Hochschule Kempten
University of Applied Sciences

Bachelorarbeit

Studiengang Informatik

A WebAssembly interpreter with integrated debugging capabilities

Raphael Isemann

Aufgabensteller:	Prof. Dr. rer. nat. Göhner
Arbeit vorgelegt:	19.02.2016

Anschrift des Verfassers:	Sudetenstr. 36, 87448 Waltenhofen
	e-Mail: teemperor@gmail.com

Abstract

One of the most crucial factors that determine the success of a new programming language are developer tools. This thesis will introduce wasmint, a software platform that allows interpreting, inspecting and debugging programs written in the upcoming programming language WebAssembly. More specifically, this paper will discuss in depth how wasmint approaches reverse execution and detection of programs that don't halt.

Contents

1 Introduction

In recent years, the role of client-side scripting in web browsers has evolved from a tool for creating basic interactive web sites to a compile target of full fledged applications. More complex APIs such as HTML5 and WebGL expanded the capabilities of client-side scripting, and even big companies like Unity Technologies or Epic Games started to provide HTML5 versions of their game engines[1][12].

With these new applications also new performance requirements on browser implementations arose and browser vendors started to work on more complex virtual machines and JIT compilers to effectivly execute JavaScript[21, p. 22]. These new engines were significantly faster than traditional interpreters and could overcome many of the negative performance characteristics of JavaScript.

These performance characteristics of JavaScript also lead to the development of new languages that have the ambition of replacing JavaScript in situations where performance is critical. Noteable examples for this are Google Native Client and asm.js.

Google Native Client was designed as a "a sandbox for untrusted x86 native code"[34] and also supports a platform independent subset of the LLVM intermediate representation bytecode. Even though it provides a runtime environment with only minimal performance overhead (on average 5 percent in the SPEC2000 benchmark), Native Client has only been supported by Google Chrome since 2016.

asm.js is a subset of JavaScript and was the result of a research project by Mozilla. It restricts JavaScript to a limited number of operations that can be effectivly translated into native code. Features that can't be expressed in JavaScript such as the type of a variable or a expression are annotated in the source code of a program as seen in Listing 1. Unlike other projects, asm.js is supported in all web browsers that can execute JavaScript and the special optimizations made possible by asm.js are currently supported by Google Chrome, Microsoft Edge and Mozillas Firefox browser.

The remaining performance problem of asm.js is the size of the source code which is critical for applications that are loaded on demand over the network. However, this problem can't be solved without breaking backward compability with JavaScript and is therefore not solveable with the current approach that asm.js has taken.

For this reason, the major web browser vendors Google, Mozilla and Microsoft started planning a new programming language that could combine the performance of asm.js with a compact serialization format. This language was announced on the 17th June

2015 under the name WebAssembly[33][13][7].

Listing 1: asm.js example[17]

```
function f(x, y) {
  // SECTION A: parameter type declarations
  x = x|0;       // int parameter
  y = +y;        // double parameter

  // SECTION B: function body
  log(x|0);      // call into FFI -- must force the sign
  log(y);        // call into FFI -- already know it's a double
  x = (x+3)|0;   // signed addition

  // SECTION C: unconditional return
  return ((((x+1)|0)>>>0)/(x>>>0))|0; // compound expression
}
```

2 A brief introduction to WebAssembly

As WebAssembly is at the time of writing still in early development and no official specification draft was released yet, this section will provide an overview over the language and its current state.

WebAssembly is a statically typed programming language. A WebAssembly program (also called a *module*) consists of a linear array of bytes called the *linear memory* and a set of functions. All instruction of a WebAssembly program reside inside a function and are encoded as a tree. In this tree each instruction is represented by a node and the operands of a instruction are the child nodes.

The instruction set and data types of WebAssembly are designed to represent common hardware capabilities. The data types itself consist of 32 and 64 bit integers and floating point types as defined in ANSI/IEEE Std 754-2008. The defined instructions allow arithmetic operations on these data types, manipulation of the control flow and accessing the linear memory.

WebAssembly has similar to traditional assembly languages multiple isomorphic representations: A binary format that is intended to be transferred over a network and a human-readable text format.

The binary format has the aim to be compact (both uncompressed and compressed) and fast to load. It is the intended form of distributing WebAssembly programs. One of main ways the format reduces its size is by using lookup tables and variable length encoded indices to those tables. For example, all instructions that are used inside a WebAssembly module are listed at the beginning of a WebAssembly module. In the rest of the module the instruction is only encoded as the variable length index into this table. Similar tables exist for each function and type that is used inside a WebAssembly module.

The text format is human readable and is intended to be used by developers using WebAssembly. It shares similarities in its syntax with Lisp and other languages that use s-expressions to represent nested expressions. An example of a WebAssembly module in the test format can be seen in listing 2.

```
(module
  ;; Recursive factorial named
  (func $fac-rec (param $n i64) (result i64)
    (if_else
      ;; condition
      (i64.eq (get_local $n) (i64.const 0))
      ;; if condition true
      (i64.const 1)
      ;; if condition false
      (i64.mul
        (get_local $n)
        (call $fac-rec (i64.sub (get_local $n) (i64.const 1)))
      )
    )
  )
) ;; end of module
```

Listing 2: WebAssembly module represented in the text format format[29]

3 Overview and design guidelines of wasmint

The goal of the project is to assist with building complex developer tools such as debuggers or analysers for the WebAssembly programming language. Additionally the project should help researching how the WebAssembly proves itself in practice.

The central element of this paper is "wasmint", a software platform for interpreting and debugging WebAssembly programs.

It provides an compiler toolchain called "wasm-module" which allows parsing and serializing WebAssembly modules and offers an C++11 API for accessing the parsed data. Additionally, it provides the WebAssembly engine wasmint VM that can execute the modules loaded by wasm-module.

To reach the goals of this study, wasmint was designed with these four design guidelines in mind:

Full control over the program execution

Many of the systems described in this paper rely on the ability to fully control all aspects of the program execution. This ability is also one of the core requirements for building efficient debuggers.

Reuseability

This guideline is inspired by a quote from Robert O'Callahan — author of *rr* (which will be discussed later in this paper): "Instead of building systems out of parts, build the parts [...] The design of the parts constrains what others can build"[24]. wasmint adheres to this idea by providing all its functionality in the form of libraries that can be reused by other programs. In fact, all executable binaries that are part of the project are just tiny wrappers around the libraries which contain the actual logic.

Platform independence

Libraries shouldn't restrict developers to certain platforms or operating systems. For this reason wasmint is only depends on the C++ Standard Library and on the cross-platform build-tool CMake. Other tools such as the testsuite or the example programs however are free to use arbitrary libraries.

Sufficient performance for real world programs

Even though wasmint doesn't aim to be a high-performance runtime environment, it should still be fast enough to handle real-world programs. The WebAssembly

reference interpreter ml-proto is considered to be a guiding value of what an acceptable execution performance is.

Future proof

WebAssembly will grow and change in the future. wasmint should be able to adapt itself to changes in the language standard without the need of major refactoring.

4 The wasm-module parser and serializer

The first step on the way to interpret a program is to parse the source code into an intermediate representation (IR) suitable for further processing. For most programming languages this problem can be solved by reusing an existing parser or generating one with a parser generator.

However, as WebAssembly was still in early development at that time, there were no existing WebAssembly parsers that could be used for parsing.

Additionally, the binary format of WebAssembly makes it quite difficult to use traditional parser generators: It allows for example that tokens consisting only of a zero byte which can't be handled by parsers that use C strings to represent tokens. Furthermore, the final grammar of the encoded instructions is defined in the header of a module and can differ between modules. This means that parser generators that only generate source code are therefore not of any use in this situation.

For these reasons this project uses a hand-written recursive descent parser called wasm-module. wasm-module parses a module like a normal recursive descent parser would with the exception of the code segments. Here it first reads the opcode table from the module header and then infers the production rules for parsing the instructions. Afterwards, the procedures that are responsible for handling the code segments can use the created production rules to correctly parse the instruction tree.

Another problem that arose during development was the introduction of the WebAssembly text format, which happened after the binary parser was finished: It was quite different from the original binary format and would require the construction of a second parser. Instead of starting all over again by building a new parser, I decided to refactor the binary parser to allow reusing the most complex part of the parsing process — the instruction parsing — for the text parser. This resulted in an instruction parser that offers a generic API that can be used as a backend for both the binary and the text parser.

The final design of the parser logic can be seen in figure 1:

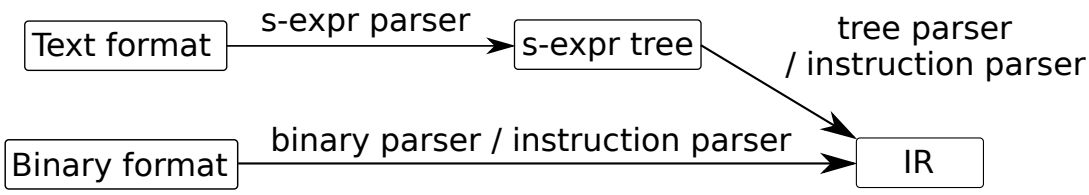

Figure 1: The wasm-module parsing process.

The text format input is first parsed into an s-expr tree data structure. This temporary data structure is necessary for handling the official testsuite format, but slows down the parsing process. Nevertheless, this technique allows writing a more flexible and simpler parser and can also be seen in projects such as WAVM[3].

Afterwards the tree data structure is transformed into the final intermediate representation. This transformation is handled by a dedicated tree parser that only handles the text format. When the tree parser encounters instructions, it uses the already mentioned instruction parser.

The binary parser is more streamlined than the text parser and doesn't use any temporary data structures but directly generates the IR and forwards any instructions to the instruction parser API.

4.1 Serializing

wasm-module also supports serializing the IR back to a WebAssembly module in either the text format or the binary format, which enables a multitude of unique applications.

The most common use case for this feature is the compilation from a WebAssembly module in the text format to the binary format. Modules in text formats are usually written by developers and are expected to be compiled to the binary format when they are published on a website. wasm-module easily compile the module by first parsing the text format into the IR, and then serializing that IR back to a binary module.

Another use case are developer tools that for example merge modules together, remove dead code or just refactor a module. Those tools need the ability to parse the code into the IR, modify it and then generate source code from the modified IR.

While parsing and serializing the same format is not that compilcated, parsing and serializing different formats can be challenging as the different formats don't contain

exactly the same information. This includes for example all lookup tables in the binary format which are missing in the text format.

While generating the lookup table itself is a trivial task, determining the order of the entries inside the table is complex and heavilly influences the size of the resulting binary module. The reason for this is that all indices referring to these lookup tables are encoded as LEB128 which consumes more memory for bigger numbers. Another factor is the compression algorithm that is used when transmitting the module over the network. If the created lookup table is taking the strengths of the used compression algorithm in aspect, it can further reduce the final size of the binary module.

At the time of writing however, wasm-module is only focusing on reducing the length of the table indices or instructions. An optimal solution for this problem is to count all uses of a specific in the module and sort them by the number of occurrences. The most frequent instruction gets the table index zero, the second most frequent instruction the table index one and so on.

It should be noted that it is not necessary for an optimal solution that the indices represent the places in the frequency ranking. As LEB128 encodes ranges of numbers with the same length (for example, the numbers 0 - 127 are all encoded as one byte), it is possible to reorder the indices after sorting them and still get an optimal solution. The only requirement for an optimal solution is that only indices in the same LEB128 encoding range are exchanged. It is for example possible to exchange the indices 33 and 120 and still get an optimal solution, but not the indices 33 and 128.

One remaining caveat when using the parser and serializer is that the current version of the parser strips the input file of any comments before further processing and makes it therefore not a praticable in scenarios where the output is intended to be human-readable.

4.2 Type checking and type inference

Even though WebAssembly doesn't offer many features to make writing software by hand easy, it is still aware of the concept of type inference for some instructions. Type inference can be described as the ability to "determine[] the type of a language construct from the way it is used"[5, p. 387].

An example of the way WebAssembly uses type inference can be seen in listing 3 in which both the loop instruction and the get_local instruction rely on type inference.

The get_local instruction receives its type from its related local variable. In this case the variable $input has the type i32 and therefore the get_local instruction also has the return type i32.

The loop construct is more complex and needs to determine its return type from its last child and all branch instructions that target its $exit label. In this example, the loop instruction first checks the i32.const instruction which returns a value of the type i32. Then it takes the br instruction and compares the type of the branch value (which is also i32) with the already known type of the last child. As both types are i32, the loop construct also has the return type i32. If the types would be different, the parser should generate an error.

```
(func $hundredOrMore (param $input i32) (result i32)
  (loop $exit $cont
    (if (i32.gt_u (i32.const 100) (get_local $input))
      (br $exit (get_local $input))
    )
    (i32.const 100)
  )
)
```

Listing 3: Example function that requires inferring type of the loop instruction and the get_local function.

An algorithm that would come to the same result as the previous paragraph can either follow a top-down or a bottom-up checking strategy.

The top-down strategy would always assume that a given instruction has the type that its parent expects it to have. For example, the loop construct in the function from listing 3 would have the type i32 because its the body statement of a function with the return type i32. The branch value of the br instruction would also have the expected type i32 because it targets the loop construct with the expected return type i32. Type errors in the top-down strategy are found whenever the return type is different from the expected type.

This strategy is called top-down because children verify that their type with the expected type of their targeted parent and the parent node therefore needs to be assigned an expected type before reaching its children. This ordering is only guarenteed when traversing the tree from top to bottom.

The WebAssembly design documents use the top-down strategy to explain the type inference rules due to its simplicity. This simplicity is also the weakness of this strategy

as it can't actually determine the type of constructs but rather only verifies if the construct evaluates to an expected type. In a situation where the parser has no expected type — for example when a debugger allows entering an expression that should be evaluated — this strategy wouldn't be able to infer the type.

The bottom-up strategy used in wasm-module solves this problem. It doesn't need an expected type but rather infers the expected type from the first relevant child instruction. Afterwards this strategy continues in the same way as the top-down strategy. As it is possible in WebAssembly that instructions infer their type from child instructions which also infer their type, this strategy requires that the parser traverses the AST from leafs at the bottom up to the root of the function body node.

5 The wasmint interpreter

wasmint is an interpreter that allows executing WebAssembly programs and uses wasm-module for parsing a WebAssembly modules. The development focus of wasmint is to provide a platform for WebAssembly development tools such as debuggers, optimizers and testing tools.

The first version of wasmint was a traditional interpreter that directly interpreted the parsed AST of a module. However, as it allocated necessary memory on demand and had to call the virtual function of the AST nodes during runtime, the runtime overhead of the initial version made it unuseable for bigger programs.

To improve performance, a second version of wasmint using just-in-time compilation was designed that uses an internal virtual machine that executes an internal byte code format. This byte code contains all the information that is necessary to run a function in a compact format and makes accessing the AST during runtime superfluous. The performance improvements from this second version reduced the runtime of the benchmark program [1] to about a tenth in comparison to the pure interpreter.

The virtual machine and the just-in-time compiler are topics of the following sections.

5.1 The wasmint virtual machine

The wasmint VM is a virtual machine that emulates its own virtual instruction set architecture (ISA) on any host system that is a supported target of a C++11 compiler.

It interprets its instruction set with a *decode-and-dispatch* interpreter as described in [30, p. 30] and allows the user to only perform an interpretation step at the time. Stepping through the program is a prerequisite for many of the debugging features of wasmint.

The most obvious example here is the common feature of executing only the next instruction of the program and then to wait for further instructions. But even more exotic features like the reverse execution engine of wasmint rely on stepping.

Stepping also allows controlling the speed at which the program executes and can be used to simulate performance-restrained runtime environments, which gives developers

[1] A WebAssembly quicksort implementation sorting the ASCII representation of first 100 000 Pi digits was used as a benchmark.

the opportunity to verify whether a program will work correctly on a variety of systems.

5.2 Structure of a VM instance

The structure of an instantiated VM strictly separates between code and data memory.

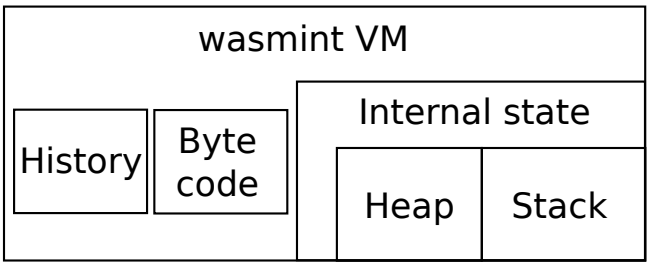

Figure 2: Structure of a wasmint VM instance inside the memory. The history is used for the reverse execution engine as discussed later in this paper.

Data memory contains all the data that can change during the execution of the program. The data memory region itself is again divided into a stack region and a heap region to conform to the idea of WebAssembly to keep variables on the stack non-addressable.

The heap region represents the linear memory of a WebAssembly program and consists only of a single continous piece of memory that allows random access. It doesn't use any kind of paging as of now (even though the WebAssembly standard permits and promotes this) and relies instead on the capabilities of the operating system.

The stack region consists of multiple function frames that are stored in a stack data structure. Each function frame contains exactly enough memory to save all temporary values which are stored in the registers and all local variables.

Only the heap memory region is addressable by the load store instructions of the VM. A positive side-effect of this design is that a program can't corrupt its own stack with malign memory accesses.

As the data memory region is the only part of the VM that changes during the execution of the program, it also known as the internal state of the VM. This internal state is designed to be write- and readable for the user. This also implies that the state can be copied by the user in a trivial way and then be copied back into the VM, which can

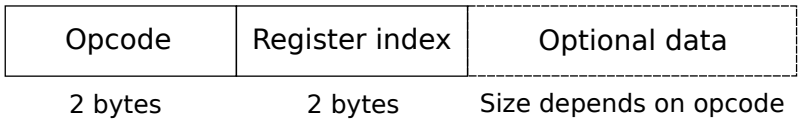

Figure 3: The internal bytecode format.

be used for more complex use cases such as taking snapshots and reverting the virtual machine.

Code memory contains the parsed IR of the loaded WebAssembly modules, compiler information necessary for debugging and any generated byte code.

The code memory is also one of the few parts of wasmint that uses operation system specific functions for security reasons: As the VM does not perform any runtime checks on the integrity of the byte code for performance reasons, manipulating the byte code inside the VM can be a viable attack vector.

To prevent that an attacker can alter the byte code, the memory pages containing the byte code are marked as read-only after the just-in-time compiler has finished. Altering the byte code during runtime would therefore result in a segmentation fault instead of executing the code of the attacker.

5.3 wasmint ISA

The instruction set of the wasmint VM specifies at the time of writing 164 different instructions and can be labeled as complex instruction set computing (CISC). Each instruction is encoded as four bytes with an additional segment containing optional data as seen in figure 3. The size of this optional data segment is not constrained in any way and only depends on the instruction itself.

There is however the requirement the size of the data segment always has to be a multiple of four bytes. This constraint is necessary as the opcode and register index of a command are always read and encoded as a single four byte word. This also means that the start address of the instruction has to be in a memory address that is a multiple of the word size as the VM would otherwise perform an unaligned memory access while reading the instruction from memory.

Performing an unaligned memory access in a program could lead to a variety of errors

as described in the following excerpt from the Linux kernel documentation:

> - Some architectures are able to perform unaligned memory accesses transparently, but there is usually a significant performance cost.
>
> - Some architectures raise processor exceptions when unaligned accesses happen. The exception handler is able to correct the unaligned access, at significant cost to performance.
>
> - Some architectures raise processor exceptions when unaligned accesses happen, but the exceptions do not contain enough information for the unaligned access to be corrected.
>
> - Some architectures are not capable of unaligned memory access, but will silently perform a different memory access to the one that was requested, resulting in a subtle code bug that is hard to detect![11]

To prevent such errors, some instructions in the wasmint ISA have padding bytes in the optional data segment that round up the size of the data segment to the next multiple of four.

5.3.1 Opcodes

The most important part of an encoded instruction is the opcode field.

It contains a 16 bit opcode that identifies the operation the VM should do if it encounters this instruction. With 16 bit opcodes, the wasmint ISA allows 2^{16} different opcodes which is more than necessary for the current 164 used instructions. The reasons for this additional byte for the opcode field are the following:

- Planned features of WebAssembly will require more opcodes than a single byte opcode field can encode.
- The opcodes are a continous set of integers and can be encoded by any interpreter with a dispatch table. This means that there is no signifcant performance drawback from having a big number of opcodes.
- The performance bottleneck of wasmint is the fact that it interprets only one instruction at a time. Merging multiple instructions into a single more complex instruction drastically improves the performance in this design.

5.3.2 Registers

The register index field specifies the registers that this instruction will use. The wasmint ISA allows to use up to 2^{16} registers to save temporary values. These registers are bound to the function frame and therefore each function frame has its own set of registers.

With 2^{16} possible registers the wasmint ISA is outlier under common ISAs but this can be explained by the lack of a stack functionality. Without a stack, all temporary values have to be stored in registers. As some computations — especially function calls — result in a high amount of temporary values, the amount of registers needs to be high enough to accommodate all values even these worst case scenarios.

The C++ standard for example recommends to support at least 256 function parameters[20, Annex B] which means that 2^8 registers is the bare minimum requirement for a WebAssembly implementation that wants to support compiled C++ code. To support even worse scenarios than the standard recommends, the wasmint ISA actually allows 2^{16} registers which should cover all real-world situations.

The reason why wasmint doesn't offer a stack for saving temporary values is that the VM registers are implemented as a chunk in memory and don't actually map to real registers in the computer as of yet. Therefore there is no reason to have a complicated stack structure in memory which has the same performance characteristics as the array of registers.

However, all 2^{16} registers are obviously far more than an average function needs to store its temporary values. This causes that each function can specify how many registers it actually needs. The VM will then guarantee that this specified amount of registers is available but trusts the function to not use more registers. By requesting less registers a function can reduce the memory useage of its related function frame.

Even though the register index field specifies all target and operand registers of an operation, it never contains multiple indexes, but only a single 16-bit index. This index identifies in the wasmint ISA both the target and all operand registers. This is possible because the target register and the first operand register are always the same. Additionally, all other operands of an instruction are in the registers following the first operand register.

This behaviour is similar to the way a stack machine identifies stack operands. The only difference is that wasmint's stack machine has the possibility to reuse higher registers to store reuseable temporary values. Also, compiling the byte code to machine code is

simpler as VM registers directly translate into hardware registers under the assumption that the function uses less registers than the hardware offers.

5.3.3 Calling convention

Calling a function is usually done by pushing the arguments onto the stack and receiving the result in a fixed register. As wasmint's ISA has no stack, the calling convention for subroutines is different. Additionallly, having a fixed register is not an option due to the way adressing operands works.

The calling convention of wasmint circumvents these problems by specifying the return register in the current function frame and the operand registers in the current frame. The operand registers are then copied into the local variables of the new function frame.

The function frame of the called function is expected to store any result in the register with the index zero. The index zero is chosen as all instructions store their result in the operand register with the lowest index, therefore is the first register with the index zero always the register with the result of the function body instruction. In situations where this is not the case — for example when using the return instruction to abort the evaluation of a function – it's the responsibility of the function frame to ensure that the result is copied into register zero.

After the execution of a function frame has terminated, the VM copies the result from register zero to the target register of the parent frame.

5.4 JIT compilation to bytecode

Even though wasmint's purpose is to interpret WebAssembly, its VM can only interpret the custom byte code format from the previous section. For this reason wasmint needs a compiler that will transform the WebAssembly input program into a format that the VM understands. As this compilation happens just before the VM actually starts, the compiler can be called a just-in-time compiler.

The main task of this compiler is to compile the instruction tree of each function into its own byte code segment.

The compilation happens in three phases:

1. In the first phase, the compiler traverses the AST of each function in the by the standardized order of evaluation and appends the compiled byte code of each instruction to the bytecode segment. Any branch instruction that is compiled in this step is assigned a placeholder branch address because the absolute addresses of all instructions are not known yet. Additionally, any function call is also assigned a placeholder function address.

2. The second phase starts directly after the first phase has compiled a function. At this point in the compilation process the compiler knows the byte code address of each compiled instruction. With this information the compiler can iterate over all branch address placeholders in the byte code and write in the real addresses.

3. The third phase happens after all modules and functions are compiled and the user indicates that he wants to create a thread inside the virtual machine. It is similar to the linking process as known from other programming languages and replaces the placeholder function addresses in each byte code segment with their real address.

5.5 Register allocation

The ability of the wasmint ISA to have a variable amount of registers allows us to create optimal function frames.

In an optimal function frame, the number of registers is equal to maximum amount of temporary values necessary to compute each instruction in the function. In other words, it is not possible to have a function frame with less registers than the optimal function frame that could be used to evaluate the given function.

The amount of actually used registers is determined by an algorithm that closely resembles the algorithm used for calculating the Horton-Strahler number of an arithmetic expression as described in [16]. Because the Horton-Strahler number is equal to the optimal number of registers for an arithmetic expression, it is an ideal template for an algorithm that calculates the optimal number of registers for a WebAssembly instruction tree.

The difficulty when using this approach is that not all WebAssembly instructions are arithmetic binary operations that need all their children at the same time. The block instruction for example returns only the return value of its last child. So it should reuse registers that were used for the evaluation of one child in the evaluation of the next

child. The traditional Horton-Strahler number however assumes that all instructions need to collect the return values of their children and thus comes to a wrong result.

This problem can be solved by classifying all instructions into two groups: Arithmetic instructions which can't reuse the registers of their child operands and non-arithmetic instructions which can reuse the registers.

The final algorithm for allocating registers would now traverse the instruction tree in the order of evaluation and would calculate the number of the necessary registers for each instruction subtree depending on the group of the specific instruction:

For arithmetic instructions the algorithm can use Horton-Strahler numbers to calculate the number of necessary registers. For non-arithmetic instructions the allocator calculates the number of registers each child instruction needs and returns the maximum as the number of necessary registers.

The computed results are saved and made available for the just-in-time compiler which uses this information to fill out the register index fields of the compiled instructions.

A remaining unsolved problem of the register allocator is limitation: Due to the way the wasmint ISA handles registers, register pressure can't be handled in a sensible way. If the register allocator comes to the conclusion that a function needs more than 2^{16} registers it will terminate the compilation and reports that it can't handle this program. Luckily this situation is, as already discussed in the previous sections concerning the ISA, far too rare to have any implications on the vast majority of the user base.

5.6 Performance

Execution speed can be considered the most important properties of an interpreter and thus this section provides some benchmarks on the performance of wasmint.

The benchmarks show the total execution time of three selected WebAssembly programs that can be retrieved from [19].

The benchmarks are similar to the benchmarks of the unfinished LLVM-based compiler wasm-to-llvm-prototype. The first benchmark program is an implementation of the quicksort algorithm that heavily uses function calls and recursion and will sort 10 KiB of memory. The second benchmark program multiplies two matrices and the third program calculuates the sum of a 10 MB vector of 8-bit integers.

Because this benchmark would only display a hardware-dependent time value that is on itself not very meaningful, two other WebAssembly interpreter will also execute above-said programs for comparison.

These two other interpreters are binaryen and ml-proto and were also developed in context of the WebAssembly project.

binaryen is a compiler toolchain and interpreter for WebAssembly written in C++. It's developed mainly by the emscripten developer Alon Zakai. Most of the features center around compiling between asm.js and the different WebAssembly formats, but it also features an interpreter called binaryen-shell.

The interpreter ml-proto serves as the reference interpreter for WebAssembly and is part of the specification effort for WebAssembly. It is written in OCaml and directly interprets any given WebAssembly modules.

Regarding the execution of programs, the main difference between ml-proto, binaryen and wasmint is that wasmint supports stepping through a program, which results in lower execution speed due to the stepping overhead. Having said that, the execution speed of wasmint is still on par with binaryen and drastically better than ml-proto as seen in figure 4.

These benchmarks should be treated with caution: None of the benchmarked interpreters were designed to be high performance WebAssembly engines. However, they reflect the current state of the WebAssembly development as implementations using mature compiler toolchains such as LLVM are still in early development and not even able to handle the benchmark programs.

5.7 Native code injection

wasmint provides an API that allows the user to inject modules into the virtual machine that contain native code as seen in listing 4. This native code is first packed into a C++11 lambda which is then packed into a function object. The function object contains the signature that the lambda expects.

The functions in these modules can then be called by WebAssembly code from inside the virtual machine in the same way as normal WebAssembly functions.

A common use case for this feature is the creation of interfaces for native libraries. WebAssembly modules can specify for example that they need the function with the

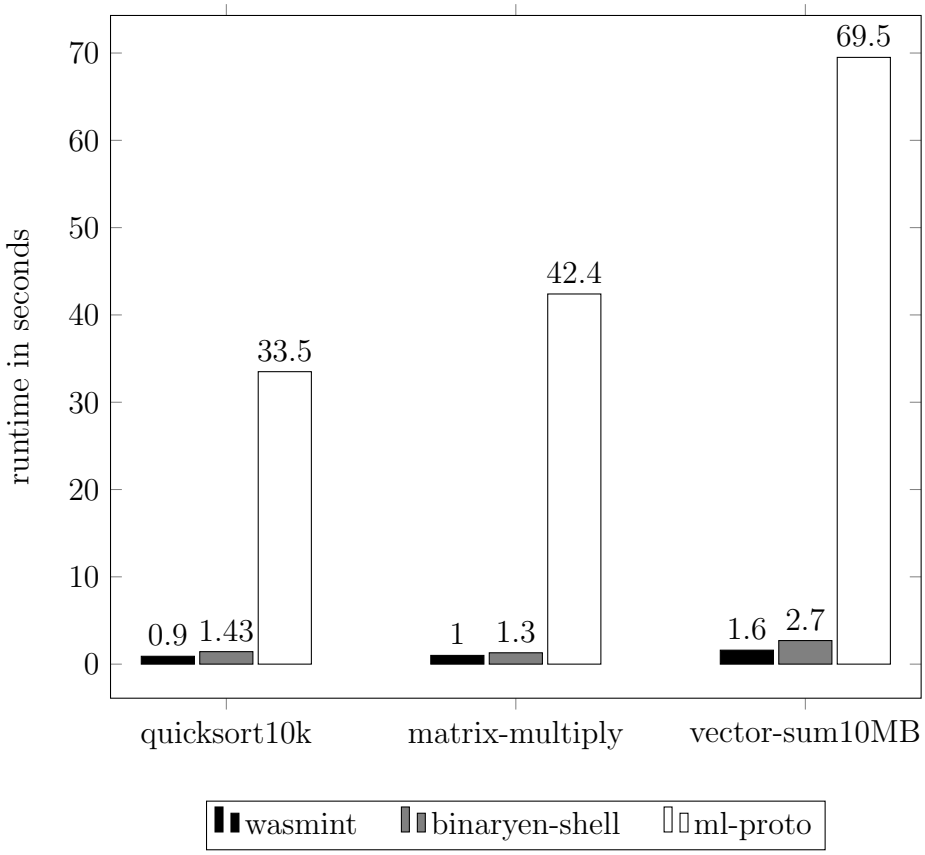

Figure 4: Performance comparison between wasmint, binaryen and the reference interpeter ml-proto. Average from 100 profiled runs.

name "sleep" from the C standard library. To fulfill this requirement without actually reimplementing each C standard library function, a wrapper function can be added to the VM that provides this functionality.

Listing 4: Code for wrapping the sleep function into a native module and loading it in the VM.

```
Module module("stdlib");
module->addFunction("sleep", Void::instance(), {Int32::instance()},
    [](std::vector<Variable> parameters) {
        usleep(Int32::getValue(parameters.at(0)));
        return Void::instance();
    }
);
vm.loadModule(module);
```

6 Reverse execution

One of the most interesting topics with regards to debugging is reverse execution which allows reverting the state of the program back to a previous version. When used in a debug environment with features like breakpoints, reverse execution is also known as reverse debugging.[14]

It can help with debugging problems which don't immediately cause observerable problems. A common bug with that characteristic is the multiple freeing of allocated memory in C where only the second free call is undefined behaviour. However, at this point all information about the perhaps problematic first free call are already lost which makes it difficult for the developer to find the cause of the problem.

In a system that supports reverse execution this information is not lost and the developer can see when and why both free calls were executed.

This information can also be retrieved by running the program again but this requires that the program is deterministic. Also, rerunning programs that take multiple hours until they are affected by the specific bug is too time consuming to be a praticable solution.

Reverse execution provides an effective way of debugging those problems and can be considered as one of the most important features of future debuggers. This can be seen for example in the Free Software Foundations high priority project list which promotes further development of the reverse debugging in gdb[28]. Also other companies are currently working on effective reverse debuggers such as Mozilla with their research project *rr* or Undo software with their UndoDB product[2]. It seems logical with these facts in mind that it should be a desireable goal for a project like wasmint to support reverse execution.

The following sections will give an overview of the history, current research and general principles of this subject. Finally, the actual implementation of reverse debugging in wasmint is discussed.

6.1 Introduction and approaches to reverse execution

In the previous section reverse execution was defined as "reverting the state of the program to a previous version". However, as the state of a real-world computer program is quite complex, it is necessary to distinguish between two kind of states in regards to

reverse execution:[14, p. 5]

Firstly, a reversible state which includes all information that should be restored when reverting the program. Usually this state only includes the memory of the current program including thread stack and allocated memory.[14, p. 5]

Secondly, a external state which includes all information that is not in the reversible state. This usually includes information that is directly associated with hardware as collecting and reverting this information is often too complex or even impossible. For example, reverting the mouse position is not possible without specialized hardware and therefore considered to be a part of the external state.[14, p. 5]

For the further discussion about the different strategies for reverse execution, reverse execution is formalized with the help of a finite-state automaton:

The program execution will be represented as a finite-state automaton. This automaton has an initial state q_0 that represents the reversible state of the program after initialization. There is also a tuple of states $(q_1, ..., q_n)$ that represent all prior states that the program execution has passed after the initialization in q_0. The current state of the program is q_n.

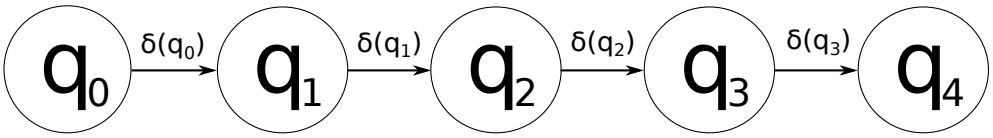

Figure 5: Deterministic finite-state automaton which represents the program execution after four steps.

The program execution behind this automaton can now be classified as reverse executable if for any state q_n there is a tuple of transitions defined that allows transitioning from q_n to any other state q_m with $m < n$.

A trivial way to make the automaton reverse executable is to record for every transition in δ the memory areas that were overwritten and then define a function r that copies the content of the records back into memory. The automaton now looks as in figure 6. [14]

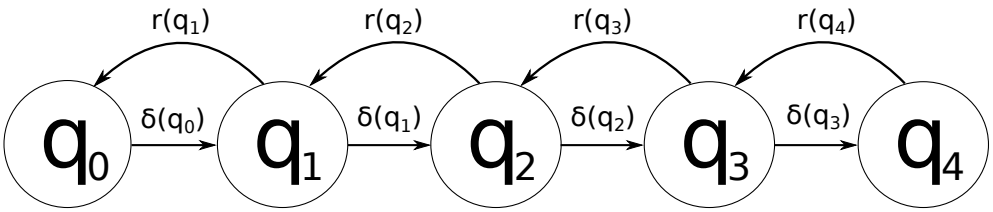

Figure 6: Reverse execution in the deterministic finite-state automaton.

Even though the automaton fully supports reverse execution with this method, it is still far from being effective. Creating a record for each executed instruction consumes too much memory to allow reverse execution for programs that run longer than a few seconds.

As an example, wasmint can execute about 100 million instructions per second and even if the system creates only records for a single 8 byte register, the records for a single second of execution already occupy 800 megabyte of memory.

In practice this can be seen at the current implementation of gdb. It is with the current implementation of reverse debugging in gdb 7.10 only possible to record very short intervals of the program execution before running out of memory.

A solution of this problem can be found in the definition of reverse execution earlier in this section, where it was only specified that a tuple of transitions which allows transitioning from the current state to the earlier state is needed. However, the definition doesn't set any restrictions about what transitions can be used to create such a tuple.

In the current approach the system only uses transitions from the r function but has ignored the δ function. The advantage of the δ function in comparison to the r function is that the transitions of the δ function don't use any additional memory for recording.

In the next model as seen in figure 7 the new r function combines several of the transitions provided by the original r function into a single transition. By also using the δ transitions for reaching the states q_2 and q_3, the automaton is still compliant with the above-said definition of reverse execution.

The advantage in memory consumption of this technique is that the r function transition between q_4 and q_1 is using less memory than the three previous r function transitions that connected q_4 and q_1. The reason for this can be illustrated by a simple example:

If the transitions $\delta(q_1)$, $\delta(q_2)$, $\delta(q_3)$ all write an arbitrary 8 byte value to the memory address 0x100, the records would consume 24 byte of memory for the three transitions as they had to record the value of the memory address in the states q_1, q_2 and q_3. The single transition however only needs 8 byte of memory as it only needs to record the value in state q_1.

This approach of reusing the δ function transitions is also known as reconstructing. All states that are the target of a r function transition are called checkpoints. [14]

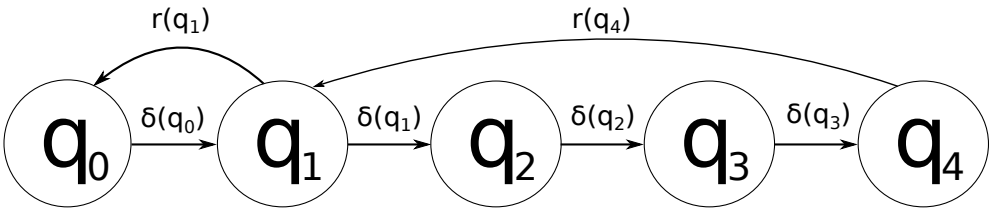

Figure 7: Reverse execution with reconstruction in the deterministic finite-state automaton. q_0 and q_1 are checkpoints.

When applying this strategy to programs in the real world, the problem arises that many programs are in fact not deterministic and that some transitions depend on the external state. Even simple operations like retrieving the current system time or reading environment variables depend on certain information in the external state. [14]

This poses a problem because the reverse execution in the current automaton relies on determinism to reconstruct states between two checkpoints. Any kind of influence from the external state during the reconstructing could alter the reconstructed state which defeats the purpose of reverse debugging.

This means that it's not possible to just rely on the transition function δ which represents the normal program flow but that it's necessary to provide a third transition function e. The function e has to fulfill the condition $e(q_n) = q_{(n+1)}$ for all states in the automaton and must not depend on the external state.

This function e is usually implemented by recording the communication between our reversible state and the external state. While reconstructing states, these records are replayed which allows the reversible state to reconstruct in a deterministic way.[14]

In figure 8 a final automaton using the e function to handle signals from the external state can be seen.

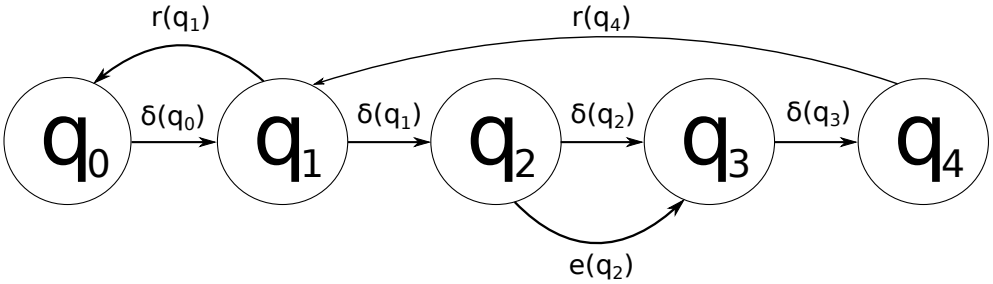

Figure 8: Handling non-determinism and external state influences while reconstruction in the deterministic finite-state automaton. Transition δ_2 is relies on the external state and is skipped during reconstruction via the e(q_2) transition.

6.2 Existing implementations

Reverse execution is implemented on a variety of different systems in many different ways – each with its own advantages and drawbacks. This section will describe three of them that are important either due to their historical significance or because of the likelihood to see them in practice.

A more detailed look at the ecosystem of reverse execution/debugging implementations is available in the article "A review of reverse debugging" by Jakob Engblom[14].

6.2.1 IGOR

The IGOR debugging system from 1989 is one of the first debuggers featuring reverse execution and targets native applications on certain hardware. It supports both checkpointing and reconstructing.

Recording memory changes in IGOR is implemented on a per page strategy: It will always record whole pages and observes the memory for write accesses to determine what memory pages need to be saved.

Reconstructing in IGOR is done by an interpreter which interprets a certain amount of native instructions. The external state during reconstructing is handled by requiring the user to reset it (e.g. reverting files to a specific state).

The developers of IGOR also suggested a generic way of recording communication with the external state by using OS features, but didn't implement this feature.[15]

6.2.2 rr

rr is a record-and-replay framework that is developed by Mozilla as a research project. The goal of the project was to develop a pratical way of replaying a non-deterministic execution of a native program.[25, section: background and motivation]

It is similar to IGOR restricted to certain operating systems because it uses library preloading to inject system calls with additional recording functionality. It also requires that each system call is manually handled in this preloaded library.[25, section: limitations]

The advantage of rr is the low performance overhead from recording on such a low level in the software stack. While exact numbers vary, the firefox testsuite only experiences a slowdown of about 20% when recording with rr.[25, section: rr in context]

rr also provides an gdb server which allows the user replaying the program execution with an gdb frontend. The gdb server functionality of rr also gained the ability to reverse execute its program via checkpointing and reconstructing at the end of 2015.

The checkpointing functionality in rr is implemented by using the copy-on-write mechanic that is usually used to reduce the peformance overhead when forking processes. rr repeatedly forks the observeable process to create snapshots and relies on the OS that it will retain the state of the fork while the original process continues to execute.

6.2.3 gdb

gdb is a popular debugger that supports a wide range of systems.

It provides reverse debugging since 2009 but uses unlike the already mentioned projects no checkpointing and reconstructing. It fully relies on tracing the execution and recording memory changes of every single instruction into a buffer[14, p. 3].

While gdb has support for reverse execution, the excessive performance overhead of the recording renders it unable to record the full execution of any non-trivial program.

6.3 Reverse execution in wasmint

The remaining open question in this section is how reverse exection is handled in wasmint. This will be answered by explaining the four steps in which the reverse

execution engine was created.

These four steps are not specific to wasmint but are generic enough to be applied to any other implementation. However, the amount and complexity of the work necessary to follow each step can drastically differ from wasmint.

The first step is to identify what belongs to the reversible state of the program. In the wasmint VM the decision was made, that the reversible state is equal to the internal state of the VM. According to figure 2 on page 16, the internal state of the VM consists of the heap and the stack. This means that developers can inspect local variables, registers and the linear memory during the execution of the program. Information that is not on the stack or heap — files and network packages for example — won't be reverted.

The second step is to implement means to record all parts of the reversible state. This is explained in more detail in section 6.3.1 and 6.3.2.

The third step is to implement the collection and grouping of those records during the program execution. These groups of records represent the checkpoints of the program.

The forth step is to allow the user navigating through the step with the strategies explained in section 6.1.

In wasmint this is implemented by enumerating each VM state with an instruction counter which increases with each VM step. The user can specify the instruction counter of the wanted state, and the VM traverses the checkpoints until it hits a checkpoint with a lower instruction counter than the one given.

Afterwards the VM repeatedly steps until the instruction counter of the VM is equal to the given counter. At this point the old state was successfully reconstructed. How the external state is handled during this reconstruction is described in section 6.3.3.

6.3.1 Recording changes to the heap

Recording changes in the wasmint VM heap is similar to the way IGOR records changes to its memory. The memory is first split up into intervals of equal size. In IGOR these intervals are identical to the pages of the underlying operating system which is necessary as it uses page flags to observe data access. [15]

In wasmint these intervals can have an arbitrary size as the capability to observer

changes to the heap is directly built into the VM. For now the size is fixed to 1024 but no effort was made so far to determine an optimal interval size as this would require a huge amount of real-world programs which are currently now available for WebAssembly.

After the intervals are set, they are observed for changes and when a write access is made to an interval that wasn't saved yet, the VM is stopped before the write access is granted and the interval is first saved. Then the VM continues as normal.

The result is a sparse array of interval backups inside each checkpoint. When reverting to a checkpoint, each interval backup is copied back to the appropriate place in the heap.

6.3.2 Recording changes to the stack

In addition to the heap, wasmint also has to record changes to the stack that it uses for temporary and function-local variables as well as the instruction pointer. WebAssembly specifies that this stack shouldn't be located inside the heap (to render for example stack manipulation impossible), so wasmint can't just reuse the recording mechanism of the heap patch data structure.

But even if it would be possible to reuse above-said recording mechanism, it wouldn't be an optimal solution as the specific way the stack is used in programs allows for unique optimizations:

Firstly, the stack is targetted more often by writing operations than the heap. In an implementation of the quicksort algorithm for example only one percent of the executed instructions inside the wasmint VM wrote values to the heap. All instructions however manipulate the current function frame as they will at least manipulate the instruction pointer inside the function frame. Also, nearly all instructions will change one or more registers when executed.

It would come with a huge performance impact to implement an observer that would peform checks before each write access. Instead, wasmint instantly saves each function frame as a whole. This makes it unnecessary to monitor write accesses as all data in the current frame is saved. Because function frames are small the worst case memory overhead from this strategy is neglectable for a single frame.

However, saving each frame on the stack in each stackpoint is still expensive and causes avoidable overhead.

This leads us to the second optimization: Because the stack doesn't offer random access to all function frames but only to the top one, the program has to indicate via pop and push operations that it wants to access another function frame. These pop and push operations happen just very rarely in normal programs and observing them comes therefore with a low overhead. For this reason wasmint can use the following mechanism to effectivly record the stack:

First the current stack size and the current function frame are saved as a whole. When a program executes an pop operation and the new stack size would be lower than the last known stack size, the function frame on top of the stack is saved and the new stack size is noted.

The result is an array of function frames and the original stack size in each checkpoint. When reverting to a checkpoint, the stack of the VM is first resized to the stack size of the checkpoint. Then the top frames in the stack are replaced with the saved function frames.

6.3.3 Handling the external state

As already discussed, it is important that all transitions used during the reconstruction process are not depending on any stimuli from the external state. This is usually implemented by recording and replaying the communication with the external state which requires that the runtime environment is able to observe all communication interfaces that allow exchanging information with the external state.

In wasmint there is only one such interface which is the functionality to call native functions. This means that recording the results of each native function call inside the VM and the instruction counter at which the call was made is enough information to provide all necessary transitions.

However, this is not an optimal solution. Some native functions for example are located in the external state but their result only depends on the internal state of the VM. Examples for this are for example mathematical functions that will always yield the same result for the same arguments.

wasmint offers for those functions a way to mark them as deterministic which causes that the VM will not create any records when such a function is called and therefore saves memory.

Additionally, functions that don't have a return value but only modify the external

state are not recorded and will be skipped during reconstructions. In the wasmint VM such function calls don't have any additional performance costs for recording.

6.3.4 Recording overhead

This section will evaluate the performance overhead when recording a program inside wasmint.

Because runtime of the original benchmark programs is too short to have meaningful execution times, they were all modified to run longer. The modified programs can also be found in [19].

The results in figure 9 show that the overhead for all programs varies between 50 and 90 percent. This overhead is worse than the overhead of rr which is about 20 percent [25, section: rr in context] but far better than for example gdb's trace-based implementation which was in tests not able to fully record any of the three benchmark programs before running out of memory on a system with 16 GiB of primary memory.

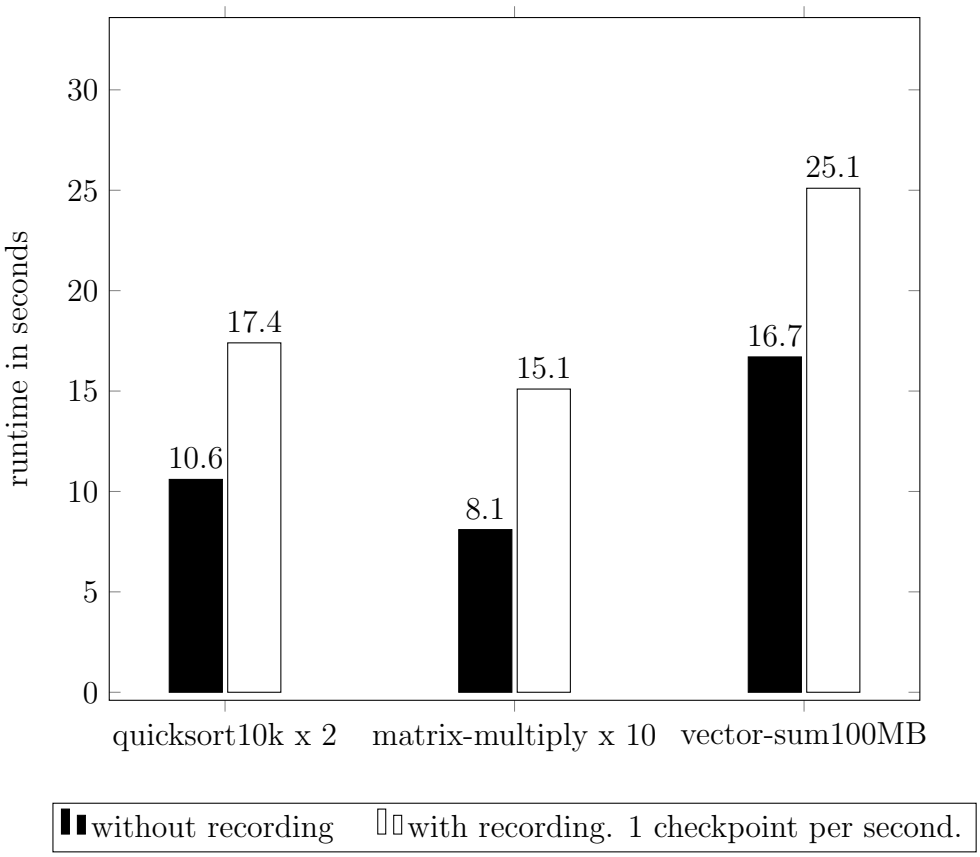

Figure 9: Recording overhead in the benchmark programs. Average from 10 profiled runs.

7 Detecting infinite running programs

The question of whether a computer program halts for a given input is one of the classic problems of computability theory and is also known under the name *halting problem*. It is also an interesting question regarding the search for undesireable behavior in programs, as most real-world computer programs are supposed to terminate after a finite amount of time.

Even though the halting problem is not decideable over Turing machines, it is theoretically decideable for finite-state machines according to Minsky:

> Now there are only a finite number of states. Clearly, if the machine is run for a long enough time, it must eventually re-enter a state it has previously been in. And this means that it muts, form that time on, continue in a periodically repeating pattern! Each chain must eventually lead to a closed loop or "cycle." Of course, different starting states may lead into the same cycle — i.e., different starting chains may *merge. But two paths, one merged (ey entering a common state) can never diverge again.*[italics in original] Hence the state diagram contains a finite number of separate, closed loops or "cycles."[p. 24][22].

As the internal state of wasmint can be considered a finite-state machine, one could in theory use this approach to detect infinite running programs.

A trivial implementation of this strategy would run an arbitrary WebAssembly program in the virtual machine and takes a snapshot of the internal VM state after each step. Then it compares the current state with all previously recorded steps: If it finds past state that is identical with the current state, the algorithm could report that the program will never halt. Otherwise, it will continue to step through the program until it either terminates or a previous identical state is found.

This implementation also makes clear what the negative aspects of this approach are: Recording each state requires an enormous amount of memory and comparing each previous state with the current state is time consuming.

For example, running a program with only 1 megabyte of memory for 1 million steps (which equals less then a second of program execution) would require 1 terabyte of memory to save all previous states. This also means that the comparison of all previous states with the current state requires in the worst case the iteration over each word in

the 1 terabyte recording storage, which results in an impratical long runtime even for small programs.

However, many of these negative aspects can be mitigated with the right strategies. These strategies are discussed in the following sections.

7.1 Storing previous states

The first problem is the memory consumption of the recorded states and is a problem that was already solved earlier in this paper in the section about reverse execution. So instead of taking a snapshot of the virtual machine after each step, the detection mechanism relies on the builtin reverse execution engine of wasmint to reproduce all previous states.

However, the programm still has to take a single snapshop of the current state to compare against, which means that it requires at least twice the VM memory of the original program. Nevertheless, the memory consumption is drastically lower than with the trivial implementation and can be considered acceptable for real-world applications.

7.2 Search for an equal past state

There are two obvious ways to iterate over all past states: From start to end and from end to start.

Iterating from end to start as seen in figure 10 means that our program has to repeatedly revert to the previous checkpoint and then reconstruct the previous state from there. This causes a significant performance overhead as each patch has to be applied n times where n is equal to the amount of states between two checkpoints.

Iterating from start to end as seen in figure 11 means that the program first applies all patches to restore the initial state of the machine and then reconstruct all states from there. Applying every patch just once is obviously more performant than applying every patch n times.

Still, iterating from start to end is not a perfect solution as the expected average amount of time until an identical state is found is longer than with the end to start strategy.

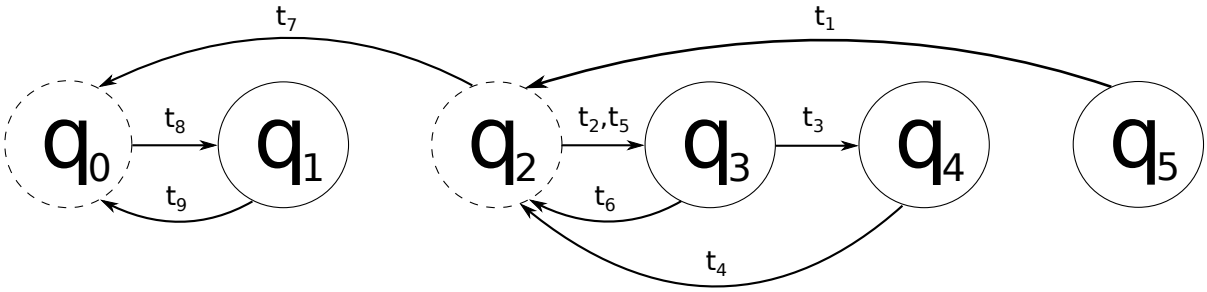

Figure 10: Seaching for an identical past state from end to start. q_0 and q_2 are checkpoints. The sequence $\{t_1, t_2, ..., t_9\}$ represents the transitions used to iterate over all states.

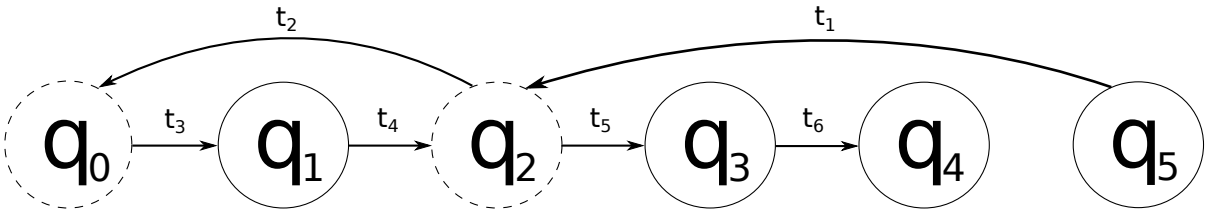

Figure 11: Seaching for an identical past state from start to end. q_0 and q_2 are checkpoints. The sequence $\{t_1, t_2, ..., t_6\}$ represents the transitions used to iterate over all states.

This can be inferred from Minsky's theory on page 35: It says that the program will eventually enter a cycle and that the program will stay in this cycle. This means that the cycle of the program is always at the end of the program execution and that the states at the end of the program execution should be preferred when checking for equal states as they are more likely to be in the wanted cycle.

The final strategy for traversing the states can be seen in figure 12 which is a hybrid from the previous two concepts. It applies every patch only twice: Once for reverting to the checkpoint and then reconstructing the following states and another time to prepare the VM for reverting to the previous checkpoint.

Even though the order of traversal is with this strategy not actually from end to start, it is a pratical approximation that has both a chance on finding cycles early and without an expensive overhead from applying the same patch millions of times.

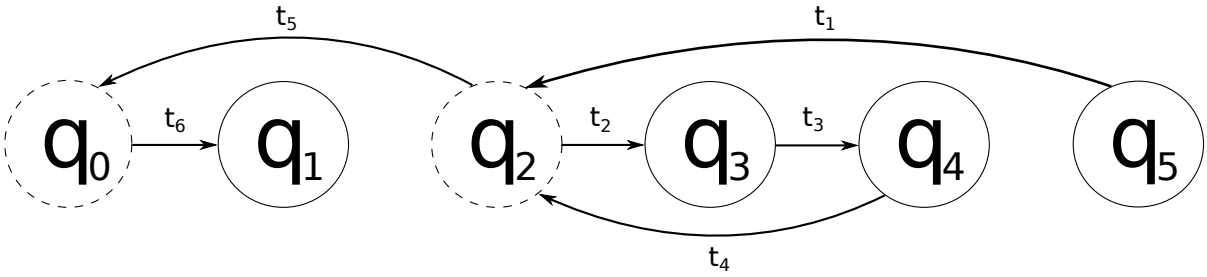

Figure 12: Seaching for an identical past state from end to start with performance improvements. q_0 and q_2 are checkpoints. The sequence $\{t_1, t_2, ..., t_6\}$ represents the transitions used to iterate over all states.

7.3 Determining changed memory regions

Even though the system is now able to drastically reduce the memory consumption with the help of reverse execution, it still faces the problem of comparing the enormous amounts of memory in a timely manner.

One way to speed up the comparison process is to reuse metadata from the patch data structures stored inside the reverse execution engine. Each patch contains information what parts of the linear memory have been changed between two checkpoints. This also implies that the rest of the linear memory hasn't been changed between the two checkpoints.

With this information the search process can be optimized by only comparing memory

regions that are known to be different according to the metadata. In the actual implementation this could be realized by making a set of modified memory regions and expand this set when the machine state is reverted to a certain checkpoint. Additionally, this set should be cleaned after each checkpoint from memory regions that are known to have the right content.

Figure 13 illustrates a scenario that profits from this optimization. The checkpoint q_{200} indicates that only the memory region with the assigned number two is modified between q_{200} and q_{300}. It is therefore only necessary to compare this memory region of each state in $[q_{200}, q_{300})$ with the memory region two of q_{300}.

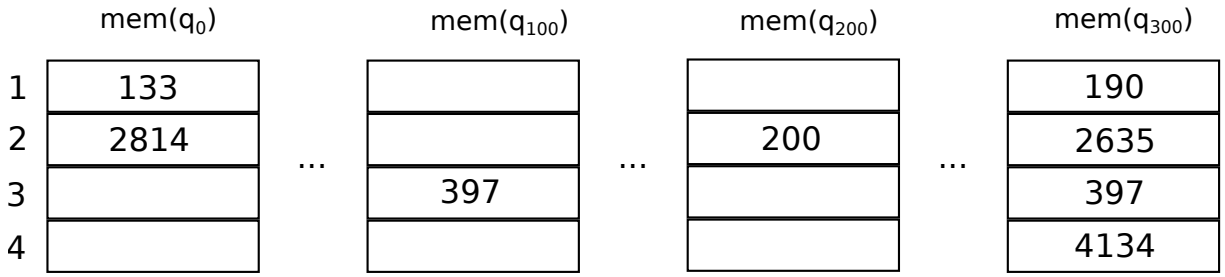

Figure 13: Linear memory with 4 regions during four different states. q_0, q_{100}, q_{200} are checkpoint states and q_{300} is the current virtual machine state. Each table cell represents a memory part and the number inside the cell the respective memory content at the given state. An empty cell represents that the content of the memory part is equal to the content of the same memory part in the next checkpoint state. An empty cell also means that the memory part can be considered immutable between this checkpoint state and the next checkpoint state.

The next checkpoint q_{100} adds the memory region with the number 3 to the set of regions that need to be compared. All states in the interval $[q_{100}, q_{200})$ therefore only need to compare their memory regions two and three with the respective regions in q_{300}.

The checkpoint q_{100} also has the special property that its memory region three is equal to the same memory region in q_{300}. With this it is legal to remove the memory page three from the set of memory regions that need to be compared before reverting to q_0.

After reverting to q_0, the memory regions one and two are added to the set of memory regions that need to be compared. When iterating over the final states in the interval $[q_0, q_{100})$, only the memory regions one and two need to be compared.

7.4 Eliminating state sequences

The figure 13 reveals another possible optimization: Without looking at the states in the interval (q_{200}, q_{300}), it can be proved that that sequence of states can't contain a state that is equal to q_{300}.

The proof relies on two observations:

The first observation is that is known that q_{200} is not equal to q_{300} because the contents of the second memory part differ.

The second observation is that q_{100} indicates that the second memory part is immutable between q_{100} and q_{200}.

This leads to the conclusion, that for all states between q_{100} and q_{200} the comparison with state q_{100} will fail due to the different contents of the second memory part. As the states are known to not pass the comparison, we also don't need to reconstruct them and can skip them altogether.

Generalizing this example into a general rule is straightforward: If the set of memory regions that a checkpoint marks as mutable doesn't contain all memory regions from the set of memory regions that need to be compared, all related states to that checkpoint can't contain a state equal to the current state.

This rationale allows us to skip reconstructing certain states just by inspecting meta-data which speeds up the search process. For example, it would reduce in the situation of figure 13 reduce the runtime by a third because checking the states in the interval $(q_{200}, q300)$ can safely be skipped.

In real-world applications the efficency of this technique depends on the locality of reference[9] of the program.

In tests, the system was able to eliminate about 60 percent of the program execution in a program[18] that sorted chunks of its memory and had a good locality of reference. However, a bigger sample size of WebAssembly programs is required to give a more precise estimate of the effect of this optimzation.

7.5 Handling the external state

The previous sections always assumed that the program runs deterministic and that there is no external state which influences the program. In real world situations, this is obviously not the case and many transitions start to depend on the external state.

This is problematic when determining whether a program halts or not, as finding a cycle in the program execution is no longer an clear proof that the program will never halt. The cycle could still be left by an transition that wasn't taken in a previous iteration because the external state has changed since then.

wasmint approaches this problem by ensuring that all transitions in a cycle are in fact deterministic and don't depend on the external state. This is done by looking for the currently only source of non-determinism or external state influence in the VM: native function calls.

If wasmint can't find a transition inside the cycle that is a native function call and not marked as deterministic (see section 6.3.3), then the cycle won't be left by the program and the program will not halt.

If wasmint finds such a function call, it will continue to execute until it finds a cycle which can't be left.

7.6 Remaining limitations

The biggest limitation of this concept in practice is that some programs have cycles with an enormous amount of states in them.

As the current strategy only works when the program has already finished the first iteration of the cycle, it needs at least as much time to find the cycle as it would take to iterate over the cycle. So if a program has a cycle which takes one day to iterate through, the current system also would need to run this program at least for one day to find the cycle.

A simple example for such a long cycle would be an infinite loop which periodically increments a 64-bit integer variable. The cycle length for this simple program is 2^{64} states which is far too big to be analysed with our current program in an acceptable amount of time.

This problem is with the current concept of running and analysing the program exe-

cution not solveable.

However, for programs that don't have such a big cycle length, this concept can effectivly determine whether they will halt or not.

7.7 Example

In the introduction a program with one megabyte of memory that runs for one million steps was mentioned as an example of a program that seems impossible to analyse with the concept of finding two equal states, as the resulting one terabyte of programm execution data are beyond the capabilities of an average computer.

To demonstrate the effectiveness of the optimzations from the previos sections, wasmint will analyse a program[18] with one megabyte of memory that runs for 30 million steps. The result are 30 terabyte of program execution compressed with the reverse execution engine into about two megabyte of memory.

When analysing these 30 terabyte with the concepts from above and a checkpoint interval of five million steps, wasmint needs about 4.9 seconds to determine that the program won't halt.

It had to analyse about 75 million execution steps in total (due to the periodic checking) and eliminated state sequences of a total of 35 million steps with the reasoning from section 7.4.

8 Other debugging capabilities of wasmint

The following sections give a brief overview over some other debugging features of wasmint that weren't mentioned so far in this paper.

8.1 Breakpoints on instructions

wasmint allows setting breakpoints to abitrary instructions. These breakpoints pause the VM when reached and are a common features in many debuggers.

All breakpoints inside the VM are identified by their address in their respective byte code segment similar to the way Intel implemented their hardware breakpoint support in the 80386 microprocessor [4].

When adding a breakpoint, wasmint uses the debug informaton generated by the just-in-time compiler and looks up the address of the related opcodes inside the byte code. This address and the related opcode are then added to an internal hash map.

While running in debug mode, wasmint checks after each instruction if the current instruction pointer value can be found in this hash map. If an entry was found, wasmint pauses the execution and calls the breakpoint handler that the user has specified.

The user can then inspect the current state of the virtual machine including its linear memory and the thread stack which includes all function frames with their local variables.

8.2 Manipulating temporary values

WebAssembly can express a wide range of programs without the use of local variables as nearly all instructions are considered expressions that can be nested

It is therefore desireable to allow inspecting and changing the return value of arbitrary expressions during runtime. wasmint implements this by providing a interface for accessing the register that is used by an instruction to store its return value. As the registers in the virtual machine can be used by multiple instructions at different times, this interface can only be accessed for the instruction that is targetted by the current breakpoint.

What register is used by the given instruction is determined during runtime from the debug information supplied by the just-in-time compiler.

8.3 Saving the virtual machine state

When reporting a bug to an upstream project it is usually encouraged by the developers to describe steps to reproduce this issue. This however requires that the bug reporter can estimate what information is essential for reproducing and how to correctly report them.

A easier way to handle this situation is to record the program execution of the problem and then only provide the recording of the program execution. This requires that all parts of the wasmint VM are serializeable and that the virtual machine can reconstruct itself from this serialized form.

This feature has many other applications such as transferring running programs between machines or hibernating them in out-of-memory scenarios.

9 Testing wasmint

Testing interpreters for programming languages is a difficult task. They accept input formats that are much more complex than those of many other programs, have to reliably sandbox their running program while still being performant and need to keep up with development of the programming language itself.

Additionally, developers have increased their expectations of their runtime environment and they demand such a high quality from them, that they can always assume that unexpected behaviour from their programs is caused by the program itself and not by the interpreter.

Assuring this kind of software quality for wasmint requires better strategies than the traditional way of writing tests by hand, as writing and maintaining such a vast collection of test would require more time than the tight schedule of the projects allows.

For this reason wasmint uses the same approach that bigger and more complex projects such as the V8 or the SpiderMonkey JavaScript engines use. Instead of having developers write test cases by themselves, they employ fuzzing programs to test their software. [8, p. 1][6]

Fuzzing can be described "as a method for discovering faults in software by providing unexpected input and monitoring for exceptions. It is typically an automated or semi-automated process that involves repeatedly manipulating and supplying data to target software for processing."[31, p. 22] and is mostly used to find security vulnerabilities in applications [6][23].

In wasmint, fuzzing is not only used to find security vulnerabilities but also to test if certain more complex features work correctly.

9.1 Identifying testable trust boundaries and testable interfaces

Fuzzing consists of supplying specially crafted data that crosses a trust boundary between the used interface and the fuzzing targets. Identifying interfaces inside the software that allow crossing such a trust boundary is essential for effective fuzzing [23]

For compilers and interpreters the most important trust boundary exists in the process that converts the raw source code files to the expected result (e.g. an executable or

the result of the computation). This means that the fuzzer has to provide the program source and then starts the interpreter to evaluate the generated source code.

9.2 Generation of test cases

The most important task of the fuzzer is to generate test cases that use as many parts of wasmint as possible. While this is in theory possible by randomly generating strings and then interpreting them as WebAssembly source code, it will take in practice far too long to even generate a single program that is accepted by the parser.

For this reason the fuzzer relies on automatic protocol generation testing [31, p. 36] which takes the protocol that the interface uses into account.

In wasmint this protocol is defined by the WebAssembly grammar which already defines a set of production rules that the generator can peform to get a syntactically valid program. The test case generation therefore begins with a start symbol and then applies randomly choosen production rules until it is left with a character string that contains no start symbol and no nonterminal symbols.

The character strings created by this strategy have a good chance to pass the parsing step and to be actually loaded into the wasmint virtual machine. However, the character strings will not produce test cases where the WebAssembly grammar was disrespected and therefore don't apply any stress on the wasm-module parser.

So the fuzzer appends a second step to the test case generation process that works in a similar way as mutation fuzzing: It takes random characters from the generated character string and performs random replace, delete and insert operations on it. It only performs this step on a small percent of the test cases as it has a high chance on breaking the grammar and could therefore drastically lower the percentage of test cases that are accepted by the parser and get a chance to test the virtual machine.[31]

9.3 Heuristics for the automatic verification of test cases

One of problems that come with fuzzing is the verification of each test run. In manually crafted test cases the computed results are usually compared with expected results that are specified by the developer.

When using fuzzing it is not practicable to verify the results of each test run by a

human developer as the output of fuzzers can reach up to fifty million test cases per day [6, p. 1]. So the decision whether a test case has found undesireable behaviour or not has to be up to the fuzzer.

The fuzzer is however in a difficult situation as it has usually no expected result which can be compared with the actual result of the test case. It is therefore not possible to make for every test case a definitive decision on whether the test case has passed or not.

The concept that most fuzzers follow is to make a decision based on heuristics. These heuristics should have a zero percent false positive rate if possible, because even a false positive rate of 0.001 percent would create hundreds of wrong results per day.

To reach such a low rate, the fuzzer has to rely on several heuristics that detect clear proof of an unsuccessful test run. These heuristics are described in the following sections.

9.3.1 Verification with ubsan

Undefined behavior in programs is hard to find and causes bugs that only show symptons when the program is ported to other compilers and systems. While such behavior is sometimes detected during the compilation, it is often "not the case and the checking has to take place at run time."[26].

These runtime checks could be implemented by hand, but this would clutter the code with complex and hard to understand checks.

A better solution is ubsan — the Undefined Behavior Sanitizer. It's a compiler feature of the GCC and clang compilers and automatically compiles such runtime checks into the code. These checks range from misaligned memory access to undefined arithmetic operations inside. [32][26]

In the fuzzer, these checks are instrumented to abort the fuzzing process as this will trigger the generation of a test case report. The fuzzer is designed to handle crashes correctly as discussed in 9.4.

9.3.2 Verification with Valgrind

The memcheck tool of Valgrind can monitor programs for memory related problems and is often seen in combination with test cases. It can detect serveral invalid memory operations such as accessing memory that was not initialized or already free'd, or not freeing allocated memory.[10]

While Valgrind's memcheck seems to be an good way to verify testruns in the fuzzer, it comes with a significant performance overhead. This performance overhead reduces the amount of test cases the fuzzer can perform from about 370 tests per second to just 9 tests per second.

In addition to that is Valgrind's memcheck also not able to inform the fuzzer about failed test cases. It only prints out information on the command line but offers no way to abort the program. This means that the fuzzer can't create a test case report for this specific case.

For these reasons Valgrind's memcheck isn't used by the fuzzer in the default configuration. However, with a specialized test case generator that tries to create programs that would trigger invalid memory behavior, memcheck could still offer a good heuristic.

9.3.3 Verification of the wasm-module parser with serializer

Parsing and serializing are two functions that complete each other. The parser accepts source code as a valid input and generates an intermediate representation. The serializer accepts this intermediate representation as a valid input and generates source code from it.

This relationship offers an unique way to automatically test both the parser and the serializer as they can verify each other in a test case.

Verifying a test case happens in four steps as illustrated in 14:

The first step is to parse the source code of the test case. This step is only necessary because the test case generator emits source code.

The second step is to serialize the IR from step one into source code. One could leap to the conclusion now that the emitted source code needs to be identical with the test case input, but as the parsing process ignores comments and formatting, this isn't the case in most situations.

However, the program that the test case and the output of the serializer represent should be identical.

The third step is therefore to parse the generated source code again and create a second IR.

As the IR only contains information that is important to the program execution and neither parser nor serializer are allowed to change the meaning of the program, both intermedate representations need to be identical.

The fourth step is to compare the IR from step one and the IR from step three and report a test case failure if they don't match.

This verification tests that the serializer and parser accept each other's output as a valid input.

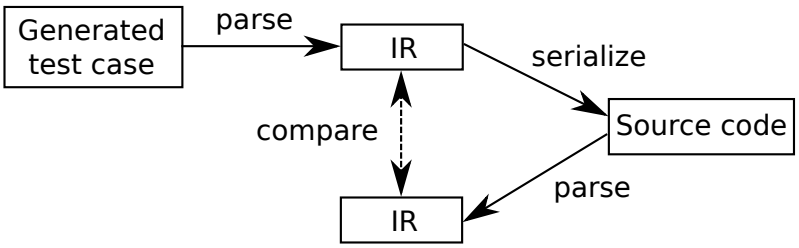

Figure 14: Testing process for the parser and serializer.

9.3.4 Verification of the monitoring debug functionality

Many of the debug features of wasmint allow the user to monitor certain parts of the running program and don't influcence the program execution.

Those features are tested by running the test case twice: Once without debugging functionality which creates an machine state that we consider as the expected result of our test. Then the test case is run a second time and the debug features such as breakpoints and reading variables are activated. Afterwards the final states of both runs are compared and have to match for the test case to be successful.

9.3.5 Verification of the reverse execution

When testing the reverse execution engine the fuzzer has the advantage that it exactly knows all expected results: The previous states.

The verification happens by first running the test case program for a certain amount of time or until it finishes and randomly taking snapshots of the VM.

Afterwards, the reverse execution engine is asked to reproduce each snapshot. If the reproduced result matches the snapshot, the reverse execution engine has passed the test.

9.4 Documentation and reproduction of test cases

If the fuzzer has generated a test case which causes suspicious behaviour it needs to inform the developer that a closer inspection of this specific test case is necessary. The developer needs for this the ability to reproduce the test case which is one of the corner stones of effective fuzzing.[31, p. 62]

The fuzzer in wasmint provides two different ways of reproducing test cases:

1. A test case number which allows reproducing the whole test case. This is possible because all randomness that is needed for generating a test case comes from a single standardized random number generator. The seed for this generator is the test case number. The advantage of using seed numbers is that it is more comfortable for the developer to reproduce and it creates smaller log files. On the other hand seed numbers also require that the algorithm for creating test cases won't change over time.
2. Printing the whole source code of the test case to the fuzzer log. This creates fairly big log files but allows changing the test case generation algorithm while keeping already found test cases intact.

As the fuzzer is at the time of writing still in development and has no stable code base, it prints the whole source code to the log. After the code base has become more mature, the fuzzer will probably start using test case numbers.

It should be noted that the fuzzer runs for performance reasons all tests in the same process as the fuzzer itself and is therefore also affected from being stopped by an

operation system signal such as a segmentation fault. This would also mean that the generated test case that caused the signal is not reported as the processes is terminated. For this reason the fuzzers attachs a signal handler that rescues at least the test case number in a situation where the whole process gets terminated by the operation system.

9.5 Reducing test cases

The programs generated by the fuzzer can be very long and complex which makes them hard to debug for humans. This is problematic as all failed test cases have to be assessed by a a human developer. To make this assessment easier, a test case should be prepared beforehand by a program which makes the program simpler without removing the characteristic that causes the test to fail.[27]

An example for such a tool that minimizes C/C++/OpenCL programs is C-Reduce. However, C-Reduce doesn't support WebAssembly at the time of writing. This makes the fuzzer not as effective as it could be. It would be desireable to have a WebAssembly version of C-Reduce that could provide this functionality in the future.

10 Conlusions

The project had two major goals: Researching on how WebAssembly performs in pratice and providing a platform for creating useful developer tools. wasmint — the final result of this project — proved to be both an interesting case study as a WebAssembly implementation and also a pratical backend for debuggers and other tools.

As one of the first interpreters for WebAssembly, wasmint shows that it is possible to execute WebAssembly with acceptable performance and without depending on complex compiler toolchains. It also shows how some of the properties of WebAssembly — such as the two different serialization formats or type inference — can be approached.

Last but not least it helped defining the WebAssembly standard with the insight I recieved from implementing it.

As the only debugging platform for WebAssembly at the time of writing, wasmint offers a wide range of functionality that covers use cases from the simple manipulation of data to reverse execution. It also introduced new debugging techniques such as the detecion of programs that don't halt based on Minsky's theories.

Future research in this area could either improve wasmint or use the knowledge gained from wasmint to improve other projects.

The most significant knowledge gained from wasmint that could be useful for other project is the detection mechanism for programs that don't halt. This detection system could be ported to other virtual machines and platforms. Most importantly, implementing this feature in LLVM would make this idea available to a wider audience.

In wasmint improving the performance is necessary to keep the project competitive in the long term. This can be done by utilizing LLVM for optimizing and running parts of the interpreted program. Also the wasmint just-in-time compiler is still only compiling debug code and would benefit from an optimizer.

A last subject of research that should be mentioned is the WebAssembly standard itself: The working group always welcomes suggestions that improve the quality of the platform and further research that evaluates existing and proposed features would benefit the working group, the WebAssembly standard and — due to the fact that WebAssembly is one of the most promising projects for a new default language of the web — the future of the internet.

Listings

List of Figures

References

[1] Mozilla is unlocking the power of the web as a platform for gaming. https://blog.mozilla.org/blog/2013/03/27/mozilla-is-unlocking-the-power-of-the-web-as-a-platform-for-gaming/. Accessed: 2015-12-28.

[2] Undodb homepage. http://undo-software.com/undodb/. Accessed: 2015-01-05.

[3] Wavm readme. https://github.com/AndrewScheidecker/WAVM/blob/d8095daac793bd51d4e4528637263c6fa480ddfd/README.md#design. Accessed: 2016-02-03.

[4] *INTEL 80386 PROGRAMMER'S REFERENCE MANUAL*. Intel Corporation, 1986.

[5] A.V. Aho. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

[6] Abhishek Arya and Cris Neckar. Fuzzing for security. http://blog.chromium.org/2012/04/fuzzing-for-security.html. Accessed: 2015-01-05.

[7] Peter Bright. The web is getting its bytecode: Webassembly. http://arstechnica.com/information-technology/2015/06/the-web-is-getting-its-bytecode-webassembly/. Accessed: 2015-12-28.

[8] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. *SIGPLAN Not.*, 48(6):197–208, June 2013.

[9] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, July 2005.

[10] Valgrind Developers. Valgrind user manual. http://valgrind.org/docs/manual/mc-manual.html, note = Accessed: 2016-02-16.

[11] Daniel Drake and Johannes Berg. Unaligned memory accesses. https://www.kernel.org/doc/Documentation/unaligned-memory-access.txt. Accessed: 2015-02-10.

[12] Jonas Echterhoff. On the future of web publishing in unity. http://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/. Accessed: 2015-12-28.

[13] Brendan Eich. From asm.js to webassembly. `https://brendaneich.com/2015/06/from-asm-js-to-webassembly/`. Accessed: 2015-12-28.

[14] J. Engblom. A review of reverse debugging. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pages 1–6, Sept 2012.

[15] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. *SIGPLAN Not.*, 24(1):112–123, November 1988.

[16] P. Flajolet, J.C. Raoult, and J. Vuillemin. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99 – 125, 1979.

[17] Dave Herman. Readme file of the asm.js github repository. `https://github.com/dherman/asm.js/blob/d46b6955ddaf627b2d0d57583727c4a674435f7c/README.md`. Accessed: 2015-12-26.

[18] Raphael Isemann. Infinite quicksort program. `https://github.com/WebAssembly/wasmint/blob/d01fbe149b6e09be8cc7a6a3a606624d6617b0f4/tools/halting_problem/infinite_tests/quicksort.wasm`, note = Accessed: 2016-02-16.

[19] Raphael Isemann. wasmint performance tests. `https://github.com/WebAssembly/wasmint/tree/d01fbe149b6e09be8cc7a6a3a606624d6617b0f4/libwasmint/tests/performance`. Accessed: 2015-02-15.

[20] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

[21] P. Lubbers, B. Albers, and F. Salim. *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*. Expert's voice in Web development. Apress, 2010.

[22] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[23] John Neystadt. Automated penetration testing with white-box fuzzing. `https://msdn.microsoft.com/en-us/library/cc162782.aspx`, 2008. Accessed: 2016-01-3.

[24] Robert O'Callahan. Changing the world. `https://ecs.victoria.ac.nz/foswiki/pub/Events/NZCSRSC2010/Keynotes/ChangingTheWorld.pdf`. Accessed: 2015-02-04.

[25] Robert O'Callahan. rr project homepage. `http://rr-project.org/`. Accessed: 2015-12-29.

[26] Marek Polacek. Gcc undefined behavior sanitizer ubsan. `http://developerblog.redhat.com/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan/`. Accessed: 2016-02-15.

[27] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. *SIGPLAN Not.*, 47(6):335–346, June 2012.

[28] Libby Reinish. Reversible debugging in gdb in the "high priority free software projects" list from the fsf. `https://www.fsf.org/campaigns/priority-projects/priority-projects/highpriorityprojects#GDB`. Accessed: 2015-12-22.

[29] Andreas Rossberg. fac.wast in the webassembly testsuite. `https://github.com/WebAssembly/spec/blob/ab726152ee45f3646f25c6ebb6053be61df25887/ml-proto/test/fac.wast`. Accessed: 2015-12-30.

[30] James E. Smith and Ravi Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. Elsevier Inc.

[31] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[32] The Clang Team. Gcc undefined behavior sanitizer ubsan. `http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`. Accessed: 2016-02-15.

[33] Luke Wagner. Webassembly. `https://blog.mozilla.org/luke/2015/06/17/webassembly/`. Accessed: 2015-12-28.

[34] B. Yee, D. Sehr, G. Dardyk, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93, May 2009.

Erklärung

Ich versichere, dass ich diese Bachelorarbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Kempten, den ....

Ermächtigung

Hiermit ermächtige ich die Hochschule Kempten zur Veröffentlichung der Kurzzusammenfassung (Abstract) meiner Arbeit, z. Bsp. auf gedruckten Medien oder auf einer Internetseite.

Kempten, den ....