QSES: Quasi-Static Executable Slices

Quentin Stiévenart
Vrije Universiteit Brussel, Belgium
quentin.stievenart@vub.be

David Binkley
Loyola University Baltimore, USA
binkley@cs.loyola.edu

Coen De Roover
Vrije Universiteit Brussel, Belgium
coen.de.roover@vub.be

***Abstract*—Program slicing aims to reduce a program to a minimal form that produces the same output for a given slicing criterion. Program slicing approaches divide into static and dynamic approaches: whereas static approaches generate an over-approximation of the slice that is valid for all possible program inputs, dynamic approaches rely on executing the program and thus generate an under-approximation of the slice that is valid for only a subset of the inputs. An important limitation of static approaches is that they often do not generate an *executable* program, but rather identify only those program components upon which the slicing criterion depends (referred to as a closure slice). In order to overcome this limitation, we propose a novel approach that combines static and dynamic slicing. We rely on observation-based slicing, a dynamic approach, but *protect* all statements that have been identified as part of the static slice by the static slicer CodeSurfer. As a result, we obtain slices that cover at least the behavior of the static slice, but that can be compiled and executed. We evaluated this new approach on a set of 57 C programs and report our preliminary findings.**

***Index Terms*—program slicing, static slicing, dynamic slicing, program dependence 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, re-engineering, refactoring, testing, reverse engineering, comprehension, tierless or multi-tier programming, commit decomposition, and vulnerability detection [1]–[7].

At its introduction program slicing was defined by Mark Weiser 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]. The initial subset is referred to as a *slicing criterion*. Key to this definition is that a slice is an *executable* program. A few years later Ottenstein and Ottenstein observed that “The program dependence graph (PDG) ... allows programs to be sliced in linear time” [8]. However this approach brings a key difference: the resulting slice is not guaranteed to be an executable program. Thus, in contrast to Weiser’s original definition, which produced *executable slices*, the slices produced from a PDG are referred to as *closure slices* as their computation involves computing a transitive closure.

This dimension of executable slice versus closure slice is one of three dimensions that are used to classify program slices [9]. The second dimension, static versus dynamic, was introduced by Korel and Laski [10] who observed that in some applications, such as debugging, it is not necessary to produce *all* of the behavior of the original program. Rather, for these applications, it suffices to preserve the behavior for only a selected set of inputs.

The final dimension, backward versus forward, was introduced by Horwitz et al. [11] who observed that the dependence edges of a PDG could be traversed in either direction. In contrast to backward slicing, which captures those program components *that affect* the slicing criterion, a forward slice identifies those program components *affected by* the slicing criterion. This paper considers only backward slicing.

Despite being efficient to compute, closure slices are not guaranteed to be executable programs. For example, Horwitz et al. provide an example in which two calls to a procedure require different subsets of the procedure’s formal parameters and different subsets of the procedure’s statements. However, it is not just closure slices that can fail to produce executable programs. A long standing challenge for executable slicing algorithms has been to *guarantee* that the resulting slice is a syntactically correct program with the correct semantics [12].

One recent variation of dynamic slicing, referred to as *observation-based slicing* [12], approaches the slicing problem from a different angle. Rather than using static or dynamic analysis to determine which components of the program to include in the slice, observation-based slicing tentatively removes components from the program and then *observes* the impact of this removal. A removal that has no impact on the behavior of the slicing criterion is made permanent. By its very nature an observational slice is *guaranteed* to be executable.

What is missing is a static slicing algorithm that can make the same executability guarantee as observation-based slicing. This paper provides such an algorithm through the combination of a static closure slicer and a dynamic observation-based slicer. The key idea is to *protect* the code of the static slice from removal during observation-based slicing. The result is a *Quasi-Static Executable Slice* (QSES), which is a safe over-approximation to the static slice that is also executable. Specifically we investigate the combination of CodeSurfer [13], which efficiently computes static closure slices, and pORBS [14], which uses a parallel algorithm to compute (dynamic) observation-based slices. While in theory the combination computes static executable slices, in practice there are some interesting corner cases related to code structure, memory layout, and termination when slicing C code. The remainder of this paper introduces ORBS, CodeSurfer, and then QSES in greater detail, provides a preliminary evaluation of QSES by considering three research questions, and finally presents some concluding remarks.

II. Background: ORBS and CodeSurfer

a) Observation-based slicing: Our approach builds on Observation-Based Slicing (ORBS) [12], the algorithm for which is identical to Algorithm 1 except that ORBS omits Line 12 and the input C. ORBS takes as input a source program P to slice, a slicing criterion identified by a program variable ν, a program location l and a set of inputs $\mathcal{I}$, and a maximum window size δ. ORBS is language agnostic so it considers the program as a sequence of lines of text, p_1 to p_n. It can be made slightly more efficient, by considering, for example, program statements, at the expense of performing minimal language-specific parsing [15].

ORBS proceeds as follows. First, the program is instrumented by Setup, which inserts a side-effect free line that tracks the value of variable ν, immediately before line l. This is to ensure that the algorithm detects any changes to that variable at that location upon the removal of other lines. The instrumented program is then run on each input in $\mathcal{I}$ and the tracked values are captured in V, which is used as an oracle of the expected output. Then Line 4 reverses the lines of the code so that they are effectively considered bottom-up, which is more efficient as it can slice out the occurrences of a variable before considering its declaration.

The rest of the algorithm iterates over the program tentatively removing lines until no more lines can be deleted. Each iteration over the program (Lines 8-25), tries to remove up to δ lines starting with the current line. After each removal, the program is compiled. If it compiles, it is executed and its output is captured in V'. If the output matches the oracle V, then the current removal can be safely made permanent. After multiple passes over the program, the process will eventually stabilize. When no more lines can be removed, S corresponds to a dynamic observation-based slice.

In principle the removal might consider arbitrary subsets of the program or, as with delta debugging [16], start by considering half the program, then quarters, etc., but these approaches can prove quite expensive [12]. Instead ORBS uses a window of up to δ *contiguous* lines. It thus computes 1-minimal dynamic slices [12] (no single line can be removed from the slice). Based on past empirical data, we use $\delta = 4$ in our experiments [12], [14].

b) Static SDG-based slicing: In addition to ORBS we make use of GrammaTech's *CodeSurfer*, to compute static closure slices. CodeSurfer builds a System Dependence Graph (SDG) [11], [17] for a program and then computes a slice by walking this graph. The result is a set of graph vertices, which we map back to lines of code in the original source. Because the SDG does not represent much of the concrete syntax (e.g., the braces delimiting a block), the identified source code is typically not an executable program.

III. Approach

One obvious advantage of the ORBS approach over static closure slicing is that the resulting slice is guaranteed to be an executable program. One obvious disadvantage is that the slice only preserves the behavior of the original program when it is run on the inputs from the set $\mathcal{I}$. Our key insight is that if *before* applying ORBS to the program we protect from deletion the lines of a (non-executable) static closure slice, then the result should preserve the behavior of the static slice, *and be executable*. In other words, the result is an executable static slice. In addition to the protected lines of the closure slice, this slice includes those parts of the code that ORBS retains to ensure the slice compiles and executes correctly. We refer to this combination as Quasi-Static Executable Slicing (QSES). Intuitively, it is not necessary to check the behavior of the protected lines (make each of them a slicing criteria) because by its very definition a closure slice includes any required supporting computations.

Algorithm 1 depicts the QSES algorithm. The two differences compared to ORBS are that QSES takes as input C all the lines of the static closure slice, and second, it will never try to remove a window of lines that overlaps with C (see Line 12). Thus QSES never removes any line that is part of the static slice. For all other lines, QSES, like ORBS, attempts to remove the lines and then observes the behavior of the resulting program. If the program without the lines produces the same behavior, then the lines can be safely removed.

```
1  QSESlice($P, \nu, l, \mathcal{I}, \delta, C$)
   Input  : Source program $P = \{p_1, \ldots, p_n\}$, slicing criterion
            $(\nu, l, \mathcal{I})$, maximum deletion window size $\delta$, and
            static slice $C$
   Output: A slice, $S$, of $P$ for $(\nu, l, \mathcal{I})$
2  $O \leftarrow$ Setup($P, \nu, l$);
3  $V \leftarrow$ Execute(Build($O$), $\mathcal{I}$);
4  $S \leftarrow$ Reverse($O$);
5  repeat
6      deleted $\leftarrow$ False;
7      $i \leftarrow 1$;
8      while $i \leq length(S)$ do
9          builds $\leftarrow$ False;
10         for $j \leftarrow \delta$ downto $1$ do
11             $S^- \leftarrow \{s_i, \ldots, s_{\min(length(S), i+j-1)}\}$;
12             if $S^- \cap C = \varnothing$ then
13                 $S' \leftarrow S - S^-$;
14                 $B' \leftarrow$ Build(Reverse($S'$));
15                 if $B'$ built successfully then
16                     builds $\leftarrow$ True;
17                     break
18         if builds then
19             $V' \leftarrow$ Execute($B', \mathcal{I}$);
20             if $V = V'$ then
21                 $S \leftarrow S'$;
22                 deleted $\leftarrow$ True
23             else
24                 $i \leftarrow i + 1$;
25 until $\neg$deleted;
26 return Reverse($S$)
```

Algorithm 1: QSES algorithm. The main change with respect to ORBS is highlighted.

Taking the intuition behind QSES one step further, the inclusion of all necessary supporting computations in a closure slice suggests that the ORBS execution check is superfluous. In other words because all the lines that are important for preserving the behavior of the program with respect to the slicing criterion are part of the static closure slice, C, any program that includes these lines and compiles should be an executable static slice. Thus, we can simplify QSES by removing the execution check (deleting Lines 19-20 and making Lines 21-24 subordinate to the if statement on Line 18). We refer to the resulting algorithm as QSES$_C$ (QSES Compile only). Program execution is expensive for ORBS and we expect QSES$_C$ to slice more efficiently than QSES.

IV. Evaluation

We implemented QSES using CodeSurfer and then modifying ORBS to not delete lines identified by CodeSurfer. Moreover, for QSES$_C$, we modified ORBS to not attempt to execute the code if it successfully compiles. To empirically evaluate the two approaches, we quantitatively compare the resulting slices and qualitatively consider some of the more interesting examples using the following three research questions.

- **RQ1: What needs to be added to a static closure slice to render it executable?** As a QSES slice is an extension of a closure slice, we look at how much code needs to be added to the closure slice and we investigate the different patterns of code additions.
- **RQ2: What is the impact on the resulting slices of disabling ORBS' execution check in QSES$_C$?** In order to understand whether the static slice's semantic guarantees are sufficient to ensure that a compilable program will execute correctly, we compare the slices produced by QSES and QSES$_C$ in terms of their size, and any semantic differences.
- **RQ3: What is the impact on the slicing time of disabling ORBS' execution check in QSES$_C$?** Running each potential slice during the ORBS slicing process is expensive, especially when a potential slice fails to terminate and ORBS must wait for a watchdog timer to expire. We investigate the performance gained by QSES$_C$ over QSES, by relying on the compilable aspect alone.

A. Subjects

We study 57 C programs gathered from three sources:

- Programs from the slicing literature: the original example of Weiser [1], the SCAM Mug [18], the Montréal Boat Example [19], and Word Count [2].
- Programs from the Mälardalen WCET research group [20], which have been used to compare and evaluate WCET analysis tools.
- Programs from the *Benchmarks Game* [21], which are designed to benchmark language implementations.

We omitted programs from the last two suites 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 which we used to normalize (i.e., parse and pretty-print) and instrument the programs with a slicing criterion (i.e., a specific printf statement that captures the criterion). After normalization, these programs range from 21 SLOC to 2988 SLOC, with an average of 192 SLOC, which renders reviewing all slices by hand feasible. For each program, we consider each assignment to a scalar variable as a separate slicing criterion. This results in a total of 2389 slices, with an average of 42 slices per source program.

B. Results

Our results are overviewed in Figure 1. Consistent with the goals of the NIER track, this section focuses on Research Question 1. To provide some context for our future research, we summarize our preliminary findings for the remaining two research questions.

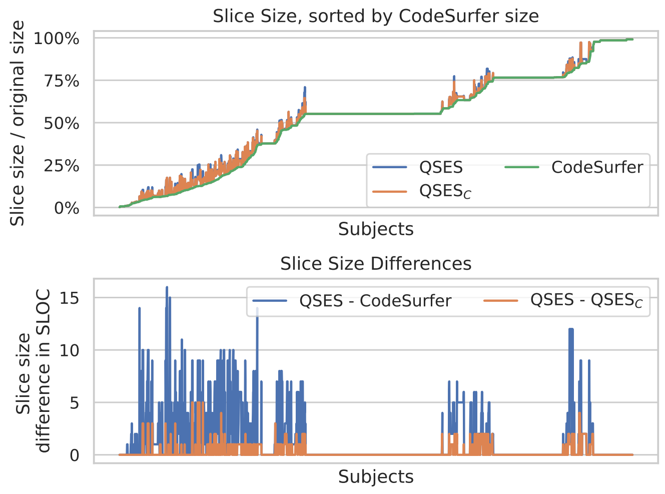

Fig. 1. Size of the slices as percentage of the original program, and differences between slice sizes in SLOC.

a) RQ1: RQ1 compares the slices produced by CodeSurfer and QSES. Most differences are caused by lines containing a single brace (i.e., '{' or '}'). While essential in maintaining the block structure of the code at the source level, such lines are not represented in the SDG used by CodeSurfer and thus not marked as protected lines of the static closure slice. Numerically, these lines account for 87.6% of the 24 625 lines showing a difference. Because these differences are otherwise uninteresting, henceforth we ignore such lines.

The inclusion of required braces alone is sufficient for 65.3% of the slices, meaning that almost two-thirds of the time the closure slice is a few braces away from being a static executable slice. To make the remaining 829 slices executable, QSES adds to the closure slice an average of 3.7 lines, with a maximum of 16 lines (see the lower graph in Figure 1). A manual inspection of the 3055 added lines found in the 829 slices reveals that 59% are related to the granularity at which CodeSurfer slices, 39% to CodeSurfer's representation of declarations and control dependence, and the remaining 2% a range of diverse causes.

In greater detail, the first group is made up of declarations not protected by CodeSurfer because it slices at a finer level of granularity than ORBS. Thus its slices can include a call without including the actual parameters or the assignment of the return value to a variable. Because ORBS must include the entire line of text that includes a call, it causes QSES to require the lines that declare the actual parameters and the variable assigned the return value. For example, the following excerpt from a CodeSurfer slice of word-count only requires the return value of the call to scanf to maintain the same loop iterations and thus does not include the two actual parameters. Therefore, the declaration char c goes unprotected but is required for the program to correctly compile and execute.

```
char c;          // unprotected
...
while (scanf("%c", &c) == 1)
```

The second group is made up of source code that has no direct representation in CodeSurfer's SDG. For example, the representation of an if statement includes control dependence edges labeled either true or false, but no explicit representation of the keyword else. Thus, the line containing else in the source code is never protected by CodeSurfer. This is exemplified by the following code. CodeSurfer, while capturing the semantic information that the printf call occurs in the false branch of the if, protects only Lines 1 and 7. Either Line 3 or Line 5 can be removed as they are each independent of the slicing criterion, but at least one needs to remain in the slice to preserve its behavior. QSES removes Line 5 as it proceeds from bottom to top; hence Line 3, even though it was not protected by CodeSurfer, is part of the QSES slice.

```
1 if (d == 0.0)
2 {
3   return 0;
4 }
5 else
6 {
7   printf("\nORBS:%f\n", w2);
8 }
```

The same lack of a direct representation applies to typedefs, structs, and any labels in the code, which also go unprotected.

Turning to the differences with more diverse causes, we first consider two examples where QSES uncovers "hidden" dependences that can be challenging to model in a static analysis tool. For example, in the slice of the following function, declared using an old-style C function declaration, MeanA is not used. Therefore CodeSurfer does not include its declaration in the closure slice. However, without the declaration MeanA defaults to type int, which, on the machine being used, is four bytes while a double is eight. As a consequence MeanB's address on the stack changes. To preserve the original behavior, ORBS retains the original declaration of MeanA as a double. The dependence of MeanB on the declaration of MeanA is a static analysis challenge.

```
void Calc_LinCorrCoef(ArrayA, ArrayB, MeanA, MeanB)
double ArrayB[];
double MeanA;
double MeanB;
```

The second hidden dependence example is one that CodeSurfer can be configured to model, which is disabled in the default configuration. The value of len in the following code is dependent on the assignment to buflen. Without the assignment, the last argument of the call to read becomes negative, effectively terminating the loop early because of an error.

```
while (len = read(in, buf + end, (buflen - 256) - end)) {
  ...
  buflen = (buflen >= _1M) ? (buflen + _1M) : (buflen * 2);
}
```

CodeSurfer can be configured to include this dependence, but in so doing becomes more conservative in its treatment of IO library calls, which can lead to increased slice size. A clear advantage of QSES is that it can cover a *middle ground* between the two configurations provided by CodeSurfer.

A final example is tooling related as it is caused by the deletion window size used by QSES. The following code can only be removed by QSES when using a window size $\delta \geq 6$ as all six lines need to be removed together in order to preserve the semantics.

```
while (1)
{
  {
    break;
  }
}
```

b) RQ2: QSES$_C$ is based on the assumption that because the static slice includes all dependent computations, these computations should have the correct behavior. Thus in theory, the QSES execution check not only wastes time, but also leads to larger than necessary slices.

Interestingly, disabling the execution check results in slices that are only marginally smaller: 80% of the slices are identical between QSES and QSES$_C$, and on average a QSES slice is only 0.24 lines longer than a QSES$_C$ slice. Furthermore, inspecting the resulting slices reveals that QSES$_C$ slices do not always preserve the correct behavior. For example, in the following code, removing the second line causes the call to process to get captured by the loop, which still compiles, but changes the behavior of the program.

```
for (from = to; (*from) != '>'; from--)
  ;
process(from, to);
```

c) RQ3: In theory, ORBS spends considerable time executing potential slices, especially when a potential slice enters an infinite loop. We might thus expect QSES$_C$ to be notably more efficient than QSES. However, we measured that QSES$_C$ only provides a 4.7% average reduction in slicing time. Looking at the individual slices reveals two patterns. First, pORBS aggressively caches past results, which greatly reduces the number of actual compilations and executions required. Second, there are slices for which the improvement is dramatic. For example, one of the SCAM mug slices showed a 63% reduction in computation time because of the number of timeouts QSES encounters while slicing. Perhaps more telling is one of the spectral-norm1 slices where QSES$_C$ is 98% faster. In this case the cause is the time the program takes to run (17 seconds), which makes this example indicative of the performance advantage when slicing larger, more complex, longer running programs.

V. Related Work

A. Observation-Based Slicing

In addition to the work on ORBS [12], discussed in Section II, Binkley et al. [15] consider ORBS and what its slices can tell us about the limits of static slicing. For example, observation can capture dependencies that arise from "back channels" such as data stored and later retrieved from a database. One indicator of such "hidden" dependences is when a static slice is strictly smaller than the corresponding ORBS slice. Similar to ORBS, QSES is able to discover such hidden dependences as illustrated in Section IV.

By its very nature a static slice is an over-approximation to the (undecidable) true slice, while a dynamic slice is an under-approximation to this slice. It is interesting to note that as the input ORBS is given approaches the set of all-possible-inputs, an ORBS slice approaches this slice from below. Investigations such as that of Binkley et al. and the work presented here help us probe and thus better understand the limits of both static and dynamic dependence analysis. For example, past comparisons between the slices of ORBS and CodeSurfer have a hard time distinguishing code related to a slice being static from code related to it being executable. QSES fills this void.

B. Combination of Static and Dynamic Slicing

Static and dynamic slicing have been combined before. *Conditioned* slicing is a generalization of static and dynamic slicing. Fox et al. present an approach to compute a conditioned slice based on symbolic execution and theorem proving to first generate a slice, which is then augmented with the information from a static slice [22]. Our approach instead uses the static slice during dynamic slicing. Gupta et al. present a hybrid slicing approach that, instead of introducing static information by protecting statements of a static slice, introduces dynamic information from breakpoints, calls, and returns, during static slicing, in order to augment the static slicing process [23]. Finally, rather than actually combining static and dynamic slicing, Ashida et al. [24] present four approaches that combine static and dynamic analyses for performing slicing, such as combining a statically computed PDG with execution histories to compute a slice.

VI. Conclusion

This paper introduces and studies *Quasi-Static Executable Slices*. We provide an algorithm to compute such slices, based on first identifying a static closure slice using CodeSurfer before applying observation-based slicing (ORBS) to add in code necessary to yield an executable program. The additions include syntactic elements that are not modeled by CodeSurfer, and statements that are required to preserve the semantics of the program due to, for example, memory dependencies. Other minor differences are related to artifacts of the dynamic slicing approach used (e.g., the difference in granularity between ORBS and CodeSurfer and the choice of the deletion window size used by ORBS).

In the future, we plan a more detailed evaluation that will further characterize the difference between QSES, QSES$_C$, and the original ORBS approach. We also plan to extend the range of slicing criteria beyond scalar assignments, with slices that involve pointers being an obvious place to start. Finally, we will consider the implication QSES brings to the interplay between static and dynamic dependence analysis.

References

[1] M. Weiser, "Program slicing," in *5th International Conference on Software Engineering*, 1981, pp. 439–449.

[2] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Trans. Software Eng.*, 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 Comput. Surv.*, vol. 44, no. 3, Jun. 2012.

[5] L. Philips, J. De Koster, W. De Meuter, and C. De Roover, "Search-based tier assignment for optimising offline availability in multi-tier web applications," *The Art, Science, and Engineering of Programming*, vol. 2, no. 2, 2018.

[6] W. Muylaert and C. De Roover, "Untangling composite commits using program slicing," in *18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM2018)*, 2018, pp. 193–202.

[7] S. Salimi, M. Ebrahimzadeh, and M. Kharrazi, "Improving real-world vulnerability characterization with vulnerable slices," in *16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2020, pp. 11–20.

[8] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984, pp. 177–184.

[9] D. W. Binkley and K. B. Gallagher, "Program slicing," in *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.

[10] B. Korel and J. W. Laski, "Dynamic slicing of computer programs," *J. Syst. Softw.*, vol. 13, no. 3, pp. 187–195, 1990.

[11] S. Horwitz, T. W. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," in *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, 1988, pp. 35–46.

[12] D. W. Binkley, N. Gold, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, "ORBS: language-independent program slicing," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, 2014, pp. 109–120.

[13] T. Teitelbaum, "Codesurfer," *ACM SIGSOFT Softw. Eng. Notes*, vol. 25, no. 1, p. 99, 2000.

[14] S. Islam and D. Binkley, "Porbs: A parallel observation-based slicer," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–3.

[15] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Orbs and the limits of static slicing," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 1–10.

[16] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[17] S. Horwitz, T. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–61, 1990.

[18] M. P. Ward, "Slicing the SCAM mug: A case study in semantic slicing," in *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, 2003, pp. 88–97.

[19] S. Danicic and J. Howroyd, "Montréal boat example," in *Source Code Analysis and Manipulation (SCAM 2002) conference resources website*, 2002.

[20] M. W. research group, "Mälardalen wcet research group's benchmarks," https://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[21] B. Fulgham and I. Gouy, "The computer language benchmarks game," https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[22] C. Fox, M. Harman, R. M. Hierons, and S. Danicic, "Consit: A conditioned program slicer," in *2000 International Conference on Software Maintenance, ICSM 2000*, 2000, p. 216.

[23] R. Gupta, M. L. Soffa, and J. Howard, "Hybrid slicing: Integrating dynamic information with static analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 4, pp. 370–397, 1997.

[24] Y. Ashida, F. Ohata, and K. Inoue, "Slicing methods using static and dynamic analysis information," in *6th Asia-Pacific Software Engineering Conference (APSEC '99)*, 1999, pp. 344–350.