RANDOM TESTING OF WEBASSEMBLY IMPLEMENTATIONS USING SEMANTICALLY VALID PROGRAMS

by

Guy Watson

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

August 2023

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Guy Watson**
has been approved by the following supervisory committee members:

Eric Norman Eide ,	Chair(s)	**May 15, 2023** Date Approved
John Regehr ,	Member	**May 18, 2023** Date Approved
Pavel Panchekha ,	Member	**May 15, 2023** Date Approved

by **Mary W. Hall** , Chair/Dean of
the Department/College/School of **Computing**
and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

WebAssembly is a relatively new language designed to be low-level and portable. Designed primarily for web browsers, its compact representation is meant to be directly executed by a browser, enabling high-performance applications on the web. Since implementations are both complex and browser-dependent, the language is a good target for differential random testing. This thesis introduces *Wasmlike*, a random generator of semantically valid WebAssembly programs. By using semantically valid programs with random differential testing, the goal is to penetrate past syntax and semantic validation, and test WebAssembly implementations for defects that cause programs to produce incorrect results. Wasmlike has found five significant semantics defects in WebAssembly implementations.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Thanks to my family and friends for their endless support.

This material is based upon work supported by the National Science Foundation under Grant Numbers 1527638 and 2027208. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 1

INTRODUCTION

WebAssembly, commonly known as *Wasm*, is a stack-based, portable, and low-level programming language. WebAssembly is defined by its specification [20] and has a single binary format that is meant to be compiled and executed by a web browser. The language shares many similarities with assembly languages, being comprised of simple instructions, very few types, and almost no frills. It is not, however, machine code. It uses stack-based execution, compared to modern processors that use registers. It enforces structured control flow, such as branches that target a nesting level instead of a program counter. These simple language structures can be easily translated into machine code.

Large pieces of software are typically tested with test suites that contain unit and integration tests to verify the behavior of the software. The complexity of these pieces of software — with unanticipated edge cases or complex interactions — makes them infeasible to exhaustively test. This is especially the case with software that processes or implements programming languages, such as compilers and interpreters, since the variability of possible programs makes it impractical to write tests for every way a language's features might be used, either singly or in combination.

Random testing, or *fuzzing*, is useful to complement existing "fixed" test suites. Random testing approaches such as AFL [26] are known as mutational fuzz testers, or *fuzzers*, and generate new test inputs by mutating an initial seed or corpus. Mutational fuzzing has several problems when applied to systems like compilers. It tends to find errors related to syntax, but has difficulty penetrating deep into the code base to find more complex errors [12], since the vast majority of mutations produce malformed code. Additionally, the test inputs created by a mutational fuzzer will often only display characteristics from the initial seed, or language constructs found randomly. This limits the possible test

inputs that the fuzzer can practically generate and makes finding bugs relying on complex interactions extremely difficult.

Random program generation based on the semantics of a language can avoid these problems and act as a complementary method to mutational fuzzing methods. For a generated program to be considered semantically valid, it must obey the grammar of the language and conform to the scoping, typing, and behavioral rules set by the language specification. A semantically valid program run by an implementation of the language can be expected to execute and produce a meaningful result. One can therefore use semantically valid test programs to find *semantic bugs* in the implementation of a programming language: i.e., errors in which a valid program does not produce the result that is correct according to the language specification.

With no ability to guarantee that a given implementation of WebAssembly is perfect, we cannot use one as an oracle to distinguish other implementations as containing or not containing defects. Instead, we use random differential testing [13] to identify test cases that trigger defects in given implementations. In short, if the same input is given to multiple systems under test and results in output that is not uniform, a defect is present in at least one system under test.

Thesis statement: Random differential testing of WebAssembly implementations using semantically valid programs allows the discovery of defects that cause programs to produce incorrect results.

To investigate this hypothesis, we present *Wasmlike*, a fuzzer that randomly generates semantically correct WebAssembly programs. We use the fuzzer to differentially test several implementations of WebAssembly: NodeJS, Wasmer, Wasmtime, Firefox, and Chromium.

To implement this fuzzer we use Xsmith [11], a framework for creating programming language fuzzers. The advantage of using a tool like Xsmith is highlighted by the restrictive nature of WebAssembly. Since browsers are meant to execute WebAssembly code natively, the language is designed to be safe. WebAssembly programs must satisfy a number of constraints in order to be semantically valid and accepted by a WebAssembly compiler. Programs that do not follow these constraints are declared invalid by the specification, which renders the program unusable for random differential testing. Because

Xsmith can express all of WebAssembly's language constraints, it is an excellent choice for constructing a fuzzer that can make guarantees about the programs it generates.

We evaluate WebAssembly implementations with the aid of a testing harness that automates differential testing and reports possible defects. Defects are signaled when a single test input produces different results across multiple implementations: if one runtime reports a different result compared to the other runtimes, we know that a semantic defect has been found.

This thesis describes our approach to testing WebAssembly implementations by generating semantically valid programs. In Chapter 2, we describe related work that informs our approach. Chapter 3 describes our methodology on generating semantically valid test programs. The constraints present in WebAssembly, as well as the need to avoid undesirable runtime behavior, guide program generation to ensure useful and semantically valid test cases. Chapter 4 covers our testing methodology and the test harness used for the experiment, and Chapter 5 discusses results, conclusions, and future work.

CHAPTER 2

RELATED WORK

Fuzz testing compilers can lead to finding bugs in areas of a language that are not often exercised [8]. These defects can be very hard to identify and test for manually, which is the main motivation for using random testing.

To generate random test cases, we take a similar approach to Csmith [25], specifying the grammar and attributes of a language in order to generate a valid program for the compiler. Csmith starts with an attribute grammar and randomly generates an AST by using the grammar to determine valid placements and type rules. Csmith then transforms that AST into the text format of a semantically valid C program. Csmith was developed specifically for the C language, and Xsmith [11] is a successor to the Csmith project. Xsmith is a tool designed for creating fuzz testers by specifying the grammar, attributes, type system, and other rules of a language. This thesis uses Xsmith to create Wasmlike, our WebAssembly fuzz tester.

Fuzzers can take many different forms, with one of the most common being a mutational fuzzer like AFL [26]. Fuzzers like these rely on an initial seed or corpus of sample data, which they mutate to produce new tests. This thesis chooses a different approach, relying on a semantic specification of the language to generate test cases. This exercises features of the language more deliberately, instead of relying on the seed data to contain language features [6].

Some previous fuzzers attempt to automatically infer highly structured inputs without a grammar specification. Grimoire [2] is a fuzz tester that attempts to synthesize language structure during the fuzzing process. However, Grimoire has issues when dealing with precise syntax and semantics. While it can approximate, WebAssembly is extremely strict with semantics. The restrictive semantics lend themselves well to a grammar specification-based fuzzer, such as the one we construct for this thesis.

Other work has been done to generate semantically valid test inputs from mutation-based fuzzing. Polyglot [7] breaks down the seed corpus into an IR language and mutates seed inputs to generate a new test input. To get a high percentage of semantically valid tests, Polyglot does a second pass with a grammar specification to fix any mismatches that arise from mutation, such as variable names and types. This presents a problem for WebAssembly programs because of the strict semantics of the language. Everything on the stack must be used, and fixing mismatches would result in WebAssembly programs that consist of mainly deterministic fixes interspersed with random instructions.

Similarly, Nautilus [1] combines a grammar specification with a coverage-guided approach to generate semantically meaningful programs. Nautilus uses coverage information to find interesting paths and mutate previously generated inputs, which guides its generation of syntactically correct programs towards being more semantically correct. However, Wasmlike is built to *always* generate semantically valid programs and instead steers the generation of programs by being able to make choices during generation based on the current structure of the program being generated. This lets Wasmlike target different features in a focused manner, albeit without coverage guidance.

A slightly different approach to semantically valid programs is Zest [17], which conducts a fuzzing campaign by guiding the choices of a program generator instead of directly mutating a seed. This requires some kind of program generator to be written, and results in semantically correct programs if the generator is written correctly. Zest is meant for use with coverage-guided fuzzing, since it uses a code coverage metric to adjust its mutations. This thesis does not instrument WebAssembly implementations and instead directly generates programs and uses differential random testing to find defects.

Finding deep semantic defects in a language through fuzzing typically requires the fuzzer to be specific to that language. Perényi and Midtgaard wrote a stack-directed fuzzer for WebAssembly [18] to target defects that would be missed by coverage-guided fuzzers. Their fuzzer uses differential testing and is focused on extreme generator freedom: any semantically valid WebAssembly program can be generated including recursive calls and nondeterministic or undefined behavior. By comparison, Wasmlike is focused on finding code-generation defects, and therefore avoids nondeterministic behavior through static and dynamic constraints on generation.

Fuzz testing is not without its problems. Because of its unpredictable nature, the rigor of random testing campaigns and tools has been called into question [12]. Klees et al. detail problems with existing random testing papers, mainly: the failure to perform multiple runs, counting unique crashes instead of distinct bugs, short timeouts, a small selection of starting seeds, and a lack of common benchmarks. This thesis attempts to follow their recommendations where applicable, e.g., appropriate timeouts and not measuring success in terms of crashes found. While Klees et al. call for care in selecting the starting seed, semantics-based fuzzing must instead take care in how often certain language features are generated in order to produce meaningful programs that test large parts of a language's implementation. Wasmlike has several tuning parameters and idiomatic structures that help generated test cases mimic real world behavior.

Fuzz testing is one of the test methods that both the WebAssembly project [10] and the makers of Wasmtime [4] use. They both provide a fuzzer that turns a stream of bytes into a WebAssembly module in order to test implementations. Their fuzzers always generate semantically valid test cases, but lack the targeting and tuning that Xsmith provides.

CHAPTER 3

GENERATING WEBASSEMBLY PROGRAMS

To test WebAssembly implementations, we constructed an Xsmith specification for WebAssembly, which we call *Wasmlike*.[1] Wasmlike encodes the grammar, types, constraints, and features described by the specification and structure of the WebAssembly language, version 1.1 [20].

We first provide an introduction to the WebAssembly specification and language features used in the rest of this thesis. We then detail how Wasmlike works, describe the constraints that guide generation, both specification constraints and generation model constraints, and discuss how Wasmlike avoids undefined behavior. We then discuss how Wasmlike generates interesting programs and avoids uninteresting ones by using the features of an attribute grammar to make decisions on what direction to guide generation. Finally, we cover the current state of Wasmlike and its limitations.

3.1 WebAssembly Specification

The WebAssembly specification is a self-contained technical document that details everything about the language, from instructions to types to program validation. Here, we summarize the important aspects and necessary information for readers to understand the rest of this thesis.

WebAssembly programs are designed to be portable and system-agnostic. A WebAssembly program is compiled into a binary form similar to Java bytecode: very compact, but the final compilation to machine code and execution is the responsibility of the browser or runtime.

The WebAssembly text format can be translated to and from its binary format with no effect on the program, although translating between the two may result in minor changes

[1] The name is based on a series of fuzzers constructed from Xsmith called *Cish* and *Pythonesque*.

such as replacing a name with its equivalent index form or desugaring convenient shortcuts. The rest of this thesis uses the text representation of WebAssembly as well as the desugared syntax forms.

In the following sections, we lay out the structure of a WebAssembly program, the type system and the run-time stack, structured control instructions and their interactions with the stack, and finally, the structure of the text format along with some examples.

3.1.1 Program Structure

A WebAssembly program is written in a Lisp-like notation and is structured in multiple parts. The outermost layer is a *module*, which defines the global variables, linear memory, functions, imports, and exports of that module.

Global variables may be referenced from anywhere in the program. Linear memory is simply a linear array of bytes that may also be referenced from anywhere in the program. WebAssembly does not provide any structures on top of memory, leaving it for programs to use as they will.

WebAssembly functions are similar to those found in other programming languages. A WebAssembly function takes parameters, produces a result, and contains a sequence of instructions. A function may also define local variables that are only available from within that function. Many WebAssembly constructs, including variables, functions, and parameters, can be given a name and may be referenced by either their name or their index based on order of appearance in the program. Comments may also be present, but are stripped out in the conversion from text to binary.

The imports and exports of a WebAssembly program are how it communicates to its runtime. A program may import or export linear memory, global variables, and functions. For example, exporting a function allows a web browser's JavaScript runtime to call that function when it wishes. Importing a linear memory allows the browser's runtime to set up the memory that the program will access when it executes.

Below is a small WebAssembly program containing all of these parts. The module declares a linear memory with a size of at least one page. The module then imports a global variable from its runtime and exports a function for the runtime to call. That function takes one parameter and returns the sum of that parameter and the global variable. Note the

nesting of different WebAssembly structures and how names are used to refer to memory, global variables, local variables, and functions. Also note how the module is dependent on the runtime to provide the import for the global variable, as well as call the function the module exports.

```
(module
  ; This is a comment
  (memory $mem 1)  ; One page of memory
  (import "env" "wasm_global" (global $arg1 i32))
  (export "_start" (func $add_two))
  (func $add_two (param $arg2 i32) (result i32)
    global.get $arg1
    local.get $arg2
    i32.add))
```

3.1.2 Types and the Runtime Stack

WebAssembly defines four types: `i32`, `i64`, `f32`, and `f64`, which represent 32- and 64-bit formats of integers and IEEE floating-point values. The integer types represent both signed and unsigned integers; some instructions treat `i32` or `i64` values as signed integers and other instructions treat them as unsigned.

When a WebAssembly program executes, it may access and store values of the above four types on a run-time stack. We refer to instructions that pop values from the stack as *consuming* values, and instructions that push values onto the stack as *producing* values. For example, an addition instruction pops two values from the stack, performs the addition, and pushes the result back onto the stack: we say that the instruction consumes two values and produces one.

WebAssembly programs are strongly typed. In particular, when a WebAssembly instruction executes, the types of the values that the instruction consumes must match the types of the values at the top of the stack. For a binary operation (binop) such as addition, both of the values consumed must be of the types required by the instruction. The value produced will be of the type specified by the instruction. The name of every instruction that modifies the stack begins with a type prefix. For example, although addition is valid for every WebAssembly type, `i32.add` consumes and produces `i32` values, while `f64.add` consumes and produces `f64` values.

Most instructions produce the same type that they consume. There are a few exceptions to this general rule, mainly centering on comparison or test operations whose result is

"true" or "false." Because WebAssembly does not have a boolean type, it uses the `i32` type to represent boolean values. For example, an equality instruction consumes two values (as long as they have the same type) and produces one `i32` value.

3.1.3 Structured Control Instructions

WebAssembly is a structured language with functions and function calls, blocks, loops, and if statements. For structured control flow, functions comprise the outer layer, and within them are `block`, `loop`, and `if` instructions. The `block` and `loop` instructions allow controlling program flow by either jumping to the end of a `block` or to the beginning of a `loop`, while the `if` instruction allows conditional if/else logic. Tying structured control flow together is the `branch` instruction and some of its conditional variants. To enforce structured control flow, a branch targets a nesting depth instead of an instruction address in the program. Using C analogies, branches may break out of a block or continue a loop, but they may not perform a `goto`. To extend this analogy, not only may a branch break out of a block, but it may choose how many blocks to break out of.

Each structured control instruction must declare the types it takes as parameters and the type of its result. To simplify type-checking, each structured control instruction has its own run-time stack, meaning that a nested structured control instruction may access local and global variables, as well as the parameters provided to it, but its run-time stack is separate from that of its parent. For the parent, a structured control instruction is simply viewed as an instruction that consumes the values specified in its parameters and produces the value of its result type. In the case of `branch` instructions, when a branch targets a structured control instruction, it must produce the correct result type for the targeted `block` or `if`, or the correct parameters for the targeted `loop`.

3.1.4 Text Format and Examples

WebAssembly programs follow a Lisp-like format, with an identifier first and a list of arguments following it. For example, the arguments of a module are function definitions. The arguments to a function are parameters and a result type followed by a list of instructions.

Functions, local variables, and global variables can have names to aid a programmer in remembering their purpose, but can be referred to by their indices. For example,

when exporting a function so that an external runtime can call it, the `export` declaration can take either a name or an index: `(export "exported_function" $wasm_func)` or `(export "exported_function" (func 0))`.

Below is a modified version of the addition function from earlier. Each `local.get` instruction pushes a value onto the stack. The following `i32.add` instruction consumes those values and produces the result of the sum. Note that the function is exported by index instead of by name. Also note that there is no explicit return instruction. When a function reaches the end of its list of instructions, it returns the value left on the stack.

```
(module
  (func $simple_add_two (param $arg1 i32) (param $arg2 i32) (result i32)
    local.get $arg1
    local.get $arg2
    i32.add)
  (export "_start" (func 0)))
```

Here is a function that takes a float parameter x and executes a loop to compute and return $(x + 2) \times 10$. Note the branch referring to nesting level by index.

```
(module
  (func $loop_func (param $x f32) (result f32) ; Nesting level 1
    (local $counter i32)

    ; Counter setup
    i32.const 10
    local.set $counter

    ; Get initial parameter
    local.get $x

    loop (param f32) (result f32) ; Nesting level 0
      ; The parameter is placed on the stack at the start of
      ; the loop.
      f32.const 2
      f32.add

      ; Decrement the counter
      local.get $counter
      i32.const -1
      i32.add
      ; A 'tee' is a simultaneous 'set' then 'get'
      local.tee $counter

      ; At this point, the conditional and the
      ; parameter value are on the stack
      ;   - If the branch is taken, it consumes the parameter
      ;     for the start of the loop
      ;   - If not, the parameter is used as the result type
      ;     for the loop

      ; The branch instruction consumes the counter value.
```

```
      ; If non-zero, branch to the start of the loop.
      br_if 0

      ; If we get past the branch, the counter was zero.
      ; The f32 result is the only thing left on the stack.
      ; We end the loop with the correct result type
    end)
  (export "_start" (func 0)))
```

The equivalent C code is:

```
float loop_func(float x) {
  float result = x;
  int i = 10;
  do {
    result += 2.0;
    i--;
  } while(i != 0);
  return result;
}
```

3.2 Semantically Valid Program Generation

There are several problems Wasmlike must solve to generate usable programs for random differential testing. First, it must generate programs that produce meaningful results that can be compared between different systems under test. A meaningful result is one where we can obtain useful information about the execution of the program from the state of the runtime after execution. Next, it must enforce WebAssembly's type system as laid out in the specification. Wasmlike does this by creating an abstract syntax tree to keep track of the structure of the program being generated and to reason about the type system. Additionally, Wasmlike must also avoid undesirable behavior to get consistent results from generated programs by implementing static and dynamic constraints. Finally, Wasmlike must generate programs that are interesting and approximate real-world programs in order to find defects caused by optimizing compilers in the systems under test. Wasmlike does this by generating idiomatic constructs such as counting loops, i.e., "for loops."

3.2.1 Generating Programs for Random Differential Testing

Random differential testing presents some challenges for determining if a WebAssembly implementation encounters a defect when it runs a generated program. First, the test program must produce something meaningful. Second, the test program must be able to be invoked by multiple WebAssembly runtimes. Third, we must be able to differentiate the result of running a program on a runtime that contains a defect from one that does not.

When generating a program with a meaningful result, WebAssembly's structure gives us several inspection points. WebAssembly runtimes have access to the return value from a function call, any imported or exported global variables, and imported or exported linear memory. Wasmlike generates programs that contain a main function with a return value, an exported linear memory, and boilerplate to expose each generated global variable. If there is a difference in execution between two systems under test that impacts any of these components, we consider it a defect.

To differentially test different runtimes, we must be able to run a generated program on each system under test. To this end, Wasmlike will always generate a main function from which all other functions are called. Each system under test has a test driver to instantiate the generated WebAssembly module, call the main function, and collect the result.

Detecting defects in runtimes comes down to comparing the printed result of each system under test. To detect as many defects as possible, the test harness combines every piece of information available to the runtime into a CRC checksum value. This value includes the return value from the main function, the contents of linear memory, and the values of each global variable after execution. The test driver can access the return value and memory easily, but the number of global variables depends on the generated program. Wasmlike solves this by adding a small amount of boilerplate code to handle it and fold the global variable values into the CRC. This is the same style of program and defect detection method that Csmith uses [25].

3.2.2 Abstract Syntax Tree Generation

Wasmlike is an Xsmith-based random program generator. It uses the library provided by Xsmith to specify the grammar and type system of WebAssembly, construct the AST representing a random program, and output the generated program.

Wasmlike starts with a single root node in the AST representing the program as a whole. The root node has unspecified children, which Xsmith calls *holes*. Wasmlike builds the AST by finding a hole and filling it with a node that satisfies the grammar, type constraints, and semantics of the WebAssembly language until all holes are filled. At that point, it transforms the AST into the text representation of the program and outputs it.

The grammar is specified in a form reminiscent of eBNF. A node specifies its children, and what types its children can be. When Wasmlike fills a hole, it chooses a node that is of the type specified by the parent node.

Wasmlike starts with a program node. The only hole it can fill is that of the main function, which it does with a new function. Wasmlike then fills the root expression of the function, by randomly choosing among the possible kinds of expressions, e.g., binary operations (binops), function calls, variable references, and literals. If Wasmlike chooses a binop, it continues by choosing another subtype, such as an addition. The addition has two children: left and right expression holes. Wasmlike continues filling in holes until there are no more to fill. If Wasmlike chooses a literal value, it immediately chooses a random value to use when printing the completed program. Generated nodes can inherit children from their supertype. In the case of the addition node, the left and right children are specified in the binop type, so that all binops inherit the requirement of having two children.

Wasmlike can generate variable definitions and references as well. To add global variables to the grammar, Wasmlike adds a list of variable definitions to the root program node, and a variable reference node to the available generation node types. Wasmlike must also tell Xsmith that the definition node is a *binder*, and that the reference node is a *reference*. When Wasmlike chooses to generate a reference to a variable, Xsmith will either choose an existing binder or create a new one, a process called *lifting*. The mechanism of lifting needs types and names to be able to match new references to existing binders. In WebAssembly, global variables are initialized when they are created. Thus, a binder consists of a type, a name, and an initialization value, while a reference consists of the name of the binder, and depending on the type of use, an expression to set the value of the variable. We discuss types in more detail in Section 3.1.2.

The choice of where to lift a definition to can be nebulous. To add local variables as well as global variables, Wasmlike adds a list of variable definitions to the function node in addition to the program node. Wasmlike distributes the choice evenly between global and local variables. It is important that not all variables are global, because one of Wasmlike's goals is to create opportunities for optimizing compilers to perform optimizations, so that a testing process can potentially discover defects in those optimizations. If every generated variable was a global variable, our testing process would capture more information about

what happened during execution, but compilers treat global variables with care, meaning fewer optimizations would be applied during compilation.

Our example grammar specification so far is in Figure 3.1.

Wasmlike can make more complex trees than just a simple binop tree. To generate more than one function, the root program node still contains the main function, but adds a list of additional functions. Functions are lifted in a manner similar to local and global variables: a function call is generated, and either an existing function is chosen as the target, or a new function is generated along with its parameters, which are binders. Since there are two new types of binders and references, Wasmlike must tell Xsmith how to differentiate between functions and variables in order to produce semantically valid programs. Because a function parameter shares the same index space as a local variable, there is no difference between a variable reference to a function parameter, a local variable, or a global variable. When lifting a new reference, Wasmlike simply differentiates the type of reference by whether or not the reference is to a function. An example of this can be seen in Figure 3.2.

To make a well-typed WebAssembly program, Wasmlike follows type constraints during generation. The simplest of these constraints can be seen in binop: a binop may produce any type, and its two operands must match that type. Xsmith allows Wasmlike to express this in terms of relationships between a node and its children. For a binop, Wasmlike declares that it can fill any hole needing a value of any type, and that the two children must be of the same type as the binop node. This is expressed in the following code.

```
(add-property wasmlike type-info
  [Binop [(fresh-number) ;; What types of holes can this node fill?
                         ;; 'fresh-number' is a type-variable composed of
                         ;; all number types: i32, i64, f32, and f64.
          (λ (n t)       ;; A function that relates the node 'n' and its
                         ;; type 't' to a mapping of its children and
                         ;; their types.
             (hash 'l t
                   'r t))]]  ;; In this case, 'l and 'r must match the
                             ;; type of this node
  ...)
```

During program generation, multiple constraints may be applied to specific nodes. For example, a unary operator node has the type constraint that its child must match the type of the unary operator node itself. The "count leading zeroes" instruction, however, adds the constraint that the type of the node must be an integer type. The combination of these

```
(add-to-grammar wasmlike
  [Program #f ([globals : VariableDef *]
               [main : Func])]
  [Func #f ([locals : VariableDef *]
            [root : Expr]
            [name])]
  [Expr #f ()
    #:prop may-be-generated #f]
  [Literal Expr ([v = (random 0 10)])]
  [Binop Expr ([l : Expr]
               [r : Expr])
    #:prop may-be-generated #f]
  [Addition Binop ()]

  [VariableExpr Expr ()
    #:prop may-be-generated #f]
  [VariableDef #f ([init : Literal] ;;only used in globals
                   [type]
                   [name])
    #:prop binder-info (#:binder-style definition)]
  [VariableGet VariableExpr ([name])
    #:prop reference-info (read)]
  [VariableSet VariableExpr ([val : Expr]
                             [name]
                             [expr : Expr])
    #:prop reference-info (read)]
...)
```

Figure 3.1: A portion of Wasmlike's grammar specification defining literals, addition operators, and variables. Wasmlike's actual grammar specification has many more components.

```
(add-to-grammar wasmlike
  [Program #f ([globals : VariableDef *]
               [functions : Func *]
               [main : Func])]
  [Func #f ([params : Param *]
            [locals : VariableDef *]
            [root : Expr]
            [name]
            [type])
    #:prop binder-info (#:binder-style definition)]
  [Param #f ([type]
             [name])
    #:prop binder-info (#:binder-style definition)]
  [Call Expr ([function : FunctionReference]
              [argnode : Arguments])]
  [FunctionReference #f ([name])
    #:prop reference-info (read)]
  ...)

(add-property wasmlike
  lift-type->ast-binder-type
  [#f (λ (type)
        (if (function-type? type)
          'Func
          'VariableDef))])
```

Figure 3.2: An portion of Wasmlike's grammar specification supporting multiple functions and function references.

type constraints means that both the "count leading zeroes" instruction node and its child must be an integer type, since the child node's type is constrained to match the type of its parent.

The constraint of child nodes does not necessarily depend on the parent, and can instead be a constraint between children. For example, the WebAssembly specification states that a comparison operation will consume any two matching types, but will always produce an `i32`. To express this, Wasmlike defines a new type variable to hold the type information, then sets the type constraint of both children to the same variable, as shown in the code below. This effectively decouples the type of the two children from the parent.

```
[Comparison [i32  ;; This node can fill i32 holes
             (λ (n t)
                ;; Create a new type-variable
                (define child-type (fresh-number))
                ;; Assign the new type-variable to both children
                (hash 'l child-type
                      'r child-type))]]
```

Manipulating types through the unification of type variables is a common pattern throughout Wasmlike when dealing with more complex type interactions. Once two type variables are unified, constraining one will constrain the other. Take for example the type constraint of an argument list node, which is generated to fill in the arguments of a newly generated function call. To match the parameters in the function signature to the supplied arguments, Wasmlike represents the arguments as a list of expressions corresponding to each argument in order. Wasmlike unifies the type of each argument in the list with the type of each parameter in the function signature. Once unification is done, generating expressions for the arguments to the function call will update the function signature, keeping the types aligned.

```
[Arguments [(product-type #f)  ;; Any product type
            (λ (n t)
               ;; Get the argument list from the target function
               (define pt (product-type #f))
               (unify! pt t)
               (when (not (product-type-inner-type-list! pt))
                 ;; Force exploration of function node to fill in args list.
                 (att-value 'xsmith_type
                            (ast-child 'function (ast-parent n))))
               ;; Set the args for the call
               (for/hash ([arg (ast-children (ast-child 'args n))]
                          [arg-type (product-type-inner-type-list! pt)])
                 (values arg arg-type)))]]
```

This is important because Xsmith deals with constraints lazily. It only concretizes types from a set of possibilities to a single type when it must. The rest of the time, Xsmith views types as a collection of possible types for each node. This allows Xsmith to quickly expand the AST while maximizing the amount of randomness present during generation. By concretizing types as late as possible, Xsmith maximizes the number of choices it has for each node in the AST.

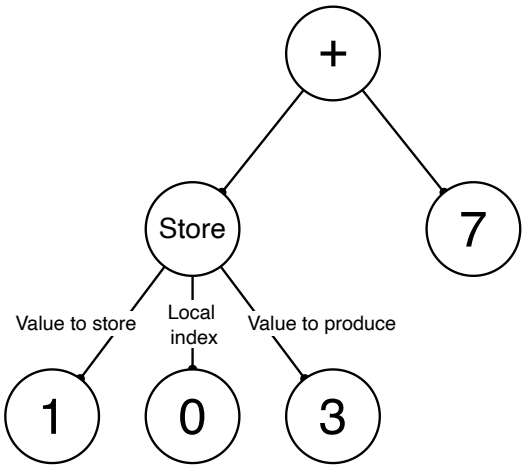
3.2.3 Wasmlike AST Generation Model

Wasmlike adds its own invariant during generation: each node in the AST that corresponds to an instruction must produce a single value on the run-time stack, regardless of what the represented WebAssembly instruction actually produces. This generation model allows Wasmlike to easily generate programs that use the run-time stack correctly, because Wasmlike only has to consider the current node being generated and its immediate surroundings.

Most instructions already fall into this model. Binary operations consume two values and produce one. Unary operations consume one value and produce one value. Even function calls produce one value. Some instructions, however, need to be adjusted so that their total effect on the stack is to produce one value.

Take for example the WebAssembly variable store instruction. A store instruction has two operands: the value to store, and the index or name of the variable to store into. When represented as an AST node, the store instruction needs two children: an expression child for the value to store, and the name of the variable. Because the store instruction does not produce a value, Wasmlike represents the store instruction in a novel way: it adds an additional expression child to the node to serve as the result of the node. This can be seen in Figure 3.3.

The generation model comes from the following reasoning. Wasmlike must generate functions with a "balanced" runtime stack, meaning that every generated value must be consumed, except for one, which is consumed by the function return. To generate code the balances the stack, Wasmlike can either generate a node that is locally guaranteed to be balanced, i.e., guaranteed to be the root of a subtree that yields a balanced stack, or it can generate some number of nodes that may be unbalanced and fix the imbalance later.



(a) AST subtree

```
;; Memory store node
i32.const 1
local.set 0  ;; Consume the 1
i32.const 3  ;; Produce a 3 for the parent

;; Literal node
i32.const 7

;; Addition node
i32.add      ;; Consume the 3 and 7
```

(b) Resulting instructions

Figure 3.3: An example memory store inside a small AST subtree.

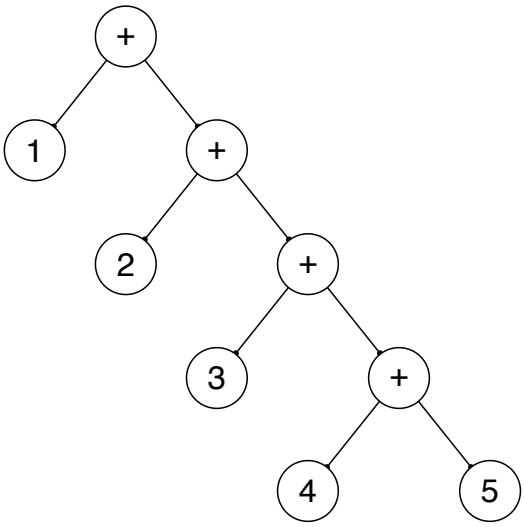
Generating a locally balanced node is the simpler option because reasoning about the state of the stack requires only considering directly adjacent nodes. Generating a truly random AST and fixing it is much more complicated. Every subtree in the AST must be balanced individually, since every instruction in the generated program must have its operands satisfied or be declared an invalid program by the specification. Fixing imbalances would mean mutating every level of the AST, not just adding to it. Because of the complexity of this approach, Wasmlike uses its generation model to guarantee the correctness of the stack without needing to mutate the AST after generation.

Because this invariant constrains programs more than the WebAssembly specification does, we must ensure that the invariant does not unduly constrain the shapes of programs being generated. As a simple proof of concept, let us take a program that sums up five numbers. One extreme is to place all five numbers on the stack, then sum them all by using four addition instructions. The other extreme is to sum them up as we go, starting with two numbers, an addition, and then an alternating sequence of numbers and additions until the five numbers are summed. Wasmlike is able to produce either of these two extremes, as shown in Figure 3.4.

When rendering an AST node into the text form of WebAssembly, Wasmlike prints the left child, then the right child, and then the parent. The corresponding ASTs of our two sum programs are long branches to the left or the right of addition nodes, surrounded by literal leaf nodes, as in Figure 3.4. Because Wasmlike can generate both extremes by changing whether the next addition in the chain is the left or the right child, we know that Wasmlike can generate any possible AST between the two extremes. From this, we can draw two conclusions. First, Wasmlike can generate any ordering of a sequence of instructions in the text format of a WebAssembly program. Second, because of this, we can be reasonably sure that Wasmlike's generation model does not constrain generated programs into similar looking patterns. As long as the type system is satisfied, any interleaving permutation of instructions is possible when generating programs.

3.2.4 Avoiding Undesirable Behavior

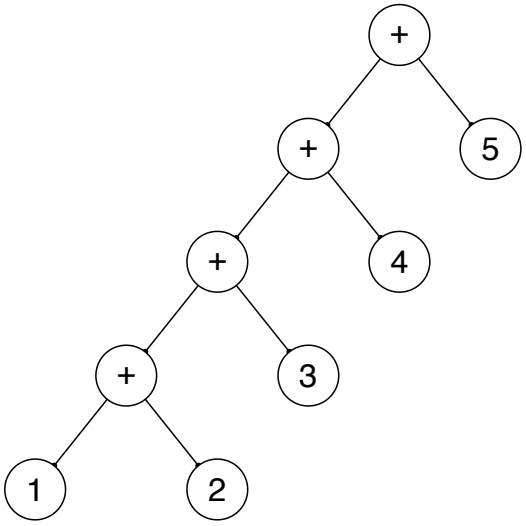
In addition to generating programs that are correctly typed, Wasmlike should also produce programs that are useful for random differential testing. There are several undesir-



(a) Push all literals, then sum

```
i32.const 1
i32.const 2
i32.const 3
i32.const 4
i32.const 5
i32.add
i32.add
i32.add
i32.add
```

(b) Resulting program from (a)



(c) Intermix pushes of literals and additions

```
i32.const 1
i32.const 2
i32.add
i32.const 3
i32.add
i32.const 4
i32.add
i32.const 5
i32.add
```

(d) Resulting program from (c)

Figure 3.4: Sum program extremes.

able behaviors to avoid in programs generated for differential testing: crashing programs, programs that do not terminate, and programs that use undefined behavior. To avoid these behaviors, Wasmlike uses a combination of static and dynamic constraints. A static constraint is enforced during generation, and is related to the choices that Wasmlike makes, such as constraining the range of a generated memory address. A dynamic constraint is code that is inserted into a generated program to detect and correct undefined behavior during execution.

The first undesirable behavior Wasmlike avoids in generated programs is crashes. It does this by using static and dynamic constraints during program generation and when the program is run. Static constraints are used when the behavior of the generated program can be determined during generation, like an out-of-bounds memory address, and dynamic constraints are used when the behavior can only be detected at run time, like a division by zero. Wasmlike's constraint on memory addresses is twofold. First, memory addresses are always generated directly as a value, never as an expression resulting in a value. Second, Wasmlike controls the range of possible memory addresses to always be in bounds. Generating the value directly allows for the static constraint during generation instead of a much more complex dynamic check during run time. Controlling the range directly lets Wasmlike do interesting things like reuse sections of memory or encourage the use of memory addresses that overlap. For instructions that would cause a run-time error, like a division by zero, Wasmlike inserts a dynamic constraint into the generated program that detects problematic operands and replaces them with similar values that will not cause the program to crash.

The second behavior we want to avoid is programs that do not terminate. We chose not to guarantee this property, in order to simplify Wasmlike and give it freedom to generate programs. Nevertheless, Wasmlike takes steps to mitigate generating these kinds of programs. First, it limits the AST depth of generated programs. By choosing a depth constraint that produces programs where most will finish quickly, it is unlikely (but still possible) that a program will terminate on one runtime but not on another. Second, Wasmlike avoids generating directly and indirectly recursive function calls. This limits the program space that Wasmlike can explore with random generation, but the likelihood of randomly generating a recursive function call that also contains a proper exit condition is

small. A recursive function call will most likely run until the program is terminated by a stack overflow, so Wasmlike enforces a static constraint of not allowing recursive calls.

The third behavior we want to avoid is programs that use undefined and nondeterministic behavior. Some floating point operations, such as taking the square root of a negative value, result in a NaN value, and the WebAssembly specification allows WebAssembly implementations to produce differing binary representations of NaN (i.e., non-canonical NaN values). Because of this, two runtimes that correctly execute a program that produces NaNs may yield two different checksum values (Section 3.2.1). To avoid these situations and produce programs useful for differential testing, Wasmlike inserts dynamic checks into generated programs in the same manner as before: one that detects problematic operands and replaces them. Another nondeterministic behavior we want to avoid in generated WebAssembly programs is when a program is unable to grow its memory because the operating system has run out of available memory to allocate. The fix is a simple static constraint to never generate the `memory.grow` instruction. Wasmlike already constrains memory addresses to a static range, so even if a `memory.grow` instruction was generated, the extra memory would go unused. Table 3.1 presents a full list of undefined behaviors along with Wasmlike's constraints to avoid those behaviors.

3.2.5 Generating Interesting Programs

To exercise the systems under test to their fullest, Wasmlike needs to generate programs that are interesting. For Wasmlike, this means generating programs that are non-trivial, varied in their layout and structure, use a variety of language features, mimic constructs of real world programs, and test edge cases. Wasmlike achieves this by using choice weights to guide program generation, a cumulative function depth limit to encourage function variety, weighted function reuse, idiomatic expression nodes, and weighted literal generation that prefers edge cases like maximum and minimum values.

Choice weights are the primary way in which Wasmlike avoids generating non-trivial programs. For Wasmlike, trivial programs are programs that are extremely short or programs that only use one type of instruction. For example, a program that immediately returns a constant value is trivial, as is a program that only uses the addition instruction. We want programs that are large and complex, to allow the compilers of various systems

Table 3.1: Static and dynamic generation constraints.

Undefined Behavior	**Constraint Type**	**Constraint**
Division by 0	Dynamic	Replace divisor with 1
Division of MININT by -1	Dynamic	Replace divisor with -2
Remainder of division by 0	Dynamic	Replace divisor with 1
Square root of a negative value	Dynamic	Apply the absolute value operator to the radicand
Out of bounds memory access	Static	Generate the memory address to always be in bounds
Out of memory	Static	Never generate `memory.grow`

under tests the opportunity to apply optimizations so that we can detect more defects. To avoid extremely short programs, Wasmlike sets the weight of terminal expressions low. To avoid extremely large programs, Wasmlike sets a depth limit, past which only terminal expressions are allowed to be generated. The combination of these two constraints produce large, varied programs. If Wasmlike were not allowed to generate terminal expressions until the depth limit was exceeded, the generated programs would look very similar: a uniformly filled out AST.

Xsmith makes choices in stages. First, it filters the nodes it is allowed to generate based on the type information of the hole it is trying to fill. For example, a node that produces an integer is not allowed to fill either child hole of a floating-point addition. After filtering, Xsmith chooses a node based on the declared choice weight. This allows easy tunability by allowing Wasmlike to specify how often a particular node is generated, while letting Xsmith ensure the correctness of the generated program as a whole. In the example of choice weights below, Wasmlike is more likely to generate a variable expression than a literal expression because of the sum of the weights of each category. When generating a variable expression, Wasmlike is more likely to get the value of a variable rather than set it.

```
(add-property wasmlike choice-weight
  ;; Variable Expressions
  [VariableGet 15]
  [VariableSet 10]
  [VariableTee 10]
  ;; Literal Expressions
  [LiteralIntThirtyTwo 5]
  [LiteralIntSixtyFour 5]
  [LiteralFloatThirtyTwo 5]
  [LiteralFloatSixtyFour 5]
...
```

Cumulative function depth is used to help generate functions with a variety of sizes. Wasmlike allows only a certain number of functions to be generated with the full depth limit available to them. Past that, every additional new function has its available depth reduced. This results in programs that are large but with a spread of large and small functions, similar to real-world applications.

To mimic real-world use cases of functions, Wasmlike encourages function reuse through weighted choice. When a function call is generated, Wasmlike has the choice to either make

a new function for the function call or to reuse an existing function. The available functions for reuse are filtered to avoid recursive calls, but this choice weight allows Wasmlike to generate more "interconnected" programs.

Idiomatic expressions are represented as single nodes in the AST, but are made up of multiple WebAssembly instructions. For example, Wasmlike can generate an idiomatic "for" loop construct consisting of a counter variable, an input parameter, a body expression that uses the input parameter, and a loop instruction. Generating a working "for" loop through random chance is very unlikely because of all the necessary individual components that have to line up, so this allows Wasmlike to generate loops consistently.

Wasmlike also generates minimum and maximum values for integers to test the edge case handling of systems under test. This is controlled by another weighted choice during generation. It is important to note that in WebAssembly, an integer literal is neither signed nor unsigned; instead, the instruction that uses the integer literal interprets the value as either signed or unsigned. Because of this, Wasmlike's integer literal generation does not target a particular instruction, but instead weights the generation of literal values to produce a small proportion of bit patterns corresponding to the maximum and minimum signed and unsigned integer values.

3.3 Current State of Wasmlike

Wasmlike supports all four WebAssembly types: i32, i64, f32, and f64. It supports binary and unary arithmetic operations, comparisons, if/elses, branches, blocks, loops, type conversion instructions, functions, direct function calls, memory operations,loads and stores from memory, and local and global variable operations. Wasmlike is able to generate programs in a restricted mode without any floating point literals or operations.

3.3.1 Limitations

There are a few limitations to the programs that Wasmlike can generate, mainly centered around certain instructions.

Wasmlike does not generate the `unreachable`, `br_table`, `return`, `call_indirect`, `drop`, `select`, `memory.size`, or `memory.grow` instructions.

The `unreachable` instruction always results in a trap during execution, which makes it harmful for differential testing since the behavior of this instruction is identical to a program that crashes.

The `br_table` and `call_indirect` instructions are layers of indirection for branch instructions and function calls. Implementing these instructions in Wasmlike would involve additional complexity to properly generate and link the necessary tables when the end result would be a branch or a function call based on a random input: something that Wasmlike already does.

The `return` instruction is never generated for the simple reason that it is an alias for a branch instruction that targets the function level.

The `drop` and `select` instructions are never generated directly because they are primarily used to discard items from the stack without producing new results. To better detect semantic defects, we prefer to use data on the stack as inputs to further computation. However, these instructions are generated as a necessary part of idiomatic expressions such as loops, conditional branches, and dynamic checks to avoid undefined behavior. Since Wasmlike only needs to guarantee the overall behavior of an AST node to maintain its generation model, the uses of `drop` and `select` are balanced by special-purpose code in Wasmlike.

The `memory.grow` instruction is avoided because it is nondeterministic. WebAssembly runtimes may not provide the additional requested memory to the module. The result of this is well-defined, but still nondeterministic. Additionally, memory access patterns are restricted to a small portion of available linear memory in the module to encourage overlap, making more memory not useful. Finally, multiple tests and runtimes are run in parallel on the testing machines. Running out of memory would result in unpredictable behavior. For example, a generated program that contains a `memory.grow` instruction inside an infinite loop could consume all available memory on the machine, leading to other systems under test failing in non-reproducible ways.

The `memory.size` instruction is avoided because if growing memory is avoided, the result of this instruction is always the same.

In addition, Wasmlike does not support newer WebAssembly proposals like multi-value returns for functions and structured control blocks. This is for two reasons: first,

because Wasmlike was under development before these proposals were approved and merged into the WebAssembly specification, and second, because of technical constraints. Wasmlike's AST generation model does not allow for multiple values produced from a single instruction. Furthermore, the programs Wasmlike generates contain a starting function that returns only one value, meaning multi-value returns would need to be condensed down to a single value. Supporting the multi-value proposal would introduce a lot of complexity to the implementation of Wasmlike for no discernible change in testing behavior.

CHAPTER 4

TESTING WEBASSEMBLY IMPLEMENTATIONS

4.1 Systems Under Test

The systems under test are Node.js [16], Wasmer [23], Wasmtime [5], Firefox [15], and Chromium [21]. Node.js is a JavaScript runtime that can directly run WebAssembly programs. Wasmer is a fully featured WebAssembly implementation with configurable optimization levels and compiler choices. Wasmtime is a JIT-style compiler focused on fast compilation speed. Firefox and Chromium are web browsers. The two browsers are run using an automated testing tool, Playwright [14]. Node.js and Chromium share the same underlying engine for running WebAssembly. The goal of including both is to test that the other components of Chromium or Node.js do not contain defects related to WebAssembly.

For each system under test, we created a small interface program to compile and execute a generated WebAssembly program and output the result. The compiler and runtime API for Wasmer and Wasmtime are accessed through Rust. For Node.js, Firefox, and Chromium, the API is accessed through JavaScript. The API for automating web browser testing with Playwright is similarly accessed through JavaScript.

4.2 Testing Process

The testing process for the systems under test is split into three major sections. The first is configuration, where the specific steps of running each system under test is defined. The configuration also defines the program-generation options given to Wasmlike as well as other necessary infrastructure for running the systems under test. The second is the differential testing phase. Here, a generated program is run with each system under test in turn. The output is collected and compared, and the test result is logged. The third section is reducing and reporting test cases that trigger defects in the systems under test. Those test cases are manually verified, reduced to a reportable size through a combination

of reduction tools and by hand, and reported to the developers of the system displaying the defect.

4.2.1 Configuration

The configuration of each system under test is flexible in regard to the sequence of tools and commands that make up that system. The configuration file details startup steps, shutdown steps, program generation options, and the steps to run each system under test.

The startup section in the configuration details any commands that need to be run before a testing campaign such as compiling the program generator, compiling system interfaces, or starting local web servers. The shutdown section contains commands for safely tearing down the testing campaign, like shutting down web servers and creating a backup of the log file with the test results. The program-generation options are fixed for all systems under test for the testing campaign, and influence the size of generated programs as well as the WebAssembly features that may be used in those programs, e.g., floating point operations. Each system under test is represented as a list of commands that will be run, with the standard output of the last command collected as the result of the test. In this way, a single runtime platform with different optimization levels, like Wasmer with and without optimizations, is considered to be multiple systems under test.

The list of systems under test is in Table 4.1. Node.js, Firefox, and Chromium are all tested without any additional optimization or runtime configuration. The systems under test for Wasmer vary between a Cranelift or LLVM based compiler, Wasmer's universal or dynamic engine, and whether or not optimizations are applied. Wasmtime only varies with optimization. For all systems under test, if optimization is an option, it is either turned up to the most aggressive setting available or down to no optimizations at all.

4.2.2 Differential Testing

The WebAssembly programs that Wasmlike generates are combined with helper functions designed to interact with the interfaces for each system under test to produce a checksum after execution. This checksum is computed with the return value of the main function, the values of any global variables, and the contents of linear memory. Each system under test is subject to a timeout: if it does not produce a checksum before the timeout occurs, the testing harness terminates the system under test and records that it

Table 4.1: Systems under test.

System	Compiler	Engine	Optimization
Node.js 16.16.0	-	-	-
Wasmer 2.3.0	Cranelift	Universal	No
Wasmer 2.3.0	Cranelift	Universal	Yes
Wasmer 2.3.0	LLVM 12	Universal	No
Wasmer 2.3.0	LLVM 12	Universal	Yes
Wasmer 2.3.0	LLVM 12	Dynamic	No
Wasmer 2.3.0	LLVM 12	Dynamic	Yes
Wasmtime 0.1.0	-	-	No
Wasmtime 0.1.0	-	-	Yes
Firefox 108.0.2	-	-	-
Chromium 110.0	-	-	-

timed out. The timeout value is generally set to a small number of seconds: enough for the most complex system under test to start up and execute the test program, plus a few seconds as a buffer.

Differential testing is performed by giving each system under test the same WebAssembly program and comparing the outputs as illustrated in Figure 4.1. There are five main categories of results, with each having its own severity: *normal*, *crash*, *timeout*, *inconsistent timeout*, and *wrong code*. A *normal* result means that each system under test successfully executed its input, and the checksums from all of the systems were identical. A *crash* indicates that at least one system under test crashed when executing the generated program. A *timeout* indicates that the generated program did not finish for any of the systems under test. An *inconsistent timeout* is when some systems under test successfully execute their program and return a result, but others time out. The final category, *wrong code*, always indicates a defect in at least one system under test, and is signaled when the checksums from the systems under test are not.

Normal and timeout results do not indicate any defect. A normal result means that all of the systems under test produced the same output. A timeout result typically indicates that the test case contains an inadvertent infinite loop. A majority of the results from a testing campaign will be in these two categories. Inconsistent timeout outcomes either indicate that (1) the generated program has a section that was slow but was optimized by some of the systems under test, or (2) that there was a defect related to a loop being treated as infinite on one system but not the other. The vast majority of inconsistent timeouts will be from the difference between an optimized executable and an unoptimized executable, where both will return the same result given sufficient time. These results may be examined further to find test cases where a timeout occurs on an optimized version instead of unoptimized versions, but to date, we have not found any such test cases.

The final two categories, crashes and wrong code bugs, signal a defect in one or more of the systems under test. Wasmlike is designed to avoid generating programs that crash, such as programs that divide by zero, meaning that a crash is most likely the result of a defect present in a system under test. A wrong code result indicates wrong code generation in at least one of the implementations since Wasmlike is also designed to avoid generating nondeterministic programs.

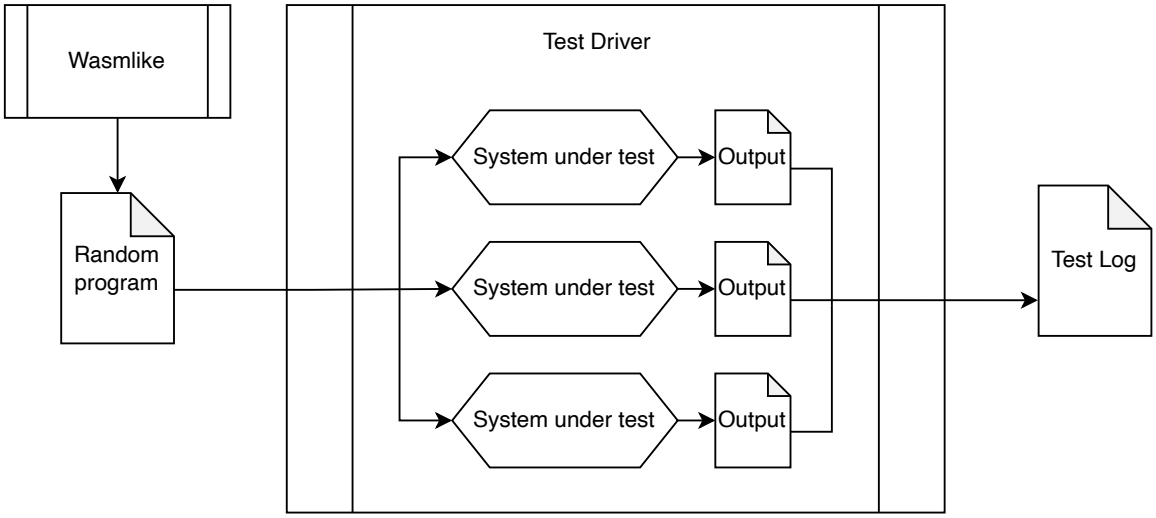


Figure 4.1: Testing procedure.

The test driver reports the result from a test case as a log entry containing the test outcome as well as the information needed to reproduce the test case. For Wasmlike this information includes the random seed, the generation options, and the version of Xsmith and Wasmlike used. The choice to use log entries instead of saving the program directly was made to avoid filling up space on the testing machines as well as being able to collect, sort, and examine the test results on a central machine.

4.2.3 Reducing and Reporting Test Cases

A test case can be reproduced entirely from information present in its log entry. A helper script is invoked to quickly convert the generation options from a log entry back into the generated program. From this point, the defect must be verified, reduced, and reported.

Verification is simple. When the helper script is passed the program generation options along with the names of systems under test that differed in their output, the script will regenerate the program, run the specified systems under test with the generated program, and print the checksum of each. If the printed checksums differ, we have verified that the defect occurs. While this step seems redundant, we have encountered extremely rare occasions in which a wrong code log entry fails verification. If Wasmlike generated nondeterministic programs, this would happen much more frequently since every nondeterministic instruction would cause this behavior. Instead, our best guess is single-bit memory flips because the checksum is sensitive to a single bit difference in a system under test's linear memory contents.

Reduction is performed by giving the generated program to a WebAssembly program reducer [3]. The reducer pares down the size of the test case by using a predicate to determine if a reduction step is still interesting. For crash bugs, the predicate is that the crashing system outputs the same crash message and that another system produces a checksum. For wrong code errors, the predicate is that the systems under test produce different results. Note that the predicate is not that the systems under test each produce the same output as the original test case, since all this would accomplish is removing dead and unreachable code. The reducer used is not as aggressive as needed to produce a minimal test case directly, but empirically, it will very reliably shrink the test case size by at least

60%. The last reduction pass is done manually by replacing structured control instructions with their result types, forcing early returns from functions, and generally eliminating big sections of code before moving on to the instruction level and eliminating instructions that do not seem to be the cause of the wrong-code error.

Once a test case is reduced as far as possible, it may be reported to the developers of the system under test. Determining the system under test that is ultimately responsible for the defect is not an easy question to answer. If the reduced test case is able to be manually checked, we can definitively blame a system under test. This is not always the case, however, especially with memory errors. Having many systems under tests makes determining blame easier, since if the difference in checksums is split with only one system producing a different checksum, we can confidently predict that the divergent system under test contains the defect. If the split in outputs is more varied, we first look at optimizing compilers, since they are more likely to contain defects than non-optimizing compilers. Then, if the blame is still not clear, we look at the groupings of output checksums. If systems under test from different developers (such as Wasmer and Firefox) agree on the result, then they are more likely correct than systems under test from the same developer (such as Node.js and Chromium).

When reporting the defect, if the behavior can be reproduced by using a standalone version of the system, like a command-line runtime, that option is preferred. If the defect cannot be reproduced by a standalone runtime, which is often the case with memory operation errors that rely on a test driver to checksum the contents of memory, a copy of the testing interface is included with purpose-built instructions and commands to reproduce the behavior.

4.3 Test Harness

A single instance of the test driver runs tests sequentially. The test driver can be run on a distributed cluster and configured to report results to a central node. When run in this distributed setup, we refer to the whole system as the test harness. The flow of data through the experiment nodes and systems is illustrated in Figure 4.2.

We designed the test harness to coordinate multiple machines, with each machine running multiple instances of the test driver. The test harness is designed to run on Em-

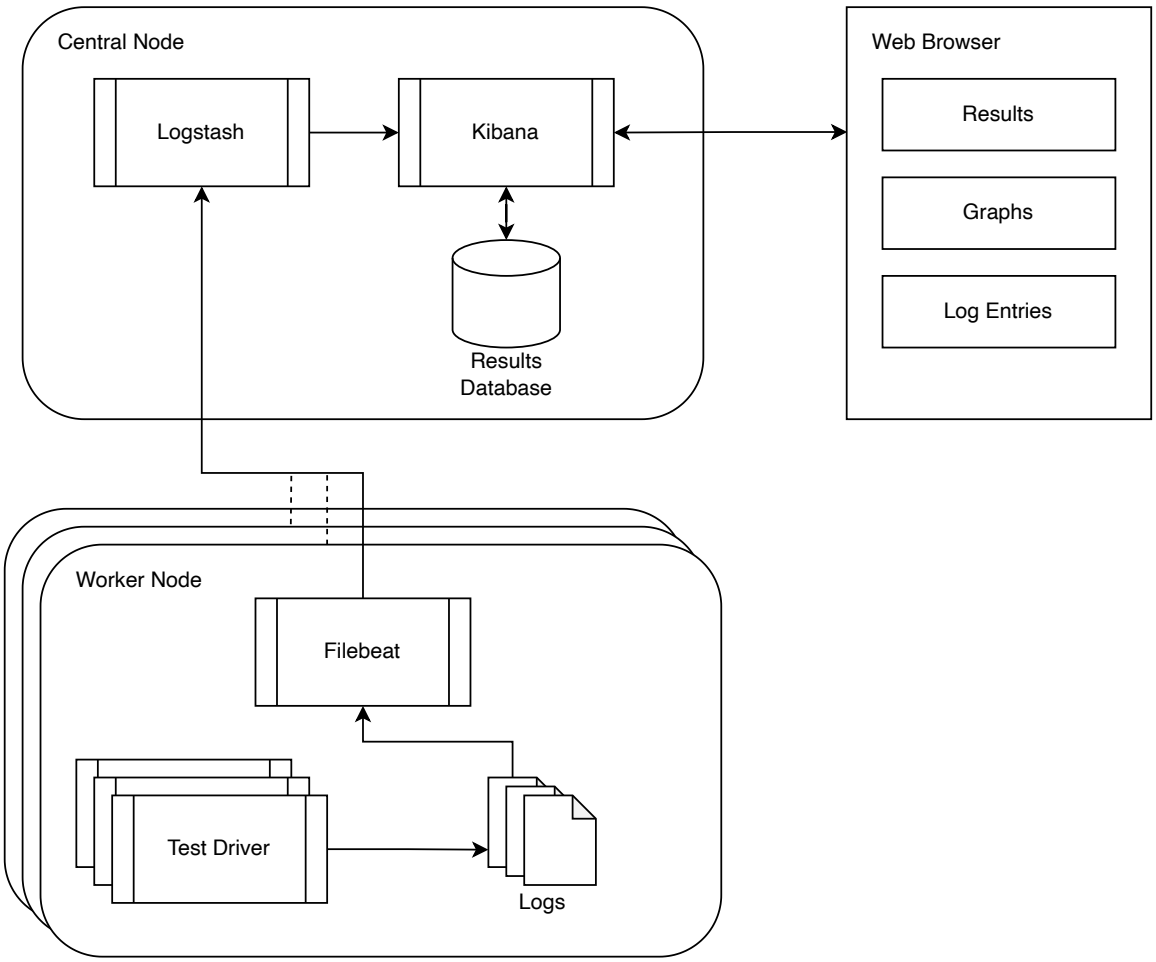


Figure 4.2: Organization of the test harness.

ulab [22] machines, with a central node responsible for collecting, sorting, and displaying the test results, and a configurable number of worker nodes responsible for running tests.

The test harness uses Ansible [19], an IT automation tool, to coordinate and distribute configurations to the machines as well as start multiple instances of the test driver per machine. A large part of the test harness is ensuring that different instances of the test driver do not overwrite each others' tests in progress. The test harness uses ElasticStack [9] to collect the generated log records from each test driver in a central database. The different components of ElasticStack relevant here are Filebeat, Logstash, and Kibana. Filebeat monitors the log files generated by the test drivers on each node and sends new log entries to Logstash on the central node. Logstash is used to tag each log entry with its result and filter out already-known false positives and defects related to the systems under test. The output is sent to the Kibana database, where the user can connect to the central node with a web browser to examine the collected data.

CHAPTER 5

RESULTS

To date, we have found and reported five defects that we discovered by testing WebAssembly implementations with semantically valid test cases generated by Wasmlike. These defects are summarized in Table 5.1 and reproductions of the test cases used in the bug reports can be found in Appendix A.

5.1 Defects Found

The first two defects found — the first two rows in Table 5.1 — were in Wasmer. They were wrong code defects related to rotating an integer value by zero bit positions, which led to incorrect output when running the test case. This is not something that a compiler would generate if asked to convert a program written in a higher-level language like C to WebAssembly, since a rotate by zero is effectively a no-op. The first of the two defects was triggered on any rotate by zero instruction, while the second was only triggered on an `i64` rotate-right instruction. The reduced test cases are presented in Figure A.1. As a result of these bug reports, the Wasmer developers added a fuzzer to their project to catch and correct similar edge cases.

The next defect found (Table 5.1, third row) was a compiler optimizer crash, again in Wasmer. The LLVM IR generated by Wasmer was incorrectly handling a branch inside of a loop, causing a crash during compilation when attempting to perform optimizations. At one time, this crash appeared in around ten percent of the programs being generated by Wasmlike. The behavior of the crash in the test harness is very distinct, both in the output and in the single system under test that crashed. Once the defect was reported, a filter was installed in the test harness to label future instances of the defect as a known bug. The reduced test case can be seen in Figure A.2.

The next two defects (fourth and fifth rows of Table 5.1) appeared only once each in a test run of 100,000 generated test cases. The first is a wrong-code error related to Cranelift,

Table 5.1: Discovered and reported defects.

System	Defect Synopsis and Link	Fixed
Wasmer 1.0.2	Rotate left wrong code error https://github.com/wasmerio/wasmer/issues/2143	✓
Wasmer 1.0.2	Rotate right wrong code error https://github.com/wasmerio/wasmer/issues/2215	✓
Wasmer 2.3.0	LLVM optimization crash https://github.com/wasmerio/wasmer/issues/3190	✓
Wasmer 2.3.0	Cranelift and LLVM wrong code error https://github.com/wasmerio/wasmer/issues/3251	
Wasmer 2.3.0	Three way wrong code error https://github.com/wasmerio/wasmer/issues/3323	

a WebAssembly-specific optimizing compiler. The defect is notable simply because no other defect was found that included Cranelift. The reduced test case is found in Figure A.3.

The second defect was a three-way difference in the output checksum between Wasmer's dynamic library engine with LLVM optimizations, Wasmer's universal engine with LLVM optimizations, and any other implementation such as Node.js or Wasmer with no optimizations. The reduced test case is quite long, mainly because any further reduction results in either the code generation defect not being triggered, the three-way difference turning into a two-way difference, or a crash. The reduced test case can be seen in Figure A.4.

Table 5.1 contains the state of each bug report. The first three defects (both rotate by zero defects and the LLVM optimization crash), have been fixed, although the LLVM optimization crash fix has yet to be merged into a versioned release of Wasmer. The last two defects (the Cranelift-related defect and the three-way code generation defect) have not been fixed yet.

5.2 Testing Metrics and Throughput

To maximize the number of defects found, the testing harness must be able to generate and test programs quickly. There are two main bottlenecks in the testing process: generating a program for each test, and running each system under test with each generated program.

Generation speed is directly related to the depth of the AST generated by Wasmlike. Each level of depth in a function can potentially double the size of the generated program if all interior AST nodes have two children, such as binops. This more than doubles the time needed to generate the program, since parts of Wasmlike that use attributes or variables have a larger AST to consider for every choice made. In practice, a good choice for the maximum AST depth seems to be nine. This depth consistently produces programs that are large, have a variety of function depths, and have a generation time of a few seconds per test. Increasing the maximum AST depth even by one increases the generation time from a maximum of around eight seconds to upwards of forty.

Running each system under test with a generated program is the second major bottleneck in the testing process. Some systems under test, like the browsers, need much longer to start up before returning a result. To address this, we attempt to use the resources of the testing machines to the fullest. Instead of running each system under test for a particular generated program in parallel, we run multiple instances of the test driver, which executes each system under test in serial. This approach has several advantages over a directly parallel approach. First, it simplifies the test driver and allows the test driver to easily sandbox each system under test. Second, it allows us to load balance a testing machine by specifying how many test drivers we want active at once. To minimize the bottleneck as much as possible, the number of test driver instances is set to a few less than the number of processor cores on the testing machines, to allow for network operations, logging, and additional overhead.

A large part of the testing harness is designed to take advantage of the configurable number of nodes in a testing campaign. Since each node contains everything necessary to run a test, from program generation to result, increasing throughput simply means scaling up the number of nodes in the experiment. This is also the primary reason for using Elastic Stack: it is designed to scale extremely well and lets us connect a variable number of nodes together with a minimal amount of configuration. Since test results are collected on a central node, viewing test results is the same when using two nodes as using twenty.

5.2.1 Analysis of Wasmlike Generation Speed

While we would like Wasmlike to generate programs quickly, Wasmlike is not focused on speed, since its primary purpose is to generate semantically valid test programs. That said, a program generator that is extremely slow is not very useful. To test the speed of generation, we ran Wasmlike in the test harness with the systems under test as a pair of simple `echo` commands to be able to measure the rate at which it generates programs.

The experiment setup is for the most part system-agnostic. More powerful machines will simply perform a testing campaign faster. The machines used for running the experiment are Dell PowerEdge R430 server provided by Emulab [22], otherwise known as “d430 nodes,” each with two 2.4 GHz 64-bit 8-Core Xeon E5-2630v3 processors, 64GB of RAM, and a 200 GB 6Gbps SSD. The CPUs are hyperthreaded for a total of 32 logical cores per

machine. To test generation speed, we use one node as a central database node, and one node to run 30 instances of the test driver, leaving 2 cores available for network operations, logging, and additional overhead.

Wasmlike generates approximately 25,000 random programs per hour with the same generation settings that were used to discover the defects listed in Table 5.1.

5.2.2 Analysis of Test Harness Throughput

Next, we test the throughput of the test harness with the systems under test as the full set of systems listed in Table 4.1, with the same d430 nodes as before. Except for the systems under test, the experiment setup is identical to the setup in Section 5.2.1. The test harness completes approximately 5,000 tests per hour, or around 80 tests per minute.

The proportion of result types from the test harness after a sufficiently long testing campaign is around 87% normal execution results, 9% results that match a previously discovered and reported defect, and around 4% time out results.

5.3 Defects Not Related to Program Generation

During the course of running fuzzing campaigns, we discovered and reported two additional defects that, while interesting, are not directly related to the generation of semantically valid programs. The defects are summarized in Table 5.2.

The first defect (first row in Table 5.2) involved Wasmer engine validation. When selecting systems under test, we created configurations with the different runtime options offered by Wasmer. One of the options selects a static library engine, designed to run precompiled WebAssembly executables. When asked to compile a WebAssembly program to an executable with the static engine, the Wasmer API would panic and crash instead of reporting a reasonable error message. We reported this bug and the developers added extra validation to their compilers and engines.

The second defect (second row in Table 5.2) was related to the WebAssembly program reducer created by Binaryen [24] that we used for creating bug reports. To create idiomatic counting loops in programs, Wasmlike uses syntax defined in the multi-value proposal for WebAssembly, which is now approved and part of the language. Due to a limitation of the reducer's IR, it could not represent the construct appropriately. This is a known limitation,

Table 5.2: Discovered and reported defects not related to program generation.

System	Defect Synopsis and Link	Fixed
Wasmer 2.0.0	Wasmer API crash when compiling with staticlib https://github.com/wasmerio/wasmer/issues/2590	✓
Binaryen 101	Unclear IR limitation error message https://github.com/WebAssembly/binaryen/issues/5047	

but we were not aware of it until after we submitted a bug report. The developers of the reducer agreed that the error message could be more detailed.

5.4 Discussion

5.4.1 Distribution of Defects Found

A notable outcome of our fuzzing campaigns is that all of the defects found by Wasmlike have been in Wasmer, using either the LLVM or Cranelift compilers with optimizations. Although this is conjecture, this result is possibly due to other implementations using more stable tooling. Node.js and Chromium both use the V8 engine, which already handles compiling and optimizing JavaScript programs to machine code. Adding the ability to compile and optimize WebAssembly programs is not a big jump in capabilities, given that once the WebAssembly program is converted to an IR form, the rest of the toolchain remains the same. Firefox uses the SpiderMonkey engine, which performs a similar role to V8. Wasmtime is built on Cranelift, which has a smaller scope than LLVM's modular system. Cranelift is an optimizing compiler specifically for WebAssembly, but does not perform mid-level optimizations, resulting in fewer corner cases and a more robust compiler.

5.4.2 Effectiveness

Wasmlike was built with a focus on generating semantically valid programs to discover defects related to wrong-code generation. Effectiveness, in Wasmlike's case, means finding defects that are both semantic in nature and difficult to find manually.

For finding semantic defects, Wasmlike performs well in our experience. Generated programs are semantically valid, which allows efforts to be focused on testing the semantics of WebAssembly implementations instead of having generated programs rejected by the compiler for being invalid. Four out of five of the defects described in Section 5.1 are wrong-code errors, meaning that Wasmlike successfully found semantic defects. In addition, most of the defects found are related to optimization, an area that is difficult to fully test because of the size and complexity of modern optimizing compilers.

In the area of finding defects that are difficult to find manually, we conclude that Wasmlike performs extremely well. It found defects that could be considered an implementation oversight as well as finding defects reliant on extremely convoluted chains of operations.

The first two defects found by Wasmlike were "rotate by zero" bugs in Wasmer, reproduced in Figure A.1. These small programs took a literal, and rotated that value by zero bits, effectively doing nothing. This defect would likely not have been found by compiling a program in some other language to WebAssembly, but because Wasmlike generated the WebAssembly directly, the defect was able to be discovered.

The next three defects, reproduced in Figures A.2, A.3, and A.4, are composed of chains of seemingly unrelated instructions that result in either a crash or a wrong-code error. Finding these defects manually would be nearly impossible given their complexity and how specific the triggering test cases are. In particular, the three-way wrong-code bug is so specific that changing any of the lines or even adding an `unreachable` instruction to the unused branch of an if statement results in the three-way difference changing to either a crash, a two-way difference, or no difference at all.

The specificity of the last three defects found by Wasmlike highlight the usefulness of a program generator that produces semantically valid test programs. This allows test cases to penetrate deep into the implementation of a compiler and test the optimizer. These defects would likely not have been found with a random mutational fuzzer.

5.4.3 Limitations and Trade-Offs

Wasmlike generates programs that assume an external API is present for testing. In particular, the generated programs export their linear memory, import an external function, and call that function for each generated global variable. The responsibility of computing a CRC value from the contents of memory and the global variables is left to the boilerplate written for each system under test. There are a few reasons for this design, as opposed to keeping the CRC computation purely in WebAssembly and injecting it into a generated program. The first and primary reason is for ease of development. Writing code for the CRC computation in Rust or JavaScript is more flexible than either writing the computation in some higher level language and compiling it to WebAssembly or writing it in WebAssembly directly. Being able to debug the CRC computation and verify that it is working correctly is invaluable. The second reason for relying on a WebAssembly API is that most contain options to modify compiler behavior, such as optimization levels. While this means that the generated programs are simpler, it also means that standalone runtimes

designed to directly run WebAssembly programs, like the Wasmer command line interface or the WebAssembly reference interpreter, are not compatible with the programs Wasmlike generates. Future work on Wasmlike could address this by moving the CRC computation into a WebAssembly snippet that is purpose-built to be injected into generated programs.

There are some limitations with the reduction of programs that trigger defects. The main limitation is that the reducer built by Binaryen does not support the multivalue proposal for WebAssembly, which Wasmlike uses as part of an idiomatic loop. Instead, we use a less effective reducer made by Bytecode Alliance [3] that requires some manual pruning to reduce a test case from mostly reduced to the smallest possible reduced size. The trade-off is more than worth it in this case, though. First, because of the small volume of defects triggered, doing the last step of reduction by hand is not particularly time-intensive. With this specific reducer, much of the manual work comes from pruning dead branches and simplifying expressions that access global state, since the reducer tries to not modify those. Second, the inclusion of a small part of the multivalue proposal increases the coverage of Wasmlike, and led to the discovery of the defects reproduced in Figures A.2 and A.4. A possible alternative is to replace the idiomatic loop construct with a version that uses local variables instead of loop parameters. This would allow us to use the more effective reducer from Binaryen, at the cost of making the generation of loops a bit more tedious within Wasmlike. Another possible alternative is to write our own reducer, but because reducers already exist for WebAssembly, we deemed this to be unnecessary.

5.4.4 Future Work

In the future, Wasmlike could be enhanced to cover recent additions to the specification of WebAssembly. Since the initial release of WebAssembly, multiple proposals have been approved and added to the language, such as multivalue returns, reference types, bulk memory operations, and tail call optimizations. Some of these proposals, like the multivalue proposal, would need a comprehensive rewrite of Wasmlike to support the feature, while others, like bulk memory operations, would simply need the instruction added to the grammar specification in Wasmlike.

Future work could also add more systems to the test harness, such as the Safari browser. Additionally, converting the checksum calculation from the runtime's responsibility to a

WebAssembly snippet to be injected into generated programs would allow standalone WebAssembly runtimes without an API to be tested.

APPENDIX

BUG REPORT TEST CASES

This appendix presents the test cases submitted with the bug reports to the developers of the systems in which we found defects. The programs in Figure A.1 trigger rotate-by-zero bugs in Wasmer 1.02 that resulted in wrong code generation. The program in Figure A.2 causes an LLVM optimizer crash in Wasmer 2.3.0. The code in Figure A.3 triggers a Cranelift and LLVM defect that produces wrong code in Wasmer 2.3.0. Finally, Figure A.4 contains the code associated with a three-way wrong-code error involving two different configurations of Wasmer 2.3.0 compared to any other system under test.

```
(module
  (func $main (result i32)
    i32.const 235
    i32.const 0
    i32.rotl)
  (export "_main" (func $main)))
```

(a) Rotate left or right with any type in Wasmer 1.0.2

```
(module
  (func $main (result i64)
    i64.const 4
    i64.const 0
    i64.rotr)
  (export "_main" (func $main)))
```

(b) Rotate right with i64 in Wasmer 1.0.2

Figure A.1: Rotate bugs in Wasmer 1.0.2.

```
(module
  (func (;0;) (type 0) (result i32)
    i32.const 0)
  (func (;1;) (type 1) (param i32 i64 i32) (result f64)  ;; label = @1
    (local i32 i64 f32 f64 i32)
    ;; load, const, and mul is important
    i32.const 1
    f64.load offset=95 align=4
    f64.const 0
    f64.mul
    ;; loop is important
    loop (param f64) (result f64)  ;; label = @0
      i32.const 1
      f64.load offset=22 align=4
      f64.add
      local.get 7
      i32.const -1
      i32.add
      local.tee 7
      br_if 0 (;@1;)
    end)
  (func (;2;) (type 2))
  (memory (;0;) 1)
  (export "_memory" (memory 0))
  (export "_main" (func 0))
  (export "_crc_globals" (func 2)))
```

Figure A.2: LLVM optimizer crash in Wasmer 2.3.0.

```
(module
  (func (;0;) (type 0) (result i32)
    (local i32 i64 f32 f64 i32 i32 i64 i64)
    i32.const 0
    call 1  ;; Stuff after the call is also important
    i32.const 2
    i32.const 1
    i32.const 0
    f64.load offset=37 align=4
    i32.const 655
    f64.load offset=40 align=4
    f64.add
    f32.demote_f64
    f32.store offset=77 align=2 ;; This store is important
    i32.const 1  ;; i32 and return is just used to return the function early
    return)      ;; without taking care of the types on the stack
  (func (;1;) (type 1) (param i32) (result i32)
    (local i32 i64 f32 f64 i32 i64)
    i32.const 663 ;; Memory address
    i32.const -2  ;; The negative for the value to be stored is important
    i32.store offset=36 align=1
    i32.const 0)
  (func (;2;) (type 2))
  (memory (;0;) 1)
  (export "_memory" (memory 0))
  (export "_main" (func 0))
  (export "_crc_globals" (func 2)))
```

Figure A.3: Cranelift and LLVM wrong code error in Wasmer 2.3.0.

```
;; A = Wasmer/LLVM/Universal -- agrees with NodeJS
;; B = Wasmer/LLVM/Universal/Optimize
;; C = Wasmer/LLVM/Dylib/Optimize

(module
  (type (;0;) (func (result i32)))
  (type (;1;) (func (param i32 f64) (result f32)))
  (type (;2;) (func (param f32) (result f32)))
  (type (;3;) (func))
  (func (;0;) (type 0) (result i32)
    i32.const 0 ;; store address
    i32.const 0 ;; arg 1
    f64.const 0 ;; arg 2
    call 1      ;; store value
    f32.store offset=59 align=1
    i32.const 21
    return) ;; Return without needing to reduce the stack
  (func (;1;) (type 1) (param i32 f64) (result f32)
    (local i32 i64 f32 f64 i32 f32)
    block (result f32)  ;; label = @1
      i32.const 1
      local.set 6 ;; This local is important
      ;; Replacing this block with an f32.const results in B and C agreeing
      ;; with each other but not with A
      block (result f32)  ;; label = @2
        i32.const 1
        f32.const 0x1.ddfd56p+81 (;=4.51448e+24;)
        f32.store offset=72 align=2
        f32.const 0x1.b01acp+9 (;=864.209;)
      end
      f32.const 0x1.86c8d6p-75 (;=4.04062e-23;)
      f32.max ;; Removing this max makes only B differ
      global.get 1 ;; Must be this specific global variable.
      ;;f32.const -0x1.2feb5p-113 (;=-1.14322e-34;)
      f32.const 1 ;; If set to 0, makes only C differ
      f32.min
      f32.sub
      loop (param f32) (result f32)  ;; label = @2
        i32.const 0
        i32.const 0
        i32.load8_u offset=9 ;; load is important
    ;; continued below
```

Figure A.4: Three way code generation difference in Wasmer 2.3.0.

```
      if (result f32)  ;; label = @3
      ;; This whole unused branch is important
        i32.const 1
        f32.const 0x1.7978d4p+9 (;=754.944;)
        f32.store offset=62 align=1
        local.get 7
        f32.const 0x1.bd60bap-4 (;=0.108735;)
        f32.min
        ;; !!! Putting an unreachable here makes all three
        ;;     implementations produce the same result !!!
        ;; unreachable
      else
        i32.const 0
        local.set 6
        i32.const 1
        f32.load offset=11 align=1
        ;; removing this loop makes B agree with C but not A
        loop (param f32) (result f32)  ;; label = @4
          f32.const -0x1.bd2fe2p+9 (;=-890.374;)
          f32.add
          local.get 6
          br_if 0 (;@4;)
        end
        global.get 1 ;;Must be a global
        f32.sub ;; Cannot replace with a drop
      end
      f32.store offset=66 align=2
      f32.const -0x1.9acb68p+56 (;=-1.15628e+17;)
      local.get 0
      br_if 1 (;@1;) ;; removing this branch results in
                     ;; a crash from B and C
      drop
      local.get 7
      f32.const 1
      f32.mul ;; Removing the mul results in C agreeing with A
      f32.add ;; Add is important
      local.get 6
      br_if 0 (;@2;)
      ;; putting a return here causes a crash in B and C
    end
    ;; same here
  end
  f32.floor) ;; Must be some kind of funop or a const and fbinop.
(func (;2;) (type 3))
(memory (;0;) 1)
(global (;0;) (mut f32) (f32.const 0x1.c701dcp+9 (;=910.015;)))
(global (;1;) (mut f32) (f32.const 0x1.0e8962p+8 (;=270.537;)))
(export "_memory" (memory 0))
(export "_main" (func 0))
(export "_crc_globals" (func 2)))
```

Figure A.4 continued.

REFERENCES

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of the 26th Network and Distributed Systems Security Symposium* (San Diego, California, Feb. 24–27, 2019), 15 pages.

[2] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2018. GRIMOIRE: Synthesizing structure while fuzzing. In *Proceedings of the 28th USENIX Security Symposium* (Santa Clara, California, Aug. 14–16, 2019). ACM, Inc., New York, NY, 1985–2002.

[3] ByteCode Alliance. 2023. wasm-shrink. `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-shrink`.

[4] Bytecode Alliance. 2023. wasm-smith. `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith`.

[5] Bytecode Alliance. 2023. Wasmtime. `https://wasmtime.dev`.

[6] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia, Aug. 26–30, 2019). ACM, Inc., New York, NY, 223–234.

[7] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz 'em all: Generic language processor testing with semantic validation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy* (Santa Francisco, California, May 24–27, 2021). Curran Associates, Red Hook, NY, 642–658.

[8] Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM InterIn Embedded Software* (Atlanta, Georgia, Oct. 19–24, 2008). ACM, Inc., New York, NY, 255–264.

[9] Elasticsearch B.V. 2023. Elastic stack. `https://www.elastic.co`.

[10] WebAssembly Community Group. 2023. Binaryen fuzzing. `https://github.com/WebAssembly/binaryen/wiki/Fuzzing`.

[11] William Hatch, Pierce Darragh, and Eric Eide. 2023. Xsmith. `https://pkgs.racket-lang.org/package/xsmith`.

[12] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and*

Communications Security (Toronto, Canada, Oct. 15–19, 2018). ACM, Inc., New York, NY, 2123–2138.

[13] William M. McKeeman. 1998. Differential testing for software. *Digit. Tech. J.*, 10, 1, 100–107.

[14] Microsoft. 2023. Playwright. `https://playwright.dev`.

[15] Mozilla Corporation. 2023. Firefox. `https://www.mozilla.org/en-US/firefox/`.

[16] OpenJS Foundation and Node.js contributors. 2023. Node.js. `https://nodejs.org/en/`.

[17] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China, July 15–19, 2019). ACM, Inc., New York, NY, 329–340.

[18] Árpád Perényi and Jan Midtgaard. 2020. Stack-driven program generation of WebAssembly. In *Proceedings of the 18th Asian Symposium on Programming Languages and Systems* (Fukuoka, Japan, Nov. 30–Dec. 2, 2020). ACM, Inc., New York, NY, 209–230.

[19] Red Hat, Inc. 2023. Ansible. `https://www.ansible.com`.

[20] Andreas Rossberg. 2020. WebAssembly specification version 1.1. `https://webassembly.github.io/threads/core/_download/WebAssembly.pdf`.

[21] The Chromium Projects. 2023. Chromium. `https://www.chromium.org/Home/`.

[22] University of Utah, Flux Research Group. 2023. Emulab d430. `https://wiki.emulab.net/wiki/d430`.

[23] Wasmer, Inc. 2023. Wasmer. `https://wasmer.io`.

[24] WebAssembly Community Group. 2023. Binaryen. `https://github.com/WebAssembly/binaryen`.

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. *ACM SIGPLAN Not.*, 46, 6, (June 2011), 283–294.

[26] Michał Zalewski. 2023. American Fuzzy Lop (AFL). `http://lcamtuf.coredump.cx/afl/`.