ORBS and the Limits of Static Slicing

David Binkley†, Nicolas Gold∗, Mark Harman∗, Syed Islam∗, Jens Krinke∗, and Shin Yoo∗
†Loyola University Maryland, 4501 N. Charles St., Baltimore, MD 21210-2699, USA
∗University College London, Gower Street, London, WC1E 6BT, UK

Abstract—Observation-based slicing is a recently-introduced, language-independent, slicing technique based on the dependencies observable from program behaviour. Due to the well-known limits of dynamic analysis, we may only compute an under-approximation of the true observation-based slice. However, because the observation-based slice captures all possible dependence that can be observed, even such approximations can yield insight into the limitations of static slicing. For example, a static slice, S that is strictly smaller than the corresponding observation based slice is guaranteed to be unsafe. We present the results of three sets of experiments on 12 different programs, including benchmarks and larger programs, which investigate the relationship between static and observation-based slicing. We show that, in extreme cases, observation-based slices can find the true static minimal slice, where static techniques cannot. For more typical cases, our results illustrate the potential for observation-based slicing to highlight unsafe static slices. Finally, we report on the sensitivity of observation-based slicing to test quality.

I. INTRODUCTION

One of the foundational scientific principles of source code analysis is the dichotomy between static and dynamic program analysis. Underpinning much of the work on static analysis is the concept of a ‘safe’ (conservative) over-approximation to static truth; by over-approximating statically determinable information, a (safe) static analysis seeks to provide safe technical conclusions to its end user (usually a software engineer). In this paper we seek to question this foundational assumption of static analysis, and to provide one technical approach for investigating the limits of static analysis.

We believe it is important to question whether there can ever exist a truly conservative and safe approximation to static truth about computation; observation based dynamic analysis is one potential way in which this important foundational scientific question can be investigated. Specifically, we study dependence analysis as realised by program slicing.

It is known to be a challenge to construct a safe conservative static slicing technique, since the slicer has to take account of the full semantics of the programming language. Much work has focused on the theoretical foundations of slicing, including correctness proofs of safe slicing algorithms and the source code analysis upon which they rest [24], [33], [40], [45]. Unfortunately, program language semantics are necessary but insufficient to capture all the possible dependencies that can arise during computation, because dependencies may arise through various interactions only partially under the control of the program itself. For example, a program p may output to some device, the state of which subsequently affects some subsequent computation in some entirely different program, which, transitively, affects p.

It seems unlikely that any static analysis, no matter how powerful, could capture all such dependencies. However, an observation-based analysis can, in principle and by definition, capture any and all observable dependencies, no matter how subtle, no matter how platform and context dependent and no matter how convoluted their transitive chain of causes may be. As such, an observation-based analysis makes a natural complementary counterpart to static analysis.

We focus on program slicing, because it has many applications, including re-engineering [11], maintenance [19], [21], debugging [32], [49], testing [8], [23], refactoring [17], reuse [4], [12], and comprehension [15], [30], [43]. Static slicing also illustrates many issues that affect attempts to construct a ‘safe’ static analysis. However, the possibility of using observation-based analysis as a complement to static analysis could be extended and applied to other static analyses, not merely program slicing.

We also chose program slicing because it has mature, safe, over-approximation algorithms, that are widely-used and implemented in both commercial and prototype research software tools. We also have available a recently-produced implementation, ORBS [9], of an observation-based slicing tool. ORBS speculatively deletes lines of code as part of its computation algorithm, attempting to remove one or more statements (lexically), compiling the result (if possible), and then executing using a suite of inputs (taken from test cases). If the execution is successful (the slice produces the same output as the original), then the reduced (sliced) program is a valid observation-based slice with respect to the criteria and input suite. Further reduction is then attempted until it is not possible to remove any further statements (technical details are provided in section II-E).

Observation-based slices are related to dynamic slices, but there is a critical difference: Observation based slices are based on observed dependencies, rather than the statically determined but dynamically occurring dependencies used by dynamic slicing. That is, a dynamic slice contains a statement if a (statically determined) dependence occurs during some execution. By contrast, an observation-based slice contains a statement, s, if a dependence is observed in which statement s affects the slicing criterion.

Although dynamic slicing only considers a dependence important if it occurs during some execution, because that dependence is, itself, computed statically, it may be one that simply cannot be observed. Furthermore, a dependence that can be observed in some execution may not correspond to any statically determined dependence. Such ‘observation only’ dependencies will be (wrongly) ignored by both dynamic and static slicing, potentially leading to incorrect slices.
In the case of dynamic slicing this under-approximation is not a problem, because dynamic slices are inherently under-approximations in any case. However, one of the primary virtues of static slicing is that the slices it produces are supposed to be safe, conservative (over) approximations to the true slice: if a statement is deleted by a static slicer then it is claimed that no possible execution could cause that statement to affect the slicing criterion. This is what it means to be ‘safe’ in the context of static slicing.

As we shall see in this paper, this belief in safe conservative static slicing, rests on extremely shaky foundations; we will give examples where observation-based slicing highlights unsafe static slices (constructed using a supposedly conservative over-approximating algorithm). This is one of the primary contributions of the paper. It illustrates, succinctly, the way in which observation-based analysis has the potential to highlight issues in existing static analysis. The other primary contributions are an investigation of the potential for observation-based techniques to produce minimal slices (in special circumstances), and the impact of test suite size on the quality of observation-based dependence analysis.

The remainder of the paper is structured as follows. Section II presents basic definitions, including the ORBS algorithm [9], section III presents the research questions followed by the results in section IV. Related work is discussed in section V and section VI concludes.

II. SLICING DEFINITIONS

Traditional program slicing can be classified as either static or dynamic. This section describes how observation-based slicing differs from the traditional forms of static and dynamic slicing.

A. Static Slicing

Static slicing [48] seeks to find an executable subset of program statements that will exhibit the same behaviour for a specified variable at a specified location (i.e. a slicing criterion) as for the original program for all possible inputs. Weiser’s formal definition [48] is as follows:

Static Slice: A static slice $S$ of a program $P$ on a slicing criterion $C$ is any executable program with the following two properties:

1) $S$ can be obtained from $P$ by deleting zero or more statements from $P$.
2) Whenever $P$ halts on input $I$ with state trajectory $T$, then $S$ also halts on input $I$ with state trajectory $T'$, and $\text{PROJ}_C(T) = \text{PROJ}_C(T')$, where $\text{PROJ}_C$ is the projection function associated with criterion $C$.

It is interesting to note that, while Weiser’s original definition of program slicing is based on statement deletion, most work on static slicing focus on dependency analysis to determine which statements can not be deleted.

B. Dynamic Slicing

Dynamic slicing [28] preserves the behaviour of the program only with respect to a specific input. Most work on dynamic slicing (e.g., the work of Agrawal and Horgan [1]) only describes their approach towards implementing dynamic slicing instead of giving a formal definition of what dynamic slicing is. Here, we present a generalized definition of dynamic slicing that extends Weiser’s definition of static slicing with specific inputs. This definition is similar to Korel and Laski’s [28] definition:

Dynamic Slice: A dynamic slice $S$ of a program $P$ on a slicing criterion $C$ and for inputs $I$ is any executable program with the following two properties:

1) $S$ can be obtained from $P$ by deleting zero or more statements from $P$.
2) Whenever $P$ halts on input $I$ from $I$ with state trajectory $T$, then $S$ also halts on input $I$ with state trajectory $T'$, and $\text{PROJ}_C(T) = \text{PROJ}_C(T')$, where $\text{PROJ}_C$ is the projection function associated with criterion $C$.

The criterion for a dynamic slice can concern either the value of variable $v$ at location $l$ only for the $i$th occurrence in the trajectory, denoted $(v_i, l, I)$, or all occurrences of $v$ in the trajectory, denoted $(v, l, I)$.

While dynamic slicing introduces a specific program input to slicing, its computation still relies on statically computed dependency between variables and statements (so that the influence of a statement on a variable can be determined).

C. Observation-Based Slicing

Observation-based Slicing is a recently introduced alternative to program slicing [9]: rather than relying on dependency analysis to only perform the allowed deletions, observation-based slicing deletes a statement in a file of interest, executes the program with a given input suite, and observes whether the projected trajectory of the criterion variable changes. If the trajectory changes, the statement cannot be deleted; if it does not change, the statement can be deleted. Consequently, it preserves the relevant part of the state trajectory from the execution of the original program $P$. The formal definition of observation-based slicing is as follows:

Observation-Based Slice: An observation-based slice $S$ of a program $P$ on a slicing criterion $C = (v, l, I)$ composed of variable $v$, line $l$, and set of inputs $I$, is any executable program with the following properties:

1) The execution of $P$ for every input $I$ in $I$ halts and produces a sequence of values $V(P, I, v, l)$ for variable $v$ at line $l$.
2) $S$ can be obtained from $P$ by deleting zero or more statements from $P$.
3) The execution of $S$ for every input $I$ in $I$ halts and produces a sequence of values $V(S, I, v, l)$ for variable $v$ at line $l$.
4) $\forall I \in \mathcal{I} V(P, I, v, l) = V(S, I, v, l)$.

Sequences $V$ are called trajectories for the criterion $(v, l, I)$. This can be produced by injecting a statement that records the value of $v$ to a file, just before line $l$. These values should be comparable across attempts to delete different statements, somewhat limiting the scope of what is observable (e.g. objects should be serialisable). On the other hand, note that the concept of ‘statement’ can be entirely language independent. Previous work has shown that, by deleting ‘lines’ in source files rather than program statements, it is possible to slice multi-language systems [9].
D. Minimal Slices

According to the definitions, the original program is a static, dynamic, and observation-based slice of itself. The aim for any implementation is to produce slices that are as small as possible but still a valid slice. A slice is considered to be minimal, if it is not possible to remove any number of statements from the slice so that the result is still a slice. If the input set \( \mathcal{I} \) is the set of all possible inputs, then the minimal static slice, the minimal dynamic slice and the minimal observation-based slice are all the same. An implementation for observation-based slicing will therefore compute static-equivalent slices for an input set consisting of all possible inputs.

Almost all implementations for static and dynamic slicing do not conform to the above definitions. The reason is that they usually identify statements that should be in the slice but they don’t actually produce executable programs.

E. ORBS

The current implementation of observation-based slicing, ORBS, continuously attempts to delete increasingly longer sequences of lines, starting from each line in the source file [9]. It increases the number of lines to be deleted together, up to the size of the so called ‘deletion window’. This is because certain lines can only be deleted simultaneously (e.g. opening and closing brackets on successive lines). If the attempt results in an observation-based slice, the lines in the deletion window are deleted, and kept otherwise, after which ORBS moves the starting position of the deletion window by one line and repeats until it reaches the end of the source file. This forms a single iteration of ORBS.

As long as the previous iteration deleted some lines, ORBS starts a new iteration, this is because certain lines become deletable only after other lines have been deleted (e.g. a loop predicate can only be deleted after the loop body is emptied). When no lines are deleted from the last iteration, ORBS terminates. The result is a \( l \)-minimal slice, i.e., it is not possible to delete any other line from the slice. It may still be possible to delete a combination of \( n \) lines, i.e., the result is not necessarily \( n \)-minimal.

To validate deletion of a set of lines, ORBS attempts to compile and execute the slice candidate with deletion in question applied. If the deletion results in compilation errors, it cannot produce a correct executable slice. Similarly, if the deletion produces an executable slice that produces a different trajectory from the original program, it cannot be a correct observation-based slice.

III. Research Questions

In prior work [9], we demonstrated that the ORBS approach to computing multi-language slices was feasible. We also compared the resulting slices with various forms of dynamic slices, all of which are ‘algorithmic cousins’ of observation-based slicing, because all share roots in dynamic analysis. In this paper, we study the relationship between observation based slicing and static slicing.

Our experiments concern 12 programs, split into three sets, each of which is specifically chosen to help us address each of the three research questions. The first research question concerns the performance of observation based slicing on benchmarks that have previously been used to exemplify static slicing challenges in the literature. For this research question we use three widely studied (tiny) benchmark programs.

Our second research question focuses on the way observation-based slicing can highlight unsafe static slices. For this research question, we use seven programs from the Siemens Suite of (relatively small) C programs, which have been widely-studied in program analysis and testing research. These programs are large enough to be non-trivial, yet small enough to allow us to establish the ground truth for dependence, thereby facilitating the comparison between observation-based and static slicing.

Finally, our third research question concerns the inherent sensitivity of observation based slicing to the test cases used. For this research question we used two larger programs, since we do not need to establish the ground truth dependence, but merely the effect of test adequacy on dependence observations. More specifically, we address the following three research questions:

**RQ1: Subtleties and surprises:** Can ORBS find minimal slices for known challenging benchmarks?

Although considering all possible inputs will usually be infeasible, the resulting slice would be a correct static slice because it will have the same behaviour as the original program on the slicing criterion for all possible inputs — the semantic requirement of a static slice. Because ORBS uses observation and deletion, such an observation-based slice would also be a minimal static slice. Of course exhaustive testing is infeasible for all but the smallest programs.

Given the key role the inputs play, the relationship between inputs and slices raises interesting questions. First, for small programs that can be tested by test suites that approach exhaustive testing, can ORBS yield minimal static slices? Of course, we can only ask such a question where we know the ground truth; the identities of all minimal slices. Furthermore, even if ORBS can yield such minimal slices, this will only be interesting if the slicing problems are, themselves, interesting and challenging in some way. Therefore, we select three programs widely used for understanding and explaining the limits of static slicing: wc, (scam) mug, and mbe.

For all three tiny benchmark programs, it is possible to construct a input suite that approximates exhaustive testing (though, we cannot, of course, test exhaustively, even in these cases). That is, there is a (small and finite) input set \( \mathcal{I} \) such that for all supersets \( \mathcal{I}' \supseteq \mathcal{I} \), the slice, ORBS\((v, l, \mathcal{I})\) is the same as ORBS\((v, l, \mathcal{I}')\). In such cases, by the definition of observation-based slicing, ORBS\((v, l, \mathcal{I})\) must also be a minimal static slice. In general finding such an input set is intractable; however, for the tiny programs, it is possible, even (relatively) straightforward.

The word count program, wc, (shown in Figure 1), computes the number of lines, words, and characters in an input text file. This makes it a good starting point, because its slices are used in so many papers on slicing [20], [41], as trivial examples of static slices. It is implicit in all treatments of this example, that the slices are trivial, and present a few interesting issues, hence its widespread use as an illustrative example. As we
shall see, observation-based slicing reveals that there are, in fact, subtleties, even in this simplest of examples.

The SCAM mug example, mug, shown in Figure 2, appeared on the souvenir mug given to delegates of the first incarnation of the SCAM conference (SCAM 2001) in Florence. It has subsequently being used as a ‘challenge’ example for slicing algorithms [44], due to its subtle semantics and the difficulty in obtaining a minimal slice, even with very sophisticated algorithmic techniques.

The Montréal Boat Example, mbe, shown in Figure 3, was formulated by Sebastian Danič and John Howroyd during a boat excursion at the 2

(SCAM 2002) in Montréal. It was discussed at length at the conference as an example of the subtleties of minimal static slicing [14].

We use these three simple examples to illustrate both the subtleties of minimal slicing, and also the power of observation-based techniques for finding slices in those special extreme cases where testing can be particularly extensive.

**RQ2: Highlighting Unsafe Static Slices.** RQ2 uses the small programs shown in Table I, known as the Siemens Suite which have been widely-studied in previous work on analysis and manipulation [50]. Each of the seven programs comes with its own pre-defined test suite. We can use these to investigate how observation-based slicing differs from traditional static slicing for a set of non-trivial programs using test suites designed by other researchers.

Where the (claimed) static slice fails to contain a statement that is included in the observation based slice produced by ORBS then, _ipso facto_, the static slice _must be unsafe_ (irrespective of the test suite). This is because a statement is only contained in the observation-based slice if there exists an observation of behaviour for which the slicing criterion depends upon it. Therefore, even if the test suite is inadequate (which is the case in all but the most trivial situations), it will have contained at least one test case that exercises the dependence observed. Such a test suite, although not necessarily adequate, will thereby raise a counter example to the safety claim made by the static slice.

We wish to experiment with the potential for observation-based slicing to expose unsafe static slices. However, it would not be reasonable of us to use specially-constructed test suites (as we have done for RQ1), since the effort required to construct such test suites may not justify the potential for exposing unsafe slices. This motivates our choice of the Siemens Suite. Since its test suites are designed without any knowledge of observation-based slicing, they are free from any bias in the selection of test cases. Therefore, they allow us to investigate the kinds of observation that can be made from ‘standard’ test suites, widely used in other research.

The Siemens Suite programs are also sufficiently small that a human can understand and investigate the underlying semantic cause for any differences in the slices constructed by traditional static slicing and those constructed by observation. However,
they are also sufficiently large that they denote nontrivial computation, thereby making these differences interesting and worthy of study.

**RQ3: Observational Sensitivity to Inadequate Testing.** RQ3 studies the impact of the sets of inputs used as test cases to compute an ORBS slice. One interest here is the question of how varying the set of inputs can provide a lower-bound for the corresponding static slice, in much the way that union slicing can [5].

For example, if \( I_1 \subseteq I_2 \) then ORBS \((v, l, I_1)\) is likely a subset of ORBS \((v, l, I_2)\). However, for larger sets, the difference is expected to be smaller because it becomes increasingly more difficult to execute previously unexecuted code. Thus the expected impact of an additional input case diminishes as the input set grows in size.

In this case the slices produced using an ever increasing input suite should approach an asymptotic limit. Furthermore, this asymptote provides a prediction for the lower bound of the corresponding minimal static slice. Observing how slice size monotonically approaches this asymptotic limit allows us to investigate the impact of inadequate testing on observation-based slicing.

For RQ3 we performed a set of experiments on the two larger programs, `ed` and `byacc`, which each come with a test suite. Table II shows some properties of the two systems: `ed` is a line-oriented text editor and `byacc` is Berkeley Yacc.

### IV. Results

In this section we present results, based on slices of each of the three sets of programs, to answer the three research questions.

We constructed all static slices using the widely available tool CodeSurfer, which implements the standard SDG algorithm for (safe) static slice computation [26]. We constructed observation based slices using our tool ORBS [9], for observation-based slicing, using the test suites provided in (with the Siemens Suite programs and the two larger programs for RQ2 and RQ3 respectively).

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>printtokens</td>
<td>733</td>
<td>81</td>
</tr>
<tr>
<td>printtokens2</td>
<td>579</td>
<td>75</td>
</tr>
<tr>
<td>replace</td>
<td>658</td>
<td>309</td>
</tr>
<tr>
<td>schedule</td>
<td>465</td>
<td>58</td>
</tr>
<tr>
<td>schedule2</td>
<td>392</td>
<td>78</td>
</tr>
<tr>
<td>tcas</td>
<td>185</td>
<td>43</td>
</tr>
<tr>
<td>totinfo</td>
<td>415</td>
<td>54</td>
</tr>
</tbody>
</table>

**TABLE II**

Size of the Programs Considered for Research Question 2

<table>
<thead>
<tr>
<th>ed</th>
<th>byacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>8</td>
</tr>
<tr>
<td>Lines</td>
<td>2836</td>
</tr>
</tbody>
</table>

1ORBS produces 1-minimal slices, but there may be multiple 1-minimal slices for the same criterion. Therefore, it may be the case that ORBS \((v, l, I_1) \not\subseteq \text{ORBS}(v, l, I_2)\).
15. when using each of the two input cases independently. In 

(backswards) the slices include Line 7, then Line 5, and finally 

reveals that the value of \( k \) of the while loop.

Multiple times. While multiple executions force the retention 

\( k \) exercises both assignment to \( x \) to take on both of its two possible values.) This single input 

cases. (The reason a single input is insufficient 

given iteration, if \( mbe \) may take an unbounded number of loop iterations. However, 

Line 7 followed by an execution of Lines 11 and 12. Only 

the execution of Line 12 impact the eventual termination and 

the unnecessary control structures.

Finally, consider the program \( mbe \). Similar to \( mug \), the code 

inclusion of a (loop-carried) flow dependence edge from \( c=g() \) to \( q(c) \) as well as a control dependence edge from predicate 

\( q(c) \) to \( x=f() \); thus while traversing these dependence edges (backwards) the slices include Line 7, then Line 5, and finally 

Line 8. In contrast the ORBS slice correctly omits Line 8 

\( c=g() \)).

It is illustrative to consider the slices that ORBS produced 

when using each of the two input cases independently.

In the first case, when \( p(j) \) is always false, the empty program 

is returned. On the other hand, the second input case alone 
generates the single line program \( x = f() \), which correctly omits the 

unnecessary control structures.

However, careful inspection of the program’s semantics 

reveals that the value of \( k \) has no impact on the final value of 

\( j \) in any terminating execution of the program. To see this, 

consider the iterations of the program’s single loop. During a 
given iteration, if \( q(k) \) is true then \( j \) remains unchanged. Thus 

only when \( q(k) \) is false is progress made towards a value of 

\( j \) that makes \( p(j) \) false. Thus in any terminating execution the 

following pattern is repeated: there are zero of more execution 

of Line 7 followed by an execution of Lines 11 and 12. Only 

the execution of Line 12 impact the eventual termination and 

thus the final value of \( j \).

For this example, there is no finite set of inputs that covers 

all possible executions because arriving at the final value of \( j \) 
may take an unbounded number of loop iterations. However, 
a single input case is sufficient to exercise all the possible 

dependence patterns. (The reason a single input is insufficient 

for the \( mug \) example is that two inputs are needed to cause \( x \) 
to take on both of its two possible values.) This single input 
exercises both assignment to \( k \) and also the assignment to \( j \) 
multiple times. While multiple executions force the retention 
of the while loop.

Using any instantiation functions for the unbound functions, 

the ORBS slice of \( mbe \) is shown in Figure 7. This slice 
correctly separates and then omits the computation of \( k \). As 

with \( mug \) the static slice for \( mbe \) computed by standard static 
slicing algorithms, such as the SDG algorithm include the 

entire program.

In summary, for RQ1, for all three programs, ORBS extracts 
a precise (minimal) static slice, illustrating the potential power of 
an observation-based approach to slicing. Although small, 

\( mug \) and \( mbe \) show just how subtle slicing can be and how 

ORBS offers a complementary alternative to traditional static 
slicing when attacking this subtlety.

B. RQ2: Highlighting Unsafe Static Slices

Research question RQ2 considers how ORBS slices compare 

to static slices on a set of small programs (the Siemens 

Suite) where the ground truth can be (manually) determined 

by examining slice differences. We constructed 743 slices in 
total. 698 of these are taken from the Siemens Suite programs 
shown in Table I. For ‘backwards compatibility’ with RQ1 and 

completeness, we also include the tiny benchmark programs, 
from which we constructed 45 slices.

The overall size difference for these 753 slices of the 

programs are graphed in Figure 8. Each point is the percent 

reduction using ORBS minus the percent reduction using 

CodeSurfer. The slice granularity effect illustrated above causes 

the 66 negative differences (where the CodeSurfer slice is 

smaller than the ORBS slice).

Most of the differences come from ORBS removing unreachable 

or untested code and from CodeSurfer’s finer granularity 

when handling parameters where, for example, it can replace 

function declarations such as \( \text{int main(int argc, char** argv) } \) 

with \( \text{int main(int argc) } \) and functions calls such as \( \text{change(pat, sub) } \) 

with \( \text{change(pat) } \). A similar granularity issue is seen when 

the slicer encounters the line \( \text{*prio = *command = -1. } \) While 

CodeSurfer sees this as two separate assignments, ORBS does
not and thus must also retain the computation of a valid address for the pointer 
 prio, to avoid the premature termination of the
 program. Finally, this pattern occurs as CodeSurfer is able to slice
 alt_sep_test() out of fprintf(stdout, "%d\n", alt_sep_test()) while ORBS is not.

In a similar example, the program tcas includes
need_upward_RA = Non_Crossing_Biased_Climb() &&
Own_Below_Thread() on a single line, which forces ORBS
to include the function definition bool Own_Below_Thread()
with an empty body because otherwise the program does not
compile. Operating at a finer level of granularity, CodeSurfer
is able to omit the call to Own_Below_Thread(). In this
case the cost of the inclusion is only the three lines of the empty
function definition. However in the case of the call
change(pat, sub) the unwanted retention of sub forces its computation to
be retained, which involves more then three lines.

These differences are interesting because they illustrate the
way in which ORBS can be used as a sanity check on the
static slice. ORBS can be used to suggest instances where the
static slice can be refined further. For example, where static
conservatism leads to an unnecessarily large slice.

However, of particular interest, and perhaps concern, is the
area of Figure 8 which lies in the negative portion of the
horizontal axis; cases where the SDG slice is smaller than
the observation based slice. In the situation, observation based
slicing may have highlighted an unsafe static slice.

An example of one such unsafe static slice is shown in
Figure 9. Static slices that include the variable ‘command’
will (incorrectly) omit the while loop. The lack of safety, in
this case, come from the way the conservative ‘safe’ SDG
algorithm does not model dependences induced by I/O streams.
Fixing this problem is more challenging than simply updating
the dependence model for fgets. Correctly slicing input (and
output) streams is subtle and challenging [3], and observation-
based slicing has merely highlighted one such instance.

As discussed in the introduction, it is questionable whether any
purely static analysis technique could ever account for all
dependencies between input and output streams, since these
may involve arbitrary ‘real’ communications as a source of
dependence. However, this example from our experiments
highlights the realisation that these issues arise even where the
input–output dependence does not involve complex interactions,
beyond the reach of any conceivable tool.

C. RQ3: Observational Sensitivity to Inadequate Testing

For both systems chosen for RQ3 (ed and byacc), we
experimented with four slicing criteria. For ed, they are:
(A): The value of *addr_cnt in line 186 of file
    main_loop.c
(B): The value of s in line 263 of file io.c
(C): The value of s in line 28 of file signal.c
(D): The value of *s in line 71 of file re.c
For byacc, the criteria are:
(A): The value of k in line 25 of file symtab.c
(B): The value of c in line 25 of file output.c
(C): The value of state in line 252 in file lalr.c
(D): The value of symbol in line 252 in file lalr.c

All eight criteria have been chosen in a way that they are points
a maintainer may be interested in.

The test suite of byacc consists of 10 different grammar files
that are used to test the functionality of byacc. The test suite of
ed consists of 80 different command sequences as input to ed.
From the 80 inputs, we have selected 52 and added three more
(smaller) inputs: (1) an empty command sequence, (2) a single
command to enable error explanations, and (3) a command to
read a file.

For the experiments, we have sorted the inputs by size so that
the sequence of inputs \( T = \{I_1, ..., I_n\} \) is increasing in
size. For each \( k = 1..n \) and for each criteria (A)...(D) we have
computed an observation-based slice for the \( k \) smallest inputs:
\( \mathcal{I}_k = \{I_1, ..., I_k\} \) \( n = 10 \) for byacc and 33 for ed.

Table III shows the resulting sizes for byacc for the 10 different
inputs and the number of nodes in the System Dependence Graph as
generated by CodeSurfer for the complete program, for the static slice computed by CodeSurfer, and the
number of nodes in the System Dependence Graph as
generated by CodeSurfer for the generated ORBS Slice for all ten inputs.

It is not surprising that for the first three smallest input sets the
number of deleted lines does not change: two of the inputs are
identical and the third input is only slightly changed. Running
ORBS with just one of the inputs or with all three inputs will
therefore not change the exercised dependences.

The addition of the next three inputs causes a drop in the
number of deleted lines as the inputs are different and larger

```
get_command(int *command, int *prio, float* ratio)
{
    char buf[CMDSIZE];
    if(fgets(buf, CMDSIZE, stdin))
    {
        *prio = *command = -1;
        sscanf(buf, "%d", command);
        while(buf[strlen(buf)-1] != \"\n\" && fgets(
            buf, CMDSIZE, stdin))
            {
            }
        }
}
```
TABLE IV
NUMBER OF DELETED LINES FOR \textit{ed} AND SIZE OF THE STATIC AND FINAL ORBS SLICE IN SDG NODES

<table>
<thead>
<tr>
<th>Input $I_k$</th>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>2836</td>
<td>2836</td>
<td>2836</td>
<td>2836</td>
</tr>
<tr>
<td>final deletions</td>
<td>1954</td>
<td>864</td>
<td>1943</td>
<td>2654</td>
</tr>
<tr>
<td>SDG Nodes</td>
<td>6605</td>
<td>6606</td>
<td>6628</td>
<td>6628</td>
</tr>
<tr>
<td>Static Slice</td>
<td>5025</td>
<td>5025</td>
<td>5025</td>
<td>5025</td>
</tr>
<tr>
<td>ORBS Slice</td>
<td>2594</td>
<td>5722</td>
<td>2644</td>
<td>505</td>
</tr>
</tbody>
</table>

The last two inputs are much larger than all the previous ones, however, the drop of deletions is very small. Despite the much large size, the added inputs will only exercise a few more dependencies and it can be assumed that the minimal static slice has been approximated by running all inputs.

For the next part of the experiment, the original program has been sliced by CodeSurfer for the four criteria. In addition, the observation-based slices as computed by ORBS over all inputs have been analysed (but not sliced by CodeSurfer). Table III shows the resulting number of nodes in the SDGs. For example, byacc has an SDG consisting of 9556 nodes and the CodeSurfer slice for criterion (A) only has 1429 nodes. The ORBS slice has only 729 nodes! It can be seen that the ORBS slices are much smaller than the CodeSurfer slices, a confirmation that static slices as computed by typical tools are not very precise.

The same experiment has been done for the set of 55 inputs for \textit{ed}. Figure 10 shows the number of deleted lines for the increasing number of $k$ inputs. It can be seen that for the four criteria the increasing number of inputs causes fewer and fewer lines to be deleted. As the first input is empty, the ORBS Slice deletes almost all of the 2836 lines of the program. Most inputs do not cause an execution of the criterion (D) and therefore the ORBS Slice stays empty. Input 24 ($g2.ed$) is the first that causes an execution of (D) and produces a small slice. As the next nine inputs do not cause an execution of criterion (D), the slice does not change. ORBS produces very similar slices for criteria (A) and (C) where the increasing number of inputs cause a steady decrease in the number of deleted lines. The slices for criterion (B) are initially empty (for the first six inputs), but most of the following inputs cause large ORBS slices, resulting in a large reduction in deleted lines for input 7 ($x2.ed$). From there on, the increasing number of inputs causes a steady decrease in the number of deleted lines. Overall, adding more inputs at the end only causes a slight decrease in deleted lines and it can be assumed that the input space is covered sufficiently. Note that the graph shows situations where adding an input increases the number of deleted lines instead of decreasing them, which is due to ORBS finding a different 1-minimal slice which is slightly larger than the one before.

Again, a comparison with static slices as produced by CodeSurfer has been made. Table IV shows the results. Due to the instrumentation for the criteria, the number of SDG Nodes for the original, unsliced, program are slightly different than the first three. The next two inputs are again only slightly different to the previous ones and only a small drop in deletions can be observed.

The static slices as produced by CodeSurfer are the same four all four criteria, they contain around 76% nodes of the original program. The final ORBS slices over all 53 inputs, however, are quite different. Criteria (A) and (C) produce slices of a typical size (39%/40% nodes of the original program), criterion (D) produces a very small slice of only 8% nodes. For criterion (B) something interesting happened: The ORBS slice is actually larger than the CodeSurfer slice, which, according to our argument, should not happen. However, CodeSurfer has a limited model of input/output and therefore does not identify dependences via input/output, e.g., via files. As \textit{ed} is mainly about input/output, a large number of dependences are not identified by CodeSurfer and the resulting slice will not only be imprecise, but incorrect! ORBS does not suffer such problems: As the inputs exercise dependencies via input/output, ORBS will not delete the corresponding lines and therefore produces correct slices, which, as in this case, can be larger than static slices which are produced by tools. This is a typical problem of static analysis which can only be correct within the assumed model which is typically limited. Dynamic analyses have a similar problem, they are also only correct within the assumed model. However, the assumed model is not as limited as the model for static analyses.

For the four criteria (6605 – 6628). The static slices as produced by CodeSurfer is the same four all four criteria, they contain around 76% nodes of the original program. The final ORBS slices over all 53 inputs, however, are quite different. Criteria (A) and (C) produce slices of a typical size (39%/40% nodes of the original program), criterion (D) produces a very small slice of only 8% nodes. For criterion (B) something interesting happened: The ORBS slice is actually larger than the CodeSurfer slice, which, according to our argument, should not happen. However, CodeSurfer has a limited model of input/output and therefore does not identify dependences via input/output, e.g., via files. As \textit{ed} is mainly about input/output, a large number of dependences are not identified by CodeSurfer and the resulting slice will not only be imprecise, but incorrect! ORBS does not suffer such problems: As the inputs exercise dependencies via input/output, ORBS will not delete the corresponding lines and therefore produces correct slices, which, as in this case, can be larger than static slices which are produced by tools. This is a typical problem of static analysis which can only be correct within the assumed model which is typically limited. Dynamic analyses have a similar problem, they are also only correct within the assumed model. However, the assumed model is not as limited as the model for static analyses.

Overall, the above experiments have clearly shown that static slices as produced by tools are not only far away in terms of precision from minimal static analyses, but they can also be incorrect. ORBS, on the other hand, can produce slices much nearer to minimal static slices if the input domain is sufficiently covered.

V. RELATED WORK

Static slicing was introduced by Weiser [46], [47]. Ottenstein and Ottenstein [38] proposed that program slicing can be viewed as a graph reachability problem and noted that the program dependence graph (PDG) was the ideal structure for program slicing. Horwitz et al. [25] introduced an algorithm which extended the idea to slice entire programs (represented as System Dependence Graphs) and later [26] introduced a two-pass static slicing algorithm. This approach remains the most pre-dominantly used and variants are widely researched.

There are many other flavours of static slicing that attempt to reduce the size of the slice. Incremental Slicing [37] allows selection of the type of data dependencies that are to be
included in a slice, by considering that all data dependencies are not of the same importance. Stop-list slicing [18] allows the programmer to define variables that are not of interest. The stop-list variable set is used to purge the dependence graph before computing slices with the standard graph reachability algorithm, causing the slice to be smaller. Barrier Slicing [31] allows the programmer to specify which parts of the program can be traversed when constructing the slice and which parts cannot. A barrier is specified with a set of nodes (or edges) of the PDG that cannot be passed during the graph traversal, also resulting in focused and smaller slices. Results presented here concerning the safety (or otherwise) of supposedly ‘safe’ static slices apply to all these (and other) forms of static slicing.

Amorphous Slicing [22] is an approach to slicing that aims to preserve the semantics of the program but not the syntax. Amorphous slices are constructed using some program transformation which simplifies the program and which preserves the semantics of the program with respect to the slicing criterion. Although ORBS only deletes lines of code, this may cause merging and this could be regarded as a form of (very slightly) amorphous slicing (depending on the precise interpretation of the phrase ‘syntax preserving’).

To the best of our knowledge no other slicing approach follows the observation-based statement-deletion approach used by our ORBS algorithm. The ORBS algorithm [10], is a dynamic form of slicing, but it constructs slices based on dynamically observed dependencies, rather than dynamically occurring (but statically determined) dependence (used in all previous dynamic slicing approaches).

Dynamic slicing is a concept introduced by Korel and Laski [28], [29]. They considered several algorithms to compute dynamic slices based on their definition. In contrast, most later work on dynamic slicing ‘defines’ dynamic slicing based on the algorithms used to compute it (e.g., Agrawal et al. [1] and Demillo et al. [16]). Although many research prototypes and approaches exist [2], [6], [7], [36], [42], [53], [54], all approaches are for a single specific programming language whereas the observation based nature of ORBS allows it to slice programs distracted from multiple programming languages [10].

Of all previous dynamic slicing formulations, the closest to our observation-based slicing is Critical Slicing [16]. However, we have found that critical slices are significantly larger than observation-based slices and are often incorrect [10].

The idea to delete parts of a program or test input is also prominent in applications of delta debugging [13], [51], [52]. As delta debugging can be very expensive, a few approaches have modified the original delta debugging formulation, so that it exploits programming language syntax and semantics. For example, Hierarchical Delta Debugging [35] exploits tree structures in inputs for a tree-based delta debugging approach. Delta [34] uses a separate tool to flatten tree structures found in programs before applying delta debugging. Regehr et al. [39] exploit the syntax and semantics of C for four delta-debugging based algorithms to minimize C programs that trigger compiler bugs.

Jiang et al. [27] presented a forward dynamic slicing approach similar in spirit with ORBS. They mutate the value of the variable at the location as given by the slicing criterion. They then observe the computed values in the state trajectory and the dynamic slice consist of all statements for which the computed values have changed compared to the trajectory of the original program. Jiang et al. compare their approach to traditional dynamic and static slicing to establish the accuracy of their approach.

Union slicing [5] is also related to observation based slicing. Like ORBS, the union slicing algorithm of Beszédes et al. [5] aims to approximate the static slice by dynamic slices for a set of test inputs. It does so by producing the union of the independently-computed dynamic slices for each test case. However, since the union slice is the union of all dynamic slices, it shares the critical difference between dynamic and observation-based slicing: The dependencies considered by union slicing are dynamically occurring (but statically determined) dependencies, rather than dynamically observed dependencies.

VI. Conclusion

Observation-based slicing is a new form of slicing in which dependencies observed during execution are used to construct slices. Previous work has compared observation-based slicing to traditional dynamic slicing. This paper has extended that analysis to compare observation-based slicing to static slicing. We have shown that observation based techniques, when guided by extremely high quality test cases can find static slices inaccessible to traditional static techniques. These include minimal slices of benchmark programs that have previously been used in the slicing literature to highlight static slicing challenges. We have also experimentally demonstrated the potential of observation-based slicing to highlight unsafe static slices. Finally, since the quality of an observation-based slice depends critically on the quality of the test suite used to guide its construction, we experimented to investigate the connection between observation-based slice size and test suite size. Overall, we believe that our results illustrate the way in which observation-based slicing provides a natural complement to traditional static slicing.

References
