Assessing the Impact of Execution Environment on Observation-Based Slicing

David Binkley* Leon Moonen§

* Loyola University Maryland, Baltimore, MD, USA – E-mail: binkley@cs.loyola.edu
§ Simula Research Laboratory, Oslo, Norway – E-mail: leon.moonen@computer.org

***Abstract*—Program slicing reduces a program to a smaller version that retains a chosen computation, referred to as a *slicing criterion*. One recent multi-lingual slicing approach, *observation-based slicing (ORBS)*, speculatively deletes parts of the program and then executes the code. If the behavior of the slicing criteria is unchanged, the speculative deletion is made permanent.**

While this makes ORBS language agnostic, it can lead to the production of some non-intuitive slices. One particular challenge is when the *execution environment* plays a role. For example, ORBS will delete the line "a = 0" if the memory location assigned to a contains zero before executing the statement, since deletion will not affect the value of a and thus the slicing criterion. Consequently, slices can differ between execution environments due to factors such as initialization and call stack reuse.

The technique considered, *n*VORBS, attempts to ameliorate this problem by *validating* a candidate slice in *n* different execution environments. We conduct an empirical study to collect initial insights into how often the execution environment leads to slice differences. Specifically, we compare and contrast the slices produced by seven different instantiations of *n*VORBS. Looking forward, the technique can be seen as a variation on metamorphic testing, and thus suggests how ideas from metamorphic testing might be used to improve dynamic program analysis.

I. Introduction

Program slicing is a program decomposition technique that has a wide range of applications in various areas such as debugging, program comprehension, software maintenance, refactoring, testing, reverse engineering, tierless or multi-tier programming, commit decomposition, and vulnerability detection [1–7]. At its introduction, Weiser defined program slicing as follows: "Starting from a subset of a program's behavior, slicing *reduces* that program to a minimal form which still produces that behavior" [1]. This behavioral subset is referred to as a *slicing criterion*. Surprisingly, most slicing algorithms try to decide which code should be *retained* to preserve the criterion.

In contrast, *observation-based slicing (ORBS)* [8] is closer to Weiser's definition. Rather than using static or dynamic analysis to find which parts of the program to include in a slice, ORBS tentatively removes parts from the program and *observes* the impact of their removal. Removals that do not impact the behavior of interest are made permanent. This enables ORBS to easily slice multi-lingual programs [8], programs with non-standard semantics [9], and handle "hidden" dependences such as those caused by reading and writing a common file [10].

For all its virtues, ORBS has two significant drawbacks. First, it requires considerable computational effort to compute a slice. Second, real-world execution environments can lead to unexpected behavior. The first of these can be mitigated by applying greater computational power and through (non-trivial) engineering work [11]. In some ways, the second drawback poses the greater challenge and is the topic of this paper.

```
1  int f() {
2    int a;
3    a = 42;
4    return a;
5  }
6  int g() {
7    int b;
8    b = 42;                  // Is this statement deletable?
9    return b;
10 }
11 main() {
12   int x, y;
13   x = f();
14   y = g();                 // Slice here on final value of y.
15 }
```

Fig. 1. An example hidden dependence cause by memory location reuse.

In theory (as opposed to in practice), this challenge does not exist. It is possible to construct a well-defined formal semantics for a programing language such that ORBS can maintain execution behavior while cleanly removing unnecessary code. Doing so in practice is less straightforward due to choices made in real-world compilers and runtime environments. For example, consider the code in Figure 1. All (correct) dependency-based slicers will (correctly) conclude that the assignment on Line 8 is in the slice taken with respect to y at Line 14 because of the transitive data dependence of y's value on the assignment to b. The same would be true of ORBS using an idealized execution environment. However, on many systems, stack activation records are reused, and thus a and b *can* share the same location in memory, which leaves b holding the correct value even if Line 8 is omitted from the slice (assuming no well-timed interrupt changes the stack).

Contributions: To mitigate the impact of a particular execution environment, this paper proposes *n*VORBS, an ORBS variant that validates candidate slices in *n* different execution environments. The idea is that the weaknesses of one execution environment are covered by at least one of the others, so if *n* is sufficiently large and a candidate slice behaves similarly in all *n* environments, we have evidence that it is a correct slice.

To gain some initial insight into how often the execution environment leads to the problems described above, we conduct an empirical study in which we compare and contrast the slices produced by seven different instantiations of *n*VORBS.

This work is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) license.

Accepted for publication in the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2022).

II. OBSERVATION-BASED SLICING

ORBS computes *dynamic backward executable slices* [4]. To do so, it repeatedly attempts to delete a window of one to four consecutive lines. Using a window of one to four lines has been empirically shown to balance the number of lines that can be deleted in a single step versus the time wasted on larger deletions that more often than not change the program's behavior [8]. In greater detail, for the slice taken with respect to variable v at program location (line) l, ORBS first instruments the program to print the value of v immediately after l. Next, the program is executed on its test suite, and the printed values of v are recorded as an oracle. ORBS then repeats its main loop where it iterates over the program, speculatively deleting a window of consecutive lines. The resulting program is compiled and executed or interpreted in an *execution environment*, and if its output matches the oracle, then the speculative deletion is made permanent. Finding motivation in metamorphic testing, nVORBS extends ORBS by checking if the output matches the oracle in n execution environments.

III. EXPERIMENT DESIGN

To investigate the impact of the execution environment on observation-based slicing, we study seven instantiations of nVORBS using three execution environments, as shown in Figure 2. All seven are identical except for the compilers used and the runtimes in which programs are executed. The bottom three instantiations $\mathcal{G}$, $\mathcal{C}$, and $\mathcal{W}$ use a single execution environment (respectively based on a selection of modern C compilers gcc, clang, and clang using the option –target=wasm32-unknown-wasi, which we refer to as wasm in the following). Both $\mathcal{G}$ and $\mathcal{C}$ build an executable that is run directly from the operating system. $\mathcal{W}$ builds a WebAssembly binary that is executed in the wasmer virtual environment.

The middle three use pairs of execution environments. Specifically, $\mathcal{GC}$ uses both gcc and clang, $\mathcal{CW}$ uses clang and wasm, and $\mathcal{GW}$ uses gcc and wasm. In each instantiation, both executables are built, and the output of each is checked against the oracle. Finally, $\mathcal{GCW}$ uses all three execution environments. $\mathcal{GCW}$ is expected to be the most strict (i.e., accept fewer deletions) and thus produce larger slices. It is also expected to be the least susceptible to unwanted slicing behavior.

IV. RESEARCH QUESTIONS

RQ1: How often does the execution environment impact the slice? In an ideal world, all execution environments would produce the same slice. This question empirically investigates how close reality comes to this ideal.

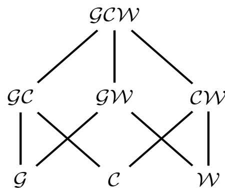

Fig. 2. Containment lattice of the seven instantiations.

RQ2: What containment relations exist between the slices produced by the different instantiations? Intuitively nVORBS instantiations that involve stricter requirements (i.e., with larger n) should produce larger slices because they can delete fewer lines. Thus we expect using both gcc and $clang$ to produce larger slices than using only one of the two.

RQ3: What qualitative patterns are found in the slices? How does the source code of slices produced by the seven instantiations for the same criterion and program compare? For example, gcc and clang are similar, so one might expect them to produce similar slices. As a second example, wasm's use of a virtual machine might be more stable because each execution starts with the virtual machine in an identical initial state.

V. EVALUATION

This section describes the subjects used in the empirical study, as well as the execution environments and hardware used to run the experiments. Next, we address the three research questions.

Subjects: The experiments consider the 69 C programs shown in Table I, which were used in prior program analysis research:

- Programs from the slicing literature: a variation on the original example of Weiser (sumprod) [1], the SCAM Mug example (scam) [12], the Montréal Boat Example (mbe) [13], and word count (wc) [2].
- Programs from the Mälardalen WCET benchmark for comparing and evaluating WCET analysis tools [14].
- Programs from the *Benchmarks Game* [15], which are designed to benchmark language implementations.
- Two multi-file programs (bc and indent) used in previous slicing studies [16].

We omit programs from the second and third sources that span multiple files, that fail to compile with -lm as the only compiler flag enabled, and that are unsupported by the pycparser Python library. We use pycparser to normalize (i.e., pretty-print) the code and to instrument it (i.e., add a printf statement that captures the criterion). For multi-file programs, ORBS slices a specified file from the program (this file is show in Table I within parenthesis). Thus there is no practical limit on the size of the program ORBS can slice.

Execution Environments: The three execution environments respectively use gcc v12.0.0 20210720 running the compiled program natively, clang v13.0.0 (50302feb) also running the compiled program natively, and clang compiling to WebAssembly and using the wasmer 2.0.0 WebAssembly runtime.

Hardware: All data was generated using a 676-core computing cluster using 2.20GHz Xeon(R) E5-2650 CPUs. The cluster has 256GB RAM per node and terabytes of HDD and fast SSD storage all connected using a 56 gigabit Infiniband network.

RQ1: To answer how often the execution environment impacts the slice, we compare the individual slices produced by each pair of nVORBS instantiations for the same program and slicing criterion. With seven instantiations and 2921 slices, this leads to ((7 * 6) / 2) * 2921 = 61 341 comparisons. In 47 269 (77%) of these, both instantiations produce the same slice. This is encouraging. It means that three-quarters of the time, the execution environment does not impact the slice. Of

TABLE I
SUBJECT SYSTEMS. FOR MULTI-FILE PROGRAMS THE SLICED FILE IS SHOWN IN PARENTHESIS. ("RC"' ABBREVIATES REVERSE-COMPLIMENT)

Program	SLoC	Slices	Program	SLoC	Slices
adpcm	585	168	ludcmp	109	25
bc (bc.c)	8 594	54	mandelbrot2	66	18
bc (execute.c)	9 140	88	mandelbrot9	66	7
binary-trees1	91	14	matmult	55	7
bs	46	9	mbe	33	12
bsort100	61	8	minver	201	49
cnt	78	17	nbody1	92	13
compress	357	74	nbody2	107	14
cover	625	199	nbody3	90	20
crc	94	16	nbody6	93	13
duff	44	1	nbody7	137	23
edn	170	42	ndes	196	39
expint	73	23	ns	31	4
fac	25	3	prime	51	1
fankuchredux1	79	11	printtokens	570	81
fankuchredux5	115	20	printtokens2	408	75
fasta1	126	19	qsort-exam	124	22
fasta2	264	34	qurt	120	16
fasta3	90	6	rc-5	83	11
fasta5	111	15	rc-6	96	15
fasta7	231	32	replace	542	309
fasta8	150	14	scam	35	16
fasta9	163	16	schedule2	292	74
fdct	138	86	schedule	314	58
fft1	128	41	select	131	22
fibcall	27	8	spectral-norm1	57	10
fir	54	15	st	98	14
hanoi_c	178	21	statemate	1 354	364
indent (indent.c)	6 680	223	sumprod	17	8
indent (parse.c)	5 213	28	tcas	142	43
insertsort	33	5	totinfo	348	54
janne_complex	38	7	triangle	59	7
jfdctint	119	65	ud	81	22
lcdnum	62	5	wc	49	17
lms	172	51			
			Total	40 311	2 921

the remaining 14 072 comparisons, in 9 279 (66%) cases one slice includes a subset of lines from the other. The remaining 4 793 cases are different slices (neither is a subset of the other).

Some slices may exhibit non-deterministic behavior in certain environments. For example, after removing Line 8 of Figure 1, if an interrupt occurs between the execution of Lines 13 and 14, and this interrupt changes the stack, then y may no longer be assigned the value 42. Because it is possible that these slices degrade the analysis, we remove from all instantiations those slices that exhibit non-deterministic behavior in any one of them. This filter turns out to have minimal impact, removing only 22 slices. Post removal there are 60 879 comparisons of which 46,954 (77%) are identical, and of the remaining 13 925 comparisons, 9 142 (66%) are subset relations.

The second filter removes slices where the criterion is not part of the slice in all seven instantiations. This typically occurs because of an inadequate test suite. This filter has a greater impact, removing 1 118 slices. Post removal there are 37 863 comparisons of which 24 907 (66%) are identical, and of the remaining 12 956 comparisons, 8 269 (64%) are subset relations.

Finally, applying both filters together leaves 1781 slices, yielding 37 401 comparisons of which 24 592 (66%) are identical, and of the remaining 12 809 comparisons 8 134 (64%) are subset relations.

TABLE II
SUBSET RELATIONS FOR RELATED ENVIRONMENTS.

comparison	$=$	$\supset$	$\subset$	$\neq$
$\mathcal{G}$ vs. $\mathcal{GC}$	1278	3	328	172
$\mathcal{G}$ vs. $\mathcal{GW}$	1013	4	658	106
$\mathcal{G}$ vs. $\mathcal{GCW}$	981	0	622	178
$\mathcal{C}$ vs. $\mathcal{GC}$	1432	9	214	126
$\mathcal{C}$ vs. $\mathcal{GW}$	1154	3	471	153
$\mathcal{C}$ vs. $\mathcal{GCW}$	1096	4	488	193
$\mathcal{W}$ vs. $\mathcal{GW}$	1243	14	289	235
$\mathcal{W}$ vs. $\mathcal{CW}$	1276	4	273	228
$\mathcal{W}$ vs. $\mathcal{GCW}$	1206	3	309	263
$\mathcal{GC}$ vs. $\mathcal{GCW}$	1178	1	446	156
$\mathcal{GW}$ vs. $\mathcal{GCW}$	1558	3	92	128
$\mathcal{CW}$ vs. $\mathcal{GCW}$	1483	9	191	98

In summary for RQ1, the execution environment affects the slice in roughly one of three cases. Note that this is a conservative result because some differences are very minor.

RQ2: The seven nVORBS instantiations validate speculative deletions using one or more execution environments. RQ1 shows that the execution environment affects roughly one in three slices. For these slices, validating in multiple execution environments can be expected to restrict the number of successful deletions. RQ2 investigates the impact this restriction has by analyzing the containment relations between the slices produced by the seven nVORBS instantiations.

Starting with the 1781 slices of the doubly filtered data created while addressing RQ1, Table II presents the comparisons in groups of three. The first three groups compare the slices of the singleton instantiations $\mathcal{G}$, $\mathcal{C}$, and $\mathcal{W}$ with those instantiations that involve additional systems. For example, the first group compares $\mathcal{G}$ with the three others that involve gcc: $\mathcal{GC}$, $\mathcal{GW}$, $\mathcal{GCW}$. The first column shows the comparison, while the second counts the number of slices where $\mathcal{G}$ alone produces the same slice as $\mathcal{GC}$, $\mathcal{GW}$, and $\mathcal{GCW}$: 1278 (72%), 1013 (57%), and 981 (55%), respectively. The third column shows how often slices produced by $\mathcal{G}$ are a superset of (i.e., larger than) the corresponding slice produced by the other instantiations (this is unexpected because $\mathcal{G}$ alone is less restrictive). The fourth column shows how often the slice produced by $\mathcal{G}$ is a subset of the other (the expected outcome), and the fifth column shows the number of slices where neither is a subset of the other. The first group shows the expected pattern where $\mathcal{G}$ is a superset in only a few cases (specifically 3, 4, and 0), while it is a subset in respectively 328, 658, and 622 cases.

The following two groups repeat these comparisons for respectively $\mathcal{C}$ and $\mathcal{W}$. Observe that the overall pattern is the same. It is interesting to note that $\mathcal{G}$, which uses gcc, produces the fewest supersets and by far the most subset relations.

The last group in Table II compares the three instantiations that use two environments (i.e., $\mathcal{CW}$, $\mathcal{GW}$, and $\mathcal{CW}$) with the instantiation that uses all three (i.e., $\mathcal{GCW}$). These comparisons show evidence that using more environments increases stability. For example, an average of 79% of the slices in the last group are identical compared to only 66% averaged over the nine singleton comparisons. Finally, the individual rows show

evidence that wasm accounts for much of the greater similarity.

In summary for RQ2, while not universal, the subset relations support the intuition that the stricter requirements of validating slices in multiple execution environments produce larger slices because fewer lines can be deleted.

RQ3: RQ3 takes a qualitative look at the slices of the filtered data to focus on differences in the slices themselves. While it is not practical to compare all slices by hand, it is possible to do this for some of the smaller programs. This section presents several interesting examples that are short enough to explain in limited space. To begin with, for 18 of the 69 test subjects, all seven instantiations produce the same set of slices.

Looking at the individual slices, we first consider the slice of wc taken with respect to the computation of inword, which is true when the current character is part of a word. The variable inword is initialized by the statement “inword = 0”. When using clang, this initialization is retained. In contrast, when using gcc and wasm, it is deleted because the memory location assigned to inword happens to initially hold the value zero.

As a related example, $\mathcal{C}$ (and $\mathcal{CW}$, $\mathcal{GC}$, and $\mathcal{GCW}$) retain the declaration “**int** lines” in the slice taken with respect to the final value of words, even though lines is not used elsewhere in the code. To understand why, the first step is to note that a previous pass deleted “words = 0” for the same reason that inword was deleted above. However, removing “**int** lines” in a subsequent pass changes the address of words to one with a non-zero value, thereby preventing the removal of the declaration. That two of the three execution environments retain this statement suggests that future work might consider voting instead of requiring equal slices in all environments.

Compiling WebAssembly puts certain constraints on code. For example, it requires that the effect on the stack must be well-typed. Each WebAssembly instruction has a specific *stack type* $t_1^* \rightarrow t_2^*$, where t_1^* is the expected sequence of types for the values on top of the stack before execution and t_2^* is the sequence of types for the values on top of the stack after execution. For instance, the wasm instruction “**i32**.const 42” has type $\rightarrow$ **i32**”, meaning that it does not need anything from the stack and pushes one value of type **i32**. These constraints result in *surprising* compilation choices when wasm compiles an **if** where ORBS is attempting to delete the **else** branch of the code. Consider the following function with conditional code:

```
1  int ishappy(...) {     // returns a boolean using an int
2    if (Cond)
3      return 0;
4    else                 // ORBS attempts to delete
5      return 1;          // these two lines
6  }
```

When gcc compiles this code without the else branch, it moves a zero into the return location when Cond is true, otherwise it does nothing, and thus the value in the return location is unchanged. Commonly this value is non-zero, which C programs interpret as true. In the wasm case, because of its well-typed stack requirement in the compiled code, both branches of an **if** statement must have the same effect on the stack. What the wasm compiler does when presented with the above code without Lines 4 and 5 is surprising, yet legal and satisfies the requirement: it replaces the **if** statement altogether by the compiled version of “**return** 0”.

A second wasm example relates to the code in Figure 1. When excluding the code on Line 8, the compiler generates:

```
1  (func $g (type 2) (result i32)
2    (local $0 i32)
3    (get_local $0)
4  )
```

When executing this in the wasmer runtime, it evaluates to zero (most likely because the runtime initializes the memory it uses on startup). However, the oracle expects 42, which means that ORBS using wasm retains Line 8 in the slice.

Next consider the slice of totinfo with respect to n at Line 199, which includes the following code *except* in the $\mathcal{W}$ instantiation:

```
79  if ( r * c > MAXTBL ) {
80    return EXIT_FAILURE;
81  }
```

If this test is omitted, the executables produced by gcc and clang both report a memory violation. However wasm allocates sufficient memory that the subsequent out of bounds array accesses do *not* generate a memory violation, and thus using $\mathcal{W}$ the statement can be deleted. (Interestingly, running the gcc produced executable in gdb does not generate a fault.)

Finally, a compiler difference can be observed in the slice of totinfo with respect to i at Line 389: for this slice, clang prevents ORBS from removing the declaration and return of the variable info, while gcc permits their removal.

```
303    double info;      /* accumulates information measure */
       ...
414  ret3:
415    return info;
416  }
```

The root cause of this difference is a rare compiler disagreement on the definition of correct C syntax, which deserves further investigation. Using gcc, the return (and subsequently the declaration of info) can be excluded without problem (even when using the stricter –ansi flag). In contrast, removing the return causes clang to issue a syntax error that a statement is expected but missing. Because it cannot remove the return, clang must also retain the declaration. While ORBS cannot do this, if the return is replaced by “**return** 42” clang produces the expected oracle output and can remove the declaration.

In summary for RQ3, the most common pattern seen occurs when a variable initialization can be removed without changing the behavior of the program. In addition, the WebAssembly stack validation requirement has the interesting effect of avoiding the non-deterministic return value found when removing a **return** statement found in an **else** branch.

Threats to validity: In addition to the standard internal threats found when using tools an analyze software, threats to external validity exist when asking if our results extend to other environments, other programs, and especially other programming languages. For example, we studied relatively small C programs from the program analysis literature. Although

these are typically aimed at covering all language features of interest, we do not know to what extent our results generalize to large systems or, for example, embedded C code. Moreover, the language C has rather "loose" semantics, so other languages that are more completely and precisely defined would likely yield more consistent results. One specific threat follows from using MD5 hashes to compare the result of executing a slice with the oracle. There is the potential for errant matches when doing more than 2^{64} comparisons. This can be easily addressed by using a longer hash, such as SHA256.

VI. Related Work

The most closely related work to ours is that related to ORBS [8], which includes the parallel implementation [11] used to implement *n*VORBS. The modification of the compile and runtime system was to some degree inspired by the work on slicing languages with non-standard semantics such as picture description languages [9]. Finally, the use of wasm finds its roots in the slicing of WebAssembly [17].

Traditional software testing relies on the presence of a *test oracle* that decides what is the correct output or expected behavior for a given test input. Automated software testing uses the oracle to identify failures, i.e., when the behavior of the software system deviates from the oracle. When a system is 'non-testable' [18], because an oracle is unavailable, or prohibitively expensive to use, *metamorphic testing* provides a way forward [19]. It is based on defining one or more relations between the outputs of a program that must hold for a series of inputs. A typical example is the sinus function for which the following metamorphic relation holds for any x: $sin(x) = sin(\pi - x)$. Another example is a cryptographic system that supports both stream and block modes. Metamorphic testing would encrypt a message using the two modes and expect the same encrypted message. Likewise, when testing autonomous vehicle software where producing an oracle is expensive, metamorphic testing can test car behavior on a set of transformed images that should result in the same behavior [20].

Closer to our use case of ensuring that slices behave the same in different execution environments, metamorphic testing has been used to ensure correct behavior of Datalog engines [21], find bugs in the implementation of compilers [22], as well as increase the validity of various simulation models [23].

VII. Concluding Remarks

Contributions: To mitigate any negative impact of a particular execution environment on observation-based slicing, we propose *n*VORBS, an ORBS variant that validates candidate slices using *n* different execution environments. By considering seven differing instantiations of *n*VORBS, we empirically investigate how widespread the impact of the execution environment is observation-based slicing in practice.

Our data suggest that the environment affected slices in roughly one of three cases. Thus, validating the correctness of slices in multiple environments helps to cover weaknesses in one of the environments, and increases the evidence that only correct slices are produced. The data also shows that validating slices in multiple environments results in larger slices. Finally, our qualitative analysis uncovers several interesting findings, such as the value that WebAssembly's more structured runtime brings to dynamic analyses such as ORBS.

Future Work: Further empirical work is needed to evaluate the benefits and drawbacks of *n*VORBS. The qualitative examples suggest some interesting directions for future work, such as the use of a voting scheme when using multiple environments. In addition, there are several opportunities for efficiency improvements. For example, a parallel algorithm can attempt the deletion of all window sizes concurrently and retain the largest successful deletion [11]. In addition, the validation in multiple execution environments can be easily parallelized. Finally, one can see *n*VORBS as a variation on metamorphic testing. Looking forward, the use of metamorphic testing ideas might find broader use in dynamic source code analysis.

Acknowledgements: This work was supported by NSF grant #1626262, which provided the cluster used to generate the empirical data. Leon Moonen was supported by the Research Council of Norway through the secureIT project (#288787).

References

[1] M. Weiser, "Program slicing," in *ICSE*, 1981, pp. 439–449.
[2] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE TSE*, vol. 17, no. 8, pp. 751–761, 1991.
[3] D. W. Binkley and M. Harman, "A survey of empirical results on program slicing," *Advances in Computers*, vol. 62, pp. 105–178, 2004.
[4] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Computing Surveys*, vol. 44, no. 3, 2012.
[5] L. Philips *et al.*, "Search-based tier assignment for optimising offline availability in multi-tier web applications," *Programming*, 2(2), 2018.
[6] W. Muylaert and C. De Roover, "Untangling composite commits using program slicing," in *SCAM*, 2018, pp. 193–202.
[7] S. Salimi, M. Ebrahimzadeh, and M. Kharrazi, "Improving real-world vulnerability characterization with vulnerable slices," in *PROMISE*, 2020.
[8] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: Language-independent program slicing," in *FSE*, 2014.
[9] S. Yoo, D. W. Binkley, and R. D. Eastman, "Observational slicing based on visual semantics," *JSS*, vol. 129, pp. 60–78, 2017.
[10] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS and the limits of static slicing," in *SCAM*, 2015, pp. 1–10.
[11] S. Islam and D. Binkley, "PORBS: A parallel observation-based slicer," in *Int'l Conf. Prog. Comprehension (ICPC)*. IEEE, 2016.
[12] M. P. Ward, "Slicing the SCAM mug: A case study in semantic slicing," in *SCAM*, 2003, pp. 88–97.
[13] S. Danicic and J. Howroyd, "Montréal boat example," in *Source Code Analysis and Manipulation (SCAM) conference resources website*, 2002.
[14] Mälardalen WCET research group, "Wcet benchmarks," https://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
[15] B. Fulgham and I. Gouy, "The computer language benchmarks game," https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
[16] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM TOSEM*, vol. 16, no. 2, pp. 1–32, 2007.
[17] Q. Stiévenart, D. W. Binkley, and C. De Roover, "Static stack-preserving intra-procedural slicing of webassembly binaries," in *ICSE*, 2022.
[18] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *ACM Conference*, 1981.
[19] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A Survey on Metamorphic Testing," *TSE*, vol. 42, no. 9, Sep. 2016.
[20] J. R. Toohey *et al.*, "From neuron coverage to steering angle: Testing autonomous vehicles effectively," *Computer*, vol. 54, no. 8, 2021.
[21] M. N. Mansur, M. Christakis, and V. Wüstholz, "Metamorphic testing of Datalog engines," in *ESEC/FSE*. ACM, Aug. 2021, pp. 639–650.
[22] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, no. 6, 2014.
[23] M. Olsen and M. Raunak, "Increasing Validity of Simulation Models Through Metamorphic Testing," *IEEE Trans. Reliability*, 68(1), 2019.