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 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 (W), extracting the function containing the slicing criterion (E), and removing (slicing out) irrelevant code (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 $E(W(S(P)))$, which we denote $EWS$ for short.

As a second example, $S(E(W(P)))$, which we denote $SWE$, 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 $ESW$, 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 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 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 SEW sits in a design sweet spot. SEW provides good performance and small, comprehensible, slices. Our dataset and empirical evaluation scripts are available in a replication package.

II. BACKGROUND

This section provides background on WebAssembly and 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 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 features 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 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 loops 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 \( h(i) \) 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 \( \text{br}_{\text{if}} \) 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 \( \text{i32.const} \ 0 \) instruction has type “ → 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 \( \text{local.get} \ 0; \text{local.get} \ 1; \text{i32.const} \ 1; \text{i32.add} \) has type “ → 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 “ → 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.

```wasm
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 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, 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 \( \nu \), a program location \( l \), a set of inputs \( \mathcal{I} \), and a maximum window size \( \delta \). The slice computed by ORBS compiles and preserves the semantics of \( \nu \) 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 \( \nu \) 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 \( \delta \) 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 (ESWS below), or to the corresponding WebAssembly binary after compilation. In the latter case, it can be applied to the whole WebAssembly file (ESW below) or to only the function that contains the slicing criterion (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:

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

We compare the three dynamic slicers, ESW, SEW, and ESWS to each other and with the static slicer, SWWS. It is worth noting that in contrast to ESWS, the slicers SEW, ESW, and SWWS 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 (ESWS), the entire WebAssembly file (ESW), and finally only the function that contains the slicing criterion (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 ESWS, ESW, and SEW best balances speed and precision?

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

<table>
<thead>
<tr>
<th>program</th>
<th>C SLoC</th>
<th>WASM SLoC</th>
<th>program</th>
<th>C SLoC</th>
<th>WASM SLoC</th>
<th>program</th>
<th>C SLoC</th>
<th>WASM SLoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>585</td>
<td>10 275</td>
<td>fasta8</td>
<td>150</td>
<td>14 903</td>
<td>nbbody6</td>
<td>93</td>
<td>14 754</td>
</tr>
<tr>
<td>bc_bc</td>
<td>8007</td>
<td>40 499</td>
<td>fasta9</td>
<td>161</td>
<td>15 007</td>
<td>nbbody7</td>
<td>137</td>
<td>15 005</td>
</tr>
<tr>
<td>binary-trees1</td>
<td>91</td>
<td>14 663</td>
<td>fdct</td>
<td>138</td>
<td>8 531</td>
<td>ndes</td>
<td>196</td>
<td>9 009</td>
</tr>
<tr>
<td>bs</td>
<td>46</td>
<td>8 130</td>
<td>fti1</td>
<td>128</td>
<td>11 004</td>
<td>ns</td>
<td>30</td>
<td>8 387</td>
</tr>
<tr>
<td>bsort100</td>
<td>61</td>
<td>8 221</td>
<td>fibcall</td>
<td>27</td>
<td>8 135</td>
<td>nischneu</td>
<td>2989</td>
<td>17 277</td>
</tr>
<tr>
<td>cnt</td>
<td>76</td>
<td>8 511</td>
<td>fir</td>
<td>54</td>
<td>8 277</td>
<td>prime</td>
<td>51</td>
<td>8 176</td>
</tr>
<tr>
<td>compress</td>
<td>357</td>
<td>9 000</td>
<td>insertsort</td>
<td>33</td>
<td>8 141</td>
<td>qsort-exam</td>
<td>124</td>
<td>8 407</td>
</tr>
<tr>
<td>cover</td>
<td>625</td>
<td>8 847</td>
<td>janne_complex</td>
<td>38</td>
<td>8 131</td>
<td>qurt</td>
<td>120</td>
<td>8 285</td>
</tr>
<tr>
<td>crc</td>
<td>94</td>
<td>8 409</td>
<td>fjdcHint</td>
<td>119</td>
<td>8 506</td>
<td>reverse-complement5</td>
<td>83</td>
<td>14 917</td>
</tr>
<tr>
<td>duff</td>
<td>44</td>
<td>8 296</td>
<td>lcdnum</td>
<td>62</td>
<td>8 227</td>
<td>reverse-complement6</td>
<td>96</td>
<td>14 770</td>
</tr>
<tr>
<td>edn</td>
<td>170</td>
<td>9 027</td>
<td>lms</td>
<td>172</td>
<td>8 759</td>
<td>scam</td>
<td>63</td>
<td>15 633</td>
</tr>
<tr>
<td>expint</td>
<td>73</td>
<td>8 268</td>
<td>ludcmp</td>
<td>109</td>
<td>9 022</td>
<td>select</td>
<td>131</td>
<td>8 307</td>
</tr>
<tr>
<td>fac</td>
<td>23</td>
<td>8 199</td>
<td>mandelbrot9</td>
<td>66</td>
<td>14 764</td>
<td>spectral-norm1</td>
<td>57</td>
<td>14 725</td>
</tr>
<tr>
<td>fankuchredux1</td>
<td>79</td>
<td>14 843</td>
<td>matmult</td>
<td>54</td>
<td>8 328</td>
<td>st</td>
<td>98</td>
<td>8 337</td>
</tr>
<tr>
<td>fankuchredux5</td>
<td>115</td>
<td>15 047</td>
<td>mbe</td>
<td>63</td>
<td>15 622</td>
<td>statemate</td>
<td>1354</td>
<td>10 613</td>
</tr>
<tr>
<td>fasta1</td>
<td>126</td>
<td>21 028</td>
<td>minver</td>
<td>201</td>
<td>9 074</td>
<td>sumprod</td>
<td>18</td>
<td>14 355</td>
</tr>
<tr>
<td>fasta2</td>
<td>264</td>
<td>15 236</td>
<td>nbbody1</td>
<td>92</td>
<td>14 756</td>
<td>ud</td>
<td>81</td>
<td>8 933</td>
</tr>
<tr>
<td>fasta3</td>
<td>90</td>
<td>14 586</td>
<td>nbbody2</td>
<td>107</td>
<td>14 819</td>
<td>wc</td>
<td>49</td>
<td>20 615</td>
</tr>
<tr>
<td>fasta5</td>
<td>109</td>
<td>14 797</td>
<td>nbbody3</td>
<td>90</td>
<td>14 757</td>
<td>fasta7</td>
<td>231</td>
<td>15 041</td>
</tr>
</tbody>
</table>

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.function, 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.

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, SWS is consistently the fastest with an average slicing time of 0.67 seconds. At the other end of the spectrum, 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, 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 ESW to EWS.

ESW and EWS are close in terms of time. ESW’s slicing times range from 14 seconds to 130 minutes with a mean of 11.1 minutes, while EWS’s slicing times range from 14 seconds to 64 minutes with a mean of 13.5 minutes. That ESW and EWS’s times are so close is surprising, given that ESW considers only the WebAssembly instructions of a single function, while EWS considers the C source code of the entire program. Finally, we note a preference for 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 ESW and EWS, respectively.

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

B. RQ2: Which of EWS, ESW, and SEW Best Balances Speed and Precision?

Building on RQ1, RQ2 factors in slice size. We first consider the slices of ESW and SEW. By focusing ORBS deletion on the lines within a given function, SEW is notably faster than 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 ESW and SEW are nearly identical. Numerically, the median slice size for ESW is 13.05% of the original code, which is essentially equivalent to SEW’s median of 13.17%. Surprisingly, in contrast to our expectation, 438 slices are larger for ESW than SEW. The
int g, *p;
foo()
{
    bar();
    *p = 42;
}
bar()
{
    p = &g;
}

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.

Finally, we consider the performance of the static SWS slicer. By its very nature SWS produces larger slices because of the need to make safe static approximations. Compared to ESW and 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: EWS produces notably larger slices than the other dynamic approaches. While at present only SWS is practical as a real-time tool, SEW satisfies our looser practicality requirement and produces slices that are less than one third the size of those produced by SWS. Thus we find that 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 ESW and SEW and then EWS and SEW. We omit the direct comparison between ESW and EWS because it is similar to that of EWS and SEW.

1) Comparison of ESW and SEW: ESW produces a smaller slice in 467 cases, the same size in 738 cases, and surprisingly SEW produces the smaller slice in 438 cases. None of the differences are large, with size(ESW) - size(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 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:

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

However, in the ESW slice the argument counting code gets sliced out leaving argc with the value zero. Thus in the EWS slice, there is no need to initialize variable i. Consequently, Lines 81 and 82 are omitted from the EWS slice.

<table>
<thead>
<tr>
<th>Slicer</th>
<th>Median</th>
<th>Mean</th>
<th>Min</th>
<th>Max</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESW</td>
<td>13.05%</td>
<td>22.25%</td>
<td>0.12%</td>
<td>92.93%</td>
<td>± 22.39</td>
</tr>
<tr>
<td>SEW</td>
<td>13.17%</td>
<td>22.82%</td>
<td>0.19%</td>
<td>92.12%</td>
<td>± 22.84</td>
</tr>
<tr>
<td>EWS</td>
<td>17.34%</td>
<td>29.57%</td>
<td>0.21%</td>
<td>267.00%</td>
<td>± 33.14</td>
</tr>
<tr>
<td>SW</td>
<td>13.05%</td>
<td>22.25%</td>
<td>0.12%</td>
<td>92.93%</td>
<td>± 36.99</td>
</tr>
</tbody>
</table>

We expect EWS to produce larger slices than ESW and SEW when, for example, the latter two remove parts of the standard function entry and exit boilerplate code. Empirically, the EWS slice is larger than the corresponding ESW slice 79% of the time and larger than the corresponding 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 EWS slices is more than 100% of the sliced function. The most extreme case is a slice of addpcm that is almost three times larger than the original function. We investigate this effect as part of RQ3’s qualitative look at the slices.

Fig. 5. 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.

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 EWS slices is more than 100% of the sliced function. The most extreme case is a slice of addpcm 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 SWS slicer. By its very nature SWS produces larger slices because of the need to make safe static approximations. Compared to ESW and 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: EWS produces notably larger slices than the other dynamic approaches. While at present only SWS is practical as a real-time tool, SEW satisfies our looser practicality requirement and produces slices that are less than one third the size of those produced by SWS. Thus we find that 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 ESW and SEW and then EWS and SEW. We omit the direct comparison between ESW and EWS because it is similar to that of EWS and SEW.

1) Comparison of ESW and SEW: ESW produces a smaller slice in 467 cases, the same size in 738 cases, and surprisingly SEW produces the smaller slice in 438 cases. None of the differences are large, with size(ESW) - size(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 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:

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

However, in the ESW slice the argument counting code gets sliced out leaving argc with the value zero. Thus in the EWS slice, there is no need to initialize variable i. Consequently, Lines 81 and 82 are omitted from the EWS slice.

<table>
<thead>
<tr>
<th>Slicer</th>
<th>Median</th>
<th>Mean</th>
<th>Min</th>
<th>Max</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESW</td>
<td>13.05%</td>
<td>22.25%</td>
<td>0.12%</td>
<td>92.93%</td>
<td>± 22.39</td>
</tr>
<tr>
<td>SEW</td>
<td>13.17%</td>
<td>22.82%</td>
<td>0.19%</td>
<td>92.12%</td>
<td>± 22.84</td>
</tr>
<tr>
<td>EWS</td>
<td>17.34%</td>
<td>29.57%</td>
<td>0.21%</td>
<td>267.00%</td>
<td>± 33.14</td>
</tr>
<tr>
<td>SW</td>
<td>13.05%</td>
<td>22.25%</td>
<td>0.12%</td>
<td>92.93%</td>
<td>± 36.99</td>
</tr>
</tbody>
</table>

We expect EWS to produce larger slices than ESW and SEW when, for example, the latter two remove parts of the standard function entry and exit boilerplate code. Empirically, the EWS slice is larger than the corresponding ESW slice 79% of the time and larger than the corresponding 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 EWS slices is more than 100% of the sliced function. The most extreme case is a slice of addpcm that is almost three times larger than the original function. We investigate this effect as part of RQ3’s qualitative look at the slices.
Another interesting example is the slices taken with respect to `fasta2_offset_49`. The `SEW` slice includes the following code while the `ESW` slice includes only the final instruction.

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

Here the `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).

```c
    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 `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 `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 `SEW` produced the preferred slice.

Turning to criteria for which the `SEW` slice is smaller, we first consider the slice taken with respect to `adpcm_inc_170`, where the `ESW` slice is three instructions longer than the `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 `ESW` slice can’t remove these instructions is twofold. First, the printing functions of the `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 `ESW` slice taken with respect to `expint_fact_57` includes the following eight instructions not present in the `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 `ESW` slice has “painted itself into a corner.” `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, `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 `ESW` and `SEW`: We next compare the slices of `SEW` and `ESW`. Comparing sizes, the `ESW` 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 `ESW` produces the largest of the dynamic slices and agrees with the quantitative data of RQ2.

Looking at examples where the `ESW` slice is smaller, the largest difference is for the slices taken with respect to `cover_c_484` for which the `SEW` slice includes 610 instructions while the `ESW` 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 `ESW` slice.

As a final illuminating example the `ESW` slice taken with respect to `lms_x_160` includes 51 instructions while the `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, `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 `SEW` produces the smaller slices. The most extreme case is for `edn_j_140` where the `SEW` slice includes 55 instruction while the `ESW` 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. For example, unrolling leads matmult_index_56’s EWS slice to include 208 instructions while the SEW slice includes only 38.

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

Unlike the comparison of EWS and SEW, the qualitative comparison of EWS and SEW is less one sided. In EWS’s favor, slicing the higher-level source enables 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 beneficial, at least in terms of slice size. An example is the slices taken with respect to edn_j_140. Some of the 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, 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 (EWS) and slicing multiple functions (EWS) 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 (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 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 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 SWS is the second argument to printf, which is called as printf("ORB5S: %s\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 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 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: 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 SWS to EWS, 92% of the static slices are larger while comparing it to 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 SWS slice on the right includes the function prologue introduced by the C compiler. The SEW and EWS slices on the left omit this code.

```
; ; prologue: ;; prologue: move
1 empty ;; stack pointer
2 global.get 0
3 i32.const 16 4
5 i32.sub 6
7 local.tee 2
8 global.set 0
9 ;; body of the function
10 local.get 2
11 i64.const 0 12
14 i32.const 1030
15 local.get 2
16 call $printf call $printf
17 drop drop
18 ;; epilogue: ;; epilogue: restore
19 ;; empty ;; stack pointer
20 local.get 2
21 i32.const 16
22 i32.add
23 global.set 0
24 ;; return value ;; return value
25 local.get 3
```

![Fig. 6. SEW and EWS remove the function prologue and epilogue, while SWS preserves it.](image)

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

```
1 global.get $sp
2 local.tee 1
3 global.set $sp
4 ...
5 local.get 1
6 local.get 0
7 $sp.const 1
8 $sp.add
9 $sp.store
10 ...
```

Another limitation of $SW$ 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 $ESW$ and $SW$ slice taken with respect to nsichneu_b_2188, we observed that both slices include a necessary store instruction but that the $SW$ slice includes 9224 instructions in total while $ESW$ includes only 22 instructions. This is due to $SW$'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 $SW$. In all but one case, all slicers agree. In the remaining case, for `fastas_l_l_89`, the static slice does not execute the slicing criterion. This indicates that $SW$ 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 $SW$. 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 $ESW$ is the preferred approach. Along with $ESW$, it satisfies our “cup of coffee” definition of practical. $EWS$ however produces larger slices than $ESW$ (and $SW$). Furthermore, our qualitative comparison finds the $ESW$ 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 $SW$ slices clearly under-approximate the true dependence. Further comparisons between $SW$ 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 $EWS$ 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
v.

VI. CONCLUSION

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

Our evaluation shows that $\mathcal{SW}$ requires much more time than the other approaches, while not reducing slice size as much as expected. $\mathcal{WS}$ 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 $\mathcal{WS}$ 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 $\mathcal{WS}$ 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


