Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries

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

David W. 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 be a portable compilation target, enabling the cross-platform distribution of programs written in a variety of languages. We propose an approach to *slice* WebAssembly programs in order to enable applications in reverse engineering, code comprehension, and security among others. Given a program and a location in that program, program slicing produces a minimal version of the program that preserves the behavior at the given location. Specifically, our approach is a static, intra-procedural, backward slicing approach that takes into account WebAssembly-specific dependences to identify the instructions of the slice. To do so it must correctly overcome the considerable challenges of performing dependence analysis at the binary level. Furthermore, for the slice to be executable, the approach needs to ensure that the stack behavior of its output complies with WebAssembly's validation requirements. We implemented and evaluated our approach on a suite of 8 386 real-world WebAssembly binaries, finding that the average size of the 495 204 868 slices computed is 53% of the original code, an improvement over the 60% attained by related work slicing ARM binaries. To gain a more qualitative understanding of the slices produced by our approach, we compared them to 1 956 source-level slices of benchmark C programs. This inspection helps to illustrate the slicer's strengths and to uncover potential future improvements.

KEYWORDS

Static program slicing, WebAssembly, Binary analysis

ACM Reference Format:
Quentin Stiévenart, David W. Binkley, and Coen De Roover. 2022. Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510070

1 INTRODUCTION

The recent inclusion of WebAssembly binaries in web applications poses new challenges with respect to their security, comprehension, and reverse engineering.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510070

WebAssembly [25] "*is a binary instruction format for a stack-based virtual machine*" [65] designed as a compilation target for high-level languages. The specification of its core has been a W3C standard since December 2019 [49]. WebAssembly was designed for the purpose of embedding binaries in web applications in a portable manner, thereby enabling intensive computations on the web. A 2021 empirical study by Hilbig et al. [30] found use cases on the web as diverse as game engines, natural language processing, and media players. Thanks to its ability to incorporate runtime functions exported by the host environment, WebAssembly has also found usage beyond web applications, broadening the value of analyses for WebAssembly. Examples include desktop applications [63], smart contracts [19], IoT back ends [27], and embedded software [52].

Program slicing [12, 66] is a program decomposition technique that, based on a specific program point called the *slicing criterion*, identifies a subprogram of the code relevant to the slicing criterion. Program slicing has numerous applications, in debugging [32, 37, 67], program comprehension [11, 16, 31, 36, 59], software maintenance [23, 26], re-engineering [14], refactoring [20], testing [4, 28, 29], reverse engineering [2, 3], tierless or multi-tier programming [45, 46], and vulnerability detection [50].

As such, there are numerous invaluable applications of slicing for WebAssembly binaries. Slicing, for example, can provide a building block for reverse engineering and for tools such as binary translators, profilers, and debuggers [14]. In terms of security, slicing can help with the inspection of WebAssembly binaries encountered in the wild where the source code is unavailable. Such often tedious and time consuming manual inspections aim to understand the code well enough to ascertain that it is free from malicious intent. Binary slicing can also serve applications such as constructing abstract program models for WCET estimation [41].

Program slicing approaches can be categorized along multiple dimensions [51]. *Static* approaches compute a slice that preserves the behavior for all possible program inputs, while *dynamic* approaches consider only a subset of the inputs. *Executable slicing* approaches produce a program that can be executed, while approaches computing a dependence *closure slice* do not. *Intra-procedural* approaches compute a slice that preserves the behavior of a given function, while *inter-procedural* approaches preserve the behavior of the entire program across function calls. Finally, program slicing can identify the portion of the program that either affects the slicing criterion (*backward slicing*) or is affected by it (*forward slicing*).

This paper presents the first *static intra-procedural backward* slicing approach for WebAssembly that is capable of producing *executable* slices. Our approach relies on control and data dependencies to identify the set of instructions that are part of a slice, given a specific instruction as the slicing criterion. This requires WebAssembly-specific data dependencies and control dependencies,

which we identify and describe in detail. Furthermore, it requires satisfying the WebAssembly *validation requirement*: WebAssembly programs must adhere to a particular stack discipline in order to be executable. Correctly removing instructions that are not part of the slice (i.e., that do not have any effect on the slicing criterion) may violate this property. Thus the slicer must identify places where the closure slice may leave the stack in an undesirable state and then compensate by including additional instructions to satisfy the validation requirement.

This paper makes the following contributions:

(1) We describe the first static intra-procedural backward slicing approach for WebAssembly. The first two phases of this three-phase algorithm compute a *closure slice* of a WebAssembly binary.
(2) To produce an *executable* slice, our algorithm's third phase implements a stack-preserving approach to produce a valid executable WebAssembly program from the closure slice.
(3) Using an implementation of our three-phase algorithm, we empirically evaluate our approach quantitatively on a real-world data set of 8 386 WebAssembly programs scraped by Hilbig et al. [30], from which we compute 495 204 868 slices, and qualitatively by comparing slices produced by our implementation against slices produced by CodeSurfer [58] on a set of 49 C programs compiled to WebAssembly.[1]

$$
\begin{aligned}
\mathit{module} &::= (\text{module } \mathit{type}^* \ \mathit{func}^* \ \mathit{table}) \\
\mathit{type} &::= (\text{type } (\text{func } \mathit{ft})) \\
\mathit{bt}, \mathit{ft} &::= t^* \rightarrow t^* \\
t &::= \text{i32} \\
\mathit{func} &::= (\text{func } (\text{type } \mathit{tidx})) \\
&\quad \mid (\text{func } (\text{type } \mathit{tidx}) \ (\text{local } t^*) \ \mathit{instr}^*) \\
\mathit{table} &::= (\text{table } n^*) \\
\mathit{instr} &::= \mathit{data} \mid \mathit{control} \\
\mathit{data} &::= \text{drop} \mid t.\text{const } n \mid t.\mathit{binop} \mid t.\mathit{unop} \\
&\quad \mid \text{local.get } n \mid \text{local.set } n \\
&\quad \mid \text{global.get } n \mid \text{global.set } n \\
&\quad \mid t.\text{load} \mid t.\text{store} \\
\mathit{control} &::= \text{block } \mathit{bt} \ \mathit{instr}^* \ \text{end} \mid \text{loop } \mathit{bt} \ \mathit{instr}^* \ \text{end} \\
&\quad \mid \text{if}_{\mathit{bt}} \ \mathit{instr}^* \ \text{else } \mathit{instr}^* \ \text{end} \\
&\quad \mid \text{call}_{\mathit{ft}} \ n \mid \text{call\_indirect}_{\mathit{ft}} \mid \text{br\_if } l \\
n, l, \mathit{tidx} &::= \text{a number}
\end{aligned}
$$

Figure 1: Syntax of MiniWasm

2 BACKGROUND: A BRIEF TOUR OF WEBASSEMBLY

WebAssembly is a stack-based assembly language. For the sake of simplicity, we describe our approach using a minimal version of WebAssembly called MiniWasm, introduced by Stiévenart and De Roover [54]. MiniWasm retains the defining features of WebAssembly, including structured control flow, the most essential assembly instructions, direct function calls, indirect function calls through function tables, unary and binary operations, and 32-bit integers. Our implementation actually supports a larger subset of WebAssembly as demonstrated by our consideration of real-world programs in our empirical evaluation, where we are able to slice 99.991% of 495 248 788 potential slicing criteria. We describe our implementation and its limitations in further detail in Section 4.1.

2.1 The MiniWasm Language

Figure 1 depicts the syntax of MiniWasm. A *module* consists of a sequence of type declarations (*type**), a sequence of function declarations (*func**), and a table (*table*) that identifies the targets of indirect function calls. As this work presents an *intra-procedural* slicing approach, we focus on function declarations. A more detailed explanation of MiniWasm and its formal semantics is given by Stiévenart and De Roover [54].

A function is declared with a type index *tidx*, which corresponds to the type declaration at that index in the sequence of type declarations. Functions first declare the types of their local variables. Parameters and local variables are anonymous and 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 declaration is the sequence of instructions that form the function's body.

Broadly speaking, there are two kinds of instructions. *Control instructions* (e.g., `loop` and `call`) structure the program's control flow, while *data instructions* manipulate the stack (`drop`, `const`), locals (`local.get` and `local.set`), and globals (`global.get` and `global.set`). Operations (*binop* and *unop*) are left unspecified as their concrete instantiation is inconsequential to program slicing. Blocks (`block`), loops (`loop`), and function calls (`call`, `call_indirect`) are annotated with their types (respectively, *bt* and *ft*). Blocks act as delimiters inside functions for identifying jump targets. Loops are basically blocks whose semantics capture the iterating behavior. We include an `if` instruction in MiniWasm that was previously treated as syntactic sugar [54]. It is a conditional that encloses one consequent branch and one alternative branch.

2.2 The SCAM Mug in MiniWasm

To illustrate programming in WebAssembly, we consider the "SCAM Mug" [61] 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, 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 code features the following main function in C where all called functions are side-effect free:

[1] Our implementation is available publicly at https://github.com/acieroid/wassail/tree/icse2022, and a full replication package is available at https://zenodo.org/record/5821007.

```
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 }
```

The equivalent function in MiniWasm is given below, with functions p, q, f, g, and h assigned the indices 0, 1, 2, 3, and 4 respectively:

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

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. Line 3 retrieves and pushes the value of the first local variable on the stack. The next line calls function 0, 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 calls function 1 with local variable 2 (q(c)) on Line 8. If the result of this call is non-zero, it calls function 2 (f) and assigns the result to local variable 1 (x) on Line 11, and does the same with function 3 (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 function 4 (h(i)). Finally, the **br_if** instruction on Line 20 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.

2.3 WebAssembly Validation Requirement

WebAssembly programs have to be *well formed*, according to Section 3 of the WebAssembly standard [49]. Of particular interest for program slicing is that the body of a function has to be well typed. 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 " → i32", meaning that it does not need anything from the stack and pushes one value of type i32. Typing extends to sequences of instructions, e.g., the sequence local.get 0, local.get 1, **i32**.const 1, **i32**.add has type " → i32 i32".

The following example illustrates the impact that slicing WebAssembly code can have on this validation requirement.

```
1  (func (type 1)
2    local.get 0      ;; push first parameter
3    if
4      local.get 0    ;; push first parameter
5      local.get 1    ;; push second parameter
6      i32.add        ;; slicing criterion
7      call 0_{i32→i32}
8      drop
9    end)
```

This function takes two parameters (assuming that type 1 is "i32,i32 → "). It first pushes the first parameter on the stack (Line 2), and, if that parameter is non-zero, executes the body of the **if** statement (Line 3). This will push both parameters on the stack (Lines 4 and 5), sum them (Line 6), and then call function 0, before dropping the return value of the call from the stack. This leaves the stack empty at the end of the function's execution. If the value of the first parameter was zero, the if body is not executed, but the condition is still removed from the stack, hence in both cases the stack is empty at the end of the function's execution.

Consider the intra-procedural closure slice taken with respect to the instruction **i32**.add. This instruction requires that two values are available on the stack. Through use-definition chains that can be computed statically, we know that the first value is computed by the instruction local.get 1 on Line 5, while the second is computed by the instruction local.get 0 on Line 4. These are the *data dependencies* of the **i32**.add instruction. The execution of the **i32**.add instruction depends on whether the **if** statement executes its *then* branch. Therefore, Line 3 is a *control dependency* and needs to be included in the slice too. Finally, the **if** instruction has a data dependency on Line 2 as the value pushed is used as the condition. As a result, we obtain the following closure slice: it contains all the instructions relevant to the evaluation of the slicing criterion.

```
1  (func (type 1)
2    local.get 0
3    if
4      local.get 0
5      local.get 1
6      i32.add
7    end)
```

However, this closure slice does not represent a valid WebAssembly program as it leaves one value on the stack in the `if` statement. Any attempt to execute such a program will lead to an error from the WebAssembly validator. Phase three of our algorithm includes a `drop` instruction after Line 6 to effectively clean up the stack.

3 THE WEBASSEMBLY SLICING ALGORITHM

Our algorithm consists of three phases: a data-gathering phase that computes the dependencies of each instruction in a function, a slicing phase that identifies the WebAssembly instructions of the closure slice, and a reconstruction phase that includes additional instructions to maintain the stack discipline and thus ensure that the slice is a valid executable WebAssembly function. This section details each of the three phases.

3.1 Data-Gathering Phase

The data-gathering phase computes the following elements.

Stack Layout. For WebAssembly programs, the layout of the stack can be computed statically for any instruction, including loops. A stack specification analysis [54] computes the impact of every instruction on the stack and assigns names to each element in the resulting stack layout. To illustrate, the following code has been annotated with the results of the stack specification analysis where, for example, [i0] denotes a stack that contains a single element, denoted by the opaque identifier i0.

```
1  local.get 0  ;; [i0]
2  i32.const 2  ;; [i1, i0]
3  i32.add      ;; [i2]
4  call 1_{i32→}    ;; []
```

Use-Definition Chains. In order to identify data dependences [43], uses of each element on the stack are linked to their respective definitions through use-definition chains [33]. For example, in the previous code listing the instruction `i32.add` *uses* i1 and i0 and *defines* i2. Use-definition chains map the use of i1 to its definition by the instruction `i32.const 2` and the use of i0 to its definition by the instruction `local.get 0`. Here a use, denoted *use*, is a pair consisting of the name of an element on the stack (e.g., i0), and the occurrence of an instruction (e.g., the position of a specific `i32.const` instruction in the binary). We denote the set of uses of instruction *instr* as uses(*instr*) and the set of instructions that contain the definitions corresponding to a use as defs(*use*).

Memory Dependences. Identifying data dependences through use-definition chains alone does not suffice for slicing as there can be indirect data dependences that arise through the use of WebAssembly's linear memory, which models the program heap. For example, the `i32.load` instruction in the following code is data-dependent on the `i32.store` instruction: a slice that includes the load instruction but not the corresponding store instruction would not preserve the semantics.

```
1  i32.const 1024 ;; stack: [1024]
2  i32.const 0    ;; stack: [1024, 0]
3  i32.store      ;; stack: [], stores 0 at 1024
4  i32.const 1024 ;; stack: [1024]
5  i32.load       ;; stack: [0], loads 0 from 1024
```

Modeling memory dependencies precisely requires some form of alias analysis. To date no alias analysis algorithm has been devised for WebAssembly. Thus we resort to a sound over-approximation: all `load` instructions are marked as potentially data-dependent on every store that may be executed before the `load` instruction (i.e., for which there exists a path in the CFG from the `store` to the `load`). A similar over-approximation is used for the memory dependencies of `call` instructions. Indeed, as the following code could represent the call of a function with a pointer as argument, any preceding modification to any pointer needs to be included in the slice.

```
1  i32.const 1024 ;; 1024 could be a pointer
2  call 1
```

`call` instructions are therefore treated as both `load` instructions (memory can be read during a function call) and `store` instructions (memory can be written during a function call). Both over-approximations could be rendered more precise with additional information from a sound alias analysis. We denote the instructions that are memory dependencies of an instruction *instr* as the set memoryDeps(*instr*).

`global.set` *Instructions.* In WebAssembly, global variables can be used to share data across function calls. For example, when producing WebAssembly, current C compilers use global variable 0 as the address in the linear memory where the stack pointer resides [38]. Incrementing or decrementing the value of global 0 is used to grow or shrink the stack. Although there might be no explicit dependence on global variable 0, it is important to include any instruction that modifies it in the slice. In the case of C programs compiled to WebAssembly, this ensures that function calls preserve the same call-stack semantics. For the same reason, and because our approach is intra-procedural and thus cannot know which global variables are required across function calls, all `global.set` instructions are collected and considered part of the slice. This over-approximation is independent of the slicing criterion. This is again a conservative over-approximation: in practice, not all `global.set` instructions are relevant for the slicing criterion. We denote the set of all `global.set` instructions as globalSetInstrs.

Control Dependences. In addition to data dependences, slices need to account for control dependences as they capture whether an instruction is executed or not. Consider the following example, where the value of local variable 0 is initially set to 0, and then set to 1 if the value of local 1 is true (i.e., any non-zero value).

```
1  i32.const 0
2  local.set 0    ;; local 0 = 0
3  local.get 1
4  if
5    i32.const 1
6    local.set 0  ;; local 0 = 1
7  end
8  ;; value of local 0 depends on
9  ;; the value of local 1
10 local.get 0
```

Here, the instruction `local.set 0` in the body of the `if` is control-dependent on Line 4, which is data-dependent on the instruction `local.get 1`. Hence, both the `if` and `local.get 1` instructions will need to be included in any slice that includes the instruction `local.set 0` on Line 6. We rely on the so-called "exact algorithm" by Ferrante et al. [21] to compute control dependences. We write $\text{controlDeps}(b) = \{a\}$ when Instruction b is control-dependent on Instruction a, or in other words, when there is a control-dependence from a to b. Hence, $\text{controlDeps}(instr)$ maps an instruction to the instructions on which it depends on due to control dependences.

3.2 Slicing Phase

Our algorithm for identifying the instructions that makeup the closure slice is inspired by traditional approaches to slicing [67] where the slicing criterion, data dependences, and control dependences are used to transitively add instructions to the slice. We include Agrawal's additions for structured control flow [1], which are required to properly support WebAssembly's structured jump instructions (e.g., `br`). Note that unlike most slicing approaches where the slicing criterion includes a program location and one or more variables of interest, we only require a program location, given as an instruction, as the slicing criterion. The variables of interest are implicitly determined by the instruction.

In order to support WebAssembly, however, two domain-specific extensions are needed. First, because we do not have inter-procedural information, `global.set` instructions are considered part of the slice as explained previously. Second, without precise aliasing information any write to the memory potentially influences any subsequent read from the memory, thus memory dependences come into play as soon as any read is included in the slice.

Our slicing algorithm, given as Algorithm 1, works as follows. The slice starts empty (Line 1) and the initial worklist contains the instruction that is the slicing criterion as well as all `global.set` instructions found in the function being sliced (Line 2). The algorithm then proceeds as a typical worklist algorithm. Instructions *instr* in the worklist that are already part of the slice are ignored (Line 6). Other instructions are added to the slice (Line 7) and their dependencies are added to the worklist:

- use-definition chains are followed to find data dependences: for each use of a value on the stack (Line 8), the instructions that define that value are added to the worklist (Line 9),
- the control dependencies of the current instruction are added to the worklist (Line 10),
- the memory dependencies of the current instruction are added to the worklist (Line 11).

Once the worklist is exhausted, the resulting slice is augmented according to Agrawal's technique, specifically the *conservative algorithm* given in Fig. 13 of their paper [1], where the `br` instructions (as unconditional jumps in WebAssembly) are treated as goto statements. For each instruction in the slice, all `br` instructions that are control-dependent on the instruction are added to the slice (Line 16). The slicing algorithm returns the slice augmented with these additions (Line 17).

```
slice(slicing criterion c, set of global.set instructions
  globalSetInstrs, set of br instructions brInstrs, use of stack
  locations uses, use-definition chains defs, control dependences
  controlDeps, memory dependences memoryDeps)
1  let slice ← {};
2  let workList ← {c} ∪ globalSetInstrs;
3  while workList ≠ {} do
4  |  let instr ∈ workList;
5  |  workList ← workList \ {instr};
6  |  if instr ∉ slice then
7  |  |  slice ← slice ∪ {instr};
8  |  |  for use ∈ uses(instr) do
9  |  |  |  worklist ← worklist ∪ defs(use);
10 |  |  worklist ← worklist ∪ controlDeps(instr);
11 |  |  worklist ← worklist ∪ memoryDeps(instr);
12 let sliceExtension ← {}
13 for instr ∈ brInstrs do
14 |  for instr' ∈ slice do
15 |  |  if instr' ∈ controlDeps(instr) then
16 |  |  |  sliceExtension ← sliceExtension ∪ {instr};
17 return slice ∪ sliceExtension;
```

Algorithm 1: Slicing Algorithm

3.3 Reconstruction Phase

The closure slice computed by the previous phase may not form a valid WebAssembly function. To illustrate why and to show the reconstruction's impact both before and after an instruction of the slice, we use the following code fragment. The fragment has type "$\rightarrow$" (i.e., it does not consume anything from the stack nor produce anything onto the stack). Its constituent instructions have been annotated with their own types:

```
1  local.get 0 ;; → i32
2  i32.const 2 ;; → i32
3  i32.add     ;; i32, i32 → i32
4  call_i32→   ;; i32 →
```

Suppose this fragment is sliced with respect to Line 2. The instruction `i32.const 2` forms a closure slice on its own because it is not influenced by any other instruction in the fragment. However, this slice has type "$\rightarrow$ i32", while the type of the original fragment is "$\rightarrow$". As the fragment may reside in a WebAssembly construct, such as a block or a function, that expects a specific type, replacing the fragment by its slice will result in an invalid WebAssembly program.

To overcome this issue, reconstruction augments the closure slice with synthetic instructions that preserve the original stack layout and thereby render the slice valid and executable. Adding these synthetic instructions is safe, because they have no side effects on the program behavior regarding the slicing criterion. The sequence of synthetic instructions that is introduced must have the same

type as the sequence of instructions it replaces. The fragment below illustrates the rewriting. It includes type summaries in place of the instructions that are not part of the slice. Note how the types of the final two instructions (`i32.add` and `call`$_{\text{i32}\rightarrow}$) have been merged into a single type summary that has the same effect on the stack.

```
1  ??           ;; → i32
2  i32.const 2  ;; → i32
3  ??           ;; i32, i32 →
```

The goal of the reconstruction phase is to add synthetic instructions that have the desired type summary. Our reconstruction phase uses `i32.const` when a summary pushes an i32 value on the stack and `drop` to remove a value from the stack. This process results in the following reconstructed slice, which is of the same type as the original fragment.

```
1  i32.const 0  ;; → i32
2  i32.const 2  ;; → i32
3  drop         ;; i32 →
4  drop         ;; i32 →
```

While the reconstructed slice has the same type as the original fragment, it is not minimal: `i32.const 2; drop` is shorter and an equally valid slice. We have opted to not attempt a global minimum but rather one that is local to each summary (e.g., `drop; drop` is the minimal set of instructions with the type "i32,i32 → " that can be inserted in place of the second summary, "`?? ;;` i32, i32 →"). This design choice has the advantage of only requiring a single linear traversal of the fragment's instructions.

Algorithm 2 depicts the reconstruction algorithm. We use the $\cdot$ operator to concatenate sequences: $[1,2]\cdot[3]$ is $[1,2,3]$, and overload it to insert an element between two sequences: $[1,2]\cdot 3\cdot[4]$ is $[1,2,3,4]$. Auxiliary function isInSlice identifies the instructions that are part of the closure slice according to Algorithm 1. Auxiliary function replace returns a sequence of synthetic `i32.const` and `drop` instructions such that the type of the entire sequence is the same as the instructions being substituted. For example, replace(`i32.const 1; i32.const 2; i32.add`) returns `i32.const 0`, because both sequences of instructions push a single value onto the stack.

Function validSlice returns a type-valid WebAssembly slice for a given sequence of instructions. To produce an executable slice it is initially called with the body of the sliced function—from which the result is built up recursively. Instructions that are part of the closure slice will be included in the result together with synthetic instructions that replace instructions that are not part of the slice while preserving their summary type. The algorithm maintains in its second argument a sequence, *removed*, of instructions that are not part of the closure slice and thus will be replaced by synthetic instructions.

If there are no instructions to process (Line 1), meaning that the last instruction of a sequence has been processed, any remaining instructions in *removed* are replaced by a type-equivalent sequence of synthetic instructions (Line 2). For instructions that contain sequences of instructions (`block`, `loop`), the corresponding body is first processed recursively (Line 7). If there are no instructions to keep in the body, the instruction itself is added to the sequence of instructions to remove (Line 9). Otherwise, its body is replaced with the sliced body (Line 11) and the subsequent instructions are processed (Line 12). The process for an `if` instruction is similar (Line 13) except that both branches need to be processed and the instruction can be removed only if both branches become empty (Line 16).

For all other instructions, if the instruction is part of the closure slice (Line 21), the previous instructions that have been removed are first replaced by their equivalent synthetic instructions, then the instruction is added to the result before the results of processing the subsequent instructions (Line 22). Otherwise, if the instruction is not part of the slice (Line 23), it is appended to *removed* (Line 24) and a recursive call is made on the remaining instructions. As a result, one obtains a valid, executable, WebAssembly program that is a superset of the closure slice that preserves the behavior of the slicing criterion and is thus *executable*.

isInSlice(*instruction instr*)
  checks whether *instr* is part of the slice;
replace(*sequence of instructions instrs*)
  return a sequence of dummy instructions that has the same type as *instrs*;
validSlice(*sequence of instructions instrs*,
  sequence of instructions removed)

```
1  if instrs = ⟨⟩ then
2  |  return replace(removed);
3  else
4  |  let instr ← head(instrs);
5  |  let instrs ← tail(instrs);
6  |  if instr is a block or loop instruction then
7  |  |  let body ←validSlice(instr.body, ⟨⟩);
8  |  |  if body = ⟨⟩ then
9  |  |  |  return validSlice(instrs, removed · instr);
10 |  |  else
11 |  |  |  let instr' ← instr with its body set to body;
12 |  |  |  return
   |  |  |    replace(removed) · instr' · validSlice(instrs, ⟨⟩);
13 |  else if instr is an if instruction then
14 |  |  let then ←validSlice(instr.then, ⟨⟩);
15 |  |  let else ←validSlice(instr.else, ⟨⟩);
16 |  |  if then = else = ⟨⟩ then
17 |  |  |  return validSlice(instrs, removed · instr);
18 |  |  else
19 |  |  |  let instr' ← instr with its branches set to then
   |  |  |    and else;
20 |  |  |  return
   |  |  |    replace(removed) · instr' · validSlice(instrs, ⟨⟩);
21 |  else if isInSlice(instr) then
22 |  |  return
   |  |    replace(removed) · instr · validSlice(instrs, ⟨⟩);
23 |  else
24 |  |  return validSlice(instrs, removed · instr);
```

Algorithm 2: Slice Reconstruction Algorithm

4 EVALUATION

We evaluate our approach through the following research questions.

RQ1: How does static stack-preserving slicing behave on classical slicing examples? The literature on program slicing has produced

several challenging examples which have each been studied extensively. We manually translate these examples to WebAssembly and inspect the output of our slicer for each.

RQ2: What is the size of closure slices built by our slicing phase? We apply our approach to the dataset collected by Hilbig et al. [30], which consists of real-world occurrences of WebAssembly from the web. We use each instruction of each function in the dataset as a slicing criterion to which we apply Algorithm 1. We present descriptive statistics about the size of the resulting closure slices.

RQ3: By how much do slices need to grow in order to render them executable? Relying on the same dataset as used in RQ2, we measure the increase in slice size after the closure slice has been rendered executable.

RQ4: How much time is needed by each phase in order to compute an executable slice? We measure the time needed by each phase when computing an executable slice.

RQ5: How does static stack-preserving slicing of WebAssembly binaries compare to slicing the original source code directly, before its compilation to WebAssembly? In cases where the source code of a WebAssembly program is available, it might be preferable to slice the source code instead of the binary. We compare the results of our approach to the static slices of C programs computed by CodeSurfer [58].

We conducted our evaluation on a machine with an AMD Ryzen Threadripper 3990X 64-Core CPU (2.9 GHz) with HyperThreading and 256 GiB of RAM, running 128 slicing jobs in parallel (one per logical core). Before detailing the research method and results for each of these research questions, we briefly describe our implementation.

4.1 Implementation

We implemented the approach described in this paper on top of the Wassail framework [55], using OCaml version 4.12.0. Its loader for WebAssembly binaries is based on the implementation that accompanies the official WebAssembly standard. Our implementation first conducts the data-gathering phase described in Section 3.1. Next, it identifies the instructions of the closure slice using Algorithm 1, before rendering the slice executable using Algorithm 2. Our implementation supports most of the WebAssembly 1.0 core specification [49], but has the following limitations:

- The `br_table` instruction is only supported when all its targets expect the same stack layout. This is a limitation of the stack specification analysis.
- Code that lies in a CFG node that is disconnected from the CFG entry node cannot be used as the slicing criterion because our implementation of use-definition chains requires a path from the entry node of the CFG to the slicing criterion.

These limitations both concern the data-gathering phase, and we leave overcoming them for future work. In our evaluation with a real-world dataset, we encountered these limitations in 0.009% of the slices, where 28% of the non-supported slices are caused by unsupported uses of the `br_table` instruction, and 72% by disconnected code.

4.2 RQ1: Behavior on Classical Examples

We first perform a qualitative evaluation of our slicing approach on examples that have been studied in the slicing literature, namely the SCAM Mug example [61], the Montréal Boat example [15], Word Count [23], and Agrawal's control examples [1]. We manually encoded each program to WebAssembly, before applying our approach. All source programs and the computed slices are available in our replication package.

Our first example is the SCAM Mug example presented in Section 2 using Line 11, `local.set 1`, as the slicing criterion. Key to understanding the challenge in this example is to realize that the value assigned to `c` does **not** effect the value of `x` at Line 11. Thus a minimal slice would exclude Lines 12 and 13. However, the expected behavior of a slicer based on transitive dependence is to include these lines [8]. Our slicer correctly produces the expected slice including these lines.

The Montréal Boat example [15] poses a similar dependence challenge. Being based on dependence closure our approach is unable to tease apart the dependences. However, it does correctly produce the expected, non-minimal, slice.

The Word Count example, introduced by Gallagher and Lyle [23], uses five different slicing criteria (e.g., a slice that computes just the number of lines in the input). After translating this example to WebAssembly, we produced the five corresponding slices and manually compared each to the expected result provided by Gallagher and Lyle, translated to WebAssembly. In each case our slicer produces the desired minimal slice.

The examples of Agrawal [1] demonstrate the need to treat unconditional jumps when slicing programs that use `goto` statements. We translated the examples from Figures 3 and 5 of their paper to WebAssembly, along with the expected slices. Our slicer correctly includes the necessary unconditional jumps (`br` instructions). It does include more instructions than the translation of Agrawal's slices. This is due to the fact that purity assumptions that are made in Agrawal's slices regarding some called functions are not made by our slicer. However, our slicer produces correct slices.

> **RQ1**: While our approach shares the limitations of other static slicers on examples such as the SCAM Mug and the Montréal Boat examples, it performs well on the more straightforward examples. It is exact when computing the Word Count slices, and the slices for each of Agrawal's examples merely include unwanted call instructions due to our over-approximation of memory dependences.

4.3 RQ2: Closure Slice Size

Hilbig et al. [30] gathered a dataset of 8 461 unique WebAssembly programs. We use this dataset in our evaluation after filtering out the 75 programs that could not be loaded. For each of the 17 88 688 functions in the remaining programs, we use each instruction as a slicing criterion yielding 495 204 868 slices. This is akin to related work [9] which used each SDG vertex as a slicing criterion. We use a timeout of 4 hours per program.

The dataset, our implementation and evaluation scripts, and the data resulting from our evaluation for RQ2, RQ3, and RQ4 are available in our replication package.

The size of the initial functions and their closure slices are summarized in Figure 2. The mean size of the slices produced by Algorithm 1 is $\mu = 52\%$ of the original function size. This is more than the 27-30% averages obtained by the C slicing techniques surveyed by Binkley and Harman [10]. There are two potential reasons for this. First, WebAssembly, being lower level than C, provides fewer semantic cues to the slicer. Second, our over-approximations regarding the memory and global.set instructions can increase slice size. However our 52% is lower than the 60% average slice size reported by Kiss et al. when slicing ARM binaries [34, 35]. In this case the likely cause is WebAssembly's use of structured control flow.

RQ2: Our slicing algorithm (Algorithm 1) computes slices that are on average 52% of the size of the original program, which is lower than related work on binary slicing, but larger than the results of source code slicers.

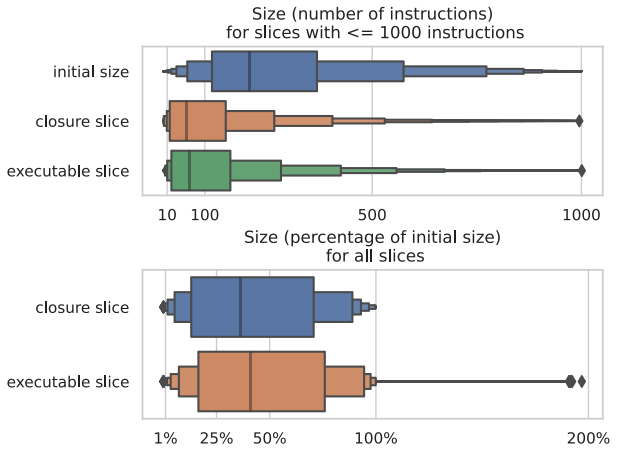

Figure 2: Sizes of the slices on a real-world dataset.

4.4 RQ3: Executable Slice Size

We now turn to the impact of Algorithm 2, which produces valid, executable slices from the closure slices studied by RQ2. As shown in Figure 2 the size of the slices increases from 52% to 53%. As evidenced by the outliers in the lower graph, some executable slices are larger than the original program. This phenomenon is limited to a handful of the slices (in total, it affects only 0.08% of the slices), where an average of 2.2 instructions are included per slice.

We manually investigated a statistically relevant sample (384 slices[2]) of these cases to identify underlying causes. The most common cause, impacting 98% of the manually investigated slices, is when an instruction that removes two or more values from the stack that is not part of the closure slice gets replaced by two or more `drop` instructions, resulting in a positive net change to the number of instructions. For example, removing a `call` to a function with 20 arguments replaces the `call` instruction with 20 `drop` instructions, thereby increasing the slice size by 19 instructions.

[2]Based on a population size of 380 364, with a confidence level of 95% and a margin of error of 5%.

Three instructions remove two or more values from the stack in WebAssembly: `select`, which pops three values and pushes one, `store`, which pops two values, and `call`, which pops as many values as there are arguments to the function call. In the worst case, a function had 32 `call` instructions to a 12-argument function, resulting in 352 `drop` instructions being added. In our manual investigation, we encountered 1881 instances of this pattern due to `call` or `call_indirect` instructions, 20 due to `store` instructions, and 9 due to `select` instructions. The only other cause we encountered for an increase in the size of the executable slice is due to type conversions: for example, the `i32.wrap_i64` instruction converts an i64 into an i32 in a single instruction, while our slicer replaces it by two instructions: `drop; i32.const 0`. We encountered this pattern only six times.

RQ3: Algorithm 2, reconstruction, has a small impact on slice size, which goes from an average of 52% to 53%.

4.5 RQ4: Time to Produce Executable Slices

We measured the time it takes for each phase in order to compute each slice and summarize the results in Figure 3.

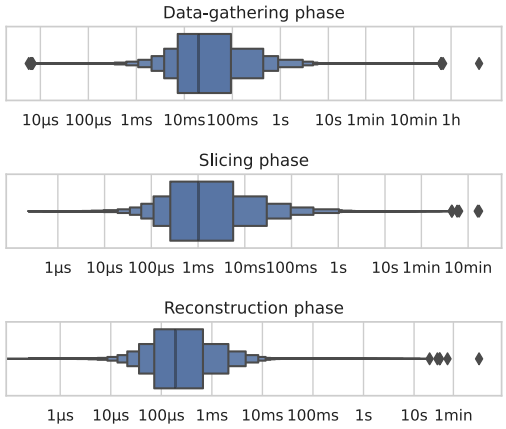

Figure 3: Slicing time.

The mean time for the entire slicing process (the data-gathering phase followed by Algorithms 1 and 2) is $\mu = 4\,094$ms. Only 14% of the slices take more than a second to compute. Looking at the breakdown between the different phases, we observe that the time to produce a slice is dominated by the data-gathering phase, taking 3 964ms on average and at most 3 hours and 42 minutes, while Algorithm 1 takes 104ms on average, and at most 44 minutes and 14 seconds, and Algorithm 2 takes 25 ms on average, and at most 3 hours and 22 minutes.

Investigating the data-gathering phase in more detail, we observe that the time is dominated by the computation of the data dependences ($\mu = 1\,672$ms), the memory dependences ($\mu = 360$ms) and the stack specification analysis ($\mu = 1\,811$ms), while the time required for the CFG construction ($\mu = 107$ms), the computation of the control dependences ($\mu = 13$ms), and listing the globals ($\mu = 247$ µs) remains small.

Regarding the outliers, we observe that they concern slices of functions of thousands of instructions. In some cases, the computation of the data dependencies takes most of the time. However, when many instructions are removed, the reconstruction algorithm may take the majority of the time. For example, the longest running time for the reconstruction algorithm is on a function that is reduced from 11k instructions to only 8.

RQ4: It takes on average 4 094ms to compute an executable slice. This time is dominated by the data-gathering phase and its computation of the data and memory dependences in particular. The time required to reconstruct an executable slice (25 ms on average) is only a fraction of the time required to construct the closure slice (104ms on average).

4.6 RQ5: Comparison to Slicing C Programs

When available, it can be beneficial to slice the source code directly rather than its WebAssembly counterpart. Furthermore, manual comparison of source slices compiled to WebAssembly with our WebAssembly slices enables characterizing the impact of the over-approximations used by our slicer. To investigate, we selected a set of 49 C programs originating from multiple sources. These include programs from the slicing literature, programs used to compare and evaluate WCET analysis tools [40], and programs designed to benchmark language implementations [22].

Each program was first normalized by pretty printing it using the pycparser Python library. After normalization these programs range from 16 to 2 988 source lines of code (SLOC), with an average of 207 SLOC, according to the tool sloccount. For each statement in these programs that modifies a scalar variable, we produced a variant of the program that includes a printf statement that prints the value of the variable, which is used as the slicing criterion.

We sliced each variant with CodeSurfer [58] and with our approach, as follows:

- We used CodeSurfer to produce a static closure slice of each C program. Because CodeSurfer does not produce executable slices, we apply quasi-static executable slicing (QSES) [53], which uses observation-based slicing (ORBS) to augment a CodeSurfer slice with the statements needed to make it executable [5]. This results in compilable slices. Each slice was then compiled to WebAssembly.
- We also compiled each variant directly to WebAssembly, and applied our approach using the instruction corresponding to the added printf statement as the slicing criterion.

In both cases, the C programs were compiled to WebAssembly using Clang 12.0.1 with the `-O2 -fno-inline-functions -lm -Wl,-demangle -Wl,-export-all` flags. We remove slices that encountered one of the limitations described in Section 4.1, as well as those whose CodeSurfer slices produce WebAssembly that cannot be loaded by our implementation. In the end, we obtain 1 956 pairs of slices to compare.

We manually investigated a statistically relevant sample of 95 slices[3] that included two slices from each program (except for three programs that only had one slice) in order to understand any differences. As described below, we identified three main root causes for the differences in instructions between each pair of slices. All of the slices as well as the details of our manual investigation are included in our evaluation package.

Memory Over-Approximation. The majority of the differences are due to our over-approximation of the memory dependences. As soon as the slice includes a `call` or a `load` instruction, our approach will include in the slice all `store` and `call` instructions that may be previously executed according to the CFG. In order to encode the slicing criterion for CodeSurfer, a call to the printf function is added to the code. This means that all slices for this RQ include a function call, and as a result our approach always includes memory dependencies in the slice. We encountered this pattern in 52% of the investigated slices.

Compiler Optimizations. The slices produced by CodeSurfer sometimes benefit from additional compiler optimization. We encountered various patterns related to this root cause. These differences can render the CodeSurfer slices quite different, and reconstructing the correspondence with the original program can require significant mental effort. In these cases, preserving the correspondence between its slices and the input program is a desirable property of our approach.

The CodeSurfer slice sometimes produces fewer instructions to manipulate memory. For example, the following code shows the slice produced by our approach on the left, and the slice produced by CodeSurfer on the right. The offset option of a store instruction enables writing at a specific offset from the target address of the store operation. Our approach preserves the offset of the store instruction from the original program, while CodeSurfer can remove it. This results in 2 superfluous instructions in our slice, for passing the written address as argument to the function call on the last line.

```
local.get 0                 local.get 0
i32.const 2000              i32.const 2000
i32.store offset=16         i32.store
...                         ...
local.get 0                 local.get 0
i32.const 16
i32.add
call 1                      call 1
```

Another instance of this pattern is that the control-flow structure of the program may be simplified, but our approach does not apply such rewritings. Here is an excerpt of a difference between a slice produced by our slicer (left), which preservers the original control-flow structure, and the corresponding CodeSurfer slice (right), which has two fewer instructions (`block` and `br 1`).

```
block
  local.get 0           local.get 0
  i32.const 1           i32.const 1
  i32.lt_s              i32.ge_s
  if                    if
    br 1
  end
  loop                    loop
    ...                     ...
  end                     end
end ;; block            end ;; if
```

[3]Based on a population size of 1 956, with a confidence level of 95% and a margin of error of 10%.

In some cases, Clang inlines loops in the WebAssembly code, resulting in an increase in size for the CodeSurfer slices. Other cases falling under this root cause include the use of fewer local variables by CodeSurfer when, for example, multiple local variables always have the same value in the slice, the reversing of a loop iteration order when it is beneficial for code size, or changing the type of a function when its return value is not needed, thereby avoiding extra instructions to return a value. We have observed this root cause in 54% of the manually investigated slices.

Inter-procedural vs. intra-procedural. CodeSurfer's computation of inter-procedural slices has the impact that, when f is recursive CodeSurfer includes code reached both directly (in the "current" invocation of f) and indirectly via recursive calls to f. For example, if f's return value is used in a control-flow decision upon which the slicing criteria is itself dependent, CodeSurfer will preserve the computation of f's return value. In contrast, our approach never includes instructions required solely to support recursive calls. This is a root cause that we encountered in 20% of the manually investigated slices.

RQ5: When the C source code of an application is available, compiling a C slice to WebAssembly may produce smaller slices. The main causes of this are the memory over-approximations made by our slicer and the extra optimizations enabled when compiling the simpler code of the slice. On the other hand, due to compiler optimizations, C slices compiled to WebAssembly may differ significantly from the corresponding portions of the directly sliced WebAssembly code. This can hamper program comprehension of the resulting slices.

4.7 Discussion

The results of our investigation indicate that slicing WebAssembly programs directly can greatly reduce their size, facilitating applications such as reverse engineering and program comprehension. Moreover, obtaining an executable slice only requires a small increase in slice size, and can be done quite quickly.

We identify two main possible improvements. First, regarding slice size, having alias information would eliminate superfluous instructions that are currently included due to our memory over-approximation. Second, regarding slice time, effort should be spent on optimizing the data-gathering phase, in particular the computation of the data dependences.

Finally, we observe that when the C source code of an application is available, even though the slice sizes may increase, it can be beneficial in terms of program comprehension to slice at the WebAssembly level directly.

4.8 Threats to Validity

To conclude our evaluation, we identify threats to validity according to the classification of Wohlin et al. [68]. A threat to internal validity comes from our evaluation setup on the real-world dataset. For some of the 8 386 programs it is infeasible to compute all of the slices; thus, we rely on a timeout of 4 hours per program and let the slicer compute as many slices as possible in that time. We were able to compute 495 204 868 in total, and the evaluation ran to completion without timeout on 83% of the programs.

Our implementation has limitations that we described in Section 4.1, which form a threat to construct validity. However, we do not expect to observe a different outcome in our results if these limitations are lifted, as we do not expect instructions within these disconnected sections or accessible from a `br_table` instruction to be too different from other instructions in the code.

A threat to external validity is that among the 8 386 programs used in our evaluation of RQ2 and RQ3, there can be duplicate functions across different binaries, which we have not filtered out and could therefore have been sliced multiple times. This is mitigated by the high number of functions sliced in total, which limits the potential impact of this threat on the results.

Our dataset for RQ5 is composed of 49 C programs that we manually gathered. Being less varied than the first dataset of 8,386 programs, it forms a threat to the external validity for RQ5. However, we ensured the diversity of this dataset by gathering it from three different sources.

5 RELATED WORK

WebAssembly. There has been interest from the research community in WebAssembly on aspects such as security [24, 38, 42, 56, 57], extensions to the language [17, 47], tooling [48], and optimizations [13]. In terms of program analysis, Lehmann and Pradel introduced a framework for dynamic analysis of WebAssembly [39], Watt et al. a first-order program logic to verify WebAssembly programs [64], and Stiévenart and De Roover a static information flow analysis [54] based upon a static analysis framework for WebAssembly [55]. Our approach is entirely static and is based on that same framework.

Perényi and Midtgaard performed property-based testing of WebAssembly runtimes [44]. The shrinking phase of their approach also faces the problem that the generated program needs to pass the validation requirement, which is solved through a different rewriting phase.

Binary Slicing. Static slicing has been applied to binary executables with a focus on register-based assembly languages whereas WebAssembly itself is stack based. Cifuentes and Fraboulet [14] perform intra-procedural static slicing on an assembly language close to x86. They argue that using basic blocks as the node granularity is more appropriate for binary executables, as the number of instructions can be large. Unfortunately, this approach is evaluated on a single example. Our approach in contrast operates on individual instructions, and its evaluation on real-world data demonstrates the feasibility of slicing at this level. Like ours, their approach does not include any alias analysis.

Inter-procedural static slicing for binaries has been achieved by Kiss et al. [35] and was later extended to include dynamic information during slicing [34]. Special care is needed to handle indirect function calls, which is not necessary for an intra-procedural approach. Similarly, the tool presented by Mangean et al. performs inter-procedural static slicing [41]. Ward et al. combined dynamic and static slicing for analyzing binaries, through conditioned slicing [62], with the goal of analyzing and migrating assembler systems. A dynamic slice is first computed, before being augmented with information from a transformation-based static slice.

In terms of slicing stack-based assembly languages, there has been related work on slicing JVM bytecode. Umemori et al. present a static approach which requires the presence of the Java source code alongside the JVM bytecode [60], while our approach does not rely on the presence of the source code. Castaldo D'Ursi et al. slice JVM bytecode after converting it to Jimple code, which eliminates the need to deal with stack-based bytecode [18]. Currently, there exists no equivalent to Jimple for WebAssembly, and our approach therefore directly handles the stack-based nature of WebAssembly. Zhao presents a dependence analysis for Java bytecode, with slicing as one of its application [69]. However, the handling of instructions that manipulate the value stack is omitted from their description.

Language-Independent Slicing. Binkley et al. present ORBS, a language-independent slicing approach that observes the program output in order to build an executable slice [5], which works at the line level [6, 7]. We rely on the QSES extension of ORBS [53] to reconstruct executable C slices from CodeSurfer for our fifth research question. ORBS could in theory be applied to WebAssembly programs directly, on the condition that the program is instrumented to capture the slicing criterion as part of the program's output. We leave a comparison with the resulting dynamic slices for future work.

6 CONCLUSION

We introduced the first static intra-procedural backward slicing approach for WebAssembly binaries. This three-phased approach consists of a data-gathering phase, which computes all the necessary information for slicing, a slicing phase, which constructs a closure slice that contains all instructions needed to preserve the behavior of the slicing criterion, and a reconstruction phase, which produces an executable slice from the closure slice. This last phase is needed because WebAssembly programs have to adhere to a validity requirement to be executable.

We evaluated our approach on a real-world dataset of 8 386 programs. We observed that our approach results in executable slices that are on average of 53% of the original function. The time needed to compute these slices averages 4 094ms where most of that time is spent on the data-gathering phase, in particular in the computation of data dependences. Through a qualitative comparison of the slices produced by our approach with executable C slices compiled to WebAssembly, we find that incorporating some form of alias analysis in the slicing process should notably reduce the average slice size.

This work forms an important stepping stone towards binary analysis applications such as reverse engineering and program comprehension.

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

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

[5] 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

[6] 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

[7] 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

[8] Binkley, D.W., Gold, N.E., Harman, M., Islam, S.S., Krinke, J., Yoo, S.: ORBS and the limits of static slicing. In: Godfrey, M.W., Lo, D., Khomh, F. (eds.) 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015. pp. 1–10. IEEE Computer Society (2015), https://doi.org/10.1109/SCAM.2015.7335396

[9] Binkley, D.W., Harman, M.: A large-scale empirical study of forward and backward static slice size and context sensitivity. In: 19th International Conference on Software Maintenance (ICSM 2003). pp. 44–53. IEEE Computer Society (2003), https://doi.org/10.1109/ICSM.2003.1235405

[10] 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

[11] 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)

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

[13] 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

[14] 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

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

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

[17] 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)

[18] 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

[19] 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)

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

[21] 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

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

[23] 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

[24] 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)

[25] 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)

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

[27] Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019. pp. 225–236 (2019)

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

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

[30] 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

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

[32] 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)

[33] Kennedy, K.: Use-definition chains with applications. Comput. Lang. **3**(3), 163–179 (1978), https://doi.org/10.1016/0096-0551(78)90009-7

[34] 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

[35] Kiss, Á., Jász, J., Lehotai, G., Gyimóthy, T.: Interprocedural static slicing of binary executables. In: 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003). p. 118. IEEE Computer Society (2003), https://doi.org/10.1109/SCAM.2003.1238038

[36] 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)

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

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

[39] 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)

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

[41] 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

[42] 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

[43] Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. pp. 177–184 (1984)

[44] Perényi, Á., Midtgaard, J.: Stack-driven program generation of WebAssembly. In: d. S. Oliveira, B.C. (ed.) Programming Languages and Systems - 18th Asian Symposium, APLAS 2020. Lecture Notes in Computer Science, vol. 12470, pp. 209–230. Springer (2020), https://doi.org/10.1007/978-3-030-64437-6_11

[45] 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)

[46] 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)

[47] 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

[48] 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

[49] Rossberg, A.: WebAssembly Core Specification. Tech. rep., W3C (2019), https://www.w3.org/TR/wasm-core-1/

[50] 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)

[51] Silva, J.: A vocabulary of program slicing-based techniques. ACM Comput. Surv. **44**(3) (Jun 2012)

[52] Singh, R.G., Scholliers, C.: WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In: Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019. pp. 27–36 (2019)

[53] 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

[54] 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

[55] Stiévenart, Q., De Roover, C.: Wassail: a WebAssembly static analysis library. In: Fifth International Workshop on Programming Technology for the Future Web (2021)

[56] 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)

[57] 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)

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

[59] Tonella, P.: Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Transactions on Software Engineering **29**(6) (2003)

[60] 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

[61] 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

[62] Ward, M.P., Zedan, H.: Combining dynamic and static slicing for analysing assembler. Sci. Comput. Program. **75**(3), 134–175 (2010)

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

[64] 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)

[65] WebAssembly. https://webassembly.org/

[66] Weiser, M.: Program slicing. In: 5th International Conference on Software Engineering. pp. 439–449 (1981)

[67] 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

[68] 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

[69] 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