Dynamic Slicing of WebAssembly Binaries

Quentin Stiévenart
Université du Québec à Montréal
Montreal, Canada
stievenart.quentin@uqam.ca

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

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

***Abstract*—The recently introduced WebAssembly standard aims to form a portable compilation target, enabling the cross-platform distribution of programs written in a variety of languages. In this paper, we propose and investigate the first dynamic slicing approaches for WebAssembly. Given a program and a location in that program, a program slice is a reduced version of the program that preserves the behavior at the given location. Slicing has numerous applications in software maintenance and evolution, including reverse engineering, code comprehension, and quality assurance.**

Our dynamic approaches are built on Observational-Based Slicing (ORBS). We explore the design space for instantiating ORBS for WebAssembly: for example, it 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 afterwards. We evaluate the slices produced quantitatively and qualitatively, and compare them to those obtained by a state-of-the-art static slicer for WebAssembly. 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.

***Index Terms*—Dynamic program slicing, WebAssembly, Binary analysis**

I. Introduction

WebAssembly is a recent binary format [28] with many diverse use cases [32] both on the Web and beyond such as in desktop applications [66] and smart contracts [22]. The academic literature has focused on the security aspects of WebAssembly [47], [40], [27], [60], [61], [46], [15] and on tools and techniques for analyzing WebAssembly binaries [41], [58], [59], [45], [15]. The rise in its use brings a growing need for tools to support the maintenance of WebAssembly code.

Program slicing [68], [14] provides a basis for such tools. Given a program point called the *slicing criterion*, program slicing identifies a reduced program that preserves the computation at the slicing criterion. Program slicing has numerous applications such as debugging [69], [35], [39], program comprehension [13], [33], [19], [38], [62], software maintenance [26], [29], re-engineering [17], refactoring [23], testing [30], [8], [31], reverse engineering [4], [2], tierless or multi-tier programming [49], [48], and vulnerability detection [53].

Program slicers exist along multiple dimensions [54] including static vs. dynamic, executable vs. closure, and backward vs. forward. To date, the only existing slicer for WebAssembly is a static backward closure slicer, which we refer to as $\mathcal{SWS}$, Static Webassembly Slicer [57].

This paper introduces and studies 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. Reducing this code to tens of instructions is a win. The design space for dynamic slicers that can slice WebAssembly is vast and until now unexplored. We consider options involving three key slicing phases: compiling the program to WebAssembly ($\mathcal{W}$), extracting the function containing the slicing criterion ($\mathcal{E}$), and removing (slicing out) irrelevant code ($\mathcal{S}$). For example, given a program P, one possible arrangement is to slice P at the source level, convert the reduced program to WebAssembly, and then extract the function that contains the slicing criterion. Mathematically, we have $\mathcal{E}(\mathcal{W}(\mathcal{S}(P)))$, which we denote $\mathcal{EWS}$ for short.

As a second example, $\mathcal{S}(\mathcal{E}(\mathcal{W}(P)))$, which we denote $\mathcal{SEW}$, first compiles the source to WebAssembly then extracts the function containing the slicing criterion, and finally removes irrelevant instructions from this function. Comparing these two, we expect slicing after compilation to enable the slicer to remove boilerplate code that compilers include at the start and end of each function. Compared with the option $\mathcal{ESW}$, 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.

This paper introduces the first dynamic slicers for WebAssembly, exploring the design space of applying ORBS [9] to WebAssembly and providing insights into the advantages and limitations of three dynamic WebAssembly slicers by empirically comparing them with each other and with $\mathcal{SWS}$ using 57 C programs and a total of 1643 slicing criteria.

The paper makes the following contributions:

- From the design space we identify, implement, and study the first dynamic slicers for WebAssembly.
- We quantitatively and qualitatively compare the slices produced by three promising options.
- We compare the dynamic approaches to the static WebAssembly slicer $\mathcal{SWS}$ 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.

In addition to an understanding of the design space, our study suggests that $\mathcal{SEW}$ sits in a design sweet spot. $\mathcal{SEW}$ provides good performance and small, comprehensible, slices. Our dataset and empirical evaluation scripts are available in a replication package[1].

II. Background

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

A. A Brief Tour of WebAssembly

This tour of WebAssembly is adapted from our previous work with $\mathcal{SWS}$ [57]. WebAssembly programs are bundled as *modules*, which contain one or more function declarations, along with other elements which 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) The SCAM Mug in WebAssembly: To illustrate programming in WebAssembly, we consider the "SCAM Mug" [64] 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.

[1]https://doi.org/10.5281/zenodo.8157269

```
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 }
```

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

```
1  (func (type 0)              ;; int main()
2      (local i32 i32 i32)     ;; declare i, x, c
3      local.get 0             ;; push local i
4      call i32→i32 $p         ;; p(i)
5      if
6        loop
7          local.get 2         ;; push local c
8          call i32→i32 $q     ;; q(c)
9          if
10           call →i32 $f      ;; f()
11           local.set 1       ;; x = result of f()
12           call →i32 $g      ;; g()
13           local.set 2       ;; c = result of g()
14         end
15         local.get 0
16         call i32→i32 $h     ;; h(i)
17         local.set 0         ;; i = result of h(i)
18         local.get 0
19         call i32→i32 $p     ;; p(i)
20         br_if 0             ;; loop if stack top
21       end                   ;; is true
22     end)
```

Fig. 2. The SCAM Mug in WebAssembly.

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 f, 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 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) WebAssembly Validation Requirement: WebAssembly programs have to be *well formed* according to Section 3 of the WebAssembly standard [52]. 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 [57]. 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 `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
```

B. Static Slicing of WebAssembly

The static WebAssembly slicer $\mathcal{SWS}$ [57] has three phases:

1) First, a data-gathering phase computes the dependencies of each instruction in a function. It computes the layout of the stack after each instruction using a stack specification analysis [58], identifies data dependences through use-definition chains, identifies control-dependences using Ferrante's exact algorithm [24], and performs a WebAssembly-specific over-approximation for memory-specific data dependencies [57].
2) Second, the slicing phase identifies the WebAssembly instructions of the closure slice relying on the dependencies identified in the first phase, taking inspiration from traditional approaches to slicing [69], and adding instructions for structured control flow [1].
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*, $\mathcal{SWS}$ 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.

C. Observation-Based Slicing

Observation-Based Slicing (ORBS) [9] is a language-agnostic slicing approach that can in theory be applied to WebAssembly. 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 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 the program as a sequence of lines of text.

The tracked values are captured in V, which is used as an oracle for the expected output.

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 program tentatively removing lines until no more lines can be deleted. Each iteration over the program 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.

III. Study Design and Methodology

This section describes our study's design and methodology, starting with the four slicers studied and then the four research questions considered. The section then describes the subject program studied, their preparation, and the two metrics used to evaluate slicer performance. Finally, it provides some implementation details.

A. The Four Slicers Studied

The design space for instantiating observation-based slicing (ORBS) to slice WebAssembly includes a range of options.

We consider three possibilities: ORBS can be applied to the C source code before its compilation to WebAssembly ($\mathcal{EWS}$ below), or to the corresponding WebAssembly binary after compilation. In the latter case, it can be applied to the whole WebAssembly file ($\mathcal{ESW}$ below) or to only the function that contains the slicing criterion ($\mathcal{SEW}$ below). We do not consider computing the slice of only the C function and then compiling that to WebAssembly because doing so would tell us more about slicing C than WebAssembly.

More precisely, using *slice* to denote the application of ORBS, *extract* to denote the extraction of the function that contains the slicing criterion, and *wasm* to denote the compilation of C code into WebAssembly by clang, the slicers operate as follows:

- $\mathcal{SEW}$– *slice(extract(wasm(*P*)))* – run clang, extract the function that contains the slicing criterion, then slice only that function.
- $\mathcal{ESW}$– *extract(slice(wasm(*P*)))* – run clang, slice the entire file, then extract the function that contains the slicing criterion.
- $\mathcal{EWS}$– *extract(wasm(slice(*P*)))* – slice the C code, compile the sliced C code using clang, then extract the function containing the slicing criterion.
- $\mathcal{SWS}$– *static-slice(wasm(*P*))* – run clang and then apply the static WebAssembly slicer.

We compare the three dynamic slicers, $\mathcal{ESW}$, $\mathcal{SEW}$, and $\mathcal{EWS}$ to each other and with the static slicer, $\mathcal{SWS}$. It is worth noting that in contrast to $\mathcal{EWS}$, the slicers $\mathcal{SEW}$, $\mathcal{ESW}$, and $\mathcal{SWS}$ have the advantage that they do not require access to the source code and thus can slice deployed binaries.

A key concern when applying ORBS to WebAssembly is that to be accepted a speculative deletion needs to not only leave the desired semantics preserved but also must satisfy the stack validation requirement. The latter might prove too restrictive because our dynamic slicers incorporate WebAssembly's validation requirement as part of the slicing process.

We consider the dynamic slicers *inter-procedural slicers* because each takes into account the calling context provided by the code external to the function that contains the slicing criteria. In contrast, an *intra-procedural* slicer would only consider the code of a single function while ignoring the code's context. When computing an inter-procedural slice using ORBS it is possible to focus the slicer on only removing code associated with a given, function, class, file, etc., but the resulting slice is still considered inter-procedural.

B. Research Questions

We evaluate our slicers using the following 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 has not been tried before and might prove prohibitively expensive. We investigate the time taken to slice the original C code ($\mathcal{EWS}$), the entire WebAssembly file ($\mathcal{ESW}$), and finally only the function that contains the slicing criterion ($\mathcal{SEW}$). In doing so, 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.

RQ2: Which of $\mathcal{EWS}$, $\mathcal{ESW}$, and $\mathcal{SEW}$ best balances speed and precision?

Because the source provides a higher-level representation of the code, we expect $\mathcal{EWS}$ to be the fastest of the three. However, it is unclear whether the precision will suffer because $\mathcal{EWS}$ is slicing at a higher level of abstraction. At the other end of the spectrum, $\mathcal{ESW}$ is expected to be the most precise, but also the slowest. The big question is does $\mathcal{SEW}$ represent a sweet spot.

RQ3: What qualitative differences are there between the slices produced by the three 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, slicing the entire compiled file may enable removing code that slicing only one function cannot because the code is required by an unnecessary computation found in another function.

RQ4: For a given binary, 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.

C. Subjects

We consider 57 C programs as the subjects of our study. These subjects are listed in Table I 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 [68], [64], [18], [26], programs from the Mälardalen WCET research group [43], programs from the *Benchmarks Game* [25], and the multi-file system `bc`. These subjects have all been used in previous slicing studies [5], [57].

We limit the complexity of the programs considered to facilitate comparison. This does not mean that our approach cannot handle more complex situations, 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.

TABLE I

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

D. 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.

The slices taken with respect to 60 of the criteria are impacted by the execution environment. For example, printing `argv[0]` of scam_argv_18 (the slice of scam taken with respect to `argv` at Line 18) outputs `scam.c.wasm` when run by wasmer but `./scam` when run as a compiled binary. The other causes include the use of a different implementation 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.

E. 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.

a) Time: We measure the time taken to compute each slice using the "real" or wall-clock time. We also gathered the CPU ("user") time, which mirrored the real-time so strongly that for clarity of presentation, we report only the real-time.

b) 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 WebAssembly code we first omit the function's declaration, its declaration of local variables, and all **`end`** lines. For compiled WebAssembly code this is the same as non-blank lines because the code is devoid of comments, except that we exclude **`end`** which marks the end of the block but is not executable.

F. 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 a single function of a WebAssembly file we split the file into three parts: prog.wat.prefix, prog.wat.function, and prog.wat.postfix. ORBS is then applied to the lines of prog.wat.function *only*. To compile and execute the program, the three parts, including the reduced prog.wat.function, are concatenated together. One motivation here is that the WebAssembly compiler includes a lot of library code (e.g., code

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

for all functions required from libc). Slicing this code impacts ORBS' running time, which is proportional to the number of lines considered. As discussed in RQ1, slicing a single function reduces the amount of work by almost two orders of magnitude.

When configuring ORBS, prior empirical work has found that the window-size value $\delta = 4$ does a good job at balancing slice size and slice computation time [9], [34], [55]. Future work will consider if this value is still optimal at the WebAssembly level.

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.

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.

IV. Evaluation

This section empirically investigates each of our four 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].

A. 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. As expected $\mathcal{SWS}$ is consistently the fastest with an average slicing time of 0.67 seconds. At the other end of the spectrum, $\mathcal{ESW}$ is the slowest slicer because it needs to consider the deletion of lines from the *entire* WebAssembly file. With a mean slicing time of 95 minutes, $\mathcal{ESW}$'s times do not fit within either of our definitions of "practical". Our average C program has 331 SLoC, while our average compiled WASM program has 12 108 SLoC. In other words, ORBS must consider 36.6 more possible deletions, resulting in an average slowdown factor of 7.0 when comparing $\mathcal{ESW}$ to $\mathcal{EWS}$.

$\mathcal{SEW}$ and $\mathcal{EWS}$ are close in terms of time. $\mathcal{SEW}$'s slicing times range from 14 seconds to 130 minutes with a mean of 11.1 minutes, while $\mathcal{EWS}$'s slicing times range from 14 seconds to 64 minutes with a mean of 13.5 minutes. That $\mathcal{SEW}$ and $\mathcal{EWS}$'s times are so close is surprising, given that $\mathcal{SEW}$ considers only the WebAssembly instructions of a single function, while $\mathcal{EWS}$ considers the C source code of the entire program. Finally, we note a preference for $\mathcal{SEW}$ because its slicing times are more stable, with a standard deviation of $\pm$16.9 minutes compared to $\pm$78.4 minutes and $\pm$21.6 minutes for $\mathcal{ESW}$ and $\mathcal{EWS}$, respectively.

[3]https://doi.org/10.5281/zenodo.8157269

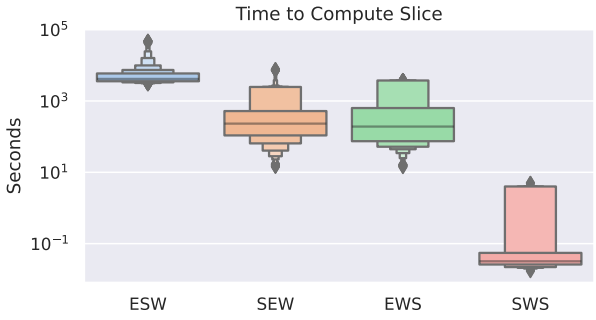

Fig. 3. Slice times for each slicer.

Looking at our "cup of coffee" practicality threshold, $\mathcal{ESW}$ fails to produce a single slice within the allotted time. $\mathcal{SEW}$ and $\mathcal{EWS}$ are more successful, producing respectively 1341 (82%) and 1362 (83%) of the 1643 slices within the time limit.

> **RQ1**: Applying ORBS to an entire WebAssembly file ($\mathcal{ESW}$) is not practical. Focusing ORBS on the function containing the slicing criterion ($\mathcal{SEW}$) reduces this time to an average of 11.1 minutes, which is similar to the average time taken for ORBS to slice the C programs ($\mathcal{EWS}$). Therefore $\mathcal{SEW}$ and $\mathcal{EWS}$ both fit within our definition of practical.

B. RQ2: Which of $\mathcal{EWS}$, $\mathcal{ESW}$, and $\mathcal{SEW}$ Best Balances Speed and Precision?

Building on RQ1, RQ2 factors in slice size. We first consider the slices of $\mathcal{ESW}$ and $\mathcal{SEW}$. By focusing ORBS deletion on the lines within a given function, $\mathcal{SEW}$ is notably faster than $\mathcal{ESW}$. 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 can enable the removal of code within the function. Figure 4 shows a source-level illustration of this phenomenon.

Figure 5 depicts the distribution of slice sizes relative to the original size of the function being sliced. Table II provides 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 $\mathcal{ESW}$ and $\mathcal{SEW}$ are nearly identical. Numerically, the median slice size for $\mathcal{ESW}$ is 13.05% of the original code, which is essentially equivalent to $\mathcal{SEW}$'s median of 13.17%. Surprisingly, in contract to our expectation, 438 slices are larger for $\mathcal{ESW}$ than $\mathcal{SEW}$. The

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

Fig. 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`.

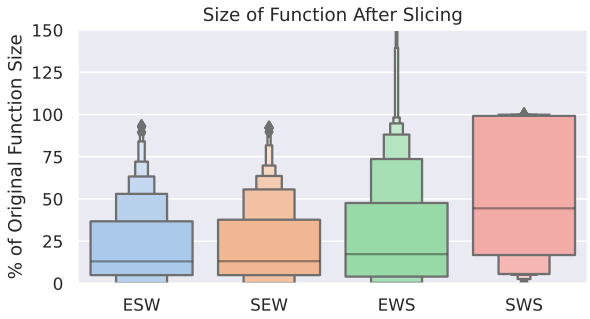

Fig. 5. Size in SLoC of the function being sliced as a percentage of the original function size. Values higher than 100% indicate that the slice is larger than the original function. A typical cause for this is compiler loop unrolling.

cause of this unexpected result is investigated as part of RQ3's qualitative investigation.

TABLE II
SLICE SIZE STATISTICS MEASURED IN TERMS OF SLOC.

Slicer	Median	Mean	Min	Max	Std. Dev.
$\mathcal{ESW}$	13.05%	22.25%	0.12%	92.93%	± 22.39
$\mathcal{SEW}$	13.17%	22.82%	0.19%	92.12%	± 22.84
$\mathcal{EWS}$	17.34%	29.57%	0.21%	267.00%	± 33.14
$\mathcal{SWS}$	44.47%	52.15%	0.19%	99.91%	± 36.99

We expect $\mathcal{EWS}$ to produce larger slices than $\mathcal{ESW}$ and $\mathcal{SEW}$ when, for example, the latter two remove parts of the standard function entry and exit boilerplate code. Empirically, the $\mathcal{EWS}$ slice is larger than the corresponding $\mathcal{ESW}$ slice 79% of the time and larger than the corresponding $\mathcal{SEW}$ slice the same 79% of the time.

There are more interesting causes. For example, 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 $\mathcal{EWS}$ slices is more than 100% of the sliced function. The most extreme case is a slice of adpcm that is almost three times larger than the original function. We investigate this effect as part of RQ3's qualitative look at the slices.

Finally, we consider the performance of the static $\mathcal{SWS}$ slicer. By its very nature $\mathcal{SWS}$ produces larger slices because of the need to make safe static approximations. Compared to $\mathcal{ESW}$ and $\mathcal{SEW}$, the average static slice size shown in Table II is 52.15%, or approximately 2.2 times as large as its dynamic counterpart.

RQ2: $\mathcal{EWS}$ produces notably larger slices than the other dynamic approaches. While at present only $\mathcal{SWS}$ is practical as a real-time tool, $\mathcal{SEW}$ satisfies our looser practicality requirement and produces slices that are less than one third the size of those produced by $\mathcal{SWS}$. Thus we find that $\mathcal{SEW}$ best balances speed and precision.

C. 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 the largest differences, both positive and negative, and report representative examples. We first compare $\mathcal{ESW}$ and $\mathcal{SEW}$ and then $\mathcal{EWS}$ and $\mathcal{SEW}$. We omit the direct comparison between $\mathcal{ESW}$ and $\mathcal{EWS}$ because it is similar to that of $\mathcal{EWS}$ and $\mathcal{SEW}$.

1) Comparison of $\mathcal{ESW}$ and $\mathcal{SEW}$: $\mathcal{ESW}$ produces a smaller slice in 467 cases, the same size in 738 cases, and surprisingly $\mathcal{SEW}$ produces the smaller slice in 438 cases. None of the differences are large, with size($\mathcal{ESW}$) - size($\mathcal{SEW}$) ranging from -126 to +195 instructions. The median difference is 0 and 74% of the slices differ by five or fewer instructions. We consider four representative examples, two from each end of the spectrum.

The first example are the slices taken with respect to sumprod_i_10 and illustrates the situation where $\mathcal{ESW}$ 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 SEW 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 $\mathcal{ESW}$ slice the argument counting code gets sliced out leaving `argc` with the value zero. Thus in the $\mathcal{EWS}$ slice, there is no need to initialize variable `i`. Consequently, Lines 81 and 82 are omitted from the $\mathcal{EWS}$ slice.

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

```
i32.const 61
i32.mul
i32.const 1
i32.add
global.set $__stack_pointer
```

Here the $\mathcal{SEW}$ slice has co-opted the first four 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 $\mathcal{ESW}$ 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 an $\mathcal{SEW}$ 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 $\mathcal{SEW}$ produced the preferred slice.

Turning to criteria for which the $\mathcal{SEW}$ slice is smaller, we first consider the slice taken with respect to adpcm_inc_170, where the $\mathcal{ESW}$ slice is three instructions longer than the $\mathcal{SEW}$ slice. The three instructions initialize local 0 with a copy of the stack pointer.

```
global.get $__stack_pointer
local.tee 0
global.set $__stack_pointer
```

The reason that the $\mathcal{ESW}$ slice can't remove these instructions is twofold. First, the printing functions of the $\mathcal{ESW}$ slice have become specialized to print the particular location used in the code as the slicing criterion. Copying the stack pointer to local 0 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.

As a final example, the $\mathcal{ESW}$ slice taken with respect to expint_fact_57 includes the following eight instructions not present in the $\mathcal{SEW}$ slice.

```
loop   ;; label = @2     ;; L0  0
  local.get 3            ;; L1 +1
  i32.const 1            ;; L2 +1
  i32.add                ;; L3 -1
  local.tee 3            ;; L4  0
  i32.const 102          ;; L5 +1
  i32.ne                 ;; L6 -1
  br_if 0 (;@2;)         ;; 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 or less. One might say that the $\mathcal{ESW}$ slicer has "painted itself into a corner." $\mathcal{SEW}$ removes code in a different order, which enables it to avoid painting itself into the same corner. Thus, it omits the loop.

Summarizing the four examples from a qualitative perspective, $\mathcal{SEW}$ seems the better approach. Despite it producing slightly larger slices, the differences are never large. On the plus side, its slices are often more easily understood.

2) Comparison of $\mathcal{EWS}$ and $\mathcal{SEW}$: We next compare the slices of $\mathcal{SEW}$ and $\mathcal{EWS}$. Comparing sizes, the $\mathcal{SEW}$ slice is smaller for 1294 slices, the same for only 13 slices, and larger for 336 slices. This is in line with the expectation that $\mathcal{EWS}$ produces the largest of the dynamic slices and agrees with the quantitative data of RQ2.

Looking at examples where the $\mathcal{EWS}$ slice is smaller, the largest difference is for the slices taken with respect to cover_c_484 for which the $\mathcal{SEW}$ slice includes 610 instructions while the $\mathcal{EWS}$ slice is only 21. Looking at the resulting WebAssembly code, slicing simplifies the control structure of the C code to the point where additional compiler optimizations become viable. Specifically constant propagation, which obviates the need for any compiled code in the $\mathcal{EWS}$ slice.

As a final illuminating example the $\mathcal{EWS}$ slice taken with respect to lms_x_160 includes 51 instructions while the $\mathcal{SEW}$ slice 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 a window size of four and the stack validation requirements, $\mathcal{SEW}$ 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 four. Slicing at the C level this call is a single line, and thus easily removed.

We next turn to the dominant case where $\mathcal{SEW}$ produces the smaller slices. The most extreme case is for edn_j_140 where the $\mathcal{SEW}$ slice includes 55 instruction while the $\mathcal{EWS}$ slice 585. This is an example where the comiler 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. For example, unrolling leads matmult_Index_56's $\mathcal{EWS}$ slice to include 208 instructions while the $\mathcal{SEW}$ slice includes only 38.

As a second example, an interesting pattern occurs in the $\mathcal{EWS}$ slice takeen 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. As a result, the $\mathcal{EWS}$ slice contains instructions that are attributable to the statements of the other function, making it larger than the $\mathcal{SEW}$ slice.

Unlike the comparison of $\mathcal{ESW}$ and $\mathcal{SEW}$, the qualitative comparison of $\mathcal{EWS}$ and $\mathcal{SEW}$ is less one sided. In $\mathcal{EWS}$'s favor, slicing the higher-level source enables $\mathcal{EWS}$ to remove code that is more difficult to remove at the finer level granularity of WebAssembly code. Furthermore, in some cases, slicing before compilation also enables additional compiler optimizations. However, at times enabled optimizations are not benificial, at least in terms of slice size. An example is the slices taken with respect to edn_j_140. Some of the $\mathcal{EWS}$ slices also included merged functions, which make the slice harder to understand, and some, such as the slices taken with respect to lms_x_160, includes merged loops. This suggests future work with hybrid techniques such as first slicing the C code and then the compiled WebAssembly code.

RQ3: Three patterns emerge from our qualitative examination of the slices. First, as with RQ2, $\mathcal{SEW}$ is often the preferred approach. Second, this preference is, in part, because slicing at a level of abstraction different from that of the final slice ($\mathcal{EWS}$) and slicing multiple functions ($\mathcal{ESW}$) leads to structural changes that make it hard to tie the slice back to the original source. WebAssembly level and slicing of a single function ($\mathcal{SEW}$), reduces the number of structural changes in the code making it easier to tie the slice back to the original source.

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

RQ1 tells us that $\mathcal{SWS}$ is on average three orders of magnitude faster than the faster dynamic slicers, while RQ2 tells us that its slices are on average over three times as large as the better dynamic slicers. These differences are in line with related work [12]. This section qualitatively investigates the differences in the slices produced by $\mathcal{SWS}$.

First, when checking for agreement in executability of the slices produced by each slicer, we noticed that our choice of slicing criterion resulted in common disagreements between the static slice and the dynamic slices. Specifically, the slicing criterion used in $\mathcal{SWS}$ is the second argument to `printf`, which is called as `printf("ORBS: %x\n", v)`. However, when inspecting the WebAssembly binary, we notice that the compiler stores `v` in the linear memory, and calls `printf` with a pointer to the memory location of `v`. Hence, our choice of slicing criterion renders the static slices inaccurate: while the slice is correct for the task asked of $\mathcal{SWS}$ (it preserves the value of the second argument to `printf`), it does not preserve the observed behavior (printing the value pointed by the second argument to `printf`). We reproduced all of the static slices after changing the slicing criterion to be the `printf` **`call`** instruction. This results in larger slices, as $\mathcal{SWS}$ needs to resort to more over-approximations when a function call is part of the slice. However, it does not invalidate the conclusions of the past research questions: $\mathcal{SWS}$ remains as fast with this new criterion and still produces larger slices than the dynamic slicers. However, in the following comparison we focus on the slices produced using the more accurate slicing criterion.

Comparing $\mathcal{SWS}$ to $\mathcal{EWS}$, 92% of the static slices are larger while comparing it to $\mathcal{SEW}$, 98% are larger. With the previous choice for slicing criterion, these numbers were respectively 59% and 72%.

A common pattern involves the function prologue and epilogue. Figure 6 demonstrates this for the slice taken with respect to `nbody1_e_51`. The $\mathcal{SWS}$ slice on the right includes the function prologue introduced by the C compiler. The $\mathcal{SEW}$ and $\mathcal{EWS}$ slices on the left omit this code.

$\mathcal{SEW}$ / $\mathcal{EWS}$	$\mathcal{SWS}$
`;; prologue:`	`;; prologue: move`
`;;  empty`	`;;  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`	`i32.const 1030`
`local.get 2`	`local.get 2`
`;; slicing criterion`	`;; slicing criterion`
`call $printf`	`call $printf`
`drop`	`drop`
`;; epilogue:`	`;; epilogue: restore`
`;;  empty`	`;;  stack pointer`
	`local.get 2`
	`i32.const 16`
	`i32.add`
	`global.set 0`
`;; return value`	`;; return value`
`local.get 3`	
`f64.const 0`	

Fig. 6. $\mathcal{SEW}$ and $\mathcal{EWS}$ remove the function prologue and epilogue, while $\mathcal{SWS}$ preserves it.

The difference is impacted by the conservative over-approximations made by $\mathcal{SWS}$ when performing data-flow analysis. In this case, it considers all globals as dependencies. On the other hand $\mathcal{SEW}$ and $\mathcal{EWS}$ can remove those parts of the prologue and epilogue that do not impact the execution.

Notice also that the $\mathcal{SWS}$ slice preserves the writing of 0 to local 2. It turns out that during execution, local 2 is always 0, so the write has no effect. A dynamic slicer can take advantage of this fact to remove the code.

Manual inspection of the $\mathcal{ESW}$ 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 $\mathcal{ESW}$ 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 dependencies precisely, whereas a dynamic slicer, by its very nature, can precisely characterize them.

```
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 $\mathcal{SWS}$ is that memory dependencies can't be tracked precisely. Hence, as soon as one memory-related instruction (`load`, `store`) is part of the slice, all preceding memory-related instructions are included in the slice. This is not the case for the dynamic slices, which only include the required memory-related instructions. For example, comparing the $\mathcal{EWS}$ and $\mathcal{SWS}$ slice taken with respect to `nsichneu_b_2188`, we observed that both slices include a necessary `store` instruction but that the $\mathcal{SWS}$ slice includes 9224 instructions in total while $\mathcal{ESW}$ includes only 22 instructions. This is due to $\mathcal{SWS}$'s static over-approximations. 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.

Finally, we investigated the executability of the slices produced by each slicer, taking the `call` instruction to `printf` as the slicing criterion for $\mathcal{SWS}$. In all but one case, all slicers agree. In the remaining case, for `fasta5_i_89`, the static slice does not execute the slicing criterion. This indicates that $\mathcal{SWS}$ has broken the control structure of the program. Such issues with static slicers are common [6]. One useful outcome of our comparison to dynamic slicers is that we can better characterize the prevalence of this problem.

RQ4: Dynamic approaches can perform more aggressive removals than $\mathcal{SWS}$. A classic example is the removal of the function prologue and epilogue. Moreover, dynamic approaches can uncover hidden memory dependencies that the static slicer misses. Looking forward, the shortcomings identified by our experiments can be used to inform future work on static slicing techniques.

E. Discussion

Our experiments show that overall $\mathcal{SEW}$ is the preferred approach. Along with $\mathcal{EWS}$, it satisfies our "cup of coffee" definition of practical. $\mathcal{EWS}$ however produces larger slices than $\mathcal{SEW}$ (and $\mathcal{ESW}$). Furthermore, our qualitative comparison finds the $\mathcal{SEW}$ slices preferable. There are however several issues worthy of consideration. The rest of this section considers the four most important.

a) Environment impact: 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 [7].

b) Window size: We identified multiple cases where further reductions could be applied if a larger window size was used. The window size influences how many lines can be removed in a single deletion. As assembly languages contain less semantic information per line than higher-level source code, some common patterns could not be removed. On the other hand, a larger window size might not be practical as it increases the slicing time [9], [34].

c) Validation requirement: Key to slicing WebAssembly programs is maintaining the validity of the slice. 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* [57], which might be used to reconstruct a valid program from an invalid slice. Improving the dynamic slicer with such a phase could potentially enable the slicer to produce smaller slices.

d) Under-approximations of the static slice: Our experiments have revealed that there are cases where the $\mathcal{SWS}$ slices clearly under-approximate the true dependence. Further comparisons between $\mathcal{SWS}$ and the dynamic slicers should help us understand this dependence under-approximation and consequently lead to improvements in the static slicer.

F. Threats to Validity

We identify threats to validity according to the classification of Wohlin et al. [70]. We instrument each subject of our evaluation for each use of a scalar variable in the program. As a threat to internal validity, 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 $\mathcal{EWS}$ 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.

V. Related Work

a) WebAssembly: There has been interest from the research community in WebAssembly on aspects such as security [47], [40], [27], [60], [61], [46], [15], extensions to the language [20], [50], [3], tooling [51], program analysis [41], [67], [58], [59], [45], [42], and optimizations [16]. WebAssembly being an assembly language, has been compared to other assembly languages such as x86 [28], [60], [61]. We compare our approach to $\mathcal{SWS}$, the only existing slicer for WebAssembly [57]. To date, no other slicers for WebAssembly exist. Relying on existing dynamic analysis frameworks [41] to implement a dynamic slicer remains to be investigated.

b) Binary Slicing: The ability to slice at the binary level is required for tasks involving binaries for which the original source code is not available. This is the case for reengineering, program comprehension, or analyses for security. 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.

Slicing approaches for binary executables, including stack-based assembly languages such as JVM Bytecode, have focused on static approaches [17], [36], [57], [63], [21], [71], [44]. Kiss et al. [37] rely on dynamic information during static slicing to approximate possible target of function calls. Conditioned slicing [65] is another approach that combines dynamic and static slicing, by first computing a dynamic slice before augmenting it with information from a static slice.

c) 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 [9], which works at the line level [10], [11]. QSES is a variant of ORBS that protects all lines of a static slice during dynamic slicing, which has been applied to C programs [56], [55], 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.

VI. Conclusion

We introduce three dynamic slicing approaches for WebAssembly binaries, namely $\mathcal{SEW}$, $\mathcal{EWS}$, and $\mathcal{ESW}$. We compared the three and the static WebAssembly slicer $\mathcal{SWS}$ using four research questions.

Our evaluation shows that $\mathcal{ESW}$ requires much more time than the other approaches, while not reducing slice size as much as expected. $\mathcal{EWS}$ 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. In summary, our evaluation finds that $\mathcal{SEW}$ yields the best trade-off in terms of running time, slice size, and inspectability of the resulting slices.

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. Another interesting avenue is to combine multiple slicing approaches. For example, one could first run the static slicer to quickly remove a portion of the code, then run $\mathcal{SEW}$ to dynamically remove the rest. Such static/dynamic hybrids have proven successful in the past at other levels of abstraction [56]. Combining multiple dynamic slicers is also interesting, for example first slicing the original source, then compiling the slice to WebAssembly, before finally slicing the compiled code.

References

[1] Agrawal, H.: On slicing programs with jump statements. In: Sarkar, V., Ryder, B.G., Soffa, M.L. (eds.) Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI). pp. 302–312. ACM (1994). https://doi.org/10.1145/178243.178456

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

[3] Bastys, I., Algehed, M., Sjösten, A., Sabelfeld, A.: Secwasm: Information flow control for webassembly. In: Singh, G., Urban, C. (eds.) Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13790, pp. 74–103. Springer (2022). https://doi.org/10.1007/978-3-031-22308-2_5

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

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

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

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

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

[9] Binkley, D.W., Gold, N., Harman, M., Islam, S.S., Krinke, J., Yoo, S.: ORBS: language-independent program slicing. In: Cheung, S., Orso, A., Storey, M.D. (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22). pp. 109–120. ACM (2014). https://doi.org/10.1145/2635868.2635893

[10] Binkley, D.W., Gold, N., Islam, S.S., Krinke, J., Yoo, S.: Tree-oriented vs. line-oriented observation-based slicing. In: 17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017. pp. 21–30. IEEE Computer Society (2017). https://doi.org/10.1109/SCAM.2017.11

[11] Binkley, D.W., Gold, N., Islam, S.S., Krinke, J., Yoo, S.: A comparison of tree- and line-oriented observational slicing. Empir. Softw. Eng. **24**(5), 3077–3113 (2019). https://doi.org/10.1007/s10664-018-9675-9

[12] Binkley, D.W., Harman, M.: A survey of empirical results on program slicing. Adv. Comput. **62**, 105–178 (2004). https://doi.org/10.1016/S0065-2458(03)62003-6

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

[14] Binkley, D.W., Harman, M.: A survey of empirical results on program slicing. Advances in Computers **62**, 105–178 (2004)

[15] Brito, T., Lopes, P., Santos, N., Santos, J.F.: Wasmati: An efficient static vulnerability scanner for webassembly. Comput. Secur. **118**, 102745 (2022). https://doi.org/10.1016/j.cose.2022.102745

[16] Cabrera-Arteaga, J., Donde, S., Gu, J., Floros, O., Satabin, L., Baudry, B., Monperrus, M.: Superoptimization of WebAssembly bytecode. In: Aguiar, A., Chiba, S., Boix, E.G. (eds.) Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming. pp. 36–40. ACM (2020). https://doi.org/10.1145/3397537.3397567

[17] Cifuentes, C., Fraboulet, A.: Intraprocedural static slicing of binary executables. In: 1997 International Conference on Software Maintenance (ICSM '97). p. 188. IEEE Computer Society (1997). https://doi.org/10.1109/ICSM.1997.624245

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

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

[20] Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A., Stefan, D.: 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. pp. 4:1–4:8 (2019)

[21] D'Ursi, A.C., Cavallaro, L., Monga, M.: On bytecode slicing and aspectj interferences. In: Harrison, W. (ed.) Proceedings of the 6th Workshop on Foundations of Aspect-Oriented Languages, FOAL 2007. ACM International Conference Proceeding Series, vol. 268, pp. 35–43. ACM (2007). https://doi.org/10.1145/1233833.1233839

[22] Ellul, J., Pace, G.J.: 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). pp. 1–4. IEEE (2018)

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

[24] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987). https://doi.org/10.1145/24039.24041

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

[26] Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Trans. Software Eng. **17**(8), 751–761 (1991). https://doi.org/10.1109/32.83912

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

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

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

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

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

[32] Hilbig, A., Lehmann, D., Pradel, M.: An empirical study of real-world WebAssembly binaries: Security, languages, use cases. In: Leskovec, J., Grobelnik, M., Najork, M., Tang, J., Zia, L. (eds.) WWW '21: The Web Conference 2021. pp. 2696–2708. ACM / IW3C2 (2021). https://doi.org/10.1145/3442381.3450138

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

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

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

[36] Kiss, Á., Jász, J., Gyimóthy, T.: Using dynamic information in the interprocedural static slicing of binary executables. Softw. Qual. J. **13**(3), 227–245 (2005). https://doi.org/10.1007/s11219-005-1751-x

[37] Kiss, Á., Jász, J., Gyimóthy, T.: Using dynamic information in the interprocedural static slicing of binary executables. Softw. Qual. J. **13**(3), 227–245 (2005). https://doi.org/10.1007/s11219-005-1751-x

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

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

[40] Lehmann, D., Kinder, J., Pradel, M.: Everything old is new again: Binary security of WebAssembly. In: 29th USENIX Security Symposium, USENIX Security 2020 (2020)

[41] Lehmann, D., Pradel, M.: 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. pp. 1045–1058 (2019)

[42] Lehmann, D., Pradel, M.: Finding the dwarf: recovering precise types from webassembly binaries. In: Jhala, R., Dillig, I. (eds.) PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. pp. 410–425. ACM (2022). https://doi.org/10.1145/3519939.3523449

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

[44] Mangean, A., Béchennec, J., Briday, M., Faucou, S.: BEST: a binary executable slicing tool. In: Schoeberl, M. (ed.) 16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016. OASICS, vol. 55, pp. 7:1–7:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/OASIcs.WCET.2016.7

[45] Marques, F., Santos, J.F., Santos, N., Adão, P.: Concolic execution for webassembly. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 11:1–11:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.11

[46] Mazaheri, M.E., Sarmadi, S.B., Ardakani, F.T.: A study of timing side-channel attacks and countermeasures on javascript and webassembly. ISC Int. J. Inf. Secur. **14**(1), 27–46 (2022). https://doi.org/10.22042/isecure.2021.263565.599

[47] Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: Twine: An embedded trusted runtime for WebAssembly. In: 37th IEEE International Conference on Data Engineering, ICDE 2021. pp. 205–216. IEEE (2021). https://doi.org/10.1109/ICDE51399.2021.00025

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

[49] Philips, L., De Roover, C., Van Cutsem, T., De Meuter, W.: 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)

[50] Pinckney, D., Guha, A., Brun, Y.: Wasm/k: delimited continuations for WebAssembly. In: Flat, M. (ed.) DLS 2020: Proceedings of the 16th

ACM SIGPLAN International Symposium on Dynamic Languages. pp. 16–28. ACM (2020). https://doi.org/10.1145/3426422.3426978
[51] Romano, A., Wang, W.: WasmView: visual testing for WebAssembly applications. In: Rothermel, G., Bae, D. (eds.) ICSE '20: 42nd International Conference on Software Engineering, Companion Volume. pp. 13–16. ACM (2020). https://doi.org/10.1145/3377812.3382155
[52] Rossberg, A.: WebAssembly Core Specification. Tech. rep., W3C (2019), https://www.w3.org/TR/wasm-core-1/
[53] Salimi, S., Ebrahimzadeh, M., Kharrazi, M.: Improving real-world vulnerability characterization with vulnerable slices. In: 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE). pp. 11–20 (2020)
[54] Silva, J.: A vocabulary of program slicing-based techniques. ACM Comput. Surv. **44**(3) (Jun 2012). https://doi.org/10.1145/2187671.2187674
[55] Stiévenart, Q., Binkley, D., De Roover, C.: An empirical evaluation of quasi-static executable slices. Journal of Systems and Software **200**, 111666 (2023)
[56] Stiévenart, Q., Binkley, D.W., De Roover, C.: QSES: quasi-static executable slices. In: 21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021. pp. 209–213. IEEE (2021). https://doi.org/10.1109/SCAM52516.2021.00033
[57] Stiévenart, Q., Binkley, D.W., De Roover, C.: 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. pp. 2031–2042. ACM (2022). https://doi.org/10.1145/3510003.3510070
[58] Stiévenart, Q., De Roover, C.: Compositional information flow analysis for WebAssembly programs. In: 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020. pp. 13–24. IEEE (2020). https://doi.org/10.1109/SCAM51674.2020.00007
[59] Stiévenart, Q., De Roover, C.: Wassail: a WebAssembly static analysis library. In: Fifth International Workshop on Programming Technology for the Future Web (2021)
[60] Stiévenart, Q., De Roover, C., Ghafari, M.: The security risk of lacking compiler protection in WebAssembly. In: 21st IEEE International Conference on Software Quality, Reliability, and Security. IEEE (2021)
[61] Stiévenart, Q., De Roover, C., Ghafari, M.: Security risks of porting c programs to WebAssembly. In: The 37th ACM/SIGAPP Symposium On Applied Computing. ACM (2022)
[62] Tonella, P.: Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Transactions on Software Engineering **29**(6) (2003)
[63] Umemori, F., Konda, K., Yokomori, R., Inoue, K.: Design and implementation of bytecode-based Java slicing system. In: 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003). pp. 108–117. IEEE Computer Society (2003). https://doi.org/10.1109/SCAM.2003.1238037
[64] Ward, M.P.: Slicing the SCAM mug: A case study in semantic slicing. In: 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003). pp. 88–97. IEEE Computer Society (2003). https://doi.org/10.1109/SCAM.2003.1238035
[65] Ward, M.P., Zedan, H.: Combining dynamic and static slicing for analysing assembler. Sci. Comput. Program. **75**(3), 134–175 (2010)
[66] Wasmer: The leading WebAssembly runtime supporting wasi and emscripten. https://github.com/wasmerio/wasmer
[67] Watt, C., Maksimovic, P., Krishnaswami, N.R., Gardner, P.: A program logic for first-order encapsulated WebAssembly. In: 33rd European Conference on Object-Oriented Programming, ECOOP 2019. pp. 9:1–9:30 (2019)
[68] Weiser, M.: Program slicing. In: 5th International Conference on Software Engineering. pp. 439–449 (1981)
[69] Weiser, M.: Program slicing. In: Jeffrey, S., Stucki, L.G. (eds.) Proceedings of the 5th International Conference on Software Engineering. pp. 439–449. IEEE Computer Society (1981), http://dl.acm.org/citation.cfm?id=802557
[70] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in Software Engineering. Springer (2012). https://doi.org/10.1007/978-3-642-29044-2
[71] Zhao, J.: Dependence analysis of Java bytecode. In: 24th International Computer Software and Applications Conference (COMPSAC 2000). pp. 486–491. IEEE Computer Society (2000). https://doi.org/10.1109/CMPSAC.2000.884771