An Empirical Evaluation of Quasi-Static Executable Slices

Quentin Stiévenart, stievenart.quentin@uqam.ca[a], David Binkley, binkley@cs.loyola.edu[b], Coen De Roover, coen.de.roover@vub.be[c]

[a]*Université du Québec à Montréal, QC, Canada*
[b]*Loyola University Maryland, MD, USA*
[c]*Vrije Universiteit Brussel, Belgium*

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, and that can be compiled and executed. We evaluated this new approach on a set of 62 C programs and report our findings.

Keywords: program slicing, static slicing, dynamic slicing, program dependence analysis

1. Introduction

Program slicing is a program decomposition technique that has a wide range of applications in areas such as debugging [1, 2, 3], program comprehension [4, 5], software maintenance [6, 7], re-engineering [8], refactoring [9], testing [10, 11, 12], reverse engineering [13, 14], comprehension [15, 16, 17], tierless or multi-tier programming [18, 19], and vulnerability detection [20].

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" [21]. 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 used to classify program slices [22]. The second dimension, static versus dynamic, was introduced by Korel and Laski [23] 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. [24] 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. Furthermore, although less problematic, the dependence graphs used to compute closure slices typically omit much of the concrete syntax found in the source code. 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 [25].

One recent variation of dynamic slicing, referred to as *observation-based slicing* [25], 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 [26], which efficiently computes static closure slices, and pORBS [27], 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. We share the dataset used for our experiments, along with all the slices produced[1].

The remainder of this paper describes ORBS and CodeSurfer, and then introduces QSES in greater detail, provides an empirical evaluation of QSES by considering four research questions, and finally presents some related work and concluding remarks.

This paper extends our earlier short paper [28] in the following ways:

- We extend our evaluation with three larger multi-file programs which increase the number of slices taken by 15%.

- We double the number of research questions considered and thus the number of experiments presented. We also expand upon the two original research questions. The short paper includes a preliminary exploration of research questions RQ1 and RQ4. In this paper we consider two new research questions: in RQ2 we compare QSES to the dynamic slices produced by ORBS, and in RQ3 we evaluate the impact of window size, a parameter of the slicing approach, on the slices and the time required to compute them. While the more technical RQ3

[1]`https://doi.org/10.5281/zenodo.7702888`

confirms that previous empirical results for ORBS also apply to QSES, RQ2 provides an important lower bound. By better understanding how QSES compares to ORBS and static slicing, we can better understand the relationship between static and dynamic source-code analysis.

2. Background: ORBS and CodeSurfer

Observation-based slicing. Our approach builds on Observation-Based Slicing (ORBS) [25], 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. The set of inputs $\mathcal{I}$ is sometimes referred to as the test suite of the program being sliced. The slice computed by ORBS compiles and preserves the semantics of program P for this set of inputs $\mathcal{I}$, so the quality of the test suite has an influence on the quality of the slice produced. ORBS is language agnostic so it considers the program as a sequence of lines of text, l_1 to l_n. It can be made slightly more efficient, by considering, for example, program statements, at the expense of performing minimal language-specific parsing [29].

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 for 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 8-24), 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 the result, S, is the dynamic observation-based slice.

In principle the removal might consider arbitrary subsets of the program or, as with delta debugging [30], start by considering half the program, then quarters, etc., but these approaches can prove quite expensive [25]. Instead ORBS uses a window of up to δ *contiguous* lines. It thus computes 1-minimal dynamic slices [25] (no single line can be removed from the slice). Past empirical work has found that the value $\delta = 4$ does a good job of balancing slice size and slice computation time [25, 27].

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) [24, 31] 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.

3. Approach

One obvious advantage of ORBS 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 expected to be 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.

3.1. Quasi-Static Executable Slicing (QSES)

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 the deletion 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 $j \leftarrow 1$ to $\delta$ 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: The QSES algorithm. The main change with respect to ORBS is highlighted.

3.2. Quasi-Static Executable Slicing, Compilation Only (QSES$_C$)

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, our speculation is that 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. Intuitively, QSES$_C$'s correctness follows from the closure slice being a sound over-approximation of the minimal slice. Any compiling subset of the original program that includes these lines should have the correct semantics. QSES$_C$ is basically identifying a minimal set of lines needed to make the closure slice compilable. We evaluate this intuition in Section 4.

3.3. Implementation concerns

The implementation used in the experiments makes use of two enhancements relative to the approach described above. First, compared to our preliminary investigation [28], we enhance the closure slices produced by CodeSurfer as follows. In the SDG the representation of an if statement includes control dependence edges labeled either true or false, but no explicit representation of the keyword else. Thus, an else in the source is never included in the set of CodeSurfer protected lines. ORBS will only remove this keyword when it is safe to do so, but the removal often leads to uninteresting, superficial, and even distracting changes to the code. The improved implementation, which produced the slices studied in this paper, includes a pass over the code where we protect else statements whose matching if statement was protected by CodeSurfer.

The second enhancement is that we make use of a parallel variant of ORBS, pORBS [27]. In short pORBS spawns a thread for each value k between 1 and δ, in which it tries to remove k lines. After all threads have finished, it removes the lines from the successful attempt by the thread with the highest k. Compared to the sequential version of ORBS given as Algorithm 1, this parallelism tends to decrease the wall-clock time required to slice a program at the expense of CPU time because after computing the result for each value of k only the largest successful deletion is kept. The wall-clock time measures the elapsed time between the instant the slicing process began and the instant it finished, while CPU time measures actual processor usage of the slicing processes.

4. Evaluation

To empirically evaluate QSES, we compare its slices to those produced by CodeSurfer, ORBS, and QSES$_C$. These comparisons enable characterizing where QSES lies on the spectrum of slicing approaches, from precise but potentially incomplete (e.g., ORBS) to complete but imprecise ones (e.g., CodeSurfer). CodeSurfer has been chosen as the static slicer in our experiments, as it implements the typical slicing algorithm based on dependence graphs [26], and is one of the few slicers that has had an available executable. We quantitatively compare the resulting slices and qualitatively consider some of the more interesting examples using the following four 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 the different patterns of code additions.
- **RQ2: How do QSES slices compare to ORBS slices?** ORBS slices, being purely dynamic, are expected to be smaller. However, by their very nature they are expected to take longer to compute. We investigate the difference in timing, size, and deletion patterns between ORBS and QSES.
- **RQ3: What is the impact of window size on the slices and the slicing time?** A smaller window size can make it hard to remove groups of statements, while larger window sizes may waste time attempting to remove large chunks of code. Prior work [25] found that for ORBS, a window size of four strikes a good balance between these two. We investigate whether the same holds for QSES.
- **RQ4: What is the impact on the resulting slices of using QSES$_C$, which disables ORBS' execution check?** In order to understand whether the static slice's semantic guarantees are sufficient

program	**SLOC**	**program**	**SLOC**	**program**	**SLOC**	**program**	**SLOC**
adpcm	585.1	fankuchredux5	115.5	jfdctint	119.0	ns	30.5
bc (bc)	8511.0	fasta1	126.1	lcdnum	62.0	nsichneu	2989.0
bc (execute)	9500.1	fasta2	264.1	lms	172.1	prime	51.0
binary-trees1	91.0	fasta3	90.0	ludcmp	109.3	qsort-exam	124.0
bs	46.0	fasta5	109.3	mandelbrot2	66.0	qurt	120.0
bsort100	61.0	fasta7	231.1	mandelbrot9	66.0	reverse-complement5	83.0
cnt	76.5	fasta8	150.6	matmult	54.1	reverse-complement6	96.0
compress	357.0	fasta9	161.4	mbe	63.1	scam	63.1
cover	625.0	fdct	138.0	minver	201.3	select	131.0
crc	94.0	fft1	128.1	nbody1	92.2	spectral-norm1	57.6
duff	44.0	fibcall	27.0	nbody2	107.0	st	98.4
ed (main_loop)	4485.0	fir	54.2	nbody3	90.1	statemate	1354.0
edn	170.1	indent (indent)	6680.0	nbody6	93.2	sumprod	17.1
expint	73.1	indent (parse)	5117.2	nbody7	137.0	ud	81.4
fac	23.7	insertsort	33.0	ndes	196.1	wc	49.0
fankuchredux1	79.4	janne_complex	38.0				

Table 1: Programs used in our evaluation. Source lines of code (SLOC) computed by sloccount_c includes only non-comment, non-blank lines of code. We report the mean SLOC across all sliced versions of each program, because the instrumentation used to perform slicing adds one or two lines depending on the slicing criterion. For multi-file programs the file sliced is shown in parentheses.

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 semantics.

We share the dataset containing our evaluation data, namely the benchmark programs, the static slices produced by CodeSurfer, the ORBS slices, the QSES slices, and the QSES_C slices[2].

4.1. Evaluation Method

Test inputs. Many of the benchmarks hard code a representative test "input". We run `wc` on itself and both `mbe` and `scam` run on sufficient inputs to cover all possibilities (thus the resulting slices are the minimal *static* slice). For the larger programs such as `bc`, we use a representative subset of the test suite that accompanied the program. As we find later, QSES is much less dependent than ORBS on the test inputs chosen.

Comparing slices. In order to compare the slices produced by the different slicers, we rely on `diff`. As we use QSES, CodeSurfer, and ORBS in a line-based setting, the only differences between slices are at the level of lines.

Normalization. We normalize the benchmark programs using the `pycparser` library. We parse each program and pretty-print it again. We also instrument the programs to add a `printf` statement that captures the slicing criterion.

Normalization accomplishes two main goals. First it removes stylistic differences from the comparison of slices from different programs. Second it places opening and closing braces each on their own line. This helps ORBS delete, for example, unnecessary conditions that must otherwise been retained if they share the same line as the opening brace of the following block.

Subjects. The experiments consider subjects written in `C` because the static slicer used, CodeSurfer, slices `C` code. Like ORBS, QSES is language agnostic: both are applicable to any programming language with textual source code, as they operate at the level of lines of source text.We study 62 `C` programs gathered from the following sources:

[2]`https://doi.org/10.5281/zenodo.7702888`

- Programs from the slicing literature: the original example of Weiser [1], the SCAM Mug example [32], the Montréal Boat example [33], and word count [6].
- Programs from the Mälardalen WCET research group [34], which have been used to compare and evaluate WCET analysis tools.
- Programs from the *Benchmarks Game* [35], which are designed to benchmark language implementations.
- Three multi-file systems used in previous slicing studies [36].

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.

Supplementing the subjects of our initial study [28], the last source provides three multi-file programs, bc, ed, and indent. For these programs QSES, like ORBS, is configured to slice a specified file from the program. Thus there is no practical limit on the size of the program it can slice. However, CodeSurfer's whole-program analysis can take considerable time to process programs over a few hundred thousand lines of code. Fortunately, this step need only be performed once as the resulting SDG can be saved and repeatedly sliced. For the three multi-file programs we consider two files from bc and indent, and one from ed.

After normalization, the files to be sliced range from 17 to 3 335 SLOC, with an average of 1 402 SLOC. For each program, we consider each assignment to a scalar variable as a slicing criterion. This results in a total of 2 741 slices, with an average of 44 slices per source program. The list of programs used in our evaluation is given in Table 1. Note that for multi-file programs only the file to be sliced is normalized, which accounts for the SLOC differences for the two versions of bc and the two version of indent.

The data for this paper was collected under CentOS Linux release 7.5.1804 running Linux version 3.10.0-862.3.3.el7.x86_64. The code was compiled using clang version 13.0.0.

4.2. Results

This section first considers the data related to each research question in turn. It then discusses the broader implications of the results. Finally, this section consider threats to the validity of the experiments.

RQ1. RQ1 compares the slices produced by CodeSurfer and QSES. Our results are overviewed in Figure 1. In aggregate, the SLOC count of the static slices totals 1 282 946 lines. QSES increases this to 1 900 340. However, most of the added lines contain a single brace (i.e., '{' or '}'). While essential to maintain 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 slices. Numerically, 582 347 (94.3%) of the 617 394 added lines contain a single brace. Because they are otherwise uninteresting, henceforth we ignore such lines.

Ignoring braces, QSES increases the total SLOC count to 1 317 993, which is only a 2.7% increase over the size of the static slices. Viewed per slice, the average CodeSurfer slice is 48.2% of the original code, while the average QSES slice is 49.9%, only 1.7 percentage points more per slice on average. Visually, the top graph of Figure 1 shows these additions normalized across all programs as the percent of the original program's size. The increase is surprisingly small, which indicates that CodeSurfer's closure slices are very close to being executable slices.

The lower graph of Figure 1 shows the size of each addition in SLOC sorted by the magnitude of the difference instead of by QSES slice size as in the top graph. Of the 2 741 slices, 1 479, or just over 50%, require no additions. In the remaining 1 262 slices QSES retains an average of 28 additional lines, with a maximum of 872 lines. Thus for more than half of the cases, the static closure slice is, in fact, already an executable slice (once any necessary braces have been added). For ten of the 62 programs *all* of the slices were executable. These ten are not limited to the smaller programs that have a limited number of slices. For example, they include the 552 slices of nsichneu and the 364 slices of statemate. However, at the other end of the spectrum twenty programs produced no executable slices.

In total QSES includes 35 047 lines not found in the closure slices. Because many of these lines (e.g., enum and struct) declarations appear in multiple slices, it turns out that the additions involve only 977 unique lines, which makes it manageable to manually consider all of them. Of the 977 unique lines, 484 (49%) are type and variable declarations while the others are various executable statements. One might expect a similar 50/50 split of the 35 047 lines, which is why it is somewhat surprising that 24 220 (69%) are declarations, while only 10 827 (31%) are executable statements.

To better understand the differences, we had a closer look at the slices in the smaller programs (829 slices in total). Inspecting the causes, 59% of the

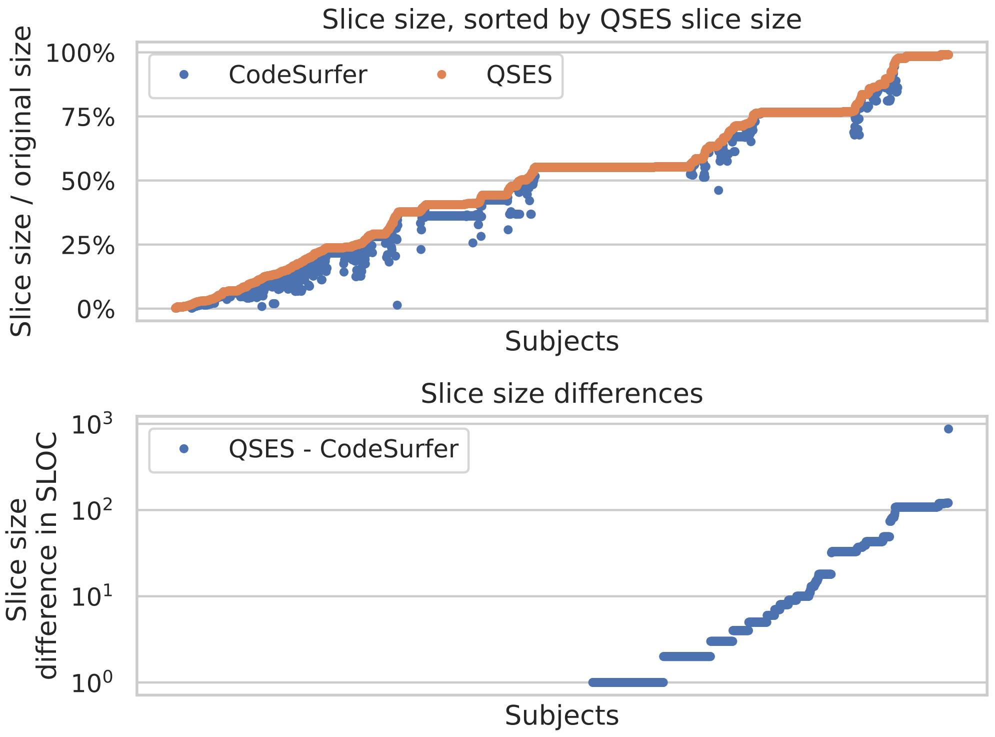

Figure 1: Size comparison for the slices produced by QSES and CodeSurfer. (Note that the y-axis of the lower graph uses a log scale.)

differences are caused by the granularity at which CodeSurfer slices, 39% by CodeSurfer's representation of declarations and control dependence, and the remaining 2% by a range of diverse causes.

In more detail, the first group is caused by CodeSurfer slicing at a finer level of granularity than ORBS. CodeSurfer works at the expression level while QSES, like ORBS, works at the line-of-text level. For example, CodeSurfer can include a call in a slice without including the actual parameters. Because ORBS must include the entire line of text, it causes QSES to include the lines that declare the actual parameters, and in some cases those that compute their values. However, the corresponding formals go unused in the called function (because they are not in the slice), thus the code computing their values need only be included if its absence leads to abnormal termination.

For example, the following excerpt from a 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 as it is not part of the closure slice, but is required for the program to compile and execute.

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

As a second granularity related example consider the following code where only the side effect h_ptr++ is required by the closure slice. In this example QSES must include the entire line and thus the declaration of ac_ptr and xa1, and in this case, the initialization of ac_ptr to avoid a memory access violation.

```
int accumc[11];
int *ac_ptr;
long int xa1;
...
ac_ptr = accumc;
...
xa1 += ((long) (*(ac_ptr++))) * (*(h_ptr++));
...
```

A less common granularity related impact found only in the multi-file programs, occurs when there is a call from another file to a function in the file being sliced. Even if such a function is not in the slice, because of the external call it is necessary to retain an empty function definition to keep the linker happy. Slicing the whole program rather than a single file from within the program would avoid this issue. Likewise, QSES must retain global declarations such as YYSTYPE yylval and its structure definition when referenced in another file.

The second group of differences is made up of source code that has no direct representation in CodeSurfer's SDG. These lines include, for example, typedefs and structs as well as lines that include only a label in the code. In the latter case, the SDG's control dependence analysis captures the control-flow impact of uses of a label, but the label itself is not explicitly represented in the SDG. If they were more prevalent, QSES could give such labels special treatment similar to the special treatment that it gives else.

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 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;

```

A second example of a hidden dependence is one that CodeSurfer's ability to model is disabled in the default configuration. CodeSurfer includes dependence models for standard library functions. For example, the conservative approach to modeling printf is to include the dependence on the initial value of stdout and the fact that printf modifies stdout. However, including these dependences means that each printf is transitively dependent on all the printfs that might execute before it. Including all these calls bloats the slice with uninteresting code.

Likewise, the default model for read does not capture the dependency between the arguments and the return value. For example, in the following code the value of end is transitively dependent on the assignment to buflen in Line 7. Without this assignment, the last argument of the call to read becomes negative, effectively terminating the loop early because read returns an error. One of ORBS strengths is that it is able to selectively include such dependences.

```
while (len = read(in, buf + end, (buflen - 256) - end))
{
  ...
  end += len;    // slice here on the value of "end"
  if (end >= (buflen - 256))
  {
    buflen = (buflen >= _1M) ? (buflen + _1M) : (buflen * 2);
    buf = realloc(buf, buflen);
```

As this example illustrates, the more dynamic treatment of library calls is a clear advantage of QSES as it can be selective between the *include all* and *include none* options provided by static slicing.

RQ1 Summary

In summary, a simple answer to RQ1's "What needs to be added to a static closure slice to render it executable?" is "very little". The bulk of additions are declarations caused by CodeSurfer slicing at a finer level of granularity than QSES. This is a fundamental difference where a closure slice need not include parts of the code that are necessary for its compilation and execution. While fewer in number, perhaps the more interesting additions are related to *hidden dependences* whose static modeling is very challenging.

RQ2. RQ2 complements RQ1. Whereas RQ1 compares QSES slices with closure slices, RQ2 compares QSES slices to ORBS slices (see Figure 2, which mirrors Figure 1). As ORBS is not required to include the lines of a static closure slice, we expect it to produce smaller slices. It does so the majority of the time. Numerically, the total size of the ORBS slices is 93 842 SLOC, just 7% of QSES' 1 317 993 SLOC. On average, an ORBS slice is only 12.4% of the original program. This is in line with the slice sizes reported in the original paper presenting ORBS [25].

Of the 2 741 slices, for 2 437 (88.9%) the ORBS slice is smaller. Of the remaining slices, 296 (10.8%) are the same size while in eight cases (0.3%) the ORBS slice is actually larger. The top graph of Figure 2 visualizes these additions normalized across all programs as a percentage of the original program's size. Compared to the top graph of Figure 1, ORBS' higher slice-size variation is evident. The lower graph shows the size of each difference, sorted by the magnitude of the difference instead of by QSES slice size as in

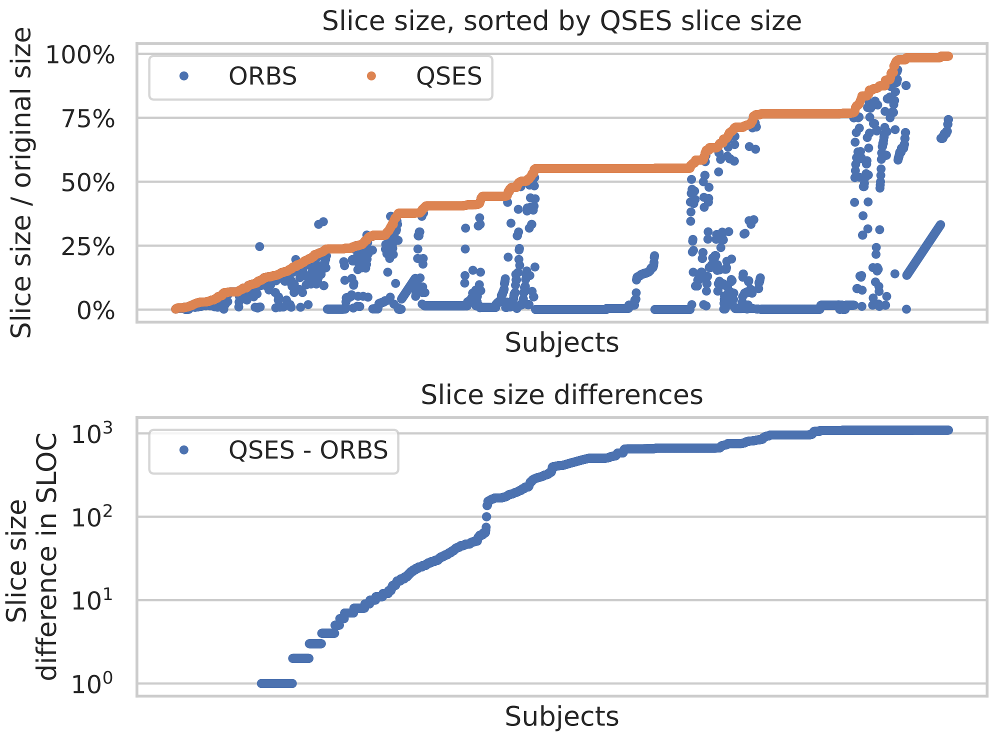

Figure 2: Size comparison for the slices produced by QSES and ORBS.

the top graph. Again compared to the lower graph of Figure 1, the overall larger per-slice difference can be clearly seen.

To provide another view of this comparison, Figure 3 depicts a scatter plot comparing the sizes. Points on the $x = y$ diagonal are slices where QSES and ORBS produce the same size slice, while points below this line are cases for which the ORBS slice is smaller, and above the line are the rare cases for which the ORBS slice is larger. Visually evident in the graph are vertically stacked points. These are caused by programs where QSES produces slices of similar size, while ORBS, by its dynamic nature, removes different portions of the code for different criteria. Thus these stacks visually illustrate how the static analysis over-approximation can vary in severity.

Also evident in this graph, as points found at the bottom of the graph, are slices where the test suite fails to execute the slicing criteria. In these cases the ORBS slice will include only the empty definition of main(). This is the case for 713 of the 2 741 slices in total. Thus the tests used are attaining approximately 74% coverage. Overall, removing these slices has a minimal impact on the big picture: numerically, the total size of the remaining ORBS slices is 93 129 SLOC, or 12.5% of 743 174 SLOC total for the corresponding QSES slices. Slices for which the QSES slice is larger are a result of the over-approximation of the static analysis of CodeSurfer. For example, consider a pointer for which static analysis determines it may point to some structure. If the pointer never points to this structure during any of the runs of the program, the code supporting that structure will not be part of the ORBS slice, but will be present in the QSES slice. Of the 2 028 remaining slices, for 1 724 (85.0%) the ORBS slice is smaller. Of the remaining slices, 296 (14.6%) are the same size while in eight cases (0.4%) the ORBS slice is actually larger.

At first glance one might expect that an ORBS slice would never be larger than the corresponding QSES slice. In theory, given a sufficiently strict semantics, this is in fact the case. However, for C programs it turns out that there are a few cases where ORBS produces larger slices than QSES. In the results of our experiment, all of these have the same underlying cause. In short, if early on ORBS performs an ill-advised deletion, it will subsequently be required to retain considerable code. In the following excerpt, for example, ORBS deletes Line 2 although it is protected in the static closure slice. This preserves the correct execution behavior as long as the stack has a 0 at the location used for cur_tid, which requires retaining other variable declarations, that are otherwise not part of the slice, so that cur_tid remains at the same stack location.

```
1 int cur_tid;
2 cur_tid = 0;
3 printf("\nORBS:%x\n", cur_tid)
```

One final difference is that QSES's use of the static closure slice helps it to preserve the structure of the code whereas ORBS changes the code's appearance. While ORBS still preserves the code's semantics, structural changes can be expensive to an engineer. In the following example, the else on Line 5 is protected as part of the closure slice. ORBS deletes it, which has no effect on the code's behavior thanks to the return on Line 4.

```
1 int p(int j)
2 {
```

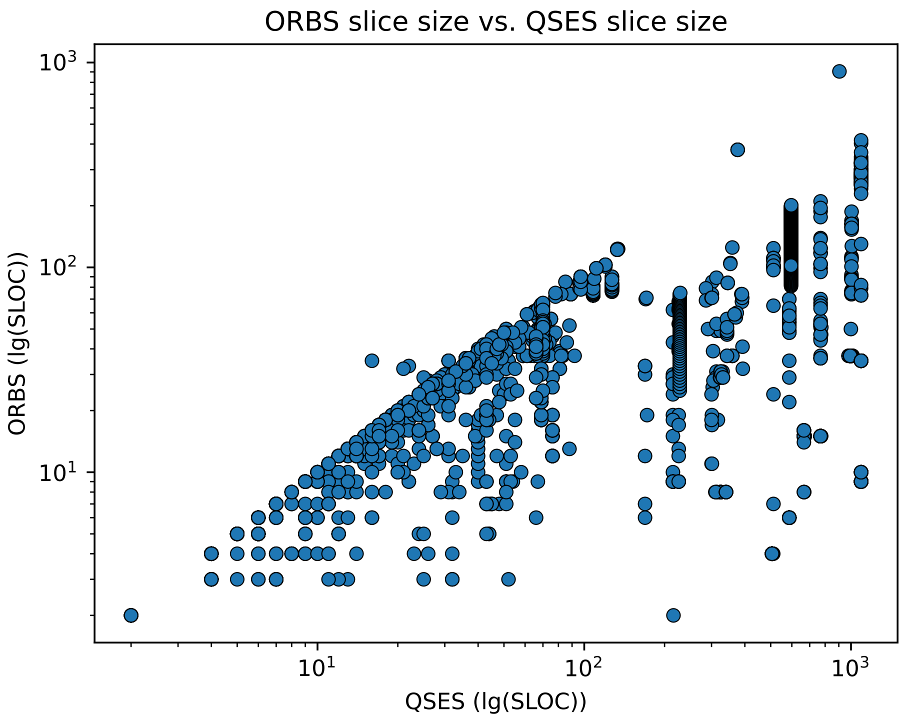

Figure 3: QSES and ORBS slice sizes. Each individual point represents a slice. Slice sizes are given in SLOC using a log scale.

```
  if(j > 0)
    return 1;
  else
    return 0;
}
```

Compared to CodeSurfer, ORBS, and thus QSES, are expensive. We consider the relative times for ORBS and QSES to understand the impact of protecting the lines of the closure slice on their relative execution time. Table 2 shows the wall-clock and CPU time taken by ORBS and QSES as well as the number of executions used by each slicer. The ORBS implementation delegates the compilation (Line 15 of Algorithm 1) to an external script, which calls a compiler such as gcc. We wanted the QSES implementation to be as similar as possible to that of ORBS, so we updated this script to first check if the deletion intersects with the static closure slice (Line 12). A more efficient, but more invasive, approach would be for QSES to modify ORBS' initialization so that all lines of the closure slice are marked as untouchable. This would eliminate the overhead of creating external processes. Based on the speedup achieved by having ORBS mark blank lines as deleted, this would no doubt be faster. However, it might influence the internal pattern of lines considered, so we have opted for the less invasive approach.

Similar to compilation, the ORBS implementation delegates Line 19 of Algorithm 1 to an external script that executes the program on the test suite. Executions can be particularly expensive when the watchdog timer is forced to kill them. Table 2 compares the number of times the execution script is invoked. It is clear from this data that QSES greatly reduces the number of executions required. The reduction in the number of executions hints at the potential performance improvement that more direct line-protection would achieve.

	ORBS	QSES	Percent Reduction
Mean Time			
per slice Wall Clock (sec)	686	581	15%
per slice CPU (sec)	1 679	1 371	18%
ORBS Executions			
Mean	1 756	637	64%
Median	547	152	72%
Total	4 813 450	1 747 642	64%

Table 2: Execution comparison

RQ2 Summary

In summary for RQ2, we find the expected pattern where ORBS, being purely dynamic, produces considerably smaller slices. The vertical columns of Figure 3 visually show how the requirements of the particular slice are more apparent when only dynamic dependences are taken into account. In the extreme case where the test suite fails to execute the slicing criteria, the ORBS slice will only include an empty definition for `main`. As QSES relies on CodeSurfer, it will suffer from the over-approximations made by CodeSurfer, generally resulting in larger slices. In QSES' favor ORBS can "gets itself into trouble" as we see in the few cases where it produces larger slices. ORBS being dependent on the execution of the code, it is able to exploit platform-specific behavior to remove lines from a slice, which can reduce the slice size, but may also prevent later deletions. In terms of timing, QSES reduces the slicing time by 15%. In summary QSES provides a more stable and faster slicer, at the cost of larger slices.

RQ3. RQ3 considers the impact of window size on slice size and on the time required to compute the slices. To that end, we computed each slice using the window sizes $\delta \in \{1, 2, 3, 4, 5, 6, 7, 8, 12, 16\}$. Figure 4 depicts the resulting slice sizes and slicing time.

We can clearly see that initially slice size decreases sharply as the window size increases. This is because for small window sizes, a number of patterns are impossible to remove from a slice. For example, with $\delta = 1$ it is impossible to remove a pair of braces on two consecutive lines. As the window size increases, slice size drops. However, numerically this reduction stabilizes at $\delta = 4$. Even though larger window sizes still result in smaller slice sizes, only a handful of extra lines get removed. For example, increasing δ from 12 to 16 results in only a total of three additional lines being removed over all slices. The differences between $\delta = 4$ and $\delta = 16$ is only 294 lines, which represents a mere 0.02% reduction. Statistically, Tukey's honestly significant difference (HSD) test using program and δ as explanatory variables, finds the QSES slice size larger for $\delta \leq 2$ than the other values ($p < 0.001$).

The lower graph of Figure 4 shows how wall-clock time decreases as δ increases while CPU time increases. The wall-clock time visually stabilizes near $\delta = 4$ or 5 while beyond $\delta = 4$ the CPU time consistently grows. Applied to the times, Tukey's HSD test shows very little difference. For wall-clock time $\delta = 4$-8 are faster than $\delta = 1$ and 2 (only 7 and 8 are faster than 3). While for CPU time $\delta = 4$-8 take more time than $\delta = 1$ while only $\delta = 8$ takes more time than $\delta = 2$ and 3.

RQ3 Summary

In summary for RQ3, both slice size and wall-clock time decrease as the window size δ increases, while CPU time increases. However, returns are diminishing for higher values of δ, where the slice size only decreases slightly. While there is no clear sweet spot, similar to previous studies [25, 27] $\delta = 4$, seems to strike a good balance.

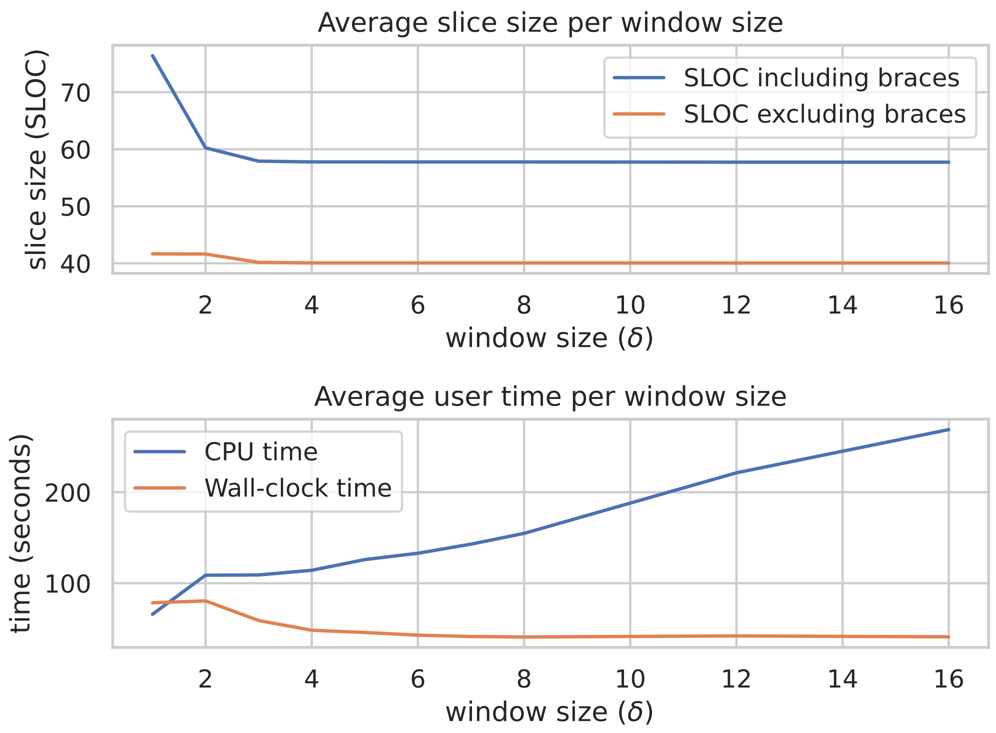

Figure 4: Effect of window size on slice size and slicing time. (Formally the windows sizes are discrete, however we treat them as continuous and draw lines to help the patters stand out.)

RQ4. RQ4 investigates whether all computations depended on by the static closure slice are included and behave correctly.

From the perspective of taint analysis, no tainted values can ever enter the computation of the slice. Thus in theory QSES_C should be sufficient because it includes all the code semantically needed from the closure slice, and it compiles. This makes the QSES execution check not only a waste of time, but undesirable because if it fails it may cause the retention of unnecessary code.

To investigate RQ4, we compute the 2 741 slices using QSES_C and then compare them with the slices produced by QSES. If the theory holds then the QSES_C slice will be no larger than the QSES slice. A QSES slice might be larger when the execution check leads to the retention of additional code. This code is either unnecessary or indicates that there are dependences that are not accounted for during static analysis. The latter suggests areas of future static analysis work. Finally, the number of such slices provides an indication of how often the theory does not hold.

The code found in the QSES slices that is not found in the QSES_C slices is minimal as the sum of the QSES_C slice sizes, 1 315 747, is 99.83% of the 1 317 993 sum of the QSES slice sizes. Size-wise 158 (5.76%) of the QSES_C slices are smaller, 2583 (94.24%) are the same size, and none are larger. The average addition per slice is less than a line, 0.81 SLOC. At the level of programs, 41 programs have identical QSES and QSES_C slices, while only one has all of its slices differ. However, inspecting the resulting slices reveals that QSES_C slices do not always preserve the correct behavior. We consider several examples.

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.

```
1 for (from = to; (*from) != '>'; from--)
2   ;
3 process(from, to);
```

This is an artifact of ORBS working at the line-of-text level while CodeSurfer works at the statement level.

As a second example, QSES_C never includes C's do keyword because the code always compiles without it. However, QSES will include this keyword when the loop executes at least two times. While not practically feasible, straightforward modification of CodeSurfer would enable the retention of the `do` keyword.

A third example is tooling related as it is caused by the QSES deletion window size. 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 execution semantics.

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

With $\delta \geq 2$ QSES$_C$ can delete all six lines although it takes several iterations.

Finally, CodeSurfer protects struct fields that are required in the slice, which means that unneeded struct fields are removed by QSES$_C$. However, the initialization of such structs is not adapted by QSES$_C$, hence the program semantics change. This is exemplified by the following code, where Line 3 is removed by QSES$_C$.

```
1 struct aminoacids
2 {
3   char c;
4   double p;
5 };
6
7 ...
8 struct aminoacids iub[] = {{'a', 0.27}, {'c', 0.12}, ...};
```

RQ4 Summary

In summary for RQ4, we find that disabling the execution check of QSES may result in semantic differences. These differences stem from two main sources: tooling artifacts, which could be resolved by straightforward updates to CodeSurfer, and the different granularity at which ORBS (line level) and CodeSurfer (expression level) operate. In other cases being able to skip the execution check enables QSES$_C$ to delete sections of the code that QSES could only remove if using a larger deletion window.

Discussion. From advanced compilation to software engineering tools, dependence analysis forms an essential component of source code analysis. Program slicing provides an excellent vehicle to study the interplay between static and dynamic dependence analysis. By definition a slice includes precisely those program elements needed to preserve the behavior of the slicing criteria. Due to their inherent limitations, static analysis tends to over-approximate this precise slice while dynamic analysis tends to under-approximates it. While not practical to compute, the ORBS slice constructed using all possible inputs is identical to the precise slice. Thus the QSES slice, constructed using all possible inputs, would include the precise slice *plus* code attributed to the static over-approximation.

For example, the vertically stacked points in Figure 3 visually illustrate how the static analysis over-approximation can vary in severity. In this case each stack represents a collection of slicing criteria for which QSES produces slices of similar size because they each include essentially the same static slice, while ORBS, by its dynamic nature, removes different portions of the code.

This observation suggests one area of future work with QSES: an exploration into the impact that the test suite has on a QSES slice. As evidenced by the points along the bottom of the graph in Figure 3, an inadequate test suite can have a huge impact on an ORBS slice. Because QSES essentially degenerates into QSES_C when using an empty suite, the results of RQ4 show how QSES is more stable in the presence of an inadequate test suite. The results for RQ1 suggest that the addition of test cases should modestly increase slice size over that of the static closure slice. Thus with a growing test suite, the difference between a QSES slice and an ORBS slice should become an ever better estimate of the static analysis over-approximation and thus provide a vehicle to study the precision of static analysis algorithms. A related question is how quickly the slice stabilizes, which provides, among other things, an indication of the diversity of the tests and thus the quality of the test suite.

Binkley et al. investigated the limits of static analysis by considering the differences between the slices produced by ORBS and those produced by CodeSurfer [29]. They report that these differences have two causes: code needed to make the slice executable and code that is part of the static over-approximation. The separate comparisons supporting RQ1 and RQ2 enable the refinement of this work where in RQ1, the comparison of QSES to CodeSurfer, focuses on the first cause, and RQ2's comparison of QSES and

ORBS focuses on the second.

An excellent example from the discussion of RQ1 comes from the modeling of library code. The safe static-analysis approach is to include a dependence model relating each function's output back to its inputs as done by the *summary graphs* introduced by Horwitz et al. [31]. QSES enables the inspection of the cost of this safety. Omitting these dependence models omits *all* of the code the call depends on. However ORBS will include the necessary subset of this code for the cases where the dependences of the called function influence the slicing criteria. Thus, one of ORBS strengths that QSES inherits is the ability to be selective between the *include all* and *include none* options provided by static analysis.

A related result is QSES' uncovering of hidden dependences. For example, the need for the slice to include a declaration such as `double MeanA` is hard to model in static analysis, especially when, as in this case, the dependences are architecture specific. This however implies that ORBS slices are platform/architecture dependent.

Furthermore, ORBS occasionally gets itself in trouble by deleting code that appears unnecessary only because the location assigned to a variable contains a fortuitous value. As seen in the examples supporting RQ2, the static slice acts as an "anchor" that helps mitigate ORBS dependence on the underlying runtime system.

QSES also provides a vehicle to study the impact of termination on dependence analysis [37, 38, 39]. For example, including all exit points of the program as additional slicing criteria requires the slice to terminate normally. In the absence of such a criteria a slice can compute the desired value and then diverge. The result is typically a smaller slice at the expense of not preserving the original program's termination behavior.

Threats to Validity. Threats to the external validity of this work comes from the dataset considered and the static slicer used for our experiments. We focused on a set of 62 programs written in the C programming language. The behavior of QSES is influenced by the underlying static slicer, CodeSurfer in our case. Should QSES be applied to programs written in a different language, the results will again depend on the static slicer used to identify the lines to protect. Moreover, as we have discussed, ORBS is platform dependent, and one might obtain different results by reproducing these experiments on a different platform. However, our manual investigation revealed that only a handful of slices are subject to such platform-specific behavior.

A threat to internal validity stems from the choice of slicing criterion for programs in the dataset. Specifically, we considered only scalar variables as the slicing criterion. We leave the extension of this experiment to non-scalar variables such as pointers to future work.

5. Related Work

5.1. Observation-Based Slicing

In addition to the work on ORBS [25], discussed in Section 2, Binkley et al. [29] 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 smaller than the corresponding ORBS slice. Similar to ORBS, QSES is able to discover such hidden dependences as illustrated in Section 4.

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 the true 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.

5.2. Static Slicing

The distinction between static and dynamic slicing was introduced by Korel and Laski [40, 23]. Static slicing approaches have been applied to numerous types of programs, such as non-deterministic programs [41], functional programs [42], WebAssembly binaries [43], or Python objects [44]. Static slicers generally rely on a static dependence analysis, which requires modeling the language being analyzed. QSES works from the results of a static slicer, but remains language agnostic. The static slicer that we use in our evaluation is CodeSurfer [26].

5.3. Dynamic Slicing

ORBS has been compared to several dynamic slicing techniques [25], as it is most closely related to dynamic slicing. Many dynamic slicing approaches exist [45, 46, 47, 48], but all require complex program analyses and target a single specific programming language.

A closely related work to observation-based slicing is *critical slicing* [49] where a statement is considered to be critical if its deletion results in a changed observed behaviour for the slicing criterion. One limitation of this approach is that it considers statements to be critical although they may not be, and thus could be deleted after another statement is deleted. For example, a critical variable declaration is no longer "critical" after all of the variable's references have been removed. Critical slices can be significantly larger than ORBS slices, but more importantly, can be incorrect [25].

5.4. 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 [50]. Our approach instead uses the static slice ahead of dynamic slicing.

Venkatesh [51] formalized and categorized various notions of program slicing. They mention *quasi-static slices*, which despite having a name similar to QSES, have a different goal. Venkatesh's quasi-static slices are static slices computed with some inputs having statically known values. Our quasi-static executable slices are computed starting from a static slice, itself computed for all possible input values.

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 [52]. Finally, rather than actually combining static and dynamic slicing, Ashida et al. [53] 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.

6. Future Work

One interesting direction for future work is applying QSES to other languages. One challenge here is that production quality static slicers are rare because they require significant static analysis. For the most part our results should hold for features of other languages that are similar to `C`. More challenging features include Java's reflection, languages such as Scheme that allow the programmer to define their own syntax, and any sufficiently dynamic language for which performing static analysis becomes a real challenge.

What is interesting about QSES is that it will work with any static slicer that *under-approximates* the statements of the slice. For example, a static slicer for Java that ignores reflection will yield a subset of the true slice. However QSES' use of ORBS means that it will include the omitted components. The same is true for dependences *transmitted* through embedded SQL, which are hard on static analysis, but will be correctly handled by QSES. This opens up the possibility of producing quasi-static slicers for a multitude of languages by computing the "easy" subset of the semantics and then letting QSES fill in the hard parts.

We also plan to investigate some of the ideas that arose in the discussion at the end of Section 4.2 such as the impact of different test suites on a QSES slice. While a challenge to answer, a key research question here is "how much execution is required?". Another interesting aspect to explore is the granularity of the slicers used. For example, we might force CodeSurfer to slice at the line (or at least statement) level. Alternatively, we might experiment with finer-grained observation-based slicing approaches such as T-ORBS [54], which can operate at granularities smaller than ORBS' line level. Other interesting possible future investigations include the possibility of a forward slicing variant of QSES and the consideration of slicing criteria beyond scalar assignments.

To provide some initial indication, we took a preliminary look at the research question "What is the impact of the test suite on QSES slices?". The study of QSES$_C$ tells us something about the impact of the test suite on a QSES slice because QSES$_C$ is equivalent to QSES using zero test cases and thus never executing the slice. At the other end of the spectrum is the likely infinite test suite that includes all possible inputs. While intractable, in theory the ORBS slice produced using all possible inputs is, by definition, the precise *static* slice. Its comparison to the corresponding QSES slice, for example, precisely captures the static over approximation.

Of course, *all possible inputs* is not feasible in practice. However, for smaller programs it is possible to hand craft a test suite that has the same effect as all possible inputs. Specifically we identify a finite test suite such that there does not exist any test whose addition would change any of the program's slices. Using first the word count program, wc, and then the triangle program, which is well studied in the testing literature, we consider the exhaustive impact of all possible test suite subsets.

For wc.c an input can either include or not include characters, lines, and words leading to eight possible inputs. However a test can't have words without having characters, nor can it have lines without having characters, thus only five of the eight are possible:

Test	Chars	Lines	Words	Test	Description
t_1	N	N	N	`""`	empty file
t_2	Y	N	N	`"   "`	three spaces
t_3	Y	Y	N	`" \n \n \n"`	three lines with one space each
t_4	Y	N	Y	`"aa bb"`	no newline
t_5	Y	Y	Y	`"aa \nbb\n"`	all three

There are 17 slices of the word count program. Using the 32 possible subsets of the five inputs yields 544 slices. Over all 32 inputs each of the 17 slices is unchanged, thus illustrating that each of the 17 static slices is precise. For comparison running the same experiment with ORBS shows expected differences. For example, using the empty test suite ORBS deletes all but main() { } while for t_1 it deletes the entire "for each character" loop which for an empty file never executes.

Turning to the triangle program, seven tests are sufficient to cover all possibles inputs, one covers the program being given something other than the expected three inputs, a second covers invalid (e.g., non-positive) values, and the remaining five each test one of the five types of triangles the program can identify. With seven tests there are 128 possible test suites. The program also has seven slices, so the experiment involved producing 896 total slices. Comparing these slices, the tests do have an impact, but only a relatively minor one. Without the declaration double atof(char *) only the invalid number of inputs and the invalid triangle tests pass. This is because without a prototype, C assumes that a function returns an int, all of which end up being zero. Thus test suites limited to subsets of these tests enable the deletion of the declaration.

In summary, similar to the 1 479 static slices that required no additions by QSES, wc's static slices include all the required statements and thus varying the test suite has no impact on the slice. In contrast the triangle program's slices did differ, but only by a single declaration statement. While it will require a larger study to understand exactly how sensitive QSES' slices are to the test suite, these two preliminary experiments suggest that QSES is quite robust to test suite differences.

7. Conclusion

This paper introduces and studies *Quasi-Static Executable Slices*, QSES. 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 in CodeSurfer's SDGs 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 with CodeSurfer and the choice of the deletion window size used by ORBS).

Acknowledgements

This work was partially supported by the *"Cybersecurity Initiative Flanders"*.

References

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

[2] M. Kamkar, N. Shahmehri, P. Fritzson, Bug localization by algorithmic debugging and program slicing, in: 2nd International Workshop Programming Language Implementation and Logic Programming, PLILP'90, Vol. 456, 1990, pp. 60–74.

[3] S. Kusumoto, A. Nishimatsu, K. Nishie, K. Inoue, Experimental evaluation of program slicing for fault localization, Empirical Software Engineering 7 (2002).

[4] D. W. Binkley, L. R. Raszewski, C. Smith, M. Harman, An empirical study of amorphous slicing as a program comprehension support tool, in: 8th International Workshop on Program Comprehension (IWPC 2000), 2000, pp. 161–170.

[5] E. Hosnieh, H. Haga, A novel approach to program comprehension process using slicing techniques, J. Comput. 11 (5) (2016) 353–364.

[6] K. B. Gallagher, J. R. Lyle, Using program slicing in software maintenance, IEEE Trans. Software Eng. 17 (8) (1991) 751–761.

[7] Á. Hajnal, I. Forgács, A demand-driven approach to slicing legacy COBOL systems, Journal of Software: Evolution and Process 24 (1) (2011).

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

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

[10] M. Harman, S. Danicic, Using program slicing to simplify testing, Softw. Test. Verification Reliab. 5 (3) (1995) 143–162.

[11] D. W. Binkley, The application of program slicing to regression testing, Inf. Softw. Technol. 40 (11-12) (1998) 583–594.

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

[13] J. Beck, D. Eichmann, Program and interface slicing for reverse engineering, in: 15th International Conference on Software Engineering, 1993, pp. 509–518.

[14] T. Akgul, V. J. M. III, S. Pande, A fast assembly level reverse execution method via dynamic slicing, in: 26th International Conference on Software Engineering (ICSE 2004), 2004, pp. 522–531.

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

[16] B. Korel, 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 29 (6) (2003).

[18] L. Philips, C. De Roover, T. Van Cutsem, W. De Meuter, Towards tierless web development without tierless languages, in: ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SPLASH/OnWard!14), 2014.

[19] L. Philips, J. De Koster, W. De Meuter, C. De Roover, Search-based tier assignment for optimising offline availability in multi-tier web applications, The Art, Science, and Engineering of Programming 2 (2) (2018).

[20] S. Salimi, M. Ebrahimzadeh, 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.

[21] K. J. Ottenstein, 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.

[22] D. Binkley, K. B. Gallagher, Program slicing, in: M. Zelkowitz (Ed.), Advances in Computing, Volume 43, Academic Press, 1996, pp. 1–50.

[23] B. Korel, J. W. Laski, Dynamic slicing of computer programs, J. Syst. Softw. 13 (3) (1990) 187–195.

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

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

[26] T. Teitelbaum, Codesurfer, ACM SIGSOFT Softw. Eng. Notes 25 (1) (2000) 99.

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

[28] Q. Stiévenart, D. W. Binkley, C. D. Roover, QSES: quasi-static executable slices, in: 21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, September 27-28, 2021, IEEE, 2021, pp. 209–213. `doi:10.1109/SCAM52516.2021.00033`.
URL `https://doi.org/10.1109/SCAM52516.2021.00033`

[29] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, 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.

[30] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Transactions on Software Engineering 28 (2) (2002) 183–200.

[31] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 26–61.

[32] 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.

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

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

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

[36] D. Binkley, N. Gold, M. Harman, An empirical study of static program slice size, ACM Transactions on Software Engineering and Methodology 16 (2) (2007) 1–32.

[37] R. Barraclough, D. Binkley, S. Danicic, M. Harman, R. Hierons, çkos Kiss, M. Laurence, A trajectory-based strict semantics for program slicing, Theoretical Computer Science 411 (11–13) (2010) 1372–1386.

[38] M. Harman, C. Fox, R. M. Hierons, D. Binkley, S. Danicic, Program simplification as a means of approximating undecidable propositions, in: 7^{th} IEEE International Workshop on Program Comprenhesion (IWPC'99), IEEE Computer Society Press, Los Alamitos, California, USA, 1999, pp. 208–217.

[39] S. Danicic, M. Harman, J. Howroyd, L. Ouarbya, A lazy semantics for program slicing, in: $1^{st.}$ International Workshop on Programming Language Interference and Dependence, Verona, Italy, 2004.
URL `http://profs.sci.univr.it/~mastroen/noninterference.html`

[40] B. Korel, J. Laski, Dynamic program slicing, Information Processing Letters 29 (3) (1988) 155–163.

[41] S. Danicic, M. R. Laurence, Static backward slicing of non-deterministic programs and systems, ACM Trans. Program. Lang. Syst. 40 (3) (2018) 11:1–11:46. `doi:10.1145/2886098`.
URL `https://doi.org/10.1145/2886098`

[42] P. K. K., A. Sanyal, A. Karkare, S. Padhi, A static slicing method for functional programs and its incremental version, in: J. N. Amaral, M. Kulkarni (Eds.), Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019, ACM, 2019, pp. 53–64. `doi:10.1145/3302516.3307345`.
URL `https://doi.org/10.1145/3302516.3307345`

[43] Q. Stiévenart, D. W. Binkley, C. D. Roover, Static stack-preserving intra-procedural slicing of webassembly binaries, in: 44th IEEE/ACM

44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, ACM, 2022, pp. 2031–2042. `doi:10.1145/3510003.3510070`.
URL `https://doi.org/10.1145/3510003.3510070`

[44] Z. Xu, J. Qian, L. Chen, Z. Chen, B. Xu, Static slicing for python first-class objects, in: 2013 13th International Conference on Quality Software, Najing, China, July 29-30, 2013, IEEE, 2013, pp. 117–124. `doi:10.1109/QSIC.2013.50`.
URL `https://doi.org/10.1109/QSIC.2013.50`

[45] A. Beszedes, T. Gergely, Z. M. Szabó, J. Csirik, T. Gyimothy, Dynamic slicing method for maintenance of large c programs, in: Proceedings of the 5th European Conference on Software Maintenance and Reengineering, 2001, pp. 105–113.

[46] A. Beszedes, T. Gergely, T. Gyimóthy, Graph-less dynamic dependence-based dynamic slicing algorithms, in: Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation, IEEE, 2006, pp. 21–30.

[47] X. Zhang, N. Gupta, R. Gupta, A study of effectiveness of dynamic slicing in locating real faults, Empirical Software Engineering 12 (2) (Apr. 2007).

[48] S. S. Barpanda, D. P. Mohapatra, Dynamic slicing of distributed object-oriented programs, IET software 5 (5) (2011) 425–433.

[49] R. A. DeMillo, H. Pan, E. H. Spafford, Critical slicing for software fault localization, in: International Symposium on Software Testing and Analysis, ACM, 1996, pp. 121–134.

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

[51] G. A. Venkatesh, The semantic approach to program slicing, in: D. S. Wise (Ed.), Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991, ACM, 1991, pp. 107–119. `doi:`

10.1145/113445.113455.
URL https://doi.org/10.1145/113445.113455

[52] R. Gupta, M. L. Soffa, J. Howard, Hybrid slicing: Integrating dynamic information with static analysis, ACM Trans. Softw. Eng. Methodol. 6 (4) (1997) 370–397.

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

[54] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, S. Yoo, Tree-oriented vs. line-oriented observation-based slicing, in: 2017 IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2017.