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
‡University of East London, University Way, London E16 2RD
§KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 305-701, Republic of Korea

***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 potentially *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 minimal static slice, where static techniques cannot. For more typical cases, our results illustrate the potential for observation-based slicing to highlight limitations in static slicers. 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 aims 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 [1]–[4].

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 later 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 or 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 [5], maintenance [6], [7], debugging [8], [9], testing [10], [11], refactoring [12], reuse [13], [14], and comprehension [15]–[17]. 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 tools. The investigation uses the recently-produced implementation, ORBS [18], of observation-based slicing. 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 program), 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 (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.

©2015 IEEE. To be published in the Proceedings 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015) in Bremen, Germany. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

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, this belief in safe, conservative, static slicing rests on extremely shaky foundations; we give examples where observation-based slicing highlights *un*safe static slices. This is one of the three 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 two 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 [18], 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 these traditional forms of slicing.

A. Static Slicing

Static slicing [19] seeks to find an executable subset of a program's statements that will exhibit the same behaviour for a specified variable at a specified location (i.e., a slicing criterion) as the original program *for all possible inputs*. Weiser's formal definition [19] 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', such that $\mathit{PROJ}_C(T) = \mathit{PROJ}_C(T')$, where 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 uses dependency analysis to determine which statements cannot be deleted.

B. Dynamic Slicing

Dynamic slicing [20] preserves the behaviour of the slicing criterion only with respect to a *specific input*. Most work on dynamic slicing (e.g., that of Agrawal and Horgan [21]) tends to describe an 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 to specific inputs. This definition is similar to Korel and Laski's [20]:

Dynamic Slice: A *dynamic* slice S of a program P on a slicing criterion C *and for inputs* $\mathcal{I}$ 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* $\mathcal{I}$ with state trajectory T, then S also halts on input I with state trajectory T', and $\mathit{PROJ}_C(T) = \mathit{PROJ}_C(T')$.

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 $(v_i, l, \mathcal{I})$, or all occurrences of v in the trajectory, denoted $(v, l, \mathcal{I})$.

While dynamic slicing introduces a specific program input to slicing, dynamic slicing tools still rely on statically computed dependency between statements.

C. Observation-Based Slicing

Observation-Based Slicing is a recently introduced alternative to dependence-based program slicing [18]: rather than relying on dependency analysis to identify allowed deletions, observation-based slicing deletes a statement of interest, executes the program with a given input suite, and observes whether the projected trajectory of the slicing criterion 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. 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 $C = (v, l, \mathcal{I})$ composed of variable v, line l, and set of inputs $\mathcal{I}$, is any executable program with the following properties:

1) The execution of P for every input I in $\mathcal{I}$ 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 $\mathcal{I}$ halts and produces a sequence of values $V(S, I, v, l)$ for variable v at line l.
4) $\forall_{I \in \mathcal{I}} V(P, I, v, l) = V(S, I, v, l)$.

The sequences 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, the

concept of 'statement' can be entirely language independent and by deleting 'lines' from source files rather than program statements, it is possible to slice multi-language systems [18].

D. Minimal Slices

According to the definitions, a 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 the removal of any statements yields an invalid 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 [18]. 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 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. When no lines are deleted from the last iteration, ORBS terminates. The result is a *1-minimal* slice and thus it is not possible to delete any single line from the slice. It may still be possible to delete a combination of n lines; consequently 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 the 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 [18], we demonstrated that the ORBS approach to computing multi-language slices was feasible. We also compared the resulting slices with various forms of dynamic slicing, 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[1] concern 12 programs, split into three sets, each of which is specifically chosen to help 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 inputs 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 static slices for known challenging benchmarks?
Although considering all possible inputs is often infeasible, the resulting ORBS slice would be a static slice because it would 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 should be a minimal static slice. Of course exhaustive testing is infeasible for all but the smallest programs. The motivation behind RQ1 is not the construction of tests that are sufficiently representative of *all inputs*, but rather to see if, using such a set of inputs, ORBS can correctly produce the minimal static slice.

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 capture 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 an input suite that captures 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

[1]Experiment data can be found at http://crest.cs.ucl.ac.uk/resources/orbs

```
1  word_count()
2  {
3    while (scanf("%c", &c) == 1)
4    {
5      characters = characters + 1;
6
7      if (c == '\n')
8      {
9        lines = lines + 1;
10     }
11
12     if (isletter(c))
13     {
14       if (inword == 0)
15       {
16         words = words + 1;
17         inword = 1;
18       }
19     }
20     else
21     {
22       inword = 0;
23     }
24   }
25 }
26
27 int isletter(char c)
28 {
29   printf("%c ", c); // slice here
30   if (((c >= 'A' ) && (c <= 'Z'))
31       || ((c >= 'a' ) && (c <= 'z')))
32   {
33     return 1;
34   }
35   else
36   {
37     return 0;
38   }
39 }
```

Fig. 1. The word count program with a printf added to slice on variable c.

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 [6], [22], 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, in Figure 2, appeared on the souvenir mug given to delegates of the first incarnation of the SCAM conference in Florence, 2001. It has subsequently been used as a 'challenge' example for slicing algorithms [23], 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 Danicic and John Howroyd during a boat excursion at the 2^{nd} incarnation of the SCAM conference in Montréal, 2002. It was discussed at length at the conference as an example of the subtleties of minimal static slicing [24].

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.

```
1  int mug(int i, int c, int x)
2  {
3     while (p(i))
4     {
5        if (q(c))
6        {
7           x = f();
8           c = g();
9        }
10       i = h(i);
11    }
12    printf("@%d\n", x); // slice here
13 }
```

Fig. 2. The SCAM'01 Mug Example. Predicates p and q, and function h depend only on their single parameter while functions f and g return (unknown) constant values. The key point in this code is that in any terminating execution the final value of x is independent of Line 8: if q(c) is initially false, it remains false and thus x retains its initial value. On the other hand, if q(c) is true one or more times then x will have the value assigned at Line 7. In the latter case, it does not matter how often q(c) is true and thus the assignment at Line 8 does not impact the value of x at Line 12.

```
1  int mbe(int j, int k)
2  {
3     while (p(j))
4     {
5        if (q(k))
6        {
7           k = f1(k);
8        }
9        else
10       {
11          k = f2(k);
12          j = f3(j);
13       }
14    }
15    printf("%d\n", j); // slice here
16 }
```

Fig. 3. The Montréal Boat Example. Predicates p and q, and functions f1, f2, and f3 are unshown. They depend only on their formal parameter. The relevant observation is that in any terminating execution, the computation of k is irrelevant to the computation of j.

RQ2: Highlighting Unsafe Static Slices. RQ2 uses the small programs shown in Table I and known as the Siemens Suite which have been widely-studied in previous work on analysis and manipulation [25]. 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. For each program the set of slices considered includes all primitive-variable assignments and predicate uses.

Where the (claimed) static slice fails to contain a statement that is included in the observation-based slice, it suggests that the static slice may be unsafe. This occurs when a statement is in the observation-based slice because there exists an observation of behaviour that the slicing criterion depends upon. Interestingly, this claim does not depend on test suite adequacy. It only requires the existence of a test case that forms a counter example to the safety claim made by the static slice.

The stronger statement "the static slice must be unsafe" would require a finer granularity; with ORBS' current line-based implementation, if a line of code includes multiple behaviors (e.g., a call with two parameters where only one

TABLE I
SIEMENS SUITE: THE SET OF SMALL PROGRAMS USED TO ANSWER RQ2

Program	LoC	Slices
printtokens	733	81
printtokens2	579	75
replace	658	309
schedule	465	58
schedule2	392	78
tcas	185	43
totinfo	415	54

impacts the slicing criteria) then ORBS must (undesirably) include both behaviors.

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 sufficiently small for a human to understand and investigate the underlying semantic cause for any differences in the slices constructed by traditional slicing and those constructed by observation. However, they are also sufficiently large that they denote nontrivial computation, 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 [26]. For example, if $\mathcal{I}_1 \subseteq \mathcal{I}_2$ then ORBS$(v, l, \mathcal{I}_1)$ is likely[2] a subset of ORBS$(v, l, \mathcal{I}_2)$. 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, which each come with their own test suite. The first, ed, is a line-oriented text editor with 8 files and 2 836 lines of code. The second program, byacc is Berkeley Yacc, which has 13 files and 7 320 lines of code.

[2]ORBS produces 1-minimal slices, but there may be multiple 1-minimal slices for the same criterion. Therefore, it may be the case that ORBS$(v, l, \mathcal{I}_1) \not\subseteq$ ORBS$(v, l, \mathcal{I}_2)$.

```
1  word_count()
2  {
3    while (scanf("%c", &c) == 1)
4    {
5       printf("%c\n", c); // slice here
6    }
7  }
```

Fig. 4. A slice of the word count program.

```
1    while (scanf("%c", &c) == 1)
2    {
3      if (isletter(c))
4      {
5        inword = 1;
6      }
7      else
8      {
9        inword = 0;
10     }
11
12     printf("%d\n", inword); // slice here
13   }
```

Fig. 5. Excerpt of a second slice of the word count program

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 [27]. The standard CodeSurfer options were used except for pointer analysis where the most precise "pa af" option was used. We constructed observation based slices with our tool ORBS [18].

A. RQ1: Subtleties and Surprises

Several ORBS slices show interesting aspects of observation based slicing on the wc benchmark program. In the first, ORBS discovers that it is possible to merge code from two functions. This slice was taken with respect to the value of c at the top of the function isletter. In this case it just so happens that the same variable name is used in the calling function. The resulting slice, shown in Figure 4, includes first three lines from the function word_count and then four from isletter.

As a second example, consider the slice taken with respect to the value of inword just after Line 23 of Figure 1. This use of the variable inword is reached by the definitions at Lines 17 and 22. The first of these definitions is control dependent on the predicate inword == 0 and thus this predicate is included in the CodeSurfer static slice. However, one can observe that this predicate does not influence the value of inword, which is either already 1 at Line 14 or set to 1 by Line 17. Thus, as ORBS correctly determines, the slice on inword at Line 23 can omit the predicate inword == 0 as shown in Figure 5.

For mug (Figure 2), the *slice of interest* is the slice taken with respect to x at Line 12. For this slice, the key observation is that because "x = f()" assigns x a constant value there are really only two interesting executions: One in which q(c) is always false and one in which it is true at least once. These two are sufficient because x=f() assigns x an (unknown) constant value and thus execution of the assignment is idempotent.

To compute ORBS slices of this code requires values for the unspecified functions and constants. For the experiment,

```
1  int mug(int i, int c, int x)
2  {
3     while (p(i))
4     {
5        if (q(c))
6        {
7           x = f();
8        }
9        i = h(i);
10    }
11    printf("@%d\n", x); // slice here
12 }
```

Fig. 6. The key slice of the SCAM'01 Mug Example.

p(i) returns i % 5 != 0 and h(i) the value i+1; thus the while loop will execute up to five times depending on the initial value of i. This is sufficient to cover the possible cases.

The test suite includes two input cases. The first executes the while loop zero times. This vacuously covers the q(c) is always false case. The second input executes the loop twice where q(c) is true the first iteration and false the second. This covers the q(c) is true at least once case. To do so c's initial value is set to 50, the function q(c) returns $c > 42$, and the function g() returns 10. For completeness, x is initialized to 10 and the function f() returns the value 20.

The reason that two iterations are used to achieve the second case is to illustrate differences with dynamic slicing, which includes the dependence of q(c) on c=g() and that of x=f() on q(c) and thus includes c=g() in the slice. In contrast the ORBS slice correctly omits c=g().

Using these two input cases produces the slice shown in Figure 6. The static slice for this example as computed by CodeSurfer, is the entire original program; the SDG for mug includes a (loop-carried) flow dependence from c=g() to q(c) as well as a control dependence from predicate q(c) to x=f(); thus while traversing these dependences (backwards) the slicer includes 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 produces 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.

Finally, consider program mbe. Similar to mug, the code includes a path of dependence edges that causes the slice to be larger than strictly necessary. Here, the control dependence of j=f3(j) on q(k) causes CodeSurfer to include the computation of k in the slice taken with respect to j at Line 15. 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. Only when q(k) is false is progress made towards a value of j that makes p(j) false. In any terminating execution the following pattern is repeated: there are zero of more executions of Line 7 followed by execution of Lines 11 and 12. Only the execution of Line 12 impacts the eventual termination and thus the final value of j.

```
1  int mbe(int j, int k)
2  {
3     while (p(j))
4     {
5           j = f3(j);
6     }
7     printf("%d\n", j); // slice here
8  }
```

Fig. 7. The key slice of the Montréal Boat Example.

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. Here multiple executions force the retention of the while loop.

Using any instantiation 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 includes 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 the dependence analysis that underlies slicing can be and how ORBS offers a complementary alternative to traditional 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 741 slices in total. 696 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.

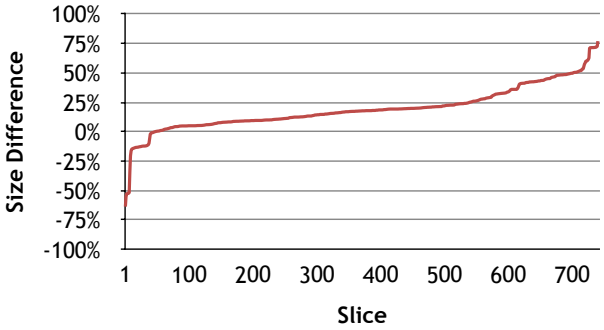

Fig. 8. Slice Size Differences

The overall size difference for these 741 slices are graphed in Figure 8. Each point is the percent reduction using ORBS minus the reduction using CodeSurfer. These differences are interesting because they illustrate the way in which ORBS can be used as a sanity check on the static slicer. It suggests

instances where the static slice can be refined further. For example, where static conservatism leads to an unnecessarily large slice.

In Figure 8 the left-most 46 slices show a greater reduction using CodeSurfer than ORBS. Each of these slices was compared by hand to determine the cause. Granularity causes 29 of the 46. For example, CodeSurfer replaces the function call change(pat, sub) with change(pat) effectively deleting the computation of sub, while ORBS must either retain or delete the entire line. A similar granularity issue occurs when the slicers encounter line *prio = *command = -1. While CodeSurfer sees this as two separate assignments, ORBS does not and must also retain the computation of a valid address for both pointers to avoid the premature termination of the program. The remaining 17 cases are caused by ORBS being inherently termination sensitive while CodeSurfer, by default, is configured to compute smaller, non-termination sensitive, slices.

In a similar example, the program tcas includes need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat() on a single line, which forces ORBS to include the function definition bool Own_Below_Threat() 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_Threat(). 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.

The granularity issues might be addressed by reimplementing ORBS to consider tokens rather than lines of text. However, of particular interest, and perhaps concern, are cases in which ORBS includes a statement omitted from the static slice because of an observed dependence. In the situation, observation based slicing may have highlighted an unsafe static slice. Thus ORBS can be used to investigate the soundness of a static slice. For example, ORBS highlights the non-termination sensitivity in the default configuration of CodeSurfer.

Another example of an unsafe static slice is shown in Figure 9. A static slice that includes the variable command should include the while loop. The lack of safety, in this case, comes from the model used to capture IO related dependences. CodeSurfer has two such models one that ignores implicit dependence through IO streams and one that does not. The ORBS slice inclusion of the while loop illustrates that CodeSurfer's default model that ignores such dependences can produce unsafe slices. Using the alternate model the while loop is included. However, this model can greatly increase the size of a slice. Computing an accurate slice is more challenging than simply refining the dependence model for fgets. Correctly slicing IO streams is subtle and challenging [28], and ORBS 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.

```
1  get_command(int *command, int *prio, float* ratio)
2  {
3    char buf[CMDSIZE];
4
5    if(fgets(buf, CMDSIZE, stdin))
6    {
7      *prio = *command = -1;
8      sscanf(buf, "%d", command);
9
10     while(buf[strlen(buf)-1] != '\n'
11           && fgets(buf, CMDSIZE, stdin))
12     {
13     }
14   }
15 }
```

Fig. 9. When slicing this code, the static slice incorrectly omits the while loop because it does not account for the *implicit* dependences caused by the input stream. This is one example of an unsafe static slice highlighted by ORBS.

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. 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 sorted the inputs by size so that the sequence of inputs $\mathcal{T} = \langle I_1, ..., I_n \rangle$ 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 is 10 for byacc and 55 for ed).

Table II shows the resulting sizes for byacc for the 10 different inputs and the number of nodes in the System Dependence Graph 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 from the ORBS slice using all ten inputs.

It is no surprise 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 dependencies.

TABLE II
NUMBER OF DELETED LINES FOR BYACC AND SIZE OF THE STATIC AND FINAL ORBS SLICE IN SDG NODES

Input I_k	Size	(A)	(B)	(C)	(D)
`error`	211	7124	7177	7303	7303
`code_error`	211	7124	7177	7303	7303
`pure_error`	233	7124	7177	7303	7303
`code_calc`	1824	6923	6846	5944	6311
`calc`	1824	6923	6846	6233	6350
`pure_calc`	1834	6923	6846	6233	6350
`calc2`	1950	6881	6828	6192	6304
`calc3`	1964	6881	6828	6192	6306
`ftp`	23819	6857	6804	6161	6294
`grammar`	27120	6828	6644	6033	6088
SDG Nodes		9556	9556	9556	9556
Static Slice		1429	3927	2817	2817
ORBS Slice		729	1120	1953	1874

The addition of the next three inputs causes a drop in the number of deleted lines as the inputs are different and larger 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 last two inputs are much larger than all the previous ones, however, the drop of deletions is very small. Despite the much larger size, the added inputs only exercise a few more dependencies and it can be assumed that the minimal static slice has been effectively approximated.

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 analyzed (but not sliced by CodeSurfer). Table II 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) has 1429 nodes (15%) while the SDG for the ORBS slice has only 729 nodes (8%). It can be seen that the ORBS slices are much smaller than the CodeSurfer slices for all four criteria, a confirmation that static slices as computed by typical tools are rather conservative.

The same experiment was undertaken using the set of 55 inputs for ed. Figure 10 shows the number of deleted lines with increasing number of inputs. It can be seen that for the four criteria, the increasing number of inputs allows progressively 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 (r2.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

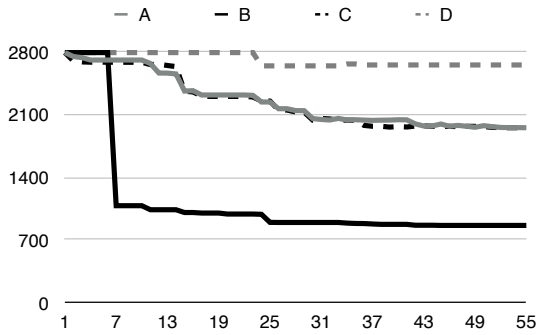

Fig. 10. Effects of Reduced Test Suites on Slices. The x-axis shows the number of tests and the y-axis shows the the number of deleted lines.

TABLE III
NUMBER OF DELETED LINES FOR ED (FROM 2836 LINES) AND SIZES OF ED AND OF THE STATIC AND FINAL ORBS SLICE IN SDG NODES

	(A)		(B)		(C)		(D)	
Deleted Lines	1954	69%	864	30%	1943	69%	2654	94%
SDG Nodes	6605		6606		6628		6628	
Static Slice	5025	76%	5025	76%	5025	76%	5025	76%
ORBS Slice	2594	39%	5722	87%	2644	40%	505	8%

is covered sufficiently. Note that the graph shows instances where adding an input increases the number of deleted lines instead of decreasing them, due to ORBS finding a different 1-minimal slice, slightly larger than the one before.

Again, a comparison with static slices as produced by CodeSurfer has been made. Table III shows the results. Due to the instrumentation for the criteria, the number of SDG nodes for the original, unsliced, program are slightly different for the four criteria (6605 – 6628). The static slices as produced by CodeSurfer are the same for all four criteria, they contain around 76% of the 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% of the nodes of the original program), criterion (D) produces a very small slice of only 8% of the 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 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 produced by tools are not only far from the precision of minimal static analyses, but they can also be incorrect.

ORBS can produce slices much nearer to minimal static slices provided the input domain is sufficiently covered.

V. Related Work

Static slicing was introduced by Weiser [29]. Ottenstein and Ottenstein [30] 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. [31] introduced an algorithm which extended the idea to slice entire programs (represented as System Dependence Graphs) and later [27] 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 the reduce the size of the slice. Incremental Slicing [32] 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 [33] 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 [34] 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 [35] is another approach that aims to preserve the semantics of the program but not the syntax. Amorphous slices use transformation to simplify programs, preserving 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').

Use of path-sensitivity analysis [36] with static slicing is another approach to reduce slice sizes by removing infeasible paths. However, such techniques suffer from their combinatorial nature and can only work precisely in the absence of certain constructs, such as loops.

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 [18], 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 [20], [37]. 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. [21] and Demillo et al. [38]). Although many research prototypes and approaches exist [39]–[45], all approaches are for a single specific programming language whereas the observation based nature of ORBS allows it to slice programs constructed from multiple programming languages [18]. Of all previous dynamic slicing formulations, the closest to our observation-based slicing is Critical Slicing [38]. However, we have found that critical slices are significantly larger than observation-based slices and are often incorrect [18].

The idea to delete parts of a program or test input is also prominent in applications of delta debugging [46]–[48]. As delta debugging can be very expensive, some approaches have modified the original delta debugging formulation, so that it exploits programming language syntax and semantics. For example, Hierarchical Delta Debugging [49] exploits tree structures for a tree-based delta debugging approach. Delta [50] uses a separate tool to flatten tree structures found in programs before applying delta debugging. Regehr et al. [51] exploit the syntax and semantics of C for four delta-debugging based algorithms to minimize C programs that trigger compiler bugs.

Jiang et al. [52] 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. Their results show that forward dynamic slicing suffers from low recall of what they call *dynamic semantic dependencies* which can have serious effects on impact analysis.

Union slicing [26] is also related to observation based slicing. Like ORBS, the union slicing algorithm 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 with 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 in its construction, we investigated 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

[1] S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," in *Principles of programming languages (POPL)*, 1988.

[2] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *SIGPLAN'92 Conf. on Programming Language Design and Implementation (PLDI'92), SIGPLAN Notices*, 1992.

[3] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up slicing," in *ACM Foundations of Software Engineering*, Dec. 1994.

[4] M. P. Ward and H. Zedan, "Deriving a slicing algorithm via FermaT transformations," *IEEE Transactions on Software Engineering*, 2010.

[5] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *Proc. Intl. Conf. on Software Maintenance (ICSM)*, 1997.

[6] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, 1991.

[7] Á. Hajnal and I. Forgács, "A demand-driven approach to slicing legacy COBOL systems," *Journal of Software: Evolution and Process*, vol. 24, no. 1, 2011.

[8] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," *Empirical Software Engineering*, vol. 7, 2002.

[9] M. Weiser and J. Lyle, "Experiments on slicing-based debugging aids," in *Empirical Studies of Programmers: First Workshop*, 1985.

[10] D. Binkley, "The application of program slicing to regression testing," *Information and Software Technology*, vol. 40, no. 11 and 12, 1998.

[11] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi, "Conditioned slicing supports partition testing," *Software Testing, Verification and Reliability*, vol. 12, 2002.

[12] R. Ettinger and M. Verbaere, "Untangling: a slice extraction refactoring," in *Proc. of the 3rd Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, 2004.

[13] J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering," in *Proc. of the 15th Intl. Conf. on Software Engineering (ICSE)*, 1993.

[14] A. Cimitile, A. De Lucia, and M. Munro, "Identifying reusable functions using specification driven program slicing: a case study," in *Proc. of the Intl. Conf. on Software Maintenance (ICSM)*, 1995.

[15] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviours through program slicing," in 4^{th} *Intl. Workshop on Program Comprehension*, 1996.

[16] B. Korel and J. Rilling, "Dynamic program slicing in understanding of program execution," in *Proc. of the* 5^{th} *Intl. Workshop on Program Comprehension (IWPC)*, 1997.

[17] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, 2003.

[18] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: Language-independent program slicing," in *Proc. 22nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*, 2014.

[19] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, 1982.

[20] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, 1988.

[21] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation (PLDI)*, 1990.

[22] T. Reps and T. Turnidge, "Program specialization via program slicing," in *Dagstuhl Seminar on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds., vol. 1110, 1996.

[23] M. Ward, "Slicing the SCAM mug: A case study in semantic slicing," in *Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

[24] S. Danicic and J. Howroyd, "Montréal boat example," in *Source Code Analysis and Manipulation (SCAM 2002) conference resources website (*`http://www.ieee-scam.org/2002/Slides_ct.html`*)*, 2002.

[25] E. Wong and V. Debroy, "A survey of software fault localization," The University of Texas at Dallas, Tech. Rep. Technical Report UTDCS-45-09, November 2009.

[26] Á. Beszédes, C. Faragó, Z. M. Szabó, J. Csirik, and T. Gyimóthy, "Union slices for program maintenance," in *Proc. of the 18th Intl. Conf. on Software Maintenance (ICSM)*, 2002.

[27] S. Horwitz, T. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, 1990.

[28] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Conf. Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[29] M. Weiser, "Program slicing," in *Proc. of the 5th Intl. Conf. on Software Engineering*, 1981.

[30] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in software development environments," in *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment*, 1984.

[31] S. Horwitz, T. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1988.

[32] A. Orso, S. Sinha, and M. J. Harrold, "Incremental slicing based on data-dependences types," in *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM)*, 2001.

[33] K. B. Gallagher, D. Binkley, and M. Harman, "Stop-list slicing," in *Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2006.

[34] J. Krinke, "Barrier slicing and chopping," in *IEEE Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

[35] M. Harman and S. Danicic, "Amorphous program slicing," in 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC)*, 1997.

[36] J. Jaffar, V. Murali, J. Navas, and A. E. Santosa, "Path-sensitive backward slicing," in *Proc. SAS'12*, vol. 7460. Springer, 2012.

[37] B. Korel and J. Laski, "Dynamic slicing in computer programs," *Journal of Systems and Software*, vol. 13, no. 3, 1990.

[38] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA)*, 1996.

[39] A. Beszedes, T. Gergely, Z. M. Szabó, J. Csirik, and T. Gyimothy, "Dynamic slicing method for maintenance of large C programs," in *Proc. of the 5th Conf. on Software Maintenance and Reengineering*, 2001.

[40] A. Beszedes, T. Gergely, and T. Gyimóthy, "Graph-less dynamic dependence-based dynamic slicing algorithms," in *Sixth IEEE Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2006.

[41] G. Mund and R. Mall, "An efficient interprocedural dynamic slicing method," *Journal of Systems and Software*, vol. 79, no. 6, 2006.

[42] A. Szegedi and T. Gyimóthy, "Dynamic slicing of Java bytecode programs," in *Proc. of the 5th IEEE Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2005.

[43] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," in *Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation*, 2004.

[44] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, vol. 12, no. 2, 2007.

[45] S. S. Barpanda and D. P. Mohapatra, "Dynamic slicing of distributed object-oriented programs," *IET software*, vol. 5, no. 5, 2011.

[46] A. Zeller, "Yesterday, my program worked. today, it does not. Why?" in *European Software Engineering Conf. and Foundations of Software Engineering*, 1999.

[47] H. Cleve and A. Zeller, "Finding failure causes through automated testing," in *Intl. Workshop on Automated Debugging*, 2000.

[48] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, 2002.

[49] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *Proc. of the 28th Intl. Conf. on Software Engineering (ICSE)*, 2006.

[50] S. McPeak, D. S. Wilkerson, and S. Goldsmith, "Heuristically minimizes interesting files." delta.tigris.org.

[51] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2012.

[52] S. Jiang, R. Santelices, M. Grechanik, and H. Cai, "On the accuracy of forward dynamic slicing and its effects on software maintenance," in *Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, 2014.