − FOC[s, i] ∗ f_p
30:               FOC[s, i] ← FOC[s, i] ∗ (1 − f_p)
31:            end if
32:         end for
33:      end if
34:   end for
35: end for
```

3.3.1 Selection Algorithm

Algorithm 1 depicts our heuristics on determining the selection order of candidate oracle data. Suppose that there are totally n candidate oracle data to be selected. In the algorithm, we use a boolean array $Selected[i]$ $(1 \leq i \leq n)$ to record whether the candidate oracle data o_i has been selected, an array $FOC[i, j]$ $(1 \leq i, j \leq n)$ to record the fault-observing capability of o_i for observing faults in the statement in o_j, and $Foc[i]$ $(1 \leq i \leq n)$ to record the total fault-observing capability of o_i.

In Algorithm 1, Lines 1 to 4 perform initialization. Lines 5 to 11 calculate the values of $Foc[i]$ $(1 \leq i \leq n)$ based on the values of $FOC[i, j]$ $(1 \leq i, j \leq n)$ according to Formula 3.

Lines 12 to 35 perform oracle data selection. In particular, the loop body repeats m $(m \leq n)$ times to select m oracle data. Within this loop, Lines 13 to 22 look for the candidate oracle data (whose index is assigned to k) with the largest fault-observing capability; Line 23 selects this oracle data by assigning true to its variable $Selected[k]$; Line 24 assigns 0 to the total fault-observing capability of selected oracle data to avoid duplicate selection.

Lines 25 to 34 adjust the fault-observing capability of other candidate oracle data on the basis that candidate oracle data o_k is selected. Here, we use f_p $(0 \leq f_p \leq 1)$ to parameterize how selected oracle data impact the fault-observing capability of other candidate oracle data. Thus, for different values of f_p, Algorithm 1 yields different oracle data selection strategies. Intuitively, the larger f_p is, the larger impact the selected oracle data o_k has on any unselected oracle data o_s, and thus the smaller the values of $Foc[s]$ and $FOC[s, i]$ are. Figure 2(b) presents the updated PSG after selecting o_6 (marked as grey node) based on Figure 2(a), where the grey areas represent adjusted FOC values. For example, the FOC value between o_3 and o_1 is now updated to $1 * (1 - f_p)$ since o_1 has been tested to some extent by the selected o_6.

3.3.2 Choice of the Parameter f_p

As f_p provides a means to control the impact of previous selections of oracle data on unselected oracle data, we provide two strategies on the choice of the value of f_p. In an optimistic perspective, since the selection of o_k in Algorithm 1 would help observe faults in the statement of any o_i where $FOC[k, i] > 0$, there is no need to consider observing faults for these statements. Thus, we can set the value of f_p to 1, so that Lines 25 to 34 in Algorithm 1 can ensure not to consider faults resulting from the statements of o_i where $FOC[k, i] > 0$.

Furthermore, since the FOC values of an oracle data already reflect how the oracle data can help observe faults in statements, we can utilize these values when setting the value of f_p. In particular, we make f_p a differentiated value for different oracle data because selected oracle data have various impact on different unselected oracle data. That is, we use an array $fp[i]$ $(1 \leq i \leq n)$ to record the f_p value of o_i in this algorithm. Initially, the f_p value for any oracle data is 0 because no oracle data is selected. In Algorithm 1, we implement this initialization by adding $fp[i] \leftarrow 0$ between Lines 3 and 4. As soon as o_k is selected, we update the f_p value for any oracle data o_i where $FOC[k, i] > 0$, by adding $fp[i] \leftarrow fp[i] + FOC[k, i]$ and $f_p \leftarrow fp[i]$ between Lines 26 and 27. That is, as o_k is selected, it adds more impact of the selected oracle data on the fault-observing capability of unselected oracle data and thus we increase the f_p value of any oracle data o_i where $FOC[k, i] > 0$. Moreover, for o_i, the more times its f_p is updated due to the selection of some oracle data, the larger its f_p value is and the smaller the value of $FOC[s, i]$ is (o_s represents any unselected oracle data where $FOC[s, i] > 0$). That is, by increasing the f_p value of o_i, Algorithm 1 prefers to not select the oracle data o_s where $FOC[s, i] > 0$ to observe faults in the statement of o_i because these faults may already be detected by the selected oracle data.

3.4 Further Extension

In practice, tests are usually not sufficient and actually test a program partially. Therefore, it is not necessary to consider all the substitution relationships between candidate oracle data because some of them are not covered by existing tests. Furthermore, considering all these substitution relationships, our basic static technique may select useless oracle data and become less effective. Fortunately, for some modern unit testing frameworks (e.g., JUnit), each test is a code snippet including sequence of method invocations. Thus it is possible to statically identify the parts of source code tested based on the call-graph analysis of the test code snippets. That is, for any program with tests in the form of analyzable code snippets (e.g., JUnit tests), we further extend the static technique to tailor the program based on static call-graph analysis.

In particular, our extended technique first extracts a static call graph of the tests (denoted as T) by using 0-1-CFA al-

gorithm [30], which has been demonstrated to be efficient and more precise than other common static algorithms, e.g., class hierarchy analysis [17] or rapid type analysis [4]. Based on the static call graph, our extended technique constructs definition-use chains by removing the candidate oracle data that are not within the static call graph of T. Based on these definition-use chains, our extended technique constructs a PSG, which is actually a tailored PSG for the whole program. Based on this PSG, our extended technique estimates the fault-observing capability of candidate oracle data (Section 3.2) and selects oracle data (Section 3.3). Further discussion on the comparison between the basic and extended static techniques is referred to Section 5.1.

3.5 Complexity Analysis

Supposing that the total number of candidate oracle data is n and the number of selected oracle data is m, we analyze the time complexity of our basic static technique as follows. The time complexity for constructing the PSG is $O(n)$. Since our basic static technique calculates paths in the PSG based on a variant of the Floyd algorithm, the time complexity for estimating the fault-observing capability is $O(n^3)$ in the worst case. When $\alpha = 0$, this complexity can be further reduced to $O(n)$. The time complexity for oracle data selection in the worst case is $O(mn^2)$. Generally speaking, the time complexity of our basic static technique is $O(n^3)$ in the worst case, since $m \leq n$.

Our extended static technique has the same stages as our basic static technique when taking definition-use chains as inputs. Thus, the time complexity of our extended static technique is also $O(n^3)$ in the worst case. In particular, our extended static technique needs to construct a static call graph of T by 0-1-CFA, which may incur extra cost [30]. On the other hand, our extended static technique may also reduce overheads due to the its tailored PSG. In particular, as the PSG used by our extended static technique is more sparse, our extended static technique may reduce overheads during calculating fault-observing capability and selecting oracle data. Furthermore, the cost of constructing a PSG is also reduced as the program under test can be viewed as being tailored. We further investigate the cost of the basic static and extended static techniques in Section 4.7.2.

4. EXPERIMENTAL STUDY

Our study addresses the following research questions.

- **RQ1**: How do different configurations (i.e., α and f_p) impact the effectiveness of SODS?
- **RQ2**: How does SODS compare to the existing dynamic approaches in terms of both effectiveness and efficiency?
- **RQ3**: How does the number of selected oracle data influence the effectiveness of SODS?

4.1 Implementation and Supporting Tools

When implementing our SODS approach, we adopted the static analysis tool Crystal[2] for C subjects and the analysis tool WALA[3] for Java subjects to identify definition-use chains. Moreover, we do not analyze the definition-use chains in libraries and take them as a black box in implementing our SODS approach. In total, we spent two months in implementing the SODS approach. More details are available at the SODS homepage[4].

We also reimplemented MAODS [62] and DODONA [46] due to the lack of existing tool support. MAODS is the first approach to oracle data selection, which runs test inputs against a large number of mutants and selects variables based on the number of mutants distinguished from the original program by these variables. Later, targeting only object-oriented programs, DODONA is proposed to select oracle data via analyzing the network centrality metrics of variable relationship graph in execution traces. In particular, we use MutGen[5] to generate mutants to implement MAODS, and use Java PathFinder [66] to analyze dynamic dataflow relations to implement DODONA strictly following the prior work [46]. In total, we spent about three months in implementing DODONA and MAODS.

Any of the preceding approaches, including our SODS and the compared DODONA and MAODS, outputs a list of selected oracle data, which can be manually augmented with expected outputs to construct complete test oracles. The same as the prior work [62, 46], testers can construct a complete test oracle for Java subjects by adding an *assertEqual* call for each oracle data in each given JUnit test, and construct program oracles for C subjects since its tests are actually system tests. Furthermore, following prior work [46], if a selected oracle variable is unstable (e.g., a random variable that may yield different values during different executions of the same test input), we removed it from the set of selected oracle data.

4.2 Subjects, Tests, and Faults

We used two C subjects and nine Java subjects in this study. The two C subjects are two unix utilities available at Software-artifact Infrastructure Repository (SIR) [1]. The nine Java subjects have been widely used in the literature of software testing [75, 32], including the prior work on oracle data selection [46]. Each subject has a test suite accumulated during software development. The tests for the C subjects are system tests, which are actually test inputs without test oracles. The tests for the Java subjects are JUnit tests, which consist of test inputs and assertions on test outputs. To evaluate fault-detection effectiveness of selected oracle data, we ignore these original assertions by using the ASM bytecode manipulation framework. Table 1 presents the basic information of these subjects, where Columns 3-6 present the LOC without libraries and test code, the total candidate oracle data number[6], test number, and test coverage. Following prior work [46], we removed the tests incuring compilation errors in Java PathFinder; the test coverage for C/Java subjects was collected using Gcov/EclEmma[7].

In the literature, mutation testing has been shown to be effective in simulating real faults for software testing experiments [40, 3]. So do the prior work on oracle data selection [46, 62]. Therefore, we also constructed faulty programs using mutation testing [76]. In particular, we used MutGen [3]/Major [41] to generate all mutants for C/Java subjects. Similar with the prior work [46], for each subject

[2]https://www.cs.cornell.edu/projects/crystal/
[3]http://wala.sourceforge.net
[4]https://github.com/JunjieChen/sods
[5]http://www.csd.uwo.ca/faculty/andrews/software/mutgen.zip
[6]Only candidate oracle data in the source code are considered here.
[7]http://www.eclemma.org/download.html.

Table 1: Subjects

Subject	Version	LOC	#OData	#Test	Coverage
Gzip	1.1.2	5,680	1,507	214	66.8%
Flex	2.4.7	10,459	1,800	525	77.4%
CSV [16]	1.1	1,382	2,233	208	88.7%
JDKIM [37]	0.2	1,777	2,600	61	48.9%
JCT [36]	1.7	1,808	2,889	148	88.5%
CLI [14]	1.2	1,978	2,871	191	94.1%
TAM [64]	0.5.1	2,372	4,643	233	87.9%
JMIME [39]	0.7.2	4,439	6,112	214	81.1%
CIO [13]	2.4	8,839	12,390	845	78.3%
DIG [18]	3.3.2	9,917	6,219	178	70.5%
JGT [38]	0.9.0	12,978	16,308	190	71.7%
Total	—	61,629	59,572	3,007	—

Table 2: Strategies of our static approach

Approach	SODS(A/B)_X							
	1	2	3	4	5	6	7	8
α	0	0	0.25	0.25	0.5	0.5	1	1
f_p	1	d	1	d	1	d	1	d

program, we constructed 20 mutant groups, each of which consists of 40 different randomly selected mutants, i.e., 800 mutation faults in total for each subject.

4.3 Independent Variables (IVs)

We consider the following independent variables.

IV1: Oracle Data Selection Approaches. We compare our SODS approach with two state-of-the-art dynamic approaches, MAODS [62] and DODONA [46].

IV2: SODS Configurations. We consider various configurations of our SODS approach by varying the values of the two parameters α and f_p. Table 2 summarizes the sixteen strategies of our static approach with various values on α and f_p. For ease of representation, we use SODSA_X /SODSB_X (X=1, 2,. . . , 8) to denote the strategies of our basic/extended static technique with various settings on α and f_p. For α, we assign its value as 0, 0.25, 0.5, 1 respectively, and for f_p, we assign a differentiated value (abbreviated as d in this table) or an uniform value (i.e., 1).

IV3: Sizes of Selected Oracle Data. This variable concerns with the number of oracle data selected when constructing a test oracle. Similar with previous work [62, 46], we use 10 oracle data as the default configuration. In addition, in practice, testers typically construct 1-10 assertions for each test input [24]. Therefore, we also investigate whether the comparison results between our approach and the existing approaches are influenced by different sizes of selected oracle data from 1 to 10.

4.4 Dependent Variables (DVs)

The dependent variables considered in our study consist of the rate of faults detected by the selected oracle data and the total time in oracle data selection. The former measures the effectiveness, whereas the latter measures the efficiency.

4.5 Experimental Process

First, we applied our basic technique to each subject, our extended technique to each Java subject, the dynamic approach MAODS to each C subject[8], and the dynamic approach DODONA to each Java subject[9], recording the oracle data selection order and the total time spent on oracle data selection.

[8]We did not apply MAODS to Java subjects because prior work has shown that DODONA is more effective than MAODS for Java programs [46]. In particular, for each C subject, we generated mutants using MutGen and then fed all the mutants and the whole test suite to MAODS.

[9]We did not apply DODONA to C subjects because it was designed for object-oriented programs [46].

To evaluate the fault-detection effectiveness of selected oracle data, we applied each test input with the selected oracle data to the faulty programs, recording the faults detected by each oracle data. In particular, we first obtained its expected value for the corresponding test input by recording its actual value through code instrumentation during the original program execution. Then we checked whether the actual value of this oracle data on the faulty program is the same as the corresponding expected value. If not, this oracle data detects the corresponding fault. For each subject, based on the faults that each oracle data detects, we calculate the rate of faults detected by the set of oracle data selected by each approach on 20 different mutant groups (i.e., 800 faults).

All the experiments were conducted on a workstation with Intel E5504 Quad-Core Processor 2.0GHz and 100G memory, running Ubuntu 12.04.

4.6 Threats to Validity

The main threat to construct validity is concerned with the metrics used to evaluate the effectiveness and efficiency of the studied approaches. To reduce this threat, we used the widely used metrics in software testing, i.g., the rate of fault detection as well as the approach overhead.

The threats to internal validity mainly lie in the implementations of the dynamic and static approaches. Since the implementation and raw experimental data of MAODS/DODONA are not accessible, with the aid of their authors, we strictly followed the prior work, and also ensured that our implementation produced similar results with prior work [62, 46]. To reduce the threat of implementing our static approach, we used existing static analysis tools (i.e., Crystal and WALA). However, we introduced the following simplifications when implementing our approach. First, we assumed that every branch within a conditional statement has the same execution probability because it is extremely hard to predict the execution probabilities of branches statically and precisely. Second, aliasing is not handled. Third, each array variable, object variable, heap variable, or member variable of a class, is treated as a single variable in constructing the set of candidate oracle data. In the future, we will further improve these simplifications by adopting more advanced analysis techniques.

The threats to external validity mainly lie in the used subjects, faults, and tests. Although we used more subjects than the prior work [62, 46] and the subjects are widely used in the literature of software testing, they may not be representative of other programs. Although mutation faults have been demonstrated to be reasonable in the evaluation of testing techniques [40, 3], mutation faults may still not be representative of real faults in practice. To reduce these threats, we will conduct experiments on practical programs with real faults in the future. Furthermore, another external threat lies in the used tests – following the DODONA work, we removed some tests that incur compiling errors on Java PathFinder.

4.7 Results and Analysis

4.7.1 RQ1: Configuration Impacts

Table 3 presents the average fault-detection rate of SODS with various values of α and f_p for a given oracle data set. The results in this section are based on 10 selected oracle data. Rows 3 to 13 present the average fault-detection rate of our static approach for each subject respectively,

Table 3: Average rate of faults detected by the oracle data selected by SODS (%)

Approach	SODSA_X								SODSB_X							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Gzip	38.3	38.3	39.9	37.5	38.8	36.5	**43.5**	**43.4**	—	—	—	—	—	—	—	—
Flex	20.5	20.5	20.4	**21.0**	20.4	**21.1**	20.5	**21.0**	—	—	—	—	—	—	—	—
CSV	34.9	34.9	38.4	37.8	38.4	37.8	**39.3**	**39.3**	32.9	32.9	39.5	37.8	39.5	37.8	**40.4**	**40.1**
JDKIM	**22.3**	**22.3**	20.9	19.0	19.0	19.0	19.4	19.4	**24.6**	**24.6**	22.0	**24.6**	22.0	22.0	22.0	22.0
JCT	**45.3**	**45.3**	35.5	35.9	35.8	35.8	35.6	35.9	**50.6**	**50.8**	38.0	35.8	37.8	35.8	37.9	37.9
CLI	60.4	**61.6**	60.9	**61.6**	59.8	**62.0**	55.9	55.9	**61.3**	**61.3**	58.6	**61.6**	57.5	61.1	53.4	53.4
TAM	**52.9**	**52.9**	47.6	**54.5**	52.4	**52.9**	45.3	45.3	53.0	53.0	47.3	**53.3**	47.8	**55.4**	50.5	50.5
JMIME	**48.5**	**48.5**	47.8	43.4	44.3	43.5	31.9	31.9	**49.9**	**49.9**	43.3	48.1	42.6	43.3	32.9	32.9
CIO	29.6	29.6	27.5	31.1	**32.4**	**33.3**	29.3	29.3	30.0	30.0	**31.0**	**33.5**	19.4	28.6	13.9	13.9
DIG	**30.0**	**30.0**	27.6	29.8	27.6	27.6	28.6	28.6	**34.6**	**34.6**	27.3	27.0	27.3	24.3	26.0	26.4
JGT	42.1	42.1	**47.4**	44.9	**46.4**	**46.4**	45.8	45.8	45.0	45.0	**46.9**	46.1	**46.4**	45.5	44.5	44.5
Avg.	**38.6**	**38.7**	37.6	37.9	37.7	37.8	35.9	36.0	**42.4**	**42.5**	39.3	40.9	37.8	39.3	35.7	35.7

Table 4: Overall fault-detection rates (%)

Subject	MAODS/DODONA		SODSA_2		SODSB_2	
	Avg.	SD.	Avg.	SD.	Avg.	SD.
Gzip	38.5	7.09	38.3	7.26	–	–
Flex	18.1	7.73	20.5	8.68	–	–
Avg.	28.3	–	29.4	–	–	–
CSV	43.6	7.09	34.9	7.45	32.9	7.66
JDKIM	23.4	4.46	22.3	6.53	24.6	6.19
JCT	44.0	8.33	45.3	8.46	50.8	8.03
CLI	60.0	7.30	61.6	8.24	61.3	8.13
TAM	51.9	9.17	52.9	7.88	53.0	8.05
JMIME	39.9	9.78	48.5	7.63	49.9	9.23
CIO	21.6	8.04	29.6	7.96	30.0	6.59
DIG	35.6	8.66	30.0	6.93	34.6	7.04
JGT	25.9	7.62	42.1	6.19	45.0	6.18
Avg.	38.4	–	40.8	–	42.5	–

and the last row presents the average fault-detection rate of our static approach for all the subjects. In particular, as SODSB targets only Java subjects, its results for the C subjects are not available. The bold numbers in Columns 2 to 9 (Columns 10 to 17) represent the top 2 largest average fault-detection rate of SODSA (SODSB) for each subject.

From Table 3, the average fault-detection rate of SODSA_2 is mostly larger than the other strategies of SODSA. Even when SODSA_2 is not within the top 2 implementations (i.e., for Gzip, Flex, CSV, CIO and JGT), the difference between SODSA_2 and the best technique is at most 5.2%. In summary, for SODSA, the best configuration of the two parameters is that $\alpha = 0$ and $f_p = d$. Similarly, for SODSB, all configurations with $f_p = d$ are usually at least as effective as all configurations with $f_p = 1$, while all configurations with $\alpha = 0$ usually perform better than the other configurations. Thus, for SODSB, the best configuration of the two parameters is also that $\alpha = 0$ and $f_p = d$. This observation indicates that, substitution relationship transfer does not tend to improve the effectiveness of oracle data selection, but considering differentiated impacts of selected oracle data on the unselected oracle data can improve the effectiveness. Through the preceding analysis, the default values of the two parameters are set to be that $\alpha = 0$ and $f_p = d$ (i.e., SODSA_2 and SODSB_2) when applying our static approach for the rest of this paper and practical usage.

4.7.2 RQ2: Comparison with Dynamic Approaches

Table 4 presents overall fault-detection rates for our default static techniques and state-of-the-art dynamic techniques. In the table, "Avg" and "SD" stand for average fault-detection rates and standard deviations. Note that the SODSB_2 technique does not apply to the C subjects. From the table, SODSA_2 is more effective than MAODS for the C subjects on average. For the Java subjects, both SODSA_2 and SODSB_2 are more effective than DODONA on average. More specifically, SODSA_2/SODSB_2 is able to outperform DODONA for 6/7 out of the 9 Java subjects.

To learn whether our static approach is significantly different from the dynamic approaches, we further perform statistical analysis. We first performed the Kolmogorov-Smirnov test [69], whose results demonstrated that the population on

Table 5: Paired sample T test ($\alpha = 0.05$)

Subject	SODSA_2-DODONA	SODSB_2-DODONA	SODSA_2-MAODS	SODSB_2-SODSA_2
Gzip	—	—	0.818	—
Flex	—	—	0.000(+)	—
CSV	0.000(-)	0.000(-)	—	0.065
JDKIM	0.330	0.234	—	0.003(+)
JCT	0.398	0.000(+)	—	0.000(+)
CLI	0.336	0.443	—	0.267
TAM	0.408	0.398	—	0.905
JMIME	0.000(+)	0.000(+)	—	0.037(+)
CIO	0.001(+)	0.000(+)	—	0.724
DIG	0.000(-)	0.402	—	0.000(+)
JGT	0.000(+)	0.000(+)	—	0.000(+)

the faults detected by the selected oracle data follows the normal distribution, which is the precondition of the paired sample T test. Then we performed a paired sample T test (with the significance level α of 0.05) [26] on the raw fault-detection rates of the selected oracle data (i.e., on 20 mutant groups) for each subject. Table 5 lists the p-values of the T test, where "(+)" indicates that the former approach significantly outperforms the latter and "(-)" indicates that the latter approach significantly outperforms the former. In other words, the results without any marks (i.e., "(+)" and "(-)") indicate that the compared approaches have no significant difference. From the table, SODSA_2 and DODONA have no significant difference for four Java subjects (i.e., JDKIM, JCT, CLI and TAM). SODSB_2 and DODONA also have no significant difference for four Java subjects. SODSA_2 and MAODS have no significant difference for Gzip. Generally speaking, for almost half subjects, our static approach has no significant difference with the dynamic approaches.

For the remaining subjects, SODS and the dynamic techniques have significant difference. In particular, SODSA_2 significantly outperforms MAODS for Flex, while MAODS cannot significantly outperform SODSA_2 for any C subject. For Java subjects, SODSA_2 significantly outperforms DODONA for 3 subjects (i.e., JMIME, CIO, and JGT), whereas DODONA significantly outperforms SODSA_2 for 2 subjects (i.e., CSV and DIG). Furthermore, SODSB_2 significantly outperforms DODONA for 4 subjects (i.e., JCT, JMIME, CIO, and JGT), whereas DODONA only significantly outperforms SODSB_2 for CSV. In summary, SODSA_2 is at least as effective as MAODS for all C subjects, while SODSB_2 is at least as effective as DODONA (significantly better than DODONA for 4 of the 9 Java subjects) except for the smallest subject CSV.

Furthermore, as our extended technique aims to improve the effectiveness of our basic static technique for programs tested under the JUnit testing framework, we further compare our basic technique and our extended technique based on the same paired sample T test. From the last column of Table 5, we observe that these two techniques have no significant difference for 4 subjects (i.e., CSV, CLI, TAM and CIO). For all the 5 remaining Java subjects, SODSB_2 significantly outperforms SODSA_2. That is, through tailoring the PSG by removing the definition-use chains that are

Table 6: Execution time (minutes)

Approach	SODSA_X		SODSB_X		MAODS	DODONA
	2	Max	2	Max		
Gzip	0.31	0.34	—	—	10,620.70	—
Flex	0.46	0.53	—	—	11,606.03	—
CSV	0.51	0.51	0.83	0.89	—	35.35
JDKIM	0.47	0.56	0.35	0.40	—	0.57
JCT	0.61	0.63	1.64	1.74	—	1.40
CLI	0.45	0.45	0.32	0.40	—	0.38
TAM	0.73	0.74	0.94	0.99	—	0.63
JMIME	1.05	1.11	0.87	0.91	—	43.45
CIO	3.00	3.20	3.73	3.85	—	192.33
DIG	1.24	1.33	3.19	3.39	—	0.45
JGT	4.07	4.15	9.45	18.27	—	317.30

not covered by tests, our extended static technique further improves the effectiveness of our basic static technique.

Table 6 presents the execution time of the static and dynamic approaches on selecting 10 oracle data. As the space is limited and the execution time for different configurations of our approach is close, we present the time cost of only our default configuration (i.e., X=2), and the maximum time cost among all configurations of our approach. The results show that the cost for DODONA ranges from 0.38 minutes to 317.30 minutes, and the cost for MAODS ranges from 10,620.70 to 11,696.03 minutes. In contrast, the execution time for our default SODSA_2 and SODSB_2 only ranges from 0.31 minutes to 9.45 minutes, and even the cost for the most expensive configuration of our approach only ranges from 0.34 minutes to 18.27 minutes. It is clear that our techniques are much more efficient than MAODS. Comparing our techniques with DODONA on Java subjects, the execution time of our static techniques with default configurations is much smaller than that of DODONA on 4 subjects (i.e., CSV, JMIME, CIO, and JGT), whereas for the remaining Java subjects the execution time of our static techniques is close to that of DODONA. We suspect the reason for the later observation to be that the source code of the latter subjects contains few complex structures such as loops so that it is less time-consuming to run these subjects with tests and analyze the execution information. In general, our SODSA_2 and SODSB_2 techniques cost less than 10 minutes for any Java subject, whereas DODONA is not stable and can cost more than 300 minutes. Therefore, our static techniques clearly demonstrate their superiority to the dynamic approaches in terms of efficiency.

4.7.3 RQ3: Impacts of Selected Oracle Data Size

In this section, we further investigate the effectiveness of all the studied techniques when selecting 1 to 10 oracle data. Table 7 presents the average fault-detection rate of each studied technique for each subject when using different numbers of oracle data[10]. In the table, each column presents the results for different techniques (M, D, SA, and SB denotes MAODS, DODONA, SODSA_2, and SODSB_2, respectively) with different sizes of oracle data set.

From this table, SODSA_2 outperforms MAODS for both C subjects when using more than 7 oracle data. When using no more than 7 oracle data, either SODSA_2 or MAODS outperforms the other only in one of the two C subjects. The reason is that MAODS is directly based on the dynamic effectiveness of oracle data, and tend to be more precise when using a smaller number of oracle data. Surprisingly, on the average of all Java programs, our SODSA_2 and SODSB_2 techniques consistently outperform DODONA for all the 10 different oracle data set sizes, demonstrating the promising future of static oracle selection. For example, when oracle data size is 9, DODONA detects 35.4% faults, while SODSA_2 and SODSB_2 detect 39.7% and 41.5% faults, respectively. Another interesting finding is that SODSA_2 is competitive comparing to SODSB_2 when using $\leq$8 oracle data, further demonstrating the effectiveness of SODS.

[10]The results of "Size=10" are the same with the average fault-detection rates in Table 4.

Table 8: Comparison using same/different test suite

Size=	1	2	3	4	5	6	7	8	9	10
Same	7	7	7	9	9	9	9	9	10	10
Different	4	4	4	5	5	5	5	5	6	6

5. DISCUSSION

5.1 General v.s. Test-Specific

Based on whether oracle data are selected for some specific tests or any tests, oracle data selection can be classified into *general* approaches (e.g., SODSA) and *test-specific* approaches (e.g., MAODS, DODONA and SODSB). General approaches may be more applicable at an early stage of software development without test inputs. For example, in practice, modern code base usually contains a large number of assertions, and the developers typically start to construct the test oracle at an early stage of software development when no test inputs have been given yet. On the contrary, test-specific approaches may be more applicable for any specific test suite consisting of only test inputs.

Furthermore, in software evolution where test suites are continuously refined, it is cost-effective to use general approaches rather than test-specific approaches, because test-specific approaches may be less effective for test suites that are not used in oracle data selection. To verify this hypothesis, we took MAODS as a representative of test-specific approaches and conducted a small experiment on Flex using two randomly constructed test suites, each of which consists of 50 randomly selected test inputs. Then, we fed one test suite to MAODS, and evaluated the oracle data selected by MAODS with this test suite as well as with the other test suite on 40 randomly selected mutants. Table 8 presents the number of faults detected by each oracle data set where "Same"/"Different" denotes the situation where the same/different test suite is used in oracle data selection and evaluation. MAODS performs much worse when different test suites are used in oracle data selection and evaluation. Therefore, in test-suite evolution, general approaches tend to be more effective than test-specific approaches because the latter require testers to re-select oracle data from time to time to keep the selected oracle data effective.

5.2 Should We Go with More Oracle Data?

Section 4.7.3 shows that our static approach is stable, and more effective than dynamic techniques in most cases for various oracle data sizes from 1 to 10. However, it is not clear how the comparison results change in case of even larger oracle data sizes. Therefore, we further investigated the effectiveness of all the studied techniques when selecting 10 to 50 oracle data. Due to the space limitation, we show the results of using 5 to 50 oracle data on two Java subject programs in Figure 3 since the other subjects follow a similar pattern. In each sub-figure, the x axis represents the sizes of selected oracle data, e.g., *s10* denotes using 10 oracle data; the y axis represents the fault-detection rates; the white, gray, and red boxplots represent the DODONA, SODSA_2, and SODSB_2, respectively. We can find that the effectiveness of different oracle data selection techniques tends to become saturate when using larger oracle data sizes. In addition, using more than 10 oracle data cannot provide

Table 7: Impacts of different oracle sizes from 1 to 9

Sub.	Size=1			Size=2			Size=3			Size=4			Size=5			Size=6			Size=7			Size=8			Size=9		
	M/D	SA	SB	M/D	SA	SB	M/D	SA	SB	M/D	SA	SB	M/D	SA	SB	M/D	SA	SB	M/D	SA	SB	M/D	SA	SB	M/D	SA	SB
Gzip	11.8	13.1	–	11.9	14.3	–	34.5	21.3	–	37.4	21.3	–	37.4	24.4	–	37.4	24.4	–	37.4	24.6	–	37.4	38.1	–	37.5	38.3	–
Flex	15.3	13.4	–	16.0	13.8	–	16.0	18.1	–	16.0	18.1	–	16.0	20.4	–	16.0	20.4	–	16.0	20.5	–	17.6	20.5	–	17.6	20.5	–
Avg.	13.6	13.2	–	14.0	14.0	–	25.2	19.7	–	26.7	19.7	–	26.7	22.4	–	26.7	22.4	–	26.7	22.6	–	27.5	29.3	–	27.6	29.4	–
CSV	9.9	14.8	9.9	29.1	16.4	17.9	39.5	19.0	17.9	39.5	24.5	19.3	39.5	25.9	21.9	39.5	27.6	22.1	43.6	32.5	26.3	43.6	34.6	29.8	43.6	34.9	31.1
JDKIM	3.8	9.6	9.6	11.3	9.6	9.6	11.3	9.6	19.0	11.3	9.6	23.5	11.3	18.8	23.5	11.3	18.8	23.5	13.4	18.8	23.5	13.4	19.5	23.5	13.4	22.3	24.3
JCT	4.0	7.6	7.6	42.5	43.9	49.3	42.5	44.5	49.9	42.5	44.5	49.9	42.9	44.5	49.9	43.0	44.5	49.9	43.0	44.5	49.9	43.0	44.5	50.5	44.0	44.5	50.6
CLI	41.4	48.6	20.3	52.6	51.3	48.6	52.8	51.4	51.3	53.0	55.1	53.0	53.0	56.5	53.5	60.0	59.6	57.0	60.0	59.6	57.1	60.0	60.1	59.0	60.0	60.1	61.3
TAM	5.9	12.3	20.3	5.9	32.0	29.4	10.5	33.6	40.6	10.5	33.6	41.9	29.9	33.6	41.9	38.8	39.4	46.9	38.8	39.4	46.9	38.8	46.9	48.4	39.3	51.5	50.8
JMIME	23.4	25.3	25.4	23.4	33.6	33.8	24.3	39.6	35.0	24.3	40.0	35.0	24.3	41.3	35.0	26.1	41.3	41.0	27.0	41.8	44.8	27.0	45.5	45.3	39.9	47.6	48.0
CIO	9.3	0.0	5.4	11.1	4.3	5.8	11.1	16.8	6.3	12.6	22.3	18.8	13.4	22.3	18.8	13.5	25.4	24.3	13.6	25.8	24.6	13.6	26.0	24.9	16.5	26.1	28.1
DIG	12.9	18.6	13.1	12.9	18.6	13.1	12.9	26.1	13.3	12.9	26.3	13.4	35.4	26.3	13.4	35.4	27.8	13.8	35.4	28.1	13.8	35.4	28.1	13.8	35.6	28.1	34.6
JGT	0.0	8.8	8.8	25.0	17.6	40.6	25.0	41.1	41.0	25.0	41.4	41.3	25.0	41.4	44.4	25.0	41.4	44.4	25.9	41.9	44.9	25.9	42.0	44.9	25.9	42.0	45.0
Avg.	12.3	16.2	13.4	23.8	25.3	27.6	25.5	31.3	30.5	25.7	33.0	32.9	30.5	34.5	33.6	32.5	36.2	35.9	33.4	36.9	36.9	33.4	38.6	37.8	35.4	39.7	41.5

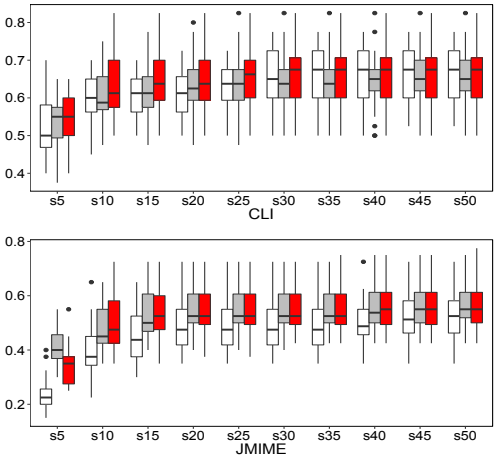


Figure 3: Results for large oracle data sizes

much improvements in fault detection. Therefore, given the costs of manually determining the expected value for each oracle data in practice, our default setting of using 10 oracle data can be cost effective.

5.3 Practical Values of Oracle Data Selection

Shown in a recent survey [5], the oracle costs of human involvement include two aspects: 1) writing test oracles and 2) evaluating test outcomes, and prior work on oracle data selection [62, 46] (same for our work) can effectively reduce efforts for the first aspect by guiding testers to code location/variables when writing test oracles [5]. With our approach, testers can identify a set of oracle data, which may serve as the variables to be observed in test assertions. To create a complete test oracle, testers need to define the expected value for each oracle data. Note that there can be little human involvement in the case of regression testing, since the values of selected oracle data for an old version can be automatically recorded, and then utilized to automatically generate the expected values of oracle data for a new version. In the future, we will further reduce the oracle cost for evaluating test outcomes beyond regression testing.

6. RELATED WORK

Our work is mostly related to oracle data selection. Staats et al. [62, 28] proposed the first dynamic approach based on mutation testing. Later, targeting object-oriented programs, Loyola et al. [46] proposed another dynamic approach, which selects oracle data via analyzing the network centrality metrics of variable relationship graph in execution traces. Similar to oracle data selection, Voas and Miller [67] proposed to identify some positions that traditional software testing can hardly detect faults by mutation analysis like Staats et al. [62] and add assertions at these positions. Our work presents the first static approach to the same oracle data selection problem. Unlike dynamic approaches, our approach does not rely on execution traces, and thus can overcome intrinsic limitations of dynamic approaches.

As the oracle data selected by our approach can be used for constructing test oracles, our work is also related to test oracle generation [45, 27, 61, 65, 51, 6, 49, 47, 63, 2, 58, 79, 78, 72, 9, 10, 43, 70]. Most work in the literature automates test oracle generation based on specifications [55, 8, 54, 20, 19]. Fraser and Zeller [25] proposed μTest to generate assertions (not oracle data) by comparing the trace information of a correct program and its mutants. Furthermore, program invariants generated by various tools (e.g., Daikon [22, 23], DySy [15] and iDiscovery [77]) can also serve as test oracles. Fully automated oracle generation techniques usually either require formal program specifications, or cannot discover faults for the current program version since those techniques summarize the behaviors of the current version as test oracles. Therefore, following existing work for oracle data selection [62, 46], this work aims to support the creation of test oracles, rather than completely generate it.

Besides, our work is also related to testing adequacy criteria based on test oracles. Ken and David [42] defined state coverage criterion, which decides whether all output defining statements are covered by an oracle through investigating program slicing. Schuler and Zeller [59, 60] proposed the concept of checked coverage, which measures the extent to which the code is checked by the test oracle through dynamic slicing. Recently, Zhang et al. [74] utilized oracle-related features (also other features) to predict test adequacy.

7. CONCLUSION AND FUTURE WORK

In this work, we propose the first static oracle data selection approach (SODS) and its extension. The experimental study demonstrates that our static approach is more effective and much more efficient than state-of-the-art dynamic approaches in most cases. For Java programs with JUnit tests, our extension further improves the effectiveness of our basic technique. In the future, we will extend our work by investigating how to help testers determine the expected values of oracle data. Furthermore, as our static approach may also suffer from the intrinsic limitation (e.g., missing dynamic class loading) of static analysis, we plan to further combine the advantageous of dynamic and static approaches.

8. ACKNOWLEDGEMENT

This work is partially supported by the National Basic Research Program of China (973) under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant No. 61421091, 91318301, 61225007, 61529201, 61522201, 61272157. This work is also partially supported by NSF Grant No. CCF-1566589, Google Faculty Research Award, and UT Dallas start-up fund.

9. REFERENCES

[1] http://sir.unl.edu/portal/index.php.

[2] J. H. Andrews. Deriving state-based test oracles for conformance testing. In *WODA*, pages 9–16, 2004.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *OOPSLA*, 31(10):324–341, 1996.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *TSE*, 41(5):507–525, 2015.

[6] L. C. Briand, M. D. Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *TSE*, 30(11):770–793.

[7] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[8] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-checking oracles from intrinsic software redundancy. In *ICSE*, pages 931–942, 2014.

[9] W. Chan, S. Cheung, J. C. Ho, and T. Tse. Reference models and automatic oracles for the testing of mesh simplification software for graphics rendering. In *COMPSAC*, pages 429–438, 2006.

[10] W. K. Chan, S.-C. Cheung, J. C. Ho, and T. Tse. Pat: A pattern classification approach to automatic reference oracles for the testing of mesh simplification programs. *JSS*, 82(3):422–434, 2009.

[11] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. Test case prioritization for compilers: A text-vector based approach. In *ICST*, pages 266–277, 2016.

[12] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. An empirical comparison of compiler testing techniques. In *ICSE*, pages 180–190, 2016.

[13] Commons IO homepage. http://commons.apache.org/proper/commons-io.

[14] Commons CLI homepage. http://commons.apache.org/cli.

[15] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008.

[16] Commons CSV homepage. http://commons.apache.org/proper/commons-csv.

[17] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, 1995.

[18] Commons Digester homepage. http://commons.apache.org/proper/commons-digester.

[19] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *FSE*, pages 106–117, 1996.

[20] L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *FSE*, pages 140–153, 1994.

[21] M. J. Dinneen, G. Gimel'farb, and M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*. Creative Commons License, 2013.

[22] M. D. Ernst. *Dynamically discovering likely program invariants*. PhD thesis, University of Washington, 2000.

[23] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *SCP*, 69(1):35–45, 2007.

[24] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *ISSTA*, pages 291–301, 2013.

[25] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *TSE*, 38(2):278–292, 2012.

[26] J. E. Freund and G. A. Simon. *Modern elementary statistics*, volume 256. Prentice-Hall Englewood Cliffs, New Jersey, 1967.

[27] K. Frounchi, L. C. Briand, L. Grady, Y. Labiche, and R. Subramanyan. Automating image segmentation verification and validation by learning test oracles. *IST*, 53(12):1337–1348, 2011.

[28] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl. Automated oracle data selection support. *TSE*, 41(11):1119–1137, 2015.

[29] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *ISSTA*, pages 302–313, 2013.

[30] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *OOPSLA*, pages 108–124, 1997.

[31] D. Hao, L. Zhang, M.-H. Liu, H. Li, and J.-S. Sun. Test-data generation guided by static defect detection. *JCST*, 24(2):284–293, 2009.

[32] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test-case prioritization approach. *TOSEM*, 24(2):10, 2014.

[33] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *TOPLAS*, 16(2):175–204, 1994.

[34] M. Höschele, J. P. Galeotti, and A. Zeller. Test generation across multiple layers. In *SBST*, pages 1–4, 2014.

[35] J. Hummel, L. J. Hendren, and A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *IPDPS*, pages 208–216, 1994.

[36] Joda Convert homepage. http://www.joda.org/joda-convert.

[37] James jDKIM homepage. http://james.apache.org/jdkim.

[38] JGraphT homepage. http://jgrapht.org.

[39] James mime4j homepage. http://james.apache.org/mime4j.

[40] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014.

[41] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *ASE*, pages 612–615, 2011.

[42] K. Koster and D. Kao. State coverage: a structural test adequacy criterion for behavior checking. In *FSE*, pages 541–544, 2007.

[43] B. P. Lamancha, M. Polo, D. Caivano, M. Piattini, and G. Visaggio. Automated generation of test oracles using a model-driven approach. *IST*, 55(2):301–319, 2013.

[44] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. *Introduction to algorithms*. MIT press, 2001.

[45] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem. *TSE*, 40(1):4–22, 2014.

[46] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel. Dodona: Automated oracle data set selection. In *ISSTA*, pages 193–203, 2014.

[47] P. McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *GECOO*, pages 1689–1696, 2009.

[48] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *TSE*, 38(6):1258–1275, 2012.

[49] C. D. Nguyen, A. Marchetto, and P. Tonella. Automated oracles: An empirical study on cost and effectiveness. In *FSE*, pages 136–146, 2013.

[50] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Kluwer Academic Publishers, 1999.

[51] C. Pacheco and M. D. Ernst. Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.

[52] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[53] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *TSE*, 20(5):385–403, 1994.

[54] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In *ISSTA*, pages 58–65, 1994.

[55] D. J. Richardson. TAOS: Testing with analysis and oracle support. In *ISSTA*, pages 138–153, 1994.

[56] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *ASE*, pages 23–32, 2011.

[57] R. Rugina, M. Orlovich, and X. Zheng. Crystal: A program analysis system for C. *URL: http://www.cs.cornell.edu/projects/crystal*, 2006.

[58] P. J. Schroeder, P. Faherty, and B. Korel. Generating expected results for automated black-box testing. In *ASE*, pages 139–148, 2002.

[59] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *ICST*, pages 90–99, 2011.

[60] D. Schuler and A. Zeller. Checked coverage: an indicator for oracle quality. *STVR*, 23(7):531–551, 2013.

[61] S. R. Shahamiri, W. M. N. W. Kadir, S. Ibrahim, and S. Z. M. Hashim. An automated framework for software test oracle. *IST*, 53(7):774–788, 2011.

[62] M. Staats, G. Gay, and M. P. E. Heimdahl. Automated oracle creation support, or: How I learned to stop worring about fault propagation and love mutation testing. In *ICSE*, pages 870–880, 2012.

[63] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *ICSE*, pages 391–400, 2011.

[64] Time and Money homepage. http://sourceforge.net/projects/timeandmoney.

[65] W.-T. Tsai, P. Zhong, J. Balasooriya, Y. Chen, X. Bai, and J. Elston. An approach for service composition and testing for cloud computing. In *ISADS*, pages 631–636, 2011.

[66] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *ASEJ*, 10(2):203–232, 2003.

[67] J. M. Voas and K. W. Miller. Putting assertions in their place. In *ISSRE*, pages 152–157, 1994.

[68] WALA: T. J. Watson Libraries for Analysis. http://wala.sourceforge.net/.

[69] R. Wilcox. Kolmogorov–smirnov test. *Encyclopedia of biostatistics*, 2005.

[70] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *TOSEM*, 16(1):4, 2007.

[71] T. Xie, L. Zhang, X. Xiao, Y.-F. Xiong, and D. Hao. Cooperative software testing and analysis: Advances and challenges. *JCST*, 29(4):713–723, 2014.

[72] Y. Xiong, D. Hao, L. Zhang, T. Zhu, M. Zhu, and T. Lan. Inner oracles: Input-specific assertions on internal states. In *FSE*, pages 902–905, 2015.

[73] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations. In *ASE*, pages 701–712, 2014.

[74] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang. Predictive mutation testing. In *ISSTA*, page To appear, 2016.

[75] J. Zhang, M. Zhu, D. Hao, and L. Zhang. An empirical study on the scalability of selective mutation testing. In *ISSRE*, pages 277–287, 2014.

[76] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *ISSTA*, pages 331–341, 2012.

[77] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *ISSTA*, pages 362–372, 2014.

[78] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, 2011.

[79] W. Zheng, H. Ma, M. R. Lyu, T. Xie, and I. King. Mining test oracles of web search engines. In *ASE*, pages 408–411, 2011.