An Empirical Evaluation of Static, Dynamic, and Hybrid Slicing of WebAssembly Binaries

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

The WebAssembly standard aims to form a portable compilation target, enabling the cross-platform distribution of programs written in a variety of languages. This paper introduces and evaluates novel slicing approaches for WebAssembly, including dynamic and hybrid approaches. Given a program and a location in that program, a program slice is a reduced program that preserves the behavior at the given location. A *static* slice does so for all possible inputs, while a *dynamic* slice does so for a fixed set of inputs. *Hybrid* slicing is a combination of static and dynamic slicing.

We build on Observational-Based Slicing (ORBS), where we explore the design space for instantiating ORBS for WebAssembly. For example, ORBS can be applied to the whole program or to only the function containing the slicing criterion, and it can be applied before compilation to WebAssembly or afterward. We evaluate the slices produced using various options quantitatively and qualitatively. Our evaluation reveals that dynamic slicing at the level of a function from a WebAssembly binary finds a sweet spot in terms of slice time and slice size, and that a combination of static and dynamic slicers achieves the best trade-off in terms of slicing time and slice size.

1. Introduction

WebAssembly is a recent binary format [1] with many diverse use cases [2] both on the Web and beyond, including desktop applications [3] and smart contracts [4]. The academic literature has focused on the security aspects of WebAssembly [5, 6, 7, 8, 9, 10, 11, 12] and on tools and techniques for

analyzing WebAssembly binaries [13, 14, 15, 16, 11, 17, 18]. The rise in its use brings a growing need for tools to support the development and maintenance of WebAssembly code.

Program slicing [19, 20] provides a basis for such tools. Given a program point called the *slicing criterion*, program slicing identifies a reduced program that captures the computation at the slicing criterion. Program slicing has numerous applications such as debugging [21, 22, 23], program comprehension [24, 25, 26, 27, 28], software maintenance [29, 30], re-engineering [31], refactoring [32], testing [33, 34, 35], reverse engineering [36, 37], tierless or multi-tier programming [38, 39], and vulnerability detection [40].

Program slicers exist along multiple dimensions [41] including static vs. dynamic, executable vs. closure, and backward vs. forward. Prior to our work, the only existing slicer for WebAssembly was a static backward closure slicer, which we refer to as $\mathcal{CSE}$ [42].

This paper extends our initial investigation [43] of the first dynamic slicers for WebAssembly. Such slicers might be applied, for example, to a bug report consisting of hundreds of WebAssembly instructions that cause the web browser's WebAssembly virtual machine to crash. Alternatively, reducing this code to tens of instructions is a clear win. Another example application is escape analysis aimed at security flaws: a web browser might run this analysis in the background against each WebAssembly binary.

The design space for dynamic slicers that can slice WebAssembly is vast and until our work has gone unexplored. We consider options involving three key slicing phases: compiling the program to WebAssembly ($\mathcal{C}$), extracting the function containing the slicing criterion ($\mathcal{E}$), and slicing out (removing) irrelevant code ($\mathcal{S}$). For example, given a program P, one possible arrangement is first apply $\mathcal{S}$ (slice P at the source level), then $\mathcal{C}$ (compile the reduced program to WebAssembly), and finally $\mathcal{E}$ (extract the function that contains the slicing criterion). We denote this arrangement $\mathcal{SCE}$ for short.

As a second example, $\mathcal{CES}$ first compiles the source to WebAssembly, then extracts the function containing the slicing criterion, and finally slices out irrelevant code from this function. Comparing these two, we expect slicing after compilation to be more *surgical*, for example, able to remove boilerplate code that compilers include at the start and end of each function. Compared with the option $\mathcal{CSE}$, we expect slicing before extraction to be slower because more code is considered but more precise because it can remove code from called and calling functions.

Building on our preliminary investigation [43], which investigated three

dynamic slicers, this paper exhaustively explores the design space of applying the dynamic slicer ORBS [44] to WebAssembly, and provides insights into the advantages and limitations of four dynamic WebAssembly slicers as well as one hybrid slicer, by empirically comparing them with each other and with $\mathcal{CS}\mathcal{E}$ using 57 C programs and a total of 1643 slicing criteria. The paper extends our preliminary investigation [43] in the following ways:

- We extend our investigation of the design space by exploring *all* possible combinations of *slice*, *extract*, and *compile*. This suggests four promising dynamic slicers for WebAssembly. Our earlier work investigated only three of these. We replicate that work here using a slicing configuration that is more appropriate to the slicing of WebAssembly (see RQ5).

- Our earlier work identified that static slicing is far faster but the slices reflect significant static over-approximation. In contrast, dynamic slicing provides smaller slices but takes considerably longer. This led us to consider a novel hybrid slicer that first applies the static slicer to quickly remove much of the unnecessary code and then applies the dynamic slicer.

- We quantitatively and qualitatively compare the slices produced by the four dynamic slicers and by one hybrid slicer uncovering several interesting patterns and suggesting directions for future improvement.

- We compare the dynamic and hybrid approaches to the static WebAssembly slicer $\mathcal{CS}\mathcal{E}$ to both study the relative cost of each slicing approach and because we hope that the dynamic slicers will suggest improvements to the static slicer.

- We investigate the ideal window size for slicing WebAssembly code. In our earlier work we relied upon prior work with high-level languages [45, 44] and used a deletion window size of four. Qualitative analysis of the resulting slices clearly indicates that there were specific WebAssembly patterns that would benefit from a larger deletion window size. We therefore hypothesize and demonstrate that a larger window size is more appropriate when slicing at the lower-level granularity of WebAssembly code.

- We expand our evaluation to include a larger application, namely SQLite. This program has 169 448 SLoC (non-blank, non-comment lines of code), which when compiled to WebAssembly produces 287 346 instructions. We slice this program according to 6 criteria, and detail the results of our experiment in RQ8.

Our dataset and empirical evaluation scripts are available in a replication package[1].

2. Background

This section provides background on WebAssembly and $\mathcal{CSE}$, the only known static slicer for WebAssembly. We also briefly describe ORBS, a language-agnostic dynamic slicing approach that we instantiate to produce our four dynamic and one hybrid slicing algorithms for WebAssembly.

2.1. A Brief Tour of WebAssembly

This tour of WebAssembly is adapted from our previous work with $\mathcal{CSE}$ [42]. WebAssembly programs are bundled as *modules*, which contain one or more function declarations, along with other elements that are less relevant for the present paper (type declarations, data segments, tables, ...) WebAssembly modules manipulate primitive types that can be 32-bit or 64-bit integers and floating point numbers (`i32`, `i64`, `f32`, `f64`). Functions are typed: they take zero, one, or more parameters of one of the primitive types, and return zero, one, or more values also of the primitive types. Functions declare the types of their local variables. Parameters and local variables can be accessed through an index. For example, a function with one formal parameter and two local variables accesses the formal parameter at index 0 and the local variables at indices 1 and 2. The remainder of a function definition is the sequence of instructions that form the function's body.

Broadly speaking, there are two kinds of instructions. *Control instructions* (e.g., `block`, `loop`, `if`, and `call`) structure the program's control flow, while *data instructions* manipulate the stack (`drop`, `i32.const`), locals (`local.get` and `local.set`), and globals (`global.get` and `global.set`). Blocks act as delimiters inside functions for identifying jump targets. Loops are blocks whose semantics capture the iterative behavior.

[1] `https://zenodo.org/records/14851417`

```
1  int main() {
2    int i = 0;
3    int x = 0;
4    int c = 0;
5    while (p(i)) {
6      if (q(c)) {
7        x = f();
8        c = g();
9      }
10     i = h(i);
11   }
12 }
```

Figure 1: The SCAM Mug program in C. All called functions are side-effects free.

2.1.1. The SCAM Mug in WebAssembly

To illustrate programming in WebAssembly, we consider the "SCAM Mug" [46] C program, which is heavily used in the slicing literature. The program, which featured on the souvenir mug given to attendees at the first SCAM workshop, shown in Figure 1, is designed to challenge static analysis tools, especially those making use of transitive dependence analysis. For example, the minimal slice at the end of the code taken with respect to the variable x does not include Line 8 despite the transitive dependence.

The equivalent function in WebAssembly is given in Figure 2, where, for convenience, we annotate function calls with the type of the target function.

On Line 2, the function declares the equivalent of local variables i (with index 0), x (index 1), and c (index 2). All local variables are initialized to zero in WebAssembly. These first two lines and those containing `end` are non-executable lines. All remaining lines represent WebAssembly instructions. Line 3 retrieves and pushes the value of the first local variable on the stack. The next line calls function p, which expects its single argument to be on the top of the stack (in this case local 0). The `if`-instruction on Line 5 checks whether the top of the stack (the function's return value) is true (differs from 0) and if so executes its then branch, which captures the body of the loop. An optional else branch is unnecessary here. This instruction should not be confused with `br_if n`, which breaks n nested blocks if the value on the top of the stack is true. The `loop` instruction on Line 6 denotes the start of a loop. When execution encounters a *break*, it re-executes the loop from the start. In WebAssembly, the "breaking" of a loop behaves like a continue

```
(func (type 0)            ;; int main()
    (local i32 i32 i32)   ;; declare i, x, c
    local.get 0           ;; push local i
    call_{i32→i32} $p     ;; p(i)
    if
      loop
        local.get 2       ;; push local c
        call_{i32→i32} $q ;; q(c)
        if
          call_{→i32} $f  ;; f()
          local.set 1     ;; x = result of f()
          call_{→i32} $g  ;; g()
          local.set 2     ;; c = result of g()
        end
        local.get 0
        call_{i32→i32} $h ;; h(i)
        local.set 0       ;; i = result of h(i)
        local.get 0
        call_{i32→i32} $p ;; p(i)
        br_if 0           ;; loop if stack top
      end                 ;; is true
    end)
```

Figure 2: The SCAM Mug in WebAssembly.

statement in C. In the example, `br_if 0` starts the next iteration if the value on the top of the stack is true. The "0" signifies which loop, in this case, the immediately enclosing loop (Line 6). If no breaks are encountered, execution continues with the instruction that follows the `loop`'s matching `end` keyword. Thus Lines 5, 6, and 20 combine to implement the `while` loop of the C program.

The body of the loop first calls the function q with local variable 2 (c) on Line 8. If the result of this call is non-zero, it calls function f and assigns the result to local variable 1 (x) on Line 11, and does the same with function g and local variable 2 (c). Finally, near the end of the loop body on Line 17, local variable 0 (i) is assigned the result of calling h(i). The loop ends with the `br_if` instruction on Line 20 which checks the loop condition (the value on the top of the stack) and jumps back to the beginning of the loop if the value is non-zero.

2.1.2. WebAssembly Validation Requirement

WebAssembly programs have to be *well formed* according to Section 3 of the WebAssembly standard [47]. This includes a stack validation requirement that is checked by the host environment when compiling and before executing a program. In brief, each instruction has a specific *stack type* $t_1^* \rightarrow t_2^*$, where t_1^* is the expected sequence of types for the values on top of the stack before the execution of the instruction and t_2^* is the sequence of types for the values on top of the stack after its execution. For example, the `i32.const 0` instruction has type " $\rightarrow$ i32", meaning that it does not need anything from the stack and pushes one value of type i32. Typing extends to sequences of instructions. For example, the sequence `local.get 0; local.get 1; i32.const 1; i32.add` has type " $\rightarrow$ i32 i32".

Validation poses a challenge to program slicing in that the body of a function has to be well formed [42]. Thus only deletions that leave the code well typed are permitted, which can prevent the removal of otherwise superfluous code. For example, in the following WebAssembly fragment, both branches of the `if` push one value on the stack and therefore have the same type " $\rightarrow$ i32". Removing the code of the `else` branch (Line 5) because it never gets executed (as the condition of the `if` is always true) would result in a WebAssembly program that *fails* the validation requirement.

```
1       i32.const 1
2       if
3         local.get 0
4       else
5         local.get 1
6       end
```

2.2. Static Slicing of WebAssembly

The static WebAssembly slicer $\mathcal{CSE}$ [42] has three phases:

1. First, a data-gathering phase computes the dependences of each instruction in a function. It computes the layout of the stack after each instruction using a stack specification analysis [14], identifies data dependences through use-definition chains, identifies control dependences using Ferrante's exact algorithm [48], and performs a WebAssembly-specific over-approximation for memory-specific data dependences [42].
2. Second, the slicing phase identifies the WebAssembly instructions of the closure slice relying on the dependences identified in the first phase, taking inspiration from traditional approaches to slicing [21], and adding instructions for structured control flow [49].
3. Third, a reconstruction phase includes additional instructions to ensure that the slice satisfies the validation requirement and is thus a valid WebAssembly program.

Given a module and an instruction as the *slicing criterion*, $\mathcal{CSE}$ produces a reduced module where the function containing the slicing criterion has been replaced by a smaller function that preserves the semantics of the slicing criterion.

2.3. Observation-Based Slicing

Observation-Based Slicing (ORBS) [44] is a language-agnostic slicing approach that can in theory be applied to WebAssembly. As originally introduced, it takes as input a source program P to slice, a slicing criterion identified by a program variable ν, a program location l, a set of inputs $\mathcal{I}$, and a maximum deletion window size δ. The slice computed by ORBS compiles and preserves the semantics of ν at l for the set of inputs $\mathcal{I}$. ORBS is language agnostic so it considers a program a sequence of lines of text.

Freed from the need to perform complex whole-program dependence analysis, Yoo et al. observed that ORBS need only focus on a subset of a program;

thus they extended ORBS' slicing criteria to include *components of interest*, *CoI* [50]. In brief, the slicer's deletion attempts are restricted to the *CoI*. This enables, for example, slicing programs that contain binary components such as third-party libraries, which can be excluded from the *CoI* and thus need not be changed by the slicer. With this addition, an ORBS slice is taken with respect to the criteria $(v, l, \mathcal{I}, CoI)$ and preserves the state trajectory for v at l for the selected inputs in $\mathcal{I}$, while deleting only code from the components of *CoI* but no other components.

Operationally, ORBS first instruments the program by inserting a side-effect free line that tracks the value of variable ν immediately before line l. This insertion enables the algorithm to detect changes to the value of the variable at the slicing location. The instrumented program is then run on each input in $\mathcal{I}$ and the tracked values are used as an oracle for the expected output.

The rest of the algorithm iterates over the *CoI* tentatively removing lines until no more lines can be deleted. Each iteration over the *CoI* attempts to remove up to δ consecutive lines starting with the current line. After each removal, the program is compiled, and if it compiles, it is executed and the output is compared with the oracle. If this output matches the oracle then the current removal is made permanent. When a fixed point is reached the result is the dynamic observation-based slice. This paper considers two *CoI*'s, the file that contains the slicing criterion and the function that contains the slicing criterion.

Finally, by its very nature, Observation-Based Slicing is not quick. Applying it to a low-level assembly representation might prove prohibitively expensive. Thus, we consider practicality at two levels: first, is slicing fast enough for real-time use within an IDE and second, is it sufficiently fast for infrequent use, which we characterize as the time it takes to get a cup of coffee. One use case for this second level is the example from the introduction, involving the simplification of a bug report that includes hundreds of instructions. Reducing this code to tens of instructions in the time it takes to get a cup of coffee is sufficient.

3. Study Design and Methodology

This section describes our study's design and methodology, starting with the slicers studied and then the research questions considered. The section then describes the subject programs studied, their preparation, and the two

metrics used to evaluate slicer performance. Finally, it provides some implementation details.

3.1. The Slicers Studied

The experiments consider several dynamic slicers, which are compared against each other and the only previously existing static slicer, $\mathcal{C}s\mathcal{E}$ [42]. In addition, we consider a static/dynamic hybrid suggested by the initial experiments. The design space for instantiating a dynamic observation-based slicer for WebAssembly is formed from combinations of three key tasks: *slicing* the code ($\mathcal{S}$), *compiling* the code to WebAssembly ($\mathcal{C}$), and *extracting* the function containing the slicing criterion ($\mathcal{E}$). The extracted function becomes the slice's *CoI* (*Component of Interest*). The order in which these tasks are performed, including replications, forms the design space. For example, $\mathcal{CES}$ first compiles the code to WebAssembly, then extracts the function containing the slicing criterion, and finally applies ORBS using the extracted function as the *CoI*.

To explore the design space we first consider the slicers obtained using a single application of each of $\mathcal{S}$, $\mathcal{E}$, and $\mathcal{C}$. We then consider all possible sequences, including repetitions. The six possible single-application orders, produce three viable slicers (primarily because the order of compile and extract does not influence the resulting slice):

- $\mathcal{SCE}$– slice the `C` code using the `C` file as the *CoI*, compile the sliced `C` code using clang, then extract the function containing the slicing criterion.
- $\mathcal{CSE}$– run clang, slice the entire resulting assembly file as the *CoI*, then extract the function that contains the slicing criterion.
- $\mathcal{CES}$– run clang, extract the function that contains the slicing criterion, then slice only that function as the *CoI*.

An exhaustive exploration of the design space requires considering all possible combinations of Slicing, Extracting, and Compiling. Fortunately, many sequences are equivalent. For example, slicing twice in succession is the same as Slicing a single time. The same is true for Extracting. Likewise Extracting before Compiling brings no real value and finally, we need only Compile once in any given slicer.

To further simplify the design space, we rely on our findings comparing $\mathcal{CSE}$ and $\mathcal{CES}$ where it is preferable to slice after extraction [43]. This preference is motivated in part by the amount of code that must be considered when slicing the entire file and by cases in which a whole-file slice removes the boundary between two functions making the resulting slice hard to comprehend.

While a bit tedious, using these equivalences and the preference of $\mathcal{CES}$ over $\mathcal{CSE}$, the set of all possible combinations boils down to six: $\mathcal{SCE}$, $\mathcal{CSE}$, and $\mathcal{CES}$ from our original experiments [43], plus $\mathcal{SCES}$, $\mathcal{ESC}$, and $\mathcal{ESCS}$. It is encouraging that our initial three choices are all retained. As for the other three, $\mathcal{SCES}$ slices at both the source and WebAssembly levels of abstraction, $\mathcal{ESC}$ slices only the C *function* that contains the criteria (in contrast to $\mathcal{SCE}$, which slices the entire C file), while $\mathcal{ESCS}$ slices only the C function that contains the criteria but at both levels.

Because slicing only a C function is a bit far removed from our goal of studying the slicing of WebAssembly binaries, we do not consider $\mathcal{ESC}$ nor $\mathcal{ESCS}$. Slicing a C function rather than the entire C file, as done with $\mathcal{SCE}$, should be faster but the resulting slices should be largely unchanged. Thus in addition to the initial three, we consider here the only multi-application combination slicer studied.

- $\mathcal{SCES}$– slice the C code using the C file as the *CoI*, compile the sliced C code using clang, extract the function containing the slicing criterion, and then slice only that function as the second slice's *CoI*.

Finally, we formalize the static slicer and a novel hybrid slicer that was suggested during the initial experiments.

- $\mathcal{C}s\mathcal{E}$– run clang, then run the static WebAssembly slicer [42] on the resulting file as the *CoI*, and finally extract the function that contains the slicing criterion.
- $\mathcal{C}s\mathcal{ES}$– same as $\mathcal{C}s\mathcal{E}$ except that we apply ORBS to the static slice using only the function that contains the slicing criteria as the *CoI*.

It is worth noting that because they slice after the code has been compiled, the slicers $\mathcal{CES}$, $\mathcal{CSE}$, $\mathcal{C}s\mathcal{E}$, and $\mathcal{C}s\mathcal{ES}$ have the advantage that they do not require access to the source code being sliced and thus can slice deployed binaries.

All of the slicers considered in the paper are *inter-procedural slicers* in the sense that each takes into account the calling context of the code external to the function that contains the slicing criteria. In contrast, an *intra-procedural* slicer ignores the context of the code and only accounts for dependences found in the code of the function being sliced. When computing an inter-procedural slice it is possible to focus ORBS on a given *CoI* (e.g., a function, class, file, etc.), but the resulting slice takes into account all of the inter-procedural dependences.

3.2. Research Questions

We evaluate our slicers using the following eight research questions.

RQ1: How practical is applying ORBS directly to WebAssembly programs? ORBS involves repetitive execution of the program being sliced. Applying it to a low-level representation such as WebAssembly might prove prohibitively expensive. This research question investigates the time taken to slice the original `C` code ($\mathcal{SCE}$), the entire WebAssembly file ($\mathcal{CSE}$), and finally only the WebAssembly function that contains the slicing criterion ($\mathcal{CES}$).

RQ2: Which of the single-application dynamic slicers $\mathcal{SCE}$, $\mathcal{CSE}$, and $\mathcal{CES}$ best balances speed and precision? Because the source provides a higher-level representation of the code, we expect $\mathcal{SCE}$ to be the fastest of the three. However, it is unclear whether the precision will suffer because $\mathcal{SCE}$ is slicing at a higher level of abstraction. At the other end of the spectrum, $\mathcal{CSE}$ is expected to be the most precise, but also the slowest. The big question is whether $\mathcal{CES}$ represents a sweet spot.

RQ3: What qualitative differences exist between the slices produced by the three single-application dynamic slicers? One expected difference is that slicing the code after compilation will enable the slicer to remove boilerplate code that most compilers include at function entry and exit. Slicing at the source level cannot remove this code. At the other end of the spectrum when slicing the compiled code where the *CoI* is the entire file, slicing should be able to remove code that cannot be removed when the *CoI* is a single function because the code is required by an unnecessary computation found in another function.

RQ4: What are the pros and cons of static slicing versus dynamic slicing? We expect the static slicer to be notably faster but lacking in two ways. First, it is forced to make conservative data-flow assumptions and second, unlike ORBS, it does not guarantee that its slices are executable.

RQ5: What is the ideal window size, δ, when applying ORBS to WebAssembly? Earlier work found that $\delta = 4$ best balances slice size and slicing time when slicing C and Java code [44, 51, 52]. Prior work slicing WebAssembly [43] identified cases where additional deletion would be possible using a window size larger than 4. Thus, $\delta = 4$ may not be the best choice when slicing low-level code such as WebAssembly. This seems reasonable because assembly language includes less semantic information per line. Counterbalancing the benefits of a larger window size is the increased slicing time a larger window size brings.

RQ6: What is the impact of slicing at multiple levels of abstraction? In an attempt to capture the *best of both worlds*, our novel slicer, $\mathcal{SCES}$, applies ORBS to the C code and then to the WebAssembly code. We expect this to combine the benefits of $\mathcal{SCE}$ and $\mathcal{CES}$, resulting in smaller slices computed in less time.

RQ7: What is the impact of a static/dynamic hybrid slicer? This research question considers the impact of first applying the static slicer $\mathcal{CsE}$ to quickly remove statically irrelevant code and then apply the dynamic slicer $\mathcal{CES}$ to remove dynamically irrelevant code. Ideally, the result, $\mathcal{CsES}$, should produce the same slice as $\mathcal{CES}$ in a fraction of the time. Such static/dynamic hybrids have proven successful in the past at other levels of abstraction [45].

RQ8: How do the slicers behave on real-world applications? This research question evaluates whether our findings generalize beyond the benchmarks used in the other research questions. We apply the slicers to a real-world application of 170kSLoC based on SQLite, with six different slicing criteria within functions of varying sizes. We expect the outcome of this experiment to reinforce the conclusions from our first seven research questions, thus improving the external validity of our results. This research question also helps us evaluate the scalability of our approach.

3.3. Subjects

We consider 57 C programs as the subjects of our study. These subjects are listed in Table 1 along with their respective sizes. Code sizes are given in *source lines of code* (SLoC), which excludes blank and comment lines. The subjects considered include classical programs from the slicing literature [19, 46, 53, 29], programs from the Mälardalen WCET research group [54], programs from the *Benchmarks Game* [55], and the multi-file system bc. These subjects have all been used in previous slicing studies [56, 42].

Table 1: Subjects used in our evaluation. Source lines of code (SLoC), computed using sloc_count_c, include 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.

Program	C SLoC	WASM SLoC	Program	C SLoC	WASM SLoC
adpcm	585	10 275	lms	172	8 759
bc__bc	8007	40 499	ludcmp	109	9 022
binary-trees1	91	14 663	mandelbrot9	66	14 764
bs	46	8 130	matmult	54	8 328
bsort100	61	8 221	mbe	63	15 622
cnt	76	8 511	minver	201	9 074
compress	357	9 000	nbody1	92	14 756
cover	625	8 847	nbody2	107	14 819
crc	94	8 409	nbody3	90	14 757
duff	44	8 296	nbody6	93	14 754
edn	170	9 027	nbody7	137	15 005
expint	73	8 268	ndes	196	9 009
fac	23	8 199	ns	30	8 387
fankuchredux1	79	14 843	nsichneu	2989	17 277
fankuchredux5	115	15 047	prime	51	8 176
fasta1	126	21 028	qsort-exam	124	8 407
fasta2	264	15 236	qurt	120	8 285
fasta3	90	14 586	reverse-complement5	83	14 917
fasta5	109	14 797	reverse-complement6	96	14 770
fasta8	150	14 903	scam	63	15 633
fasta9	161	15 007	select	131	8 307
fdct	138	8 531	spectral-norm1	57	14 725
fft1	128	11 004	st	98	8 337
fibcall	27	8 135	statemate	1354	10 613
fir	54	8 277	sumprod	18	14 355
insertsort	33	8 141	ud	81	8 933
janne_complex	38	8 131	wc	49	20 615
jfdctint	119	8 506	fasta7	231	15 041
lcdnum	62	8 227			

We limit the complexity of the programs considered to facilitate comparison. This does not mean that our approach cannot handle larger codes, only that such complexity makes patterns harder to identify. As an example, we consider one multi-file program, bc. Including this program allows us to illustrate that larger programs do not have any material effect on the analysis.

3.4. Subject Preparation

ORBS captures the slicing criterion by annotating the program with a statement that prints the variable of interest. The static slicer also uses this print statement to identify the slicing criterion. For each program, we annotate the use of each scalar variable in the program. In addition, for the classical slicing examples, we consider slices with respect to pointers such as argv, which is possible because we instrument these programs by hand. This process yields 1646 slicing criteria. We subsequently removed three criteria whose slices exhibited non-deterministic behavior, leaving 1643 slicing criteria.

When considering RQ4 and RQ7, we further cull 98 slices whose static slice fails to preserve the behavior of the slicing criteria. The static slicer produces *closure slices*, which are not guaranteed to preserve the original execution behavior. When a static closure slice preserves all of the original program's behavior for the slicing criteria then it is also a static *executable* slice. We use a lower bar. In our experiments, it is sufficient for the static slice to preserve the behavior only on the input(s) used to produce the dynamic slice.

Finally, we note that 60 of the included criteria are impacted by the execution environment. For example, the criteria scam_argv_18 (the slice of the scam mug program taken with respect to `argv` at Line 18) outputs `scam.c.wasm` when run by the wasmer WebAssembly runtime, but `./scam` when run as a compiled binary. The other causes include, for example, different implementations of `read` by wasmer and WebAssembly's use of 32-bit long ints in contrast to the native machine's 64-bit long ints. The discussion section considers the impact of this final difference.

3.5. Metrics

We collect time and size metrics for each slice on a 676-core computing cluster using 2.20GHz Xeon(R) E5-2650 CPUs. The cluster has 256GB RAM

per node and runs Centos 7. We use clang version 13.0.0, wasmer version 4.2.5, wasm2wat version 1.0.34, and pORBS version 5.0.

Time. We measure the time taken to compute each slice using both the CPU (user) time, to reflect the computational effort required to compute the slice, and the wall clock ("real") time, which is sometimes significantly lower. Unless otherwise specified, when providing timing data, we are reporting the user time.

Size. When reporting sizes we report the number of non-comment, non-blank lines of code as reported by the tool sloc_count_c applied to the C code and to the WebAssembly code of the function containing the slicing criterion. For compiled WebAssembly code this is the same as non-blank lines because the code is devoid of comments, except that we omit the function's declaration, its declaration of local variables, and all `end` lines.

3.6. Implementation

This subsection provides implementation details regarding the configuration of the experiments. First, to create a WebAssembly module from C source code we use clang with the target wasm32-unknown-wasi and then convert the binary to its textual representation using wasm2wat[2]. We use the compiler options -O2 -lm -fno-inline-functions, the latter prevents the compiler from inlining the function that contains the slicing criterion. We then prepend the closing parenthesis of each function in WebAssembly by a newline in order to enable ORBS to remove the last instruction of a function without breaking the syntax of the WebAssembly code.

Second, to slice using a single function as the *CoI* we split the WebAssembly file into three parts: file.wat.prefix, file.wat.function, and file.wat.postfix. ORBS is then applied to the lines of file.wat.function *only*. Thus *extracting* the function containing the slicing criterion. To compile and execute the program, the three parts, including the reduced file.wat.function, are concatenated together.

One motivation here is that the WebAssembly compiler includes a lot of library code (e.g., code for all functions required from libc). Slicing this code impacts ORBS' running time, which is proportional to the number of

[2] `https://github.com/WebAssembly/wabt`

lines considered. As discussed with RQ1, slicing a single function reduces the amount of work by almost two orders of magnitude.

Most of the code we analyze is from compiler benchmarks that include a single prescribed input. The exception to this is the classic slicing examples, where we use sufficient inputs to cause the dynamic slice to be equivalent to the static slice, and `bc` where we use a random sample of sixteen tests from the `bc` test suite.

While we evaluate the ideal window size in RQ5, we use a window size of $\delta = 6$ in the other research questions.

Finally, we consider slices based on a single slicing criterion. While it would require some engineering work, creating a single slice based on multiple slicing criteria would not impact the slicing algorithms nor should it impact our insights.

4. Evaluation

This section empirically investigates each of our seven research questions. All source programs and the slices computed by each slicer, as well as the scripts used in this evaluation, are available in our replication package[3].

4.1. RQ1: How Practical is Applying ORBS Directly to WebAssembly Programs?

For RQ1 we consider the time taken by each slicer. Of particular interest here is the performance of applying ORBS to WebAssembly *functions*. Because the average function size is reasonably constant and much smaller than the average program size, slicing only the code of a specific function is hoped to prove practical. For the evaluation, we define our "time to get a cup of coffee" practical time as 1000 seconds or approximately 15 minutes.

Figure 3 summarizes the times taken by each slicer. Of the three, $\mathcal{CSE}$ is the slowest slicer, which is unsurprising given that it considers deletions over the *entire* WebAssembly file, which has a median SLoC of 9 496 instructions. With a minimum slicing time of 92 minutes, none of $\mathcal{CSE}$'s times fit within our definitions of "practical". Compare this with $\mathcal{SCE}$ where the median size of the `C` programs is 201 SLoC, requiring ORBS to consider approximately 50 times fewer deletions.

[3] `https://zenodo.org/records/14851417`

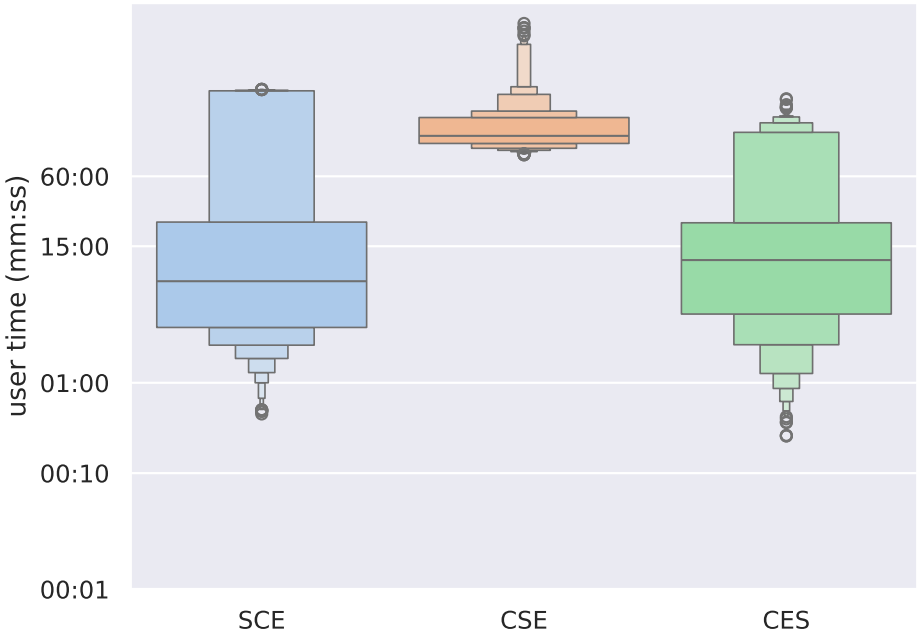

Figure 3: Slice times for each slicer. The graph uses a log scale on the y-axis.

$\mathcal{SCE}$ and $\mathcal{CES}$ are close in terms of time. $\mathcal{SCE}$'s slicing times range from 32 seconds to 337 minutes with a median of 7.49 minutes, while $\mathcal{CES}$'s slicing times range from 22 seconds to 279 minutes with a median of 11.40 minutes. That $\mathcal{SCE}$ and $\mathcal{CES}$'s times are so similar comes as a bit of a surprise given that $\mathcal{SCE}$ considers the entire `C` source file, while $\mathcal{CES}$ considers only the WebAssembly instructions of a single function.

Looking at our "cup of coffee" practicality threshold, $\mathcal{CSE}$ fails to produce a single slice within the allotted time. $\mathcal{SCE}$ and $\mathcal{CES}$ are more successful, producing respectively 1032 (63%) and 1024 (62%) of the 1643 slices considered. In terms of wall-clock time, which may be more relevant in terms of practicality, $\mathcal{CSE}$ still fails to produce a single slice within the allotted time, while $\mathcal{SCE}$ and $\mathcal{CES}$ respectively produce 1359 (83%) and 1228 (75%) slices within the time limit.

```
1  int g, *p;
2
3  foo()
4  {
5    bar();
6    *p = 42;
7  }
8
9  bar()
10 {
11   p = &g;
12 }
```

Figure 4: The slice of `bar` has to retain `p = &g` when slicing just `bar`. However, when slicing the entire program, if the slicer can remove `*p = 42` then it can subsequently safely remove `p = &g`.

> **RQ1**: Applying ORBS to an entire WebAssembly file ($\mathcal{CSE}$) is not practical. With a median time of 7.49 minutes, slicing an entire `C` program ($\mathcal{SCE}$) and, with a 11.40 minutes median focusing ORBS on the *function* containing the slicing criterion ($\mathcal{CES}$), both largely fit within our definition of practical.

4.2. RQ2: Which of $\mathcal{SCE}$, $\mathcal{CSE}$, and $\mathcal{CES}$ Best Balances Speed and Precision?

Building on RQ1's consideration of slicing times, RQ2 factors in slice size. We first consider the slices of $\mathcal{CSE}$ and $\mathcal{CES}$. By focusing ORBS deletion on the lines within a given function, $\mathcal{CES}$ is notably faster than $\mathcal{CSE}$. The question is, does faster come at the expense of slice size? For example, when the slicer has access to the entire file the removal of code outside a function might enable the removal of code within the function. Figure 4 illustrates this phenomenon at the source level.

Figure 5 depicts the distribution of slice sizes relative to the original size of the function being sliced. Table 2 provides some descriptive statistics for the slice sizes. The mid-point lines of Figure 5 show the median. That the median is less than the mean indicates the presence of a small number of very large slices and makes the median the better representative of the expected size.

The slice size distribution of $\mathcal{CSE}$ and $\mathcal{CES}$ are similar. Numerically, the median slice size for $\mathcal{CSE}$ is 13.18% of the original code, which is 50% higher

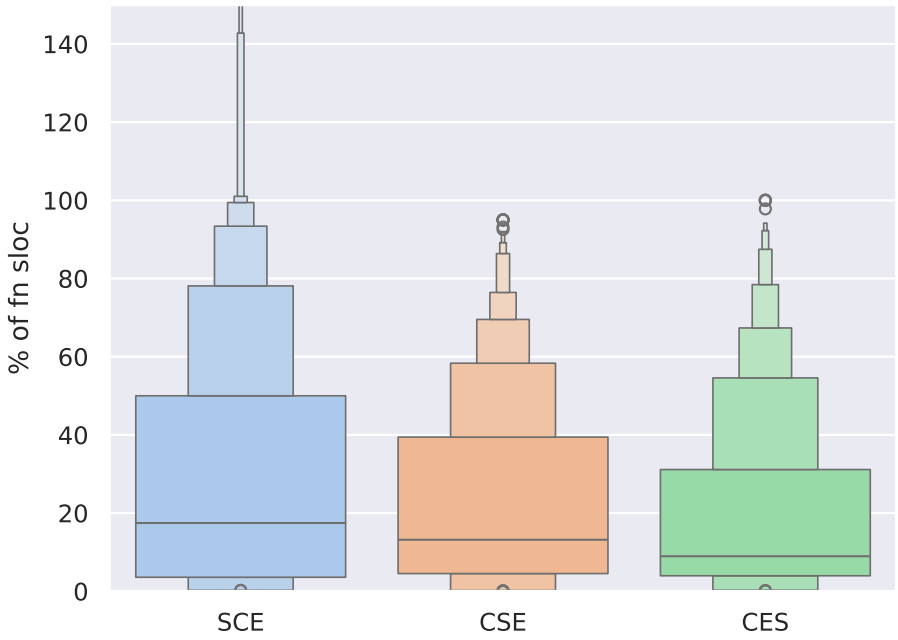

Figure 5: Size in SLoC of the function being sliced as a percentage of the original function size (cropped at 150%). Values higher than 100% indicate that the slice is larger than the original function. A typical cause for this is compiler loop unrolling.

than the $\mathcal{CES}$ median of 8.96%. We expected the two to be similar and furthermore that $\mathcal{CSE}$ would always produce the smaller slice. Surprisingly, this is often not the case: 709 (43.15%) of the $\mathcal{CSE}$ slices are actually larger than the corresponding $\mathcal{CES}$ slice. The cause of this unexpected result is investigated as part of RQ3's qualitative investigation.

We expect $\mathcal{SCE}$ to produce larger slices than $\mathcal{CSE}$ and $\mathcal{CES}$ when, for example, the latter two remove parts of the standard function entry and exit boilerplate code. Empirically, an $\mathcal{SCE}$ slice is larger than the corresponding $\mathcal{CSE}$ slice 83% of the time and larger than the corresponding $\mathcal{CES}$ slice 76% of the time. One of the more interesting causes is when the sliced `C` code is reduced to the point where the compiler opts to perform loop unrolling. Such optimizations trade compiled function size for performance and can result in larger slices. This also explains why the size of some $\mathcal{SCE}$ slices is more than

Slicer	Median	Mean	Min	Max	Std. Dev.
$\mathcal{SCE}$	17.46%	31.03%	0.21%	289.13%	± 34.76
$\mathcal{CSE}$	13.18%	23.37%	0.00%	95.00%	± 24.36
$\mathcal{CES}$	8.96%	20.35%	0.11%	100.00%	± 23.65

Table 2: Slice Size Statistics

100% of the sliced function. The most extreme case is adpcm_wd_402_expr, which is almost three times larger than the original function. We investigate this effect as part of RQ3's qualitative look at the slices.

RQ2: $\mathcal{SCE}$ produces notably larger slices than $\mathcal{CES}$ and $\mathcal{CSE}$. Combining this with the results from RQ1, we find that $\mathcal{CES}$ best balances speed and precision.

4.3. RQ3: What Qualitative Differences are There Between the Slices Produced by the Three Slicers?

RQ3 takes a qualitative look at the slices. To do so, we compute the size difference for each slicing criterion and then sort the differences. We then inspected random examples from the largest and the smallest differences, both positive and negative, and report interesting patterns. We first compare $\mathcal{CSE}$ and $\mathcal{CES}$ and then $\mathcal{SCE}$ and $\mathcal{CES}$. We omit the direct comparison of $\mathcal{CSE}$ and $\mathcal{SCE}$ because it is similar to that of $\mathcal{SCE}$ and $\mathcal{CES}$.

4.3.1. Comparison of $\mathcal{CSE}$ and $\mathcal{CES}$

$\mathcal{CSE}$ produces a smaller slice in 562 cases, the same size in 372 cases, and surprisingly $\mathcal{CES}$ produces the smaller slice in 709 cases. Interestingly the median difference is 0 lines, and 71% of the slices differ by five or fewer instructions. We consider four representative examples, two from each end of the spectrum.

The first example compares the slices taken with respect to sumprod_i_10 and illustrates the situation where $\mathcal{CSE}$ removing code from outside the function containing the slicing criterion enables it to remove code from within the function. In this example, the compiler reuses the stack location of main's first parameter, argc, to hold the value of index variable i. In a C program, argc is always at least one because the count of the arguments includes the name of the program. The $\mathcal{CES}$ slice does not slice outside of `main` and thus the counting code is retained and the location initially holds a non-zero

value. Therefore, in the compiled version of for (i = 0; ...) the initialization of i on Lines 81 and 82 of the following code must be retained:

```
37      (func $main (type 3) (param i32 i32) ....
...
81          i32.const 0
82          local.set 0           ;; i = 0
```

However, in the $\mathcal{CSE}$ slice the argument counting code gets sliced out leaving argc with the value zero. Thus in the $\mathcal{SCE}$ slice, there is no need to initialize variable i. Consequently, Lines 81 and 82 are omitted from the $\mathcal{SCE}$ slice.

Another interesting example is the slices taken with respect to fasta2_offset_49. The $\mathcal{CES}$ slice includes the following code while the $\mathcal{CSE}$ slice includes only the final instruction.

```
i32.const 61
i32.mul
global.set $__stack_pointer
```

Here the $\mathcal{CES}$ slice has co-opted the first two instructions from the size computation found in the following C code (for typesetting reasons the names have been shortened).

```
need = string_length * 60;
buffer = malloc(((need + (need / 60)) + 1);
```

The co-opting is safe because the required activation record size is less than the amount allocated. While the changes are intricate, in brief, the $\mathcal{CSE}$ slice omits from *each* function all of the standard entry code for creating a stack activation record. Doing so causes all functions to share a common activation record. For this particular slice, doing so is safe because each function call is the final instruction of its function body (a situation similar to tail recursion). In contrast, because a $\mathcal{CES}$ slice removes nothing from other functions, it must maintain the stack discipline and allocate stack space for its local variables.

Reflecting on these two representative examples, in the first retaining the initialization of i makes the slice larger but more straightforward to understand. In the second slice, the co-opted code is not the standard stack allocation code but its presence is better than the absence of any such code. Thus, in both cases $\mathcal{CES}$ produced the preferred slice.

Turning to criteria for which the $\mathcal{CES}$ slice is smaller, we first consider the slices taken with respect to adpcm_inc_170, where the $\mathcal{CSE}$ slice is three instructions longer than the $\mathcal{CES}$ slice. The three instructions initialize local

1 with a copy of the stack pointer.

```
global.get $__stack_pointer
local.tee 1
global.set $__stack_pointer
```

The reason that the $\mathcal{CSE}$ slice can't remove these instructions is twofold. First, the printing functions of the $\mathcal{CSE}$ slice have become specialized to print the particular location used in the code as the slicing criterion. Copying the stack pointer to local 1 ensures that these specialized functions continue to find the value where they expect it. Second, the call to printf is the last instruction in the function being sliced which precludes the need to maintain a separate activation record for this function.

Our fourth example, shown in Figure 6, considers the slice taken with respect to spectral-norm1_i_10. In the figure the comments show the three deletions used by the $\mathcal{CES}$ slicer to remove the ten instructions from within the loop: $\mathcal{CES}$ first removes lines annotated with 1, then lines annotated with 2 which have become contiguous after the first deletion, and then finally the lines annotated with 3. The code is the compiled version of the C statement `Au[i] += eval_A(i, j) * u[j]`, which $\mathcal{CES}$ correctly omits from the slice. In the code of Figure 6 the update of `Au[i]` is done through a pointer held in local variable `5`.

The removal is prevented in the $\mathcal{CSE}$ slice by changes in the calling functions that affect the allocation of the array `Au`. The net effect is that the address of `Au[i]` overlaps with the loop counter `i`. In the sliced code, the value of `u[j]` is always zero, and thus the additions to `Au[i]` do not change its value (nor the value of `i`). However, the removal of the three instructions labeled "*;; 1*" leads to the repeated incrementing of `i`, which causes the loop to terminate after a single iteration. Inserting the following instructions before the loop enables the deletion because it assigns local `5` an unused memory address.

```
i32.const 4242
local.set 5
```

Summarizing the four examples, from a qualitative perspective, $\mathcal{CES}$ seems the better approach. $\mathcal{CES}$ sometimes produces larger slices, but the difference is never large. On the plus side, its slices are often more easily understood. In contrast, the smaller $\mathcal{CSE}$ slices are often more fragile because of the specialization of other functions, most notably those involved in program IO.

```
loop  ;; label = @3
  local.get 5      ;;   3     p = &Au[i]
  local.get 6      ;;   3
  local.get 4      ;;  2
  local.get 8      ;;  2
  call $eval_A     ;;  2
  local.get 7      ;; 1
  f64.load         ;; 1
  f64.mul          ;; 1
  f64.add          ;;  2
  f64.store        ;;   3     *p = ...
```

Figure 6: Code in $\mathcal{CSE}$ slice but not that $\mathcal{CES}$ slice.

4.3.2. Comparison of $\mathcal{SCE}$ and $\mathcal{CES}$

We next compare the slices of $\mathcal{CES}$ and $\mathcal{SCE}$. Comparing sizes, the $\mathcal{CES}$ slice is smaller for 1249 slices, the same for only 22 slices, and larger for 372 slices. This is in line with the expectation that $\mathcal{SCE}$ produces the largest of the dynamic slices and agrees with the quantitative data of RQ2.

Looking at examples where the $\mathcal{SCE}$ slice is smaller, the largest difference is for the slices taken with respect to nbody7_x1_24 for which the $\mathcal{CES}$ slice includes 322 instructions while the $\mathcal{SCE}$ slice is only 238 instructions long. The main difference between $\mathcal{SCE}$ and $\mathcal{CES}$ is how the following loop gets compiled:

```
for (i = 0; i < 5; ++i)
{
    x[i] += DT * vx[i];
    y[i] += DT * vy[i];
    z[i] += DT * vz[i];
}
```

In the case of $\mathcal{CES}$, the compiler unrolls the loop and results in 15 multiplications spanning 134 instructions, while in the case of $\mathcal{SCE}$ the loop does not get unrolled, requiring only 26 instructions. Slicing at the C level changed the optimizing behavior of the compiler.

As an illuminating example, the $\mathcal{SCE}$ slice taken with respect to lms_x_160 includes 51 instructions, while the $\mathcal{CES}$ slice contains 93. The C code includes two loops that ORBS can *fuse* together because none of the intervening code is in the slice. In contrast, up against the window size of six and the stack validation requirements, $\mathcal{CES}$ is unable to merge the two

loops. Its efforts are in part thwarted by its inability to remove a call to a function with six arguments because the call requires six push instructions followed by the call, which pops the six elements from the stack. Satisfying the validation requirement necessitates removing the six pushes and the call in a single deletion, which is not possible using a window size of less than seven. Slicing at the C level this call is a single line and thus easily removed.

We next turn to the dominant case where $\mathcal{CES}$ produces the smaller slices. The most extreme case is for edn_j_140 where the $\mathcal{CES}$ slice includes 55 instructions while the $\mathcal{SCE}$ slice 585. This is an example where the compiler, faced with the simplified C code, opted to unroll a nested loop, producing 64 copies of the simplified loop body. This pattern dominates the larger differences.

As a second example, an interesting pattern occurs in the $\mathcal{SCE}$ slice taken with respect to adpcm_wd_402. In the C slice, ORBS merges the function containing the criterion with the function proceeding it in the source code. The merged function involves considerably more code than the original function. When the "function" containing the slicing criteria is naively extracted, the $\mathcal{SCE}$ slice contains instructions that are attributable to the statements of the other function, making it much larger than in the $\mathcal{CES}$ slice.

Unlike the comparison of $\mathcal{CSE}$ and $\mathcal{CES}$, the qualitative comparison of $\mathcal{SCE}$ and $\mathcal{CES}$ is less one sided. In $\mathcal{SCE}$'s favor, slicing at a higher level of abstraction enables $\mathcal{SCE}$ to remove code that is more difficult to remove at the finer granularity level of WebAssembly code. Furthermore, in some cases, slicing before compilation also enables additional compiler optimizations. However, some enabled optimizations are not beneficial, at least in terms of slice size. An example is the slice taken with respect to edn_j_140. Some of the $\mathcal{SCE}$ slices also included merged functions, which make the slice harder to understand, while some include merged loops, such as the slices taken with respect to lms_x_160. These challenges suggest hybrid techniques such as first slicing the C code and then the compiled WebAssembly code.

RQ3: Our qualitative examination of the slices reveals that, as with RQ2, $\mathcal{CES}$ is often the preferred approach. This preference is, in part, because slicing at a level of abstraction different from that of the final slice ($\mathcal{SCE}$) and slicing multiple functions ($\mathcal{CSE}$) leads to structural changes that make it hard to tie the slice back to the original source. Slicing a single function at the WebAssembly level ($\mathcal{CES}$) reduces the number of structural changes in the code making it easier to tie the slice back to the original source.

4.4. RQ4: For a Given Binary, what are the Pros and Cons of Static Slicing Versus Dynamic Slicing.

RQ4 compares the static slices of $\mathcal{C}s\mathcal{E}$ with those of $\mathcal{CSE}$ and $\mathcal{CES}$. The comparison is first done quantitatively using $\mathcal{CES}$, as RQ1 through RQ3 suggest that it has the most desired trade-off, and then qualitatively using both $\mathcal{CES}$ and $\mathcal{CSE}$. For this research question, we cull from our dataset of 1643 slices 98 where the static slice fails to capture the correct semantics.

Quantitative evaluation. The chart on the left of Figure 7 compares the slice sizes. By its very nature, $\mathcal{C}s\mathcal{E}$ produces larger slices because of the need to make safe static approximations. Compared to $\mathcal{CES}$, the median static slice size of 99.28% is approximately 11.4 times larger than the median $\mathcal{CES}$ slice. This large median slice size is due to configuring $\mathcal{C}s\mathcal{E}$ to slice the printf call that includes the slicing criterion, which results in multiple memory dependences being over-approximated. We detail this effect in our qualitative discussion. As seen in the chart on the right of Figure 7, the median time taken by $\mathcal{C}s\mathcal{E}$ is 0.70 seconds, making it 992 times faster than $\mathcal{CES}$'s 695s. These differences are in line with related work [57].

Qualitative evaluation. Turning to qualitative differences, a common pattern involves the function prologue and epilogue. Figure 8 illustrates this for the slice taken with respect to `nbody1_e_51`. The $\mathcal{C}s\mathcal{E}$ slice on the right includes the function prologue introduced by the C compiler. The $\mathcal{CES}$ and $\mathcal{CSE}$ slices on the left omit this code. The difference is impacted by the conservative over-approximations made by $\mathcal{C}s\mathcal{E}$ when performing data-flow analysis. In this case, it considers all globals as dependences. On the other hand $\mathcal{CES}$ and $\mathcal{CSE}$ can remove those parts of the prologue and epilogue that do not impact execution. Notice also that the $\mathcal{C}s\mathcal{E}$ slice preserves the writing of 0 to local 2. It turns out that during execution, local 2 is always 0, so the write

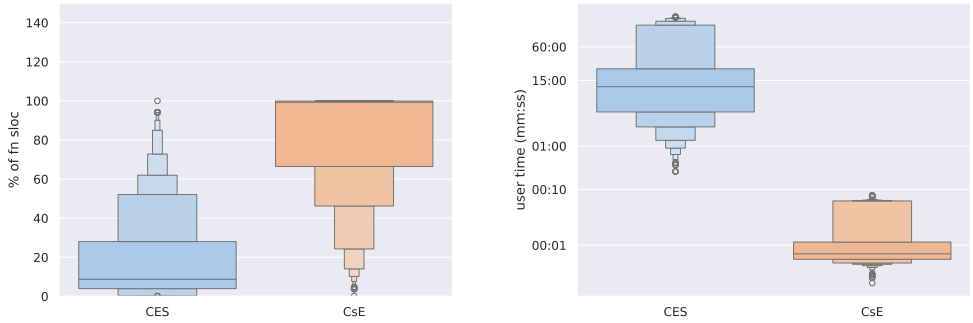

Figure 7: Slice size and times for the static slicer and $\mathcal{CES}$. Note that the time graphs uses a log scale on the y-axis

has no effect. A dynamic slicer can take advantage of this fact to remove the associated code. Finally, the last instruction of both slices is different: $\mathcal{CES}/\mathcal{CSE}$ may only remove instructions, while $\mathcal{CsE}$ can replace instructions with "dummy" instructions. In this case, `local.get 3` was replaced by `f64.const 0`. In effect, the static slice more accurately reflects the return type, `f64`.

Manual inspection of the $\mathcal{CSE}$ slices finds that they preserve the function prologue only 1% of the time. We also noticed several cases where the prologue is partially removed. All are similar to the following $\mathcal{CSE}$ slice taken with respect to `cover_c_112` where the stack pointer is saved in a local and later used to store a value in the linear memory. This may overwrite data used later in the original computation but that is not required by this specific slice. In general, a static slicer is unable to model memory dependences precisely, whereas a dynamic slicer, by its very nature, can precisely characterize them.

$\mathcal{CES}$ / $\mathcal{CSE}$

```
;; prologue:
;;   empty




i32.const 1030
local.get 2
;; slicing criterion
call $printf
drop
;; epilogue:
;;  empty



;; return value
local.get 3
```

$\mathcal{CsE}$

```
;; prologue:
;;  move stack pointer
global.get 0
i32.const 16
i32.sub
local.tee 2
global.set 0
;; body of the function
;; write 0 to local 2
local.get 2
i64.const 0
i64.store
i32.const 1030
local.get 2
;; slicing criterion
call $printf
drop
;; epilogue: restore
;;   stack pointer
local.get 2
i32.const 16
i32.add
global.set 0
;; return value
f64.const 0
```

Figure 8: $\mathcal{CES}$ and $\mathcal{CSE}$ remove the function prologue and epilogue, while $\mathcal{CsE}$ does not.

```
global.get $sp
local.tee 1
global.set $sp
...
local.get 1
local.get 0
i32.const 1
i32.add
i32.store
...
```

Another limitation of $\mathcal{CsE}$ is that memory dependences can't be tracked precisely. Hence, as soon as one memory-related instruction (`load`, `store`) or a call to an external function is part of the slice, all preceding memory-

related instructions are included in the slice. The use of the call to `printf` as the slicing criterion results in many over-approximations. This is not the case for the dynamic slices, which only include the required memory-related instructions. For example, comparing the $\mathcal{CSE}$ and $\mathcal{CsE}$ slice taken with respect to `nsichneu_b_2188`, we observed that both slices include a necessary `store` instruction but that the $\mathcal{CsE}$ slice includes 9224 instructions in total while $\mathcal{CSE}$ includes only 22 instructions. This is due to $\mathcal{CsE}$'s static over-approximation. Such examples are valuable because they inform future work on static slicing where there is a trade-off between the quality of the static approximation and the effort spent.

An interesting case is the slice taken with respect to lms_arg_149. The function lms has six parameters: float lms(float x,float d,float *b,int l, float mu,float alpha). Unfortunately, calls to this function that need not be in the slice cannot be removed by $\mathcal{CES}$ because doing so would require a window size of at least $\delta = 7$ to remove the six instruction-pushing parameters and the call $lms instruction. $\mathcal{CsE}$ does not have the notion of window size and is therefore able to remove calls to the lms function. As a result, $\mathcal{CsE}$ can reduce the function from 105 SLoC to 25 SLoC, while $\mathcal{CES}$ reduces it to 31 SLoC.

Executability of static slices. Finally, we look at the semantics of the static slices. Recall that $\mathcal{CsE}$ produces *closure slices*, which are not guaranteed to preserve the original execution behavior. That 1545 of the 1643 slices (94%) do so is impressive. It does suggest that $\mathcal{CsE}$ is being overly conservative.

Of the 98 slices that fail to exhibit the correct semantics, 81 produce the same value for the slicing criterion at least the first time it is reached in the program but diverge later in the program execution. This is expected because $\mathcal{CsE}$ is focused on preserving the semantics of the program up to the first time the criterion is reached, which is sufficient, for example, in debugging. In cases where the function containing the criterion is called multiple times, $\mathcal{CsE}$ attempts to preserve the semantics of only the first call.

Through manual investigation, we can categorize the remaining 17 cases into two distinct classes of bugs. Thirteen cases are due to the slicer breaking stack discipline for specific patterns. For example, when the stack-polymorphic `unreachable` instruction is being sliced, the slicer does not replace it with instructions that preserve stack validity. The four remaining cases include multiple calls to the function containing the slicing criterion, where the criterion is not reached upon the first call to the function. In these

cases, the return value of the function is not preserved, and the remainder of the program's execution diverges from its original execution, without ever reaching the criterion.

> **RQ4**: Dynamic approaches can perform more aggressive removals than $\mathcal{CSE}$. A classic example is the removal of the function prologue and epilogue. Also, dynamic approaches ensure the executability of the resulting slice. $\mathcal{CSE}$ on the other hand, is not limited by a particular window size and may remove large blobs of consecutive instructions that could not be removed only partially. However, $\mathcal{CSE}$, even if it has a focus on executability, sometimes breaks executability. Looking forward, the shortcomings identified by our experiments can be used to inform future work on static slicing techniques.

4.5. RQ5: What is the ideal window size?

In this research question, we consider the impact of window size (δ), which trades off slice size against slice time. In principle, a larger window size will enable more deletions but will pay for this in increased running time. Previous work [44, 51, 52] identified $\delta = 4$ as an ideal window size for code written in high-level languages. We expect this might be small for lower-level code such as WebAssembly.

To illustrate this expectation we consider the slice taken with respect to spectral-norm1_i_10_for. When using the window size $\delta = 4$, the slice includes the following eight instructions.

```
loop    ;; label = @3      ;; L0  0
  local.get 7              ;; L1 +1
  local.get 8              ;; L2 +1
  i32.const 1              ;; L3 +1
  i32.add                  ;; L4 -1
  local.tee 8              ;; L5  0
  i32.ne                   ;; L6 -1
  br_if 0 (;@3;)           ;; L7 -1
end
```

We have annotated each instruction with its impact on the stack (`end` is not an instruction). Mandating that each deletion satisfies the WebAssembly validation requirement renders the deletion of this loop impossible using a window size of four. It is not until $\delta = 7$ that this loop can be removed.

To investigate window size's impact on the slices we computed the $\mathcal{CES}$ slices using window sizes of 2, 4, 5, 6, 7, 8, 16, and 32. Figure 9 summarizes

the results. Visually we can see that the size of the slice has a drastic decrease from $\delta = 2$ (median 41.57%) to $\delta = 4$ (median 14.34%), but soon thereafter stabilizes. For example, $\delta = 5, 6, 7$, and 8 all have the same median size, 8.96%. Furthermore, the difference between $\delta = 5$ and $\delta = 32$, where the percentage is 8.93%, is quite small.

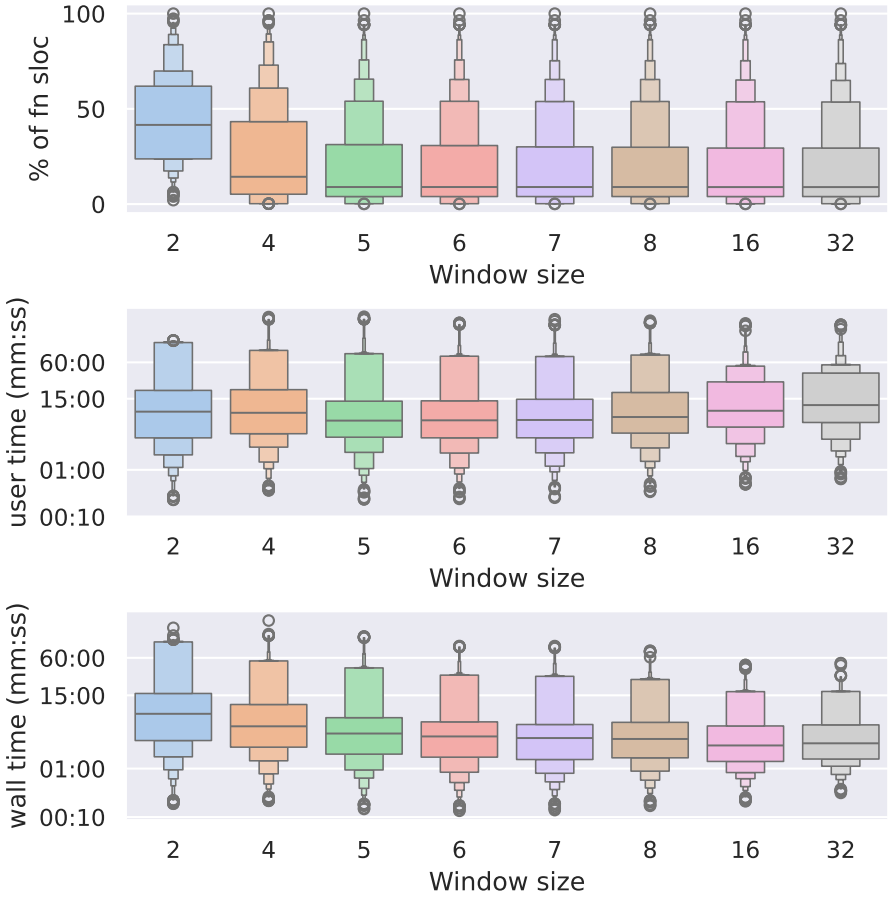

Figure 9: Median slice size and times for a range of window sizes.

In terms of time, we can see from Figure 9 that user time visibly increases after $\delta = 7$. Numerically it bottoms out at $\delta = 5$ where the median time is

391 seconds. For smaller values of δ, ORBS has to perform multiple deletions to remove code that takes a single deletion with larger values of δ. Values larger than $\delta = 5$ incur the cost of starting additional processes for each deletion attempt, which empirically is not worth it from a user-time perspective. Recall that we rely on a parallel version of ORBS and a larger window size involves greater parallelization. Finally, the wall-clock time trend shows an initial reduction before visually stabilizing near $\delta = 6$. Numerically, the bottom is at $\delta = 16$ with a median of 141 seconds. Given unlimited processors, wall-clock time would monotonically decrease, but the machine used in the experiments has only 24 cores per node, so with $\delta = 32$ the computation is spread across multiple nodes, which incurs the cost of increased coordination.

RQ5: Based on slice size a window size larger than $\delta = 4$ is appropriate for slicing lower-level WebAssembly code. Balancing the greater size reduction and diminishing wall-clock time against increasing user time, puts the balance point between 5 and 7. In experiments other than this one, we use a window size of 6.

4.6. RQ6: What is the impact of slicing at multiple levels of abstraction?

We now consider $\mathcal{SCES}$, where we first slice the C source code and then the compiled WebAssembly; thus $\mathcal{SCES}$ can be seen as a combination of $\mathcal{SCE}$ and $\mathcal{CES}$. First, the slice size distribution of $\mathcal{SCES}$ falls between $\mathcal{SCE}$ and $\mathcal{CES}$. Visually, as seen on the left of Figure 10, elements of the two distributions are clearly evident in the $\mathcal{SCES}$ distribution. The results in terms of time are summarized on the right of Figure 10. As expected, because there are two slicing steps, $\mathcal{SCES}$ is slower than both $\mathcal{SCE}$ (where the median slowdown is 15%) and $\mathcal{CES}$ (where it is 45%). The 15% suggests that slicing the WebAssembly code after slicing the C code is comparatively quick. Finally, considering the wall-clock time against our "cup of coffee" practicality goal, 1333 (81%) of the $\mathcal{SCES}$ slices satisfy this requirement. This is only slightly less than $\mathcal{SCE}$'s 83%, again suggesting that $\mathcal{SCES}$'s second slicing phase is quick.

Turning to the individual slices, by construction, an $\mathcal{SCES}$ slice can never be larger than the corresponding $\mathcal{SCE}$ slice. Empirically, in one case the two are the same size and in the other 1642 cases, the $\mathcal{SCES}$ slice is, as expected, smaller. This indicates that the extra slicing step almost always brings value in terms of slice size. Comparing the median sizes shows that it is actually an effective step: $\mathcal{SCE}$ has a median slice size of 17.5%, which drops to 10.0% for

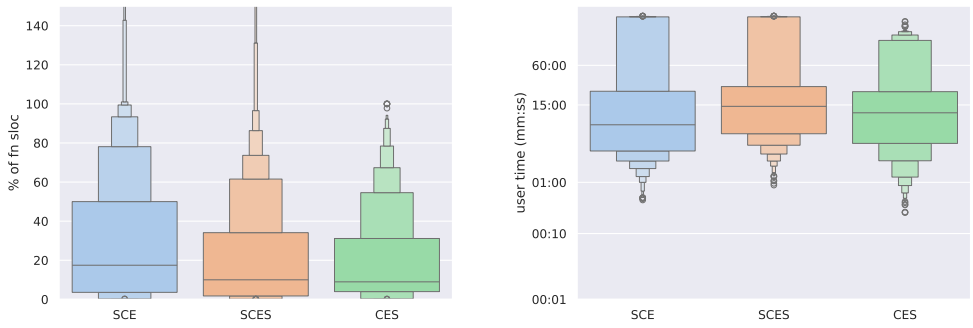

Figure 10: Slice sizes and times comparing with $\mathcal{SCES}$.

$\mathcal{SCES}$. Intuitively, the comparison with $\mathcal{CES}$ is less interesting but follows the expected pattern with 1137 $\mathcal{SCES}$ slices (69%) being smaller, 198 (12%) having the same size and 308 (19%) being larger, which accounts for the tall spire seen in Figure 10.

Looking qualitatively at the largest differences, one of the most extreme cases for a $\mathcal{CES}$ slice being larger is `nbody7_R_31`, where the $\mathcal{SCES}$ slice is 231 lines, while the $\mathcal{CES}$ slice is 339 lines. The explanation here is the presence of the same loop unrolling as in `nbody7_x1_24` considered in Section 4.3 where we compared $\mathcal{SCE}$ and $\mathcal{CES}$.

On the other end, we have the `edn_j_140` slice being 496 lines for $\mathcal{SCES}$ but only 55 for $\mathcal{CES}$. As discussed in Section 4.3 the explanation here is the sliced `C` encouraging compiler optimizations. Note that the $\mathcal{SCES}$ slice has been reduced compared to the $\mathcal{SCE}$ slice in this case (from 585 to 496 lines).

RQ6: $\mathcal{SCES}$ is typically a bit more costly but often results in smaller slices. Outside those cases where compiler optimization makes the slice larger, $\mathcal{SCES}$ proves the more stable slicer —effectively shaving the rough edges off of $\mathcal{SCE}$ and $\mathcal{CES}$. Thus, when the source code is available, it makes a better choice than just $\mathcal{SCE}$ or $\mathcal{CES}$.

4.7. RQ7: What is the impact of a static/dynamic hybrid?

We next consider the $\mathcal{CsES}$ static/dynamic hybrid, which applies the static slicer to the WebAssembly binary and then applies ORBS to the function containing the slicing criterion. The motivation here is to quickly remove code with the static slicer, before applying the more expensive ORBS

approach to the reduced program. $\mathcal{CsES}$ brings value when the static slice captures the correct execution semantics, which is not something that a closure slicer such as $\mathcal{CsE}$ guarantees. In this regard, the static slicer does an impressive job of producing slices with the correct semantics for all but 98 of the original slices.

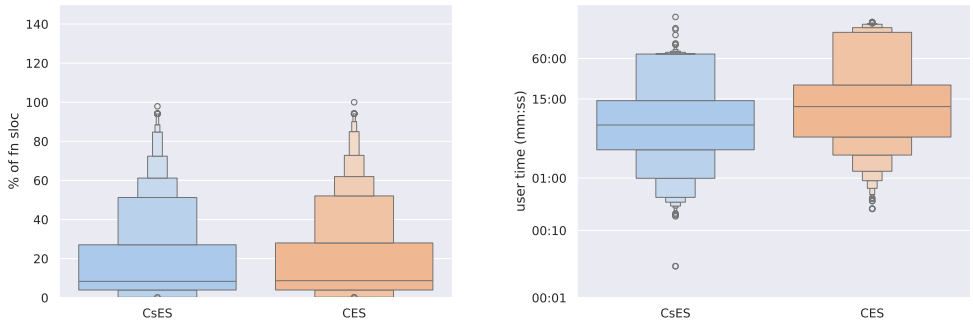

Figure 11: Slice size and time comparison for $\mathcal{CsES}$

We compare $\mathcal{CsES}$ with $\mathcal{CES}$ because consideration of our previous research questions suggests that $\mathcal{CES}$ provides the best trade-offs. The chart on the left of Figure 11 shows the size of the slices. Here $\mathcal{CsES}$ produces the smaller slices, with a median size of 8.37%, compared to $\mathcal{CES}$'s 8.72%. Comparing the individual slices, 130 (8.41%) of the $\mathcal{CsES}$ slices are larger than their $\mathcal{CES}$ counterpart, while 985 (64.75%) are the same size, and 410 (27.83%) and smaller.

Turning to slice time, as shown on the right of Figure 11, $\mathcal{CsES}$ brings the expected drop in user time relative to $\mathcal{CES}$. With a median wall-clock time of 4.33 minutes, it produces 1226 (79%) of its slices in under 15 minutes. With a median user slicing time of 6.16 minutes, $\mathcal{CsES}$ is the fastest of the slicers that dynamically slice WebAssembly. Slicing just the C code and computing a static slice are both faster. Finally, for the majority of the slices (88%), $\mathcal{CsES}$ produces the slice faster than $\mathcal{CES}$. $\mathcal{CsES}$ is also more stable, with a standard deviation of 27.18 minutes compared to $\mathcal{CES}$'s 59.88.

We divide our quantitative inspection of the slices into three groups: the cases where $\mathcal{CES}$ produces a larger slice than both $\mathcal{CsE}$ and $\mathcal{CsES}$ (6 cases), the cases where $\mathcal{CES}$ produces a larger slice than $\mathcal{CsES}$ but for which the $\mathcal{CsE}$ slice was larger than its $\mathcal{CES}$ counterpart (422 cases), and the cases where the $\mathcal{CES}$ slice is smaller than its $\mathcal{CsES}$ counterpart (130 cases).

In the first group, the most extreme case where $\mathcal{CES}$ produces a larger slice is ndes_ietmp2_177, with 286 SLoC for $\mathcal{CES}$ and only 105 SLoC for $\mathcal{CsES}$. In this case, the $\mathcal{CsE}$ slice was already smaller than its $\mathcal{CSE}$ counterpart, with 199 SLoC. It is one case where $\mathcal{CsE}$ has only preserved the semantics for the first few executions of the slicing criterion but eventually diverges. Hence, $\mathcal{CsES}$ inherits this limitation from $\mathcal{CsE}$. $\mathcal{CsES}$ is able to remove function prologue and epilogue which is maintained by $\mathcal{CsE}$, as well as memory dependences that were over-approximated by $\mathcal{CsE}$.

In the same group, looking at slices where the $\mathcal{CsE}$ slice produced a valid slice, an interesting case is lms_arg_149 discussed in RQ4, where $\mathcal{CES}$ would require a higher window size to remove instructions, and therefore gets stuck with a slice of 31 instructions. $\mathcal{CsE}$ is not limited by window size and removes a group of 22 lines, getting its slice size to 25 SLoC because it over-approximates in other places. $\mathcal{CsES}$ is able to further reduce the slice size to only 8 instructions by removing the function prologue as well as other over-approximations made by $\mathcal{CsE}$.

In adpcm_diff_163 we see a similar pattern with extra instructions that are preserved by $\mathcal{CES}$ because they are required by a sequence of instructions that can't be removed due to the window size. In this case, the $\mathcal{CsES}$ slice is 40 SLoC, reduced from the 71 SLoC of the $\mathcal{CsE}$ slice, while the $\mathcal{CES}$ slice is 107 SLoC.

The second group corresponds to cases where $\mathcal{CES}$ is between $\mathcal{CsES}$ and $\mathcal{CsE}$. For example, lms_flag_39 provides a similar situation to lms_arg_149: there is a group of ten lines (five instructions in nested blocks) that cannot be removed by $\mathcal{CES}$ due to the window size. $\mathcal{CsE}$ is able to eliminate the code but over-approximates in other places resulting in a slice that is 19 instructions long. $\mathcal{CsES}$ is then able to remove unneeded memory dependences, as well as the function prologue and epilogue, resulting in a slice of seven instructions. A similar case arises in mandelbrot9_xy_14, where a complex condition finishing with seven consecutive i32.and instructions needs to be removed in order to be able to remove a loop that is not part of the slice: $\mathcal{CES}$ preserves this condition and all its instructions, while $\mathcal{CsE}$ is able to remove them, which enables $\mathcal{CsES}$ to further reduce the slice size.

Finally, the last group corresponds to the 130 cases where $\mathcal{CES}$ produces a smaller slice than $\mathcal{CsES}$. One pattern that can be seen in fasta3_seed_5 is the following. The return value of the sliced function, a floating point number, is irrelevant to the slice, hence $\mathcal{CsE}$ removes the instruction that computes the return value and replaces it by f32.const 0. Moreover, the call to printf

that forms the slicing criterion leaves an integer on the stack, which has to be removed to ensure slice validity, hence $\mathcal{CsE}$ adds a drop instruction. $\mathcal{CsE}$ also over-approximates but $\mathcal{CsES}$ manages to remove all over-approximation code and ends up with a slice whose end is call $printf; drop; f32.float 0. $\mathcal{CES}$ however keeps a f32.convert_i32_u instruction, which effectively converts the return value of printf into a float, resulting in a slice ending with call $printf; f32.convert_i32_u. In the end, the $\mathcal{CES}$ slice is one instruction shorter than its $\mathcal{CsE}$ counterpart.

Another common pattern can be seen with fft1_xp2_64. $\mathcal{CsES}$ is able to reduce the $\mathcal{CsE}$ slice from 98 instructions to 33 instructions. However, this is larger than the 10 instructions of the $\mathcal{CES}$ slice. The cause is that when applied to the $\mathcal{CsE}$ slice it is possible to remove an instruction whose removal requires the subsequent retention of the roughly 23 additional instructions.

While less dramatic, this same pattern occurs in the lms_inc_86 slices where the smaller sizes render the pattern easier to explain. Figure 12 shows the $\mathcal{CsE}$ slice, the $\mathcal{CsES}$ slice, and the $\mathcal{CES}$ slice. The latter two are spaced out to show the correspondence. The pre-slice version of the $\mathcal{CES}$ slice is not shown because it is 118 instructions long.

Looking first at the $\mathcal{CsE}$ slice, Lines 11-14 are the compiled version of the ORBS `printf`, where the address of the variable printed is retrieved by `local.get 1`. Lines 7-9 store a `2` at this address. The $\mathcal{CsE}$ slice is annotated with the three deletions performed when computing the $\mathcal{CsES}$ slice. The key deletion is the second, which removes the instruction `local.get 1`. This deletion is possible because Line 2 places the same value on the top of the stack and Line 5 copies that value into Location 1. However, having deleted Line 7, the slice must retain Lines 2 and 5 as shown in the center column.

Finally, in the $\mathcal{CES}$ slice shown in the rightmost column, Line 7 cannot be removed. Unlike the $\mathcal{CsE}$ slice where the intervening code is only Line 6, the compiled version of `lms_sin` includes over 50 instructions between the `local.tee 1` and the `local.get 1`; thus `local.get 1` is retained and the other instructions eventually all deleted.

$\mathcal{C}s\mathcal{E}$

```
(func (;10;) ...
  global.get 0
3 i32.const 16
3 i32.sub
  local.tee 1
2 global.set 0
2 local.get 1
  i32.const 2
  i32.store

  i32.const 1024
  local.get 1
  call 19
  drop

1 local.get 1
1 i32.const 16
1 i32.add
1 global.set 0
  f32.const 0
)
```

$\mathcal{C}s\mathcal{E}\mathcal{S}$

```
(func (;10;) ...
    global.get 0

    local.tee 1

    i32.const 2
    i32.store

    i32.const 1024
    local.get 1
    call 19
    drop

    f32.const 0
)
```

$\mathcal{C}\mathcal{E}\mathcal{S}$

```
(func $lms_sin ....

        local.get 1
        i32.const 2
        i32.store

        i32.const 1024
        local.get 1
        call $printf
        drop

        local.get 2
)
```

Figure 12: Unexpected larger $\mathcal{C}s\mathcal{E}\mathcal{S}$ slice example.

RQ7: First applying the static slicer brings the anticipated benefit: slices are smaller and are computed faster than the dynamic slicer with the best trade-off, $\mathcal{CES}$. Looking at the slices qualitatively, the main pattern that emerges is that $\mathcal{CsES}$ inherits both benefits and limitations of $\mathcal{CsE}$ and $\mathcal{CES}$: it is able to remove a large range of instructions for which a higher-window slice would be required by $\mathcal{CES}$, but it may break the executability of the slices. Furthermore, to $\mathcal{CsE}$, $\mathcal{CsES}$ has the benefit of overcoming the over-approximating nature of $\mathcal{CsE}$.

4.8. RQ8: How do the slicers behave on real-world applications?

Finally, we evaluate the different slicers on a real-world system. In line with other work that considers WebAssembly [5], we selected the real-world application SQLite, which can easily be compiled to WebAssembly. More precisely, we used the `sqllogictest` program[4], one of the many test drivers for SQLite. This application takes a test script as argument, composed of SQL queries along with the expected results. It runs all queries in the test script one by one and checks that the results are as expected. SQLite is embedded within `sqllogictest`. In total, `sqllogictest` has 169 448 SLoC (non-blank, non-comment, LoC) of C code. Compiled to WebAssembly, `sqllogictest` counts 287 346 instructions.

We wrote one simple test script to be given as input to `sqllogictest`. The test script exercises several parts of SQLite: it creates a table with two columns, inserts two rows, selects all elements from the table, performs an update to one row, and computes an average of the values. We selected six slicing criteria that were reached at least once by the test script. We then applied $\mathcal{CES}$, $\mathcal{SCE}$, $\mathcal{SCES}$, $\mathcal{CsE}$, and $\mathcal{CsES}$ to each criterion. We left out $\mathcal{CSE}$ because RQ1 established that it is not a practical slicer.

The results in terms of slice size are shown in Table 3. The two-phase slicers $\mathcal{CsES}$ and $\mathcal{SCES}$ achieve the highest reductions, followed by the one-phase dynamic slicers $\mathcal{CES}$ and $\mathcal{SCE}$, and finally by the static slicer $\mathcal{CsE}$. These results mostly align with the quantitative results from our other experiments: in agreement with RQ6, $\mathcal{SCES}$ outperforms $\mathcal{SCE}$. However, it also outperforms $\mathcal{CES}$ in this experiment, contrary to the results of RQ6. We explain why below. In agreement with RQ7, $\mathcal{CsES}$ also outperforms $\mathcal{CES}$ and $\mathcal{SCE}$. $\mathcal{CsE}$ unsurprisingly results in large slices, although the median size

[4] `https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki`

of 69.54% is more encouraging than RQ3's median of 99.28%.

There are two factors that lead to $\mathcal{SCES}$ outperforming $\mathcal{CES}$ when slicing the larger SQLite code. First, recall that when considering RQ3 we encountered slices in which $\mathcal{SCE}$ reduced the code to the point that the compiler applied optimizations such as loop unrolling. In the larger SQLite codebase none of the $\mathcal{CSE}$ slices enable such size increasing optimizations and thus the subsequent slicing of the WebAssembly always leads to a smaller slice. The second factor is evident when comparing the SQLite WebAssembly code for the $\mathcal{SCES}$ and $\mathcal{CES}$ slices. The $\mathcal{CES}$ slices include code that is impossible to remove with a window size of six without violating the stack validation requirement. The $\mathcal{SCE}$ slice removes this code at the C level making the combination more effective. While less pronounced, a similar effect is seen when comparing the $\mathcal{CES}$ and $\mathcal{CsES}$ slices. The static slicer is capable of removing rejoins of code that are not in the slice but that are too large for $\mathcal{CES}$ to remove (again without violating the static validation requirement). In essence the static slicer first violates this requirement and then uses a reconstruction step to satisfy the requirement. Doing so with ORBS would mean modifying the WebAssembly runtime during slicing. This suggests there is value in dynamic slicing while ignoring the stack validation requirement and then running the static slicer's reconstruction algorithm.

The results in terms of slicing time are shown in Table 4. The static slicer $\mathcal{CsE}$ is the fastest by a large margin, followed by $\mathcal{CsES}$, $\mathcal{SCE}$, $\mathcal{SCES}$, and finally $\mathcal{CES}$. This is in line with our prior findings. Overall, $\mathcal{CsE}$ is the only slicer that produces all slices in under 15 minutes. $\mathcal{CsES}$ is the only other slicer producing a slice in under 15 minutes. Some of this larger run time is the inevitable cost of executing a large system, but some is a limitation of our current prototype tooling uncovered by this experiment. This limitation is not inherent to ORBS. The slicer needs to convert the text representation of the WebAssembly (a .wat file) into its executable form (a .wasm file), which is costly for a large system. More sophisticated WebAssembly tools would enable the slicer to convert all but the CoI to executable form as a preliminary step ahead of the actual slicing. Doing so would speed up the slicing process.

We investigated the executability of the slices. ORBS' slices are valid by construction and this is indeed the case for all six slices when computed by $\mathcal{CES}$, $\mathcal{SCE}$, and $\mathcal{SCES}$. The executability of $\mathcal{CsE}$ slices depends on the correctness of the static approach: we notice two slices where the executability is broken, namely inner and select. In both cases, the slicing criterion is

reached at least the first time. Hence, $\mathcal{CsE}$ fulfills its promised guarantee of preserving its execution at least once. The executability of the $\mathcal{CsES}$ slices is directly impacted by the executability of the corresponding $\mathcal{CsE}$ slice, and the results are the same.

Comparing the slices themselves, three patterns emerge. These include the two noted above where the $\mathcal{SCE}$ and $\mathcal{CsE}$ slicers are able to remove code that the $\mathcal{CES}$ slicer cannot. In addition, the $\mathcal{SCES}$ slices make good use of the optimizer. A pattern that was not prevalent in the earlier experiments that occurs in four of the six SQLite slices involves the $\mathcal{SCE}$ slicer removing one branch of a conditional, such as Lines 4 and 5 in the following code. When clang compiles the sliced code it essentially ignores Lines 3, 6, and 8, exploiting the fact that r is undefined in the true branch and thus might as well take on the value p->iSum. The resulting code omits a level of nesting which enables the $\mathcal{CES}$ slicer to remove more code than it can when the nesting structure is present because the removal of this structure is difficult without violating the stack validation requirement.

```
1    if( p && p->cnt>0 ){
2      double r;
3      if( p->approx ){
4        r = p->rSum;
5        if( !sqlite3IsOverflow(p->rErr) ) r += p->rErr;
6      }else{
7        r = (double)(p->iSum);
8      }
9      printf("\nORBS: %.0f\n", r);
10     sqlite3_result_double(context, r/(double)p->cnt);
11   }
```

RQ8: On a larger application, our findings remain in line with the previous research questions. Size-wise, the hybrid slicers ($\mathcal{SCES}$ and $\mathcal{CsES}$) perform better. Time-wise, they also generally perform well, with $\mathcal{CsES}$ being the fastest slicer aside from $\mathcal{CsE}$. Performance results however are on the lower side, with only one non-static slice taking less than our 15-minute goal (by $\mathcal{CsES}$), and none by the other slicers that include a dynamic component. These are due to limitations in the tooling and not in the approach itself.

Table 3: Slice Size for the `sqllogictest` experiments in WebAssembly SLoC. Percentages represent the size of the final slice as a proportion of the size of the function containing the slicing criterion measured in WebAssembly instructions.

Criterion	Function SLoC	$\mathcal{CES}$ %	$\mathcal{SCE}$ %	$\mathcal{SCES}$ %	$\mathcal{C}s\mathcal{E}$ %	$\mathcal{C}s\mathcal{ES}$ %
avg	57	35.09%	54.39%	26.32%	92.98%	36.84%
create	267	9.36%	11.99%	6.37%	94.76%	10.86%
drop	234	2.56%	11.54%	2.56%	29.06%	2.99%
inner	982	10.59%	9.78%	4.99%	57.03%	3.26%
insert	1944	3.96%	1.08%	0.31%	82.05%	1.70%
select	3387	2.83%	4.07%	1.09%	43.22%	1.24%
Average		10.73%	15.47%	6.94%	66.52%	9.48%
Median		6.66%	10.66%	3.78%	69.54%	3.13%

Table 4: Slicing time for the `sqllogictest` experiments. Times are given in seconds and represent the total slicing time in user time, as well as the time to slice per SLoC of the function containing the criteria, in terms of WebAssembly instructions. Underlined entries are the ones below our 15-minute threshold.

Criterion	Function SLoC	$\mathcal{CES}$ time	$\mathcal{SCE}$ time	$\mathcal{SCES}$ time	$\mathcal{C}s\mathcal{E}$ time	$\mathcal{C}s\mathcal{ES}$ time
avg	57	1150 (20.18)	5377 (94.34)	6161 (108.09)	9 (0.16)	1050 (18.41)
create	267	4137 (15.49)	5784 (21.66)	6606 (24.74)	9 (0.03)	4201 (15.73)
drop	234	2392 (10.22)	6088 (26.02)	6680 (28.55)	9 (0.04)	854 (03.65)
inner	982	17222 (17.54)	3867 (03.94)	6186 (06.30)	9 (0.01)	7012 (07.14)
insert	1944	21082 (10.84)	8905 (04.58)	9182 (04.72)	10 (0.00)	14323 (07.37)
select	3387	33248 (09.82)	13322 (03.93)	16347 (04.83)	10 (0.00)	13636 (04.03)
Average		13205 (14.02)	7224 (25.75)	8527 (29.54)	9 (0.04)	6846 (09.39)
Median		10680 (13.17)	5936 (13.12)	6643 (15.52)	9 (0.02)	5607 (07.25)

4.9. Discussion

Our initial experiments repeatedly pointed to $\mathcal{CES}$ as the preferred approach. $\mathcal{CSE}$'s slicing of the entire WebAssembly file is clearly too expensive. $\mathcal{SCE}$'s slicing at the C-code level suffers from some undesirable compiler interactions and what might be called a lack of boundaries. Furthermore, our qualitative comparison finds the $\mathcal{CES}$ slices preferable to those of $\mathcal{CSE}$ and $\mathcal{SCE}$.

In terms of slice size, $\mathcal{CES}$ produces smaller slices than $\mathcal{SCES}$, which finds a middle ground between $\mathcal{SCE}$ and $\mathcal{CES}$. However, $\mathcal{SCES}$ does a better job on a real-world scenario, yielding smaller slices than $\mathcal{CES}$ in our SQLite experiments. It is interesting that in this scenario the two slicing phases seem to better compliment each other. $\mathcal{CsES}$ produces the smallest slices thanks to the static slicer enabling deletions that would not be possible for $\mathcal{CES}$ due to the window size. This comes at the cost of inheriting the limitations of the static slicer in terms of the executability guarantee.

In terms of our "cup of coffee" definition of practical, $\mathcal{CsE}$ produces all the slices of our subjects of Table 1 in under 15 minutes. $\mathcal{SCE}$ is the next most successful slicer, producing 83% of the slices in under 15 minutes of wall-clock time. It is followed by $\mathcal{SCES}$ (81%), $\mathcal{CsES}$ (79%), and $\mathcal{CES}$ (74%). On a real-world scenario, however, the dynamic slicers require much more time, with only $\mathcal{CsES}$ producing a slice in under 15 minutes. This could be solved by improving the WebAssembly tooling without requiring changes in the slicers. While static slicing and slicing the C source are faster, when it comes to slicers that include slicing the WebAssembly code, our two hybrid approaches are both improvements over the non-hybrid slicers.

There are however several issues worthy of consideration. The rest of this section considers the four most prominent. The first issue concerns the impact of the execution environment. We observed during our experiments that the slice characteristics can be influenced by the environment. Notably, the optimization setting of the C compiler has an important impact on the WebAssembly code. More work needs to be done to better characterize the impact of the environment. There is a link here with recent work that studied how the execution environment impacts slicing [58].

Next, we revisited past experiments that consider OBRS' window size. At the WebAssembly level with a window size of four, it is impossible to remove a call with four or more parameters because the code pushes all the arguments before the call pops them. This observation motivated RQ5

and also the design-space search that led to the value in $\mathcal{SCES}$'s multi-level slicing.

Third, one key challenge when slicing WebAssembly programs is maintaining the validity of the code. We do this by requiring the slice to be valid and executable after each removal. This prevents ORBS from removing certain patterns, which would temporarily violate validity but preserve the semantics. The static slicer includes a *reconstruction phase* [42], which might be used to reconstruct a valid program from an "invalid" dynamic slice. Improving the dynamic slicer with such a phase could potentially enable the slicer to produce smaller slices.

Finally, our experiments reveal that there are cases where the $\mathcal{CsE}$ slices clearly under-approximate the true dependence. Further comparison between the $\mathcal{CsE}$ and dynamic slices will help us better understand this dependence under-approximation and consequently should suggest improvements to the static slicer.

4.10. Threats to Validity

We identify threats to validity according to the classification of Wohlin et al. [59]. We instrument each subject of our evaluation for each use of a scalar variable in the program. As a threat to internal validity, other than a few subjects, we did not consider the use of pointer variables as slicing criterion. We leave this for future work. To enable comparing the different slices, we have removed criteria whose slices exhibited non-deterministic behavior and for which the $\mathcal{SCE}$ slice does not contain the criterion because it was removed during the optimization phase of the `C` compiler.

A threat to external validity is our selection of subjects. These are all `C` programs and our results may not generalize to other source languages. We selected our subjects from various sources, given that there is no standard benchmark of `C` programs that compile to WebAssembly.

5. Related Work

WebAssembly. There has been interest from the research community in WebAssembly on aspects such as security [5, 6, 7, 8, 9, 10, 11], extensions to the language [60, 61, 62], tooling [63], program analysis [13, 64, 14, 15, 16, 65], and optimizations [66]. WebAssembly being an assembly language, has been compared to other assembly languages such as x86 [1, 8, 9]. We compare our approach to $\mathcal{CsE}$, the only existing slicer for WebAssembly [42]. To date, no

other slicers for WebAssembly exist. Relying on existing dynamic analysis frameworks [13] to implement a dynamic slicer remains to be investigated.

Slicing Applications. Application of program slicing [21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40] stem from its ability to reduces the side of the program under consideration. In their survey of empirical results in program slicing [20], Binkley and Harman report that the typical static slice is approximately one third of the program, while the typical dynamic slice is approximately one fifth of the program. The value this reduction brings in some cases, for example when the source code is not available, requires the ability to slice at the *binary level*. This is the case for re-engineering, program comprehension, and security analyses. A web browser may, for example, run an analysis against a WebAssembly binary before running the binary. Another use case is debugging WebAssembly virtual machines, where a bug report containing hundreds of WebAssembly instructions could be reduced to tens of instructions through program slicing.

Language-Independent Slicing. Our work relies on ORBS, a language-independent slicing approach that observes the program output in order to build an executable slice [44], which works at the line level [67, 68]. QSES is a variant of ORBS that protects all lines of a static slice during dynamic slicing, which has been applied to C programs [45, 52], but not at the assembly level. It would be more beneficial for binary slicing to first apply static slice to quickly eliminate a large portion of the code, before applying dynamic slicing to further remove instructions. The use of ORBS stands alone as all other dynamic slicers first instrument either the program itself or its underlying runtime environment, and then use this instrumentation to collect an execution trace of the running program [69].

Hybrid Slicing. Dynamic slicing traces its roots back to the work of Korel and Laski [70] who proposed a slicer based on the analysis of execution traces. Parallel to the early development of static slicing [21, 71], Agrawal and Horgan followed with a slicer that cached dependence information in a dependence graph [72]. One of their four algorithms was essentially three steps: compute the program's program dependence graph (PDG), remove all graph vertices that do not show up in an execution trace, and then slice the reduced graph using the standard PDG static slicing algorithm. To the best of our knowledge this is the first static/dynamic hybrid slicing algorithm.

Almost a decade later, Nishimatsu et al.'s call-mark slicing [73] again exploits both static and dynamic information. Their motivation is that dynamic slicing has a large time and space overhead because of the execution trace size and the resulting large numbers of dynamic dependences. On the other hand, conservative static data-flow analysis leads to imprecision. In comparison, these two hybrid approaches are more integrated than ours, which essentially combines static and dynamic slicers as black boxes. These differences suggest a possible spectrum of possibilities where our black-box approach is likely one end of the spectrum.

Slicing Binary Executables. Cifuentes and Fraboulet describe the first slicer for binary executables [31]. Their work outlines the necessary modifications to conventional slicing techniques needed to slice machine code and assembly-type languages. A few years later Kiss et al. [74, 75] built on this to produce an *interprocedural* static slicer for binary executables. The size of their interprocedural slices was between 56% to 68% of the instructions in the original program. Similar to $\mathcal{CSE}$, this implementation contains an imprecise (but safe) memory dependence model.

Mangean et al. [76] built on the work of Kiss et al. [74], specifically the latter's graph-based approach, but widen the tools applicability to dynamic slicing of binaries, independent from the target platform, by exploiting a hardware simulator to extract data-flow information about each instruction. This work is in some ways the most directly comparable to ours as they too study the Mälardalen benchmark suite [54].

They report slice sizes for a number of different compiler configurations. The average slice size is 22% of the code using `gcc` with no optimization, `-O0`, and 37% when (greater) optimization is used. While we don't consider the influence of compiler optimization, in a simple experiment we see the same pattern where the larger unoptimized code enables greater reduction. Using the same optimization level, $\mathcal{SCE}$'s mean of 31% shows a similar performance. With mean sizes of 23% and 20%, $\mathcal{CSE}$ and $\mathcal{CES}$ produce smaller slices which reflects the greater precision that ORBS brings to the slicing challenge. Mangean et al. do not provide detailed timing information, other than to note that all slices were computed in under a second. The higher slicing time of ORBS is clearly the cost paid for its greater precision.

Mangean et al. note some limitations of their tool such as programs containing floating point arithmetic or switch-case statements and recursive programs. By directly observing the program in execution, observation based

slicing is not challenged by these features, nor other common challenging features such as reflection and third party libraries.

Slicing Android Apps. We now discuss two major tools to slice Android apps, another form of binary executables that operate on a register-based virtual machine (the Dalvik VM before Android 5.0, and ART afterwards).

Ahmed et al. study Mandoline, a dynamic slicer for Android apps that slices compiled Jimple code [69], motivated by past work with the tool Android Slicer [77], which makes several decisions that sacrifice accuracy for low instrumentation overhead. In contrast, Mandoline, aims to use minimal, low-overhead instrumentation followed by sophisticated, on-demand execution trace analysis. In comparison, this trade-off pays off and their approach is faster and more accurate than its predecessor.

Mandoline is evaluated on 12 Android apps with known faults. Impressively, the authors construct ground truth slices for these apps, a task that took a daunting 30 work days. Each slice reproduces a bug, which leads to the app crashing. The size of the slices (on average 0.04% of the code) is orders of magnitude smaller than other slices. Since this includes the manual slices and the output of two tools there must be something in the nature of the criteria that leads to such small slices. For example, one artifact of choosing a crash as the criterion is that it truncates the trace. Shorter traces clearly reduce slicing time. It is also possible that early termination means that the buggy statement executes fewer times and thus less of its dependence gets uncovered. Evidence for this comes from comparing the size of the program to the size of the traces. The number of instruction instances is typically four to five *orders of magnitude* larger [78] but in this study is only 2.5 times as many.

The slice times are also interesting. Android Slicer takes about 4000 seconds on average, while Mandoline about 725. This is a far cry from the "less than a second" reported by Mangean et al. Assuming Jimple instructions are comparable to those of WebAssembly, $\mathcal{SCE}$ takes an average of 450 seconds and $\mathcal{CES}$ 700 seconds. However, the twelve Android apps are about an order of magnitude larger.

Given their ground truth slices, the authors present the precision and recall for each slice. Using this metric, ORBS is a clear winner. For example, its slices all attain 100% recall as removal of any statement from an ORBS slice will change the computation of the slicing criteria. Likewise, while not nearly at the same level, we include comparison with ground truth slices for

examples such as the MBE example, with its illustration of the fact that data dependence is non-transitive. For these examples we attain 100% precision.

Similar to the other dependence based slicers, Mandoline slices include unnecessary statements because it makes approximations. For example, when looking for a definition of an element in a certain position within an array, Mandoline includes definitions of all array elements. The authors also note the following challenges of slicing in the Android environment: programs are event-driven, asynchronous, and execute framework code. Observation based slicing again sidesteps these challenge.

Slicing Java byte code. We conclude by considering three Java byte code slicers, which like WebAssembly uses a stack-based architecture.

Wang et al. [78], present a dynamic slicer for Java that performs trace collection using a modified Kaffe Virtual Machine. They make use of "reverse" stack simulation to identify implicit data dependences between byte codes involving data transfer via the operand stack.

Their experimental subjects average about 10 000 byte codes, which, assuming Java byte codes are comparable to WebAssembly instructions, is essentially the same size as the programs we study.

Their overall average slice includes 16% of the original byte codes, which is slightly less than $\mathcal{CES}$'s 20%. The most likely explanation for this is WebAssembly's stack validation requirement. Our manual inspection of the WebAssembly slices includes multiple examples of code that can be removed without affecting program behavior except that it violates the stack validation requirement. Comparison with the work of Wang et al. suggests that the cost of this requirement is no more than 4% modulo imprecision in their analysis. Finally, the paper provides only relative timing comparison, so it is unclear how much time the slicer takes.

The authors note that their tool is challenged by Java language features such as reflection, native methods, and multi-threaded code. All challenges that observation-based slicing sidesteps.

Ahmed et al. [79] present Slicer4J and compare it with the older JavaSlicer [80]. They note that JavaSlicer only supports programs up through Java 6, while Slicer4J supports up through Java 9. One clear advantage to observation based slicing is that it supports current *and future* versions of Java provided there is a compiler for the language.

To support Java framework methods and native code, Slicer4J relies on a set of pre-constructed data-flow summaries of framework methods. Here

again our dynamic slicers do not incur the cost of the construction of such models.

Unlike JavaSlicer, which modifies the JVM, Slicer4J instruments JAR files. Thus, Slicer4J is unable to slice classes that are dynamically generated at runtime. For programs that are large but have small execution traces, JavaSlicer's instrumenting on-demand saves a substantial portion of instrumentation cost, leading to better performance. Likewise, observation based slicers benefit when the tests execute only a small portion of a (large) program.

Empirically, the authors compare Slicer4J and JavaSlicer using three programs from the Defects4J benchmark [81] again using failing test assertion statements as the slicing criteria. These three are an order of magnitude larger than the subjects of Table 1. Slicer4J takes an order of magnitude less time than JavaSlicer requiring between one and four minutes to instrument, execute, and slice. Given that the subjects are an order of magnitude larger, this is still notably faster than the dynamic slicers we study. As with the slices computed by Mandoline, the average slice size ranges from 0.13% to 4.07% of the program's LoC, likely for the same reason.

6. Conclusion

We introduce four dynamic slicing approaches for WebAssembly binaries, namely $\mathcal{CES}$, $\mathcal{CSE}$, $\mathcal{SCE}$, and $\mathcal{SCES}$, as well as a hybrid slicer, $\mathcal{CsES}$. We compared these and the static WebAssembly slicer $\mathcal{CsE}$ using eight research questions. Our evaluation using a set of benchmark programs shows that $\mathcal{CSE}$ requires much more time than the other approaches, while not reducing slice size as much as expected. $\mathcal{SCE}$ itself may result in slices that grow in size and have a different structure, due to the compilation phase happening after slicing, which leaves room for extra optimization. A static approach is favorable in terms of running time but results in the largest slices due to its over-approximation. $\mathcal{SCES}$, a combination of both $\mathcal{SCE}$ and $\mathcal{CES}$, improves the stability of the slices, reducing the size of $\mathcal{SCE}$ slices. In terms of the purely dynamic approaches, for the benchmark programs, $\mathcal{CES}$ yields the best trade-off in terms of running time, slice size, and inspectability of the resulting slices. It is interesting that for our larger, non-benchmark, system, we see that $\mathcal{SCES}$ (and $\mathcal{CsES}$) benefit from the two slicing phases. Thus given the modest increase in computation time that $\mathcal{SCES}$ brings, it may be preferred when slicing larger systems. Likewise, combining static and

dynamic slicing, $\mathcal{CSES}$ brings a considerable reduction in slicing time and may be preferred in applications where semantic preservation is not crucial.

Our empirical investigation suggests several avenues for future work. The validation requirement could be lifted to create smaller slices as long as the slices are later *reconstructed* using an algorithm akin to the one used by the static slicer. This should enable the dynamic slicers to produce smaller slices.

Regarding window size, our experiments have shown that $\delta = 6$ is a sweet spot, which is a notable difference to slicing at the source level where $\delta = 4$ yields best results [44, 51, 52]. However, we still notice multiple patterns where a higher window size is desirable. Our experiments have shown that this has a cost in slicing time. Experimenting with heuristics to adapt δ dynamically could benefit slice size with a lesser impact on slicing time. We noticed for example that most large deletions happen in the first pass of the slicer, indicating that subsequent passes might limit window sizes to smaller values. Another interesting heuristic would be to "back up" by one line after a deletion, instead of continuing slicing with the next line.

Finally, some of the patterns we encountered involved the `local.tee` instruction that could not be removed as it was needed to write over a local. A `local.tee x` instruction is equivalent to `local.set x; local.get x`. In multiple slices, an amorphous slice [82] would be helpful as it could replace the `local.tee` instruction with `local.set`.

In summary, compared to our earlier investigation of dynamic slicing for WebAssembly [43], this work shows that a larger window size is desirable when slicing at a lower level of abstraction, that slicing at multiple levels of abstraction further reduces slice size in most cases at a small cost in additional run time, and that a combination of static and dynamic slicing improves both slicing time and slice size compared to the previously established best trade-off, $\mathcal{CES}$.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, 2017, pp. 185–200.

[2] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world WebAssembly binaries: Security, languages, use cases," in *WWW*

'21: The Web Conference 2021, J. Leskovec, M. Grobelnik, M. Najork, J. Tang, and L. Zia, Eds. ACM / IW3C2, 2021, pp. 2696–2708.

[3] "Wasmer: The leading WebAssembly runtime supporting wasi and emscripten." https://github.com/wasmerio/wasmer.

[4] J. Ellul and G. J. Pace, "Alkylvm: A virtual machine for smart contract blockchain connected internet of things," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–4.

[5] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An embedded trusted runtime for WebAssembly," in *37th IEEE International Conference on Data Engineering, ICDE 2021*. IEEE, 2021, pp. 205–216.

[6] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *29th USENIX Security Symposium, USENIX Security 2020*, 2020.

[7] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting," in *Proceedings of the 20th International Middleware Conference, Middleware 2019*, 2019, pp. 123–135.

[8] Q. Stiévenart, C. De Roover, and M. Ghafari, "The security risk of lacking compiler protection in WebAssembly," in *21st IEEE International Conference on Software Quality, Reliability, and Security*. IEEE, 2021.

[9] ——, "Security risks of porting C programs to WebAssembly," in *The 37th ACM/SIGAPP Symposium On Applied Computing*. ACM, 2022.

[10] M. E. Mazaheri, S. B. Sarmadi, and F. T. Ardakani, "A study of timing side-channel attacks and countermeasures on javascript and webassembly," *ISC Int. J. Inf. Secur.*, vol. 14, no. 1, pp. 27–46, 2022.

[11] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Comput. Secur.*, vol. 118, p. 102745, 2022.

[12] J. Cabrera-Arteaga, M. Monperrus, T. Toady, and B. Baudry, "Webassembly diversification for malware evasion," *Comput. Secur.*,

vol. 131, p. 103296, 2023. [Online]. Available: https://doi.org/10.1016/j.cose.2023.103296

[13] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing WebAssembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019*, 2019, pp. 1045–1058.

[14] Q. Stiévenart and C. De Roover, "Compositional information flow analysis for WebAssembly programs," in *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020*. IEEE, 2020, pp. 13–24.

[15] Q. Stiévenart and C. De Roover, "Wassail: a WebAssembly static analysis library," in *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[16] F. Marques, J. F. Santos, N. Santos, and P. Adão, "Concolic execution for webassembly," in *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, ser. LIPIcs, K. Ali and J. Vitek, Eds., vol. 222. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 11:1–11:29.

[17] K. Brandl, S. Erdweg, S. Keidel, and N. Hansen, "Modular abstract definitional interpreters for webassembly," in *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, ser. LIPIcs, K. Ali and G. Salvaneschi, Eds., vol. 263. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 5:1–5:28.

[18] Y. Xia, P. He, X. Zhang, P. Liu, S. Ji, and W. Wang, "Static semantics reconstruction for enhancing javascript-webassembly multilingual malware detection," in *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part II*, ser. Lecture Notes in Computer Science, G. Tsudik, M. Conti, K. Liang, and G. Smaragdakis, Eds., vol. 14345. Springer, 2023, pp. 255–276.

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

[20] D. W. Binkley and M. Harman, "A survey of empirical results on program slicing," *Advances in Computers*, vol. 62, pp. 105–178, 2004.

[21] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, S. Jeffrey and L. G. Stucki, Eds. IEEE Computer Society, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=802557

[22] M. Kamkar, N. Shahmehri, and 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.

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

[24] D. W. Binkley, L. R. Raszewski, C. Smith, and 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.

[25] E. Hosnieh and H. Haga, "A novel approach to program comprehension process using slicing techniques," *J. Comput.*, vol. 11, no. 5, pp. 353–364, 2016.

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

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

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

[29] 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.

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

[31] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *1997 International Conference on Software Maintenance (ICSM '97)*. IEEE Computer Society, 1997, p. 188.

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

[33] M. Harman and S. Danicic, "Using program slicing to simplify testing," *Softw. Test. Verification Reliab.*, vol. 5, no. 3, pp. 143–162, 1995.

[34] D. W. Binkley, "The application of program slicing to regression testing," *Inf. Softw. Technol.*, vol. 40, no. 11-12, pp. 583–594, 1998.

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

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

[37] T. Akgul, V. J. M. III, and 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.

[38] L. Philips, C. De Roover, T. Van Cutsem, and 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.

[39] 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.

[40] 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.

[41] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Comput. Surv.*, vol. 44, no. 3, Jun. 2012.

[42] Q. Stiévenart, D. W. Binkley, and C. De 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.

[43] Q. Stiévenart, D. W. Binkley, and C. D. Roover, "Dynamic slicing of webassembly binaries," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2023, Bogotá, Colombia, October 1-6, 2023*. IEEE, 2023, pp. 84–96.

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

[45] Q. Stiévenart, D. W. Binkley, and C. De Roover, "QSES: quasi-static executable slices," in *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021*. IEEE, 2021, pp. 209–213.

[46] 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)*. IEEE Computer Society, 2003, pp. 88–97.

[47] A. Rossberg, "WebAssembly Core Specification," W3C, Tech. Rep., 2019. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[48] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[49] H. Agrawal, "On slicing programs with jump statements," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, V. Sarkar, B. G. Ryder, and M. L. Soffa, Eds. ACM, 1994, pp. 302–312.

[50] S. Yoo, D. W. Binkley, and R. D. Eastman, "Observational slicing based on visual semantics," *J. Syst. Softw.*, vol. 129, pp. 60–78, 2017.

[51] 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.

[52] Q. Stiévenart, D. Binkley, and C. De Roover, "An empirical evaluation of quasi-static executable slices," *Journal of Systems and Software*, vol. 200, p. 111666, 2023.

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

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

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

[56] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 2, pp. 1–32, 2007.

[57] D. W. Binkley and M. Harman, "A survey of empirical results on program slicing," *Adv. Comput.*, vol. 62, pp. 105–178, 2004.

[58] D. Binkley and L. Moonen, "Assessing the impact of execution environment on observation-based slicing," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 40–44.

[59] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer, 2012.

[60] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position paper: Progressive memory safety for WebAssembly," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2019*, 2019, pp. 4:1–4:8.

[61] D. Pinckney, A. Guha, and Y. Brun, "Wasm/k: delimited continuations for WebAssembly," in *DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, M. Flat, Ed. ACM, 2020, pp. 16–28.

[62] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, "Secwasm: Information flow control for webassembly," in *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings*, ser. Lecture Notes in Computer Science, G. Singh and C. Urban, Eds., vol. 13790. Springer, 2022, pp. 74–103.

[63] A. Romano and W. Wang, "WasmView: visual testing for WebAssembly applications," in *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 13–16.

[64] C. Watt, P. Maksimovic, N. R. Krishnaswami, and P. Gardner, "A program logic for first-order encapsulated WebAssembly," in *33rd European Conference on Object-Oriented Programming, ECOOP 2019*, 2019, pp. 9:1–9:30.

[65] D. Lehmann and M. Pradel, "Finding the dwarf: recovering precise types from webassembly binaries," in *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, R. Jhala and I. Dillig, Eds. ACM, 2022, pp. 410–425.

[66] J. Cabrera-Arteaga, S. Donde, J. Gu, O. Floros, L. Satabin, B. Baudry, and M. Monperrus, "Superoptimization of WebAssembly bytecode," in *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming*, A. Aguiar, S. Chiba, and E. G. Boix, Eds. ACM, 2020, pp. 36–40.

[67] D. W. Binkley, N. Gold, S. S. Islam, J. Krinke, and S. Yoo, "Tree-oriented vs. line-oriented observation-based slicing," in *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017*. IEEE Computer Society, 2017, pp. 21–30.

[68] ——, "A comparison of tree- and line-oriented observational slicing," *Empir. Softw. Eng.*, vol. 24, no. 5, pp. 3077–3113, 2019.

[69] K. Ahmed, M. Lis, and J. Rubin, "Mandoline: Dynamic slicing of android applications with trace-based alias analysis," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 105–115.

[70] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, Oct. 1988.

[71] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[72] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1990, pp. 246–256.

[73] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue, "Call-mark slicing: An efficient and economical way of reducing slices," in *Proceedings of the 21st International Conference on Software Engineering*. ACM Press, May 1999, pp. 422–431.

[74] Á. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy, "Interprocedural static slicing of binary executables," in *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*. IEEE Computer Society, 2003, p. 118.

[75] Á. Kiss, J. Jász, and T. Gyimóthy, "Using dynamic information in the interprocedural static slicing of binary executables," *Softw. Qual. J.*, vol. 13, no. 3, pp. 227–245, 2005.

[76] A. Mangean, J. Béchennec, M. Briday, and S. Faucou, "BEST: a binary executable slicing tool," in *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016*, ser. OASICS, M. Schoeberl, Ed.,

vol. 55. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 7:1–7:10.

[77] T. Azim, A. Alavi, I. Neamtiu, and R. Gupta, "Dynamic slicing for android," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 1154–1164. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00118

[78] T. Wang and A. Roychoudhury, "Dynamic slicing on java bytecode traces," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 2, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1330017.1330021

[79] K. Ahmed, M. Lis, and J. Rubin, "Slicer4j: a dynamic slicer for java," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1570–1574. [Online]. Available: https://doi.org/10.1145/3468264.3473123

[80] C. Hammacher, "Design and implementation of an efficient dynamic slicer for java," Bachelor's Thesis, University of Saarland Saarbrucken, Germany, 2008.

[81] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 437–440.

[82] M. Harman, D. W. Binkley, and S. Danicic, "Amorphous program slicing," *Journal of Systems and Software*, vol. 68, no. 1, Oct. 2003.