Reverse Engineering WebAssembly

Nicolas Falliere, PNF Software - nico@pnfsoftware.com
Last revision: July 17 2018 (#2)
PDF version: http://www.pnfsoftware.com/reversing-wasm.pdf

This article is an introduction to WebAssembly geared towards reverse-engineers. It focuses on understanding the binary format, virtual machine, execution environment, implementation details and binary interfaces, in order for the reader to acquire the skills to analyze wasm binary modules. The annex details the representation of WebAssembly in JEB and how to use it to analyze wasm binary modules.

Introduction

WebAssembly (wasm) is a binary instruction format for a virtual machine (VM) whose primary goal is to run in-browser application code. It is meant to complement JavaScript, for instance to enhance the performance of CPU intensive components of a web app. Source languages that can currently be compiled to WebAssembly bytecode include common strongly typed languages such as C, C++, Java, or even TypeScript.[1] A wasm binary file defines an application module, and its code can be loaded and executed by external components, including regular JavaScript files.[2] All major browsers are shipping with WebAssembly support, and its prime backers include Mozilla and Google. WebAssembly is not limited to client code execution though. The VM is general-purpose, and can be used to run server code, such as node.js applications, or distributed application code, such as smart contracts.[3]

From an auditing - and by extension security - perspective, executing binary blobs of machine code or bytecode has implications as well: increased potential for obfuscation means readability may decrease. Although one could argue that minified JavaScript can be difficult to comprehend, using a beautifier, formatter, and refactoring tool is an easy way to start making sense of a piece of code. Matters are different when it comes to binary code: the verbosity, the lack of evident structure, the ability to easily modify the shape (eg, by modifying the control flow) and the form (eg, by substituting code by equivalent code) of a piece of code makes any reverse engineering endeavor theoretically and practically more complicated.[4] Static and dynamic analysis tools are invaluable when it comes to tackling such programs, malicious or otherwise.

People involved in WebAssembly development have published a variety of tools that can be used to examine compiled wasm modules. JEB ships with a WebAssembly plugin comprising a wasm parser and disassembler, as well as a wasm decompiler. We will detail how to use them in the annex of this article.

WebAssembly Modules

Since WebAssembly is a relatively unknown topic, we will first present the wasm binary format and bytecode. A wasm binary starts with the four magic bytes `\x00asm`, followed by a 32-bit integer version number, 1, the version of the current Minimum Viable Product (MVP)

[1] JavaScript may be an accepted source language in the future; however weakly typed languages pose additional challenges that are not resolved by the current WebAssembly MVP.
[2] This paper does not focus on the JavaScript parts and glue of WebAssembly.
[3] As an example, the EOS blockchain uses WebAssembly bytecode for its smart contracts.
[4] The aforementioned techniques could also be applied to script files; however in practice, they are not applied to the degree and extent on which they commonly are on binary files.

implementing the WebAssembly specifications. Note that multi-byte integers are stored in little-endian format.

Sections immediately follow the header.

Sections

A wasm module contains sections. Well known sections are identified by an integer id, they are: [5]

- `Type`: function signatures
- `Import`: import declarations
- `Export`: exported declarations
- `Function[*]`: function declarations of imported and internally defined functions (bodies of internal functions are defined in the Code section)
- `Memory[*]`: a list of memory element specifications (initial size, maximum size) - currently, a single memory element is allowed per wasm module
- `Data`: bytes used to initialize memory elements
- `Table[*]`: function pointer table element specifications - currently, a single table element is allowed per wasm module
- `Element`: references used to initializers of a table element
- `Global[*]`: global variables (not mapped in the data section)
- `Start`: the entry-point function reference (optional)
- `Code`: the internal function bodies (incl. bytecode)

Additional sections (custom) may be present, and are identified by name. Two well-known custom sections include:

- `"name"`: debugging information such as symbol names
- `"dylink"`: dynamic linking information generated when compiling modules with SIDE_MODULE=1

Indexed Spaces

The differences between a wasm binary and a traditional binary encompass the typical differences found between a virtual machine executing bytecode and a common processor:[6] code and data are residing on two or more memory spaces, or on a single memory space, respectively. In the case of wasm, we are effectively dealing with four spaces. Those spaces can be further defined as *indexed spaces*, as their items can be accessed by a 0-based integer index.

[5] The *[*]* suffix denotes a section defining an indexed space, as explained below
[6] By common processor, we mean a processor having an almost-von-Neumann architecture.

- The **memory space** (read-write[7]), a list of linear memories, containing data elements, accessible by the load/store operators
- The **table space** (read-only), a list of tables, containing function pointers used for indirect invocation of functions
- The **global space** (read-only or read-write), containing read-write immutable global variables
- The **function space** (read-only), containing all imported and internal functions (and their bodies)

From an execution environment standpoint, two additional spaces exist:

- The local and parameter variables space, holding temporary, per-function variables as well as input function parameters
- The operand stack[8], used by instructions to push and pop its operands

Primitive types

The typing system is relatively more compact than other VM's. There are four primitive types:

- i32: 32-bit integer (the signedness depends on the manipulating operator)
- i64: 64-bit integer (the signedness depends on the manipulating operator)
- f32: 32-bit float (IEEE 754 signed)
- f64: 64-bit float (IEEE 754 signed)

The WebAssembly is specified to operate within 32-bit or 64-bit memory environments. The runtimes are called wasm32 or wasm64, respectively. Pointers in wasm32 are stored on i32 typed variables; pointers in wasm64 are stored on i64 typed variables. As of the MVP, only wasm32 has been implemented, and the rest of this article assumes four-byte pointer types.

Primitives smaller than 32-bit, such as an 8-bit byte or a 16-bit character can be stored on and manipulated on i32 variables.

WebAssembly Instructions

The WebAssembly VM is stack-based - with a twist: control can only flow to explicit labels.[9] It defines 172 operators. The instruction opcodes are encoded on a single byte. The immediate operands (referred to as *immediates*) follow. Other instruction operands (simply referred to as *operands*) are pushed on the operand stack. An instruction consumes (pops) its operand(s)

[7] The rw/ro annotations are from a user code execution point of view.
[8] Not to be confused with the memory stack, optionally used by functions for memory-mapped locals, as we will see later.
[9] The VM is deemed *structured stack-based* by WebAssembly authors. An example of non-structured stack-based VM would be the JVM executing Java bytecode.

from the stack and produces (pushes) zero or one result. Refer to the above link for a complete list.

Example:

```
21 04   set_local $L4
```

- The instruction is encoded as 0x21 0x04
- The operator set_local takes one immediate operator, the local variable index
- It consumes a stack operand, let's call it $S0
- $S0 is assigned to $L4

Operator categories

The instruction set can be broken down into the following important categories:

- Control flow operators
    - Labels definition instructions (via branch/loop/if instructions)
    - Unconditional branch instruction
    - Conditional branch instruction
    - Switch-like branch instruction
    - Branch to functions (directly or indirectly) instructions
- Locals access operators
    - Read/write instructions for function parameters and local variables
- Memory access operators
    - Load/store instructions, down to a byte granularity with support for sign- or zero-extension
- Const operators
    - Instructions to push hard-coded integers and floats on the stack
- Comparison operators
    - Equality and inequality checking instructions, etc.
- Arithmetic operators
    - Common operators
        - add/sub/mul/div/etc.
    - Integer operators
        - and/or/xor/shift/etc.
    - Float operators
        - ceil/trunc/nearest/etc.
- Conversion and cast operators
    - Truncate/extend from float to int, int to float
    - Promote/demote for floats
    - Reinterpret from int to float

Most instructions are self-explanatory. The control flow operators, however, require additional explanation.

Control flow

Reverse engineers used to dealing with machine code, register-based stack machines, or even regular stack-based machines, may be slightly confused at first. Let's recall that the WebAssembly VM is structured stack-based: control cannot flow to arbitrary locations. It flows:

- Either to the next instruction, if the current instruction is not branching
- Or a "labeled" instruction

Labels can be defined using one of the three operators:

- BLOCK: define a block of code and a break-like (as in C's break) label
- LOOP: define a block of code and a continue-like (as in C's continue) label
- IF: define a conditional-entry block

Intra-function branching can be done using:

- BR: unconditional branching
- BR_IF: conditional branching
- BR_TABLE: switch-like branching
- IF blocks IF/ELSE blocks

Inter-function branching can be done using:

- CALL for static call invocation
- CALL_INDIRECT for dynamic callsite invocation (dynamic dispatch is detailed later)

Defining a label can be seen as defining a nested block of instructions. Each block is explicitly closed by the END operator. The function body itself is an implicit block, and therefore is terminated by the END operator as well.

Have a look at the following wasm bytecode snippet:

```
Function _crc32_init: (-)-
    +0000h: get_global $g10
    +0002h: set_local $30
    +0004h: get_global $g10
    +0006h: i32.const 16
    +0008h: i32.add
    +0009h: set_global $g10
    ......................
    +001Dh: i32.const 5243024
    +0022h: i32.add
    +0023h: i32.const 0
    +0025h: i32.store @0h(4)
    +0028h: i32.const 128
    +002Bh: set_local $0
```

```
+002Dh: loop $1
+002Fh:   block $2
+0031h:     get_local $0
+0033h:     set_local $22
+0035h:     get_local $22
+0037h:     i32.const 0
+0039h:     i32.ne
+003Ah:     set_local $23
+003Ch:     get_local $23
+003Eh:     i32.eqz
+003Fh:     if $3
+0041h:       br $2       (---> break out of $2 (BLOCK))
+0043h:       end
+0044h:     get_local $12
+0046h:     set_local $24
+0048h:     get_local $24
.......................
+0106h:     get_local $20
+0108h:     i32.const 1
+010Ah:     i32.shr_u
+010Bh:     set_local $21
+010Dh:     get_local $21
+010Fh:     set_local $0
+0111h:     br $1         (---> continue to $1 (LOOP))
+0113h:     end
+0114h:   end
+0115h: get_local $30
+0117h: set_global $g10
+0119h: return
+011Ah: end
```

This function implements a CRC table initializer. In the snippet above:

- Branching to label $1, defined by a LOOP operator, means jumping to 2Dh, the address of the LOOP operator itself
- Branching to label $2, however, means jumping to 113h (not 2Fh), the address where the END of the block $2 is defined!
- Notice the IF operator at address 3Fh: the instructions in the if-block are executed only if the condition previously pushed on the stack is true
- Also have a look at BR $2: this instruction branches to the end of block $2 as well

Overall, one may understand:

- “BR to BLOCK-block X” as the C-like instruction “break labelX;”
- “BR to LOOP-block X” as the C-like instruction “continue labelX;”

These constructs restrain the flow of execution to well-known labels, just like a high-level programming language would. The idea behind this is to have functions that can always be represented as reducible flow graphs - unlike goto-supporting, lower level languages allowing

branching inside blocks, and yielding non-reducible flow graphs. In practice though, everything could be made a block, at the expense of extra BLOCK/LOOP instructions.[10]

Step-by-step example

We will now have a look at complete function and step through it:

```
Function f: (i32,i32)i32
    0 get_local $0                    [1]
    0 get_local $1                    [2]
    0 i32.le_s                        [1]
    0 set_local $2                    [0]
    0 get_local $2                    [1]
    0 if $I1(i32)                     [0]
    1   i32.const 3                   [1]
    1   else                          [0]
    1   i32.const 4                   [1]
    1   end                           [1]
    0 set_local $3                    [0]
    0 get_local $3                    [1]
    0 return                          [0]
    0 end                             [0]
```

Details:

- The prototype of this function `(i32,i32)i32` means it takes two i32 parameters, and returns an i32.
- The first number indicates the depth level in blocks; the function starts at depth zero.
- The trailing number in angle brackets “[x]” indicates the size of the operand stack after execution of the instruction; the function starts with an empty stack.
- The function parameters are mapped to the first local variable slots. Therefore, the first parameter is local $0 and the second one, local $1
- GET_LOCAL $0 pushes the first parameter onto the stack
- GET_LOCAL $1 pushes the second parameter onto the stack
- I32.LE_S pops two operands, does a signed less-than-or-equals comparison, and pushes the result onto the stack
- Note that the IF instruction specifies a return type as well: an integer is pushed and the result of the if-else-block
- If $0<=$1, the value 3 is pushed onto the stack; else, the value 4 is pushed onto the stack
- SET_LOCAL $3 pops the result-value of the if-block and assigns it the the local $3

[10] Another constraint is enforced by the WebAssembly VM though: a block cannot access the operand stack variables that were pushed by the outer block operators. This restriction does not apply to locals, obviously.

- GET_LOCAL $3 pushes it back onto the stack, while the return instruction pops it as the return value of the function
- Note that after the final return is executed, the stack level is back to 0, just like it was at the function entry-point

The above assembly snippet is equivalent to the high-level statement:

```
int f(int a, int b) {
    return a <= b ? 3: 4;
}
```

S-Expressions

A common representation of WebAssembly modules, including function bodies and instructions, is using a folded symbolic expression syntax. Here's the equivalent S-expression of the linear listing of f.[11]

```
(func $_f (type $t1) (param $p0 i32) (param $p1 i32) (result i32)
  (set_local $l2
    (i32.le_s
      (get_local $p0)
      (get_local $p1)))
  (set_local $l3
    (if $I1 (result i32)
      (get_local $l2)
      (then
        (i32.const 3))
      (else
        (i32.const 4))))
  (return
    (get_local $l3)))
```

The wasm2wat tool generates an S-expression equivalent representation of a wasm binary module.[12]

Throughout the rest of this document, we will alternate between S-expressions, useful to represent full wasm modules, and linear listings, closely tied to the underlying byte sequences of function bodies.

Accessing the memory

Recall that the current WebAssembly specifications allow the definition of a single linear memory space. Modules may import a reference to an already existing memory space. The load

[11] The reader accustomed to practicing Lisp or its derivatives will feel right at home manipulating sexps.
[12] An online version of the tool is available as well.

and store operators are the sole operators a program can use to read or write memory, down to a single byte granularity.

- `<type>.load[ext]` specify how to load from memory to the stack.
  - Example: i32.load8_s will load an 8-byte integer and sign extend it into 32-bit integer pushed on the stack
- `<type>.store[ext]` specify how to store to memory from the stack.
  - Example: i64.store32 will pop an i64, truncate it to an i32, and store it to memory

The load/store operators work on the *default linear memory*, which is the first and currently unique memory space of a WebAssembly instance.[13]

WebAssembly Implementation Details

This function focuses on implementation details relevant to the compilation of C and C++ programs to WebAssembly. Compiling from another source language, such as Rust, will certainly yield additional constructs and idioms.[14]

Let's compile the simplest C program to WebAssembly with the emcc compiler:

```
1.c: int main(void){return 0;}
```

using the command-line:

```
$ emcc 1.c -s WASM=1 -o 1.html
```

We can use wasm2wat to get an S-expression representation of the compiled 1.wasm module:

```
$ wasm2wat 1.wasm
(module
  (type $t0 (func (param i32 i32 i32) (result i32)))
  (type $t1 (func (param i32) (result i32)))
  (type $t2 (func (result i32)))
  (type $t3 (func (param i32)))
  (type $t4 (func (param i32 i32) (result i32)))
  (type $t5 (func (param i32 i32)))
  (type $t6 (func))
  (type $t7 (func (param i32 i32 i32 i32) (result i32)))
  (import "env" "memory" (memory $env.memory 256 256))
  (import "env" "table" (table $env.table 10 10 anyfunc))
  (import "env" "memoryBase" (global $env.memoryBase i32))
  (import "env" "tableBase" (global $env.tableBase i32))
  (import "env" "DYNAMICTOP_PTR" (global $env.DYNAMICTOP_PTR i32))
  (import "env" "tempDoublePtr" (global $env.tempDoublePtr i32))
  (import "env" "ABORT" (global $env.ABORT i32))
  (import "env" "STACKTOP" (global $env.STACKTOP i32))
  (import "env" "STACK_MAX" (global $env.STACK_MAX i32))
  (import "global" "NaN" (global $global.NaN f64))
```

[13] Load/store operators have a reserved immediate operand that may be used by future additions to specify on which memory space the operation should be performed.

[14] Future updates of this paper may cover {other language}-specific binary details in the context of wasm.

```
  (import "global" "Infinity" (global $global.Infinity f64))
  (import "env" "enlargeMemory" (func $env.enlargeMemory (type $t2)))
  (import "env" "getTotalMemory" (func $env.getTotalMemory (type $t2)))
  (import "env" "abortOnCannotGrowMemory" (func $env.abortOnCannotGrowMemory (type $t2)))
  (import "env" "abortStackOverflow" (func $env.abortStackOverflow (type $t3)))
  (import "env" "nullFunc_ii" (func $env.nullFunc_ii (type $t3)))
  (import "env" "nullFunc_iiii" (func $env.nullFunc_iiii (type $t3)))
  (import "env" "___lock" (func $env.___lock (type $t3)))
  (import "env" "___setErrNo" (func $env.___setErrNo (type $t3)))
  (import "env" "___syscall140" (func $env.___syscall140 (type $t4)))
  (import "env" "___syscall146" (func $env.___syscall146 (type $t4)))
  (import "env" "___syscall54" (func $env.___syscall54 (type $t4)))
  (import "env" "___syscall6" (func $env.___syscall6 (type $t4)))
  (import "env" "___unlock" (func $env.___unlock (type $t3)))
  (import "env" "_emscripten_memcpy_big" (func $env._emscripten_memcpy_big (type $t0)))
  (func $stackAlloc (export "stackAlloc") (type $t1) (param $p0 i32) (result i32)
    (local $l0 i32)
    (set_local $l0
      (get_global $g12))
      ... )
  (global $g9 (mut i32) (get_global 2))
  (global $g10 (mut i32) (get_global 3))
  (global $g11 (mut i32) (get_global 4))
  (global $g12 (mut i32) (get_global 5))
  (global $g13 (mut i32) (get_global 6))
  (global $g14 (mut i32) (i32.const 0))
  (global $g15 (mut i32) (i32.const 0))
  (global $g16 (mut i32) (i32.const 0))
  (global $g17 (mut i32) (i32.const 0))
  (global $g18 (mut f64) (get_global 7))
  (global $g19 (mut f64) (get_global 8))
  (global $g20 (mut i32) (i32.const 0))
  (global $g21 (mut i32) (i32.const 0))
  (global $g22 (mut i32) (i32.const 0))
  (global $g23 (mut i32) (i32.const 0))
  (global $g24 (mut f64) (f64.const 0x0p+0 (;=0;)))
  (global $g25 (mut i32) (i32.const 0))
  (global $g26 (mut f32) (f32.const 0x0p+0 (;=0;)))
  (global $g27 (mut f32) (f32.const 0x0p+0 (;=0;)))
  (elem (get_global 1) $f43 $f24 $f44 $f44 $f30 $f26 $f25 $f44 $f44 $f44)
  (data (i32.const 1024) "\05\00\00\00\..." [ trimmed, total 0x7E bytes])
)
```

This wasm file has seven sections:

- The function types, highlighted in red
- The imports, highlighted in dark green
- The exports, highlighted in blue (only functions are exported in this file)
- The internal function bodies, in black
- The list of internal globals, in dark red
- Table elements in light blue, used to initialize the imported table
- Data bytes in dark blue, used to initialize the imported linear memory

In the first section, we explained that a wasm file contains four separate indexed spaces. The spaces of 1.wasm are:

- functions: 14 imported (index 0...13), 31 internal (index 14...44)
- globals: 9 imported (index 0...8), 19 internal (index 9...27)
- memories: 1 imported ($env.memory)
- tables: 1 imported ($env.table)

Note that indexing of the elements of a space is zero-based. It starts with imported elements, then with internally-defined elements.

Implicit Memory Initialization

The memory space is initialized by the byte arrays contained in the Data section. In the case of 1.wasm:

- The linear memory is 256 pages long (a wasm page being 64Kb, that's 16Mb)
- 7Eh bytes of data "\05\00\00\00\..." are copied to address 1024 in that memory

Note that the destination address of a byte array defined in the Data section is not hardcoded as an integer; instead, it is the product of an initializer expression consisting of wasm instructions. The list of allowed operators supported by the MVP is quite limited:

- CONST-like operators (effectively providing the same functionality as a hardcoded address, as is the case in 1.wasm)
- GET_GLOBAL operator, allowing the target to be defined by a global variable

The first imported global, that could be named $g0, appears to always be the base of the default linear memory; it is named $env.memoryBase. The memoryBase global is virtually used by all load or store operations.

Implicit Table Initialization

The MVP specifications currently allow a single table space to be defined. The sample module 1.wasm imports its table element.

The table is initialized with elements that are pointers to any function type (referred to as *anyfunc* in wasm terminology, a C type equivalent would be *void**). Those functions may be the targets of indirect calls. Not all module functions are listed in the table space; only those susceptible of being invoked indirectly by the CALL_INDIRECT operator are.

In the case of 1.wasm, the table is initialized with 10 function pointers:
$f43 $f24 $f44 $f44 $f30 $f26 $f25 $f44 $f44 $f44

Explicit Module Initialization

A WebAssembly module can export a function entry named `__post_instantiate`. This function is executed by the WebAssembly runtime after its initialization.

Currently, emcc uses `__post_instantiate` to initialize:

- The memory stack pointers: Two globals that will hold the memory stack current pointer and maximum pointer are initialized.[15]
- Function pointers: The internal routine `runPostSets` is invoked to port over references to the function pointers in linear memory.

Stack pointer initialization

Depending on compilation options, the globals holding the stack boundaries are initialized using one of two ways.

1. Stack boundaries are pre-initialized by copying the values provided by two imported globals, aptly named env.STACKTOP and env.STACK_MAX. Example:

```
(module
  ...
  (import "env" "memoryBase" (global $env.memoryBase i32))
  (import "env" "tableBase" (global $env.tableBase i32))
  (import "env" "DYNAMICTOP_PTR" (global $env.DYNAMICTOP_PTR i32))
  (import "env" "tempDoublePtr" (global $env.tempDoublePtr i32))
  (import "env" "ABORT" (global $env.ABORT i32))
  (import "env" "STACKTOP" (global $env.STACKTOP i32))    <--- global $g5
  (import "env" "STACK_MAX" (global $env.STACK_MAX i32))  <--- global $g6
  ...
  (global $g9 (mut i32) (get_global 2))
  (global $g10 (mut i32) (get_global 3))
  (global $g11 (mut i32) (get_global 4))
  (global $g12 (mut i32) (get_global 5))  <--- init: $g12=$g5(=$env.STACKTOP)
  (global $g13 (mut i32) (get_global 6))  <--- init: $g13=$g6(=$env.STACK_MAX)
  ...
```

When allocating space for memory-mapped locals such as buffers and structures, the stack allocator routine verifies that ($g13-$g12) is large enough to accomodate the request. If so, the current value of $g12 is provided as the base for the function stackframe, and $g12 is updated; else, the function throws. When a function terminates, $g12 is reset to its original value. In this instance, $g12 is effectively the memory stack pointer of the routine.

[15] As foot-noted earlier, the memory stack is not to be confused with the operand stack.

2. Stack boundaries are initialized by `__post_instantiate` itself: the two globals holding the stack top and max values are derived from the memory base and placed after the space reserved initialized by the Data section (ie, C program's globals). Example:

```
(func $__post_instantiate (export "__post_instantiate") (type $t5)
  (set_global $g10
    (i32.add
      (get_global $env.memoryBase)
      (i32.const 112)))
  (set_global $g11
    (i32.add
      (get_global $g10)
      (i32.const 5242880)))
  (call $runPostSets))
```

Which translate to a five Mb stack:

```
/*TOP*/ $g10 = $memoryBase + 112;
/*MAX*/ $11 = $10 + 0x500000;
```

The layout of the linear memory can therefore be seen as:

```
0       TOP                                                    MAX
[-------|-------------------------------------------------------|........]
 ^ Data  ^ Memory Stack
 (C globals)
```

The load/store operators access the memory stack area just as they access the Data bytes.

Function pointer table initialization

Let's go back to the last snippet, and have a look at the `runPostSets` function body. It consists of a repeated sequence of instructions like the following:

```
  (i32.store
    (i32.add
      (get_global $env.memoryBase)
      (i32.const OFFSET))
    (i32.add
      (get_global $env.tableBase)
      (i32.const INDEX)))
```

The listing equivalent is:

```
  get_global $env.memoryBase
  i32.const OFFSET
  i32.add
```

```
    get_global $env.tableBase
    i32.const INDEX
    i32.add
    i32.store @0h(4)
```

The idiom translates to:[16]

```
*(void*)(memoryBase + OFFSET) = tableBase + INDEX
```

The indexes, offset by the fixed `tableBase`, of the function references of the Table entries are simply copied at the beginning of the linear memory space.

To summarize:

- This stub of code initialize all function pointer data; other non-pointer data (e.g. strings, immediates, initialized structures, etc.) were already initialized using the Data section.
- Operators never manipulate function pointers directly. Instead, they are referring to well-known function targets by index (in reality, tableBase-offset indices).[17]

Indirect Function Invocation

The binary implementation details necessary to understand how indirect invocation works have been laid out. The CALL_INDIRECT operator has two immediate operands:

- a type index, referencing a function prototype,
- and a reserved operand.

The target function operands are pushed on the operand stack; an additional operand is the index of the target routine.

Let's have a look at a second sample, 2.c:

```
int x0(int val) {return val + 8800;}
int x1(int val) {return val - 8811;}
int x2(int val) {return val * 8822;}
int x3(int val) {return val / 8833;}

typedef int (* PFUNC)(int);
PFUNC pfuncs[] = {x0, x1, x0, x2, x2};

int z(int val, int index) {
  PFUNC f = pfuncs[index + 2];
  return f(val);
}
```

[16] We are assuming an ILP32/wasm32 runtime - the only implementation supported by the MVP.
[17] The security implication in terms of code safety are out of scope here: refer to the WebAssembly design goals document for additional details.

compiled with emcc as a dynamic module (this allows us to keep all function names as is):

```
$ emcc 2.c -s WASM=1 -s SIDE_MODULE=1 -o 2.wasm
```

See below an S-expression representation of `z` and `runPostSets`:

```
(module
  ...
  (func $_z (export "_z") (type $t4) (param $p0 i32) (param $p1 i32) (result i32)
    ...
    (set_local $l2
      (get_local $p0))
    (set_local $l3
      (get_local $p1))
    (set_local $l5
      (get_local $l3))
    (set_local $l6
      (i32.add
        (get_local $l5)
        (i32.const 2)))
    (set_local $l7
      (i32.add
        (i32.add
          (get_global $env.memoryBase)
          (i32.const 0))
        (i32.shl
          (get_local $l6)
          (i32.const 2))))
    (set_local $l8
      (i32.load
        (get_local $l7)))
    (set_local $l4
      (get_local $l8))
    (set_local $l9
      (get_local $l4))
    (set_local $l0
      (get_local $l2))
    (set_local $l1
      (call_indirect (type $t0)
        (get_local $l0)
        (get_local $l9)))
    (set_global $g10
      (get_local $l11))
    (return
      (get_local $l1)))
  ...
  (func $runPostSets (export "runPostSets") (type $t5)
    (local $l0 i32)
    (i32.store
      (i32.add
        (get_global $env.memoryBase)
```

```
        (i32.const 0))
      (i32.add
        (get_global $env.tableBase)
        (i32.const 1)))
    (i32.store
      (i32.add
        (get_global $env.memoryBase)
        (i32.const 4))
      (i32.add
        (get_global $env.tableBase)
        (i32.const 2)))
...
(elem (get_global 1) $f14 $_x0 $_x1 $_x2 $_x3 $_z $f14 $f14))
```

Function `runPostsSets` copies the values {tableBase+}[1, 2, 1, 3, 4] to five i32 spots at `$memoryBase`. They are the indexes of x0, x1, x0, x2, x3, per the Elements section.

Function `z` could be represented as the following pseudo-code. The number of explicitly resolved indirection remains one. While native code would retrieve a pointer and dispatch the flow of execution to its address, the bytecode retrieves an index and lets the call operator retrieve the pointer associated with it. That amounts to another, implicit indirection to be resolved.

```
int FUNC_INDEX = *(i32*)(memoryBase + 0 + (2 + index) << 2);
return CALL_INDIRECT(val, FUNC_INDEX);
```

Function Locals

Local primitives can be mapped to *locals* and declared as such by the function header. Locals may be seen as the registers of a native program. They are manipulated by the {SET,GET,TEE}_LOCAL operators.

Memory-mapped locals such as buffers, however, may and most certainly will be allocated on the memory stack, bounded by the globals STACKTOP and STACK_MAX. For instance, this function allocates 1040 bytes on the memory stack:

```
(func $_fbu (type $t0) (param $p0 i32) (result i32)
  (local $l0 i32) ... (local $l14 i32)
  (set_local $l14       <--- SP($l14)
    (get_global $g10))
  (set_global $g10
    (i32.add
      (get_global $g10)
      (i32.const 1040)))
  (if $I0
    (i32.ge_s
```

```
        (get_global $g10)
        (get_global $g11))
      (then
        (call $env.abortStackOverflow
          (i32.const 1040))))
    ... routine body ...
    (set_global $g10
      (get_local $l14))
    (return ...
```

Here, $g10 holds the current memory stack pointer (SP). $g11 holds the memory stack upper limit. Note that the stack grows upward.[18] The above snippet can be translated to:

```
SP = STACKTOP;
STACKTOP += 1040;
if(SP >= SP_MAX) { abortStackOverflow(1040); }
...
STACKTOP = SP;
return ... ;
```

Prologue and epilogue idiomatic code can be found in most routines[19], even if they do not use SP (eg, a function using local registers exclusively will still allocate 16 bytes). The comparison check may not be performed on release code.

System calls

The subsystem environment emulates POSIX system calls. Since no explicit operator is provided to perform syscalls, programs can call imported wrapper routines named `__syscallN` to invoke them directly.

Conclusion

WebAssembly is expected to gain traction over the coming years. Its support in all four major Internet browsers, as well as stable prototype and impressive demos, are elements indicating that the MVP of WebAssembly 1.0 is a success. It is easy to envision this technology misused by malicious programs wanting to:

1) Get more performance out the browser.[20]
2) Cloak and undermine reverse engineering process by taking advantage of a binary format.

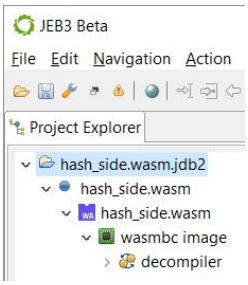
---

[18] My thoughts go to reversers of x86, arm, and most other machine code architectures with downward-growing stacks.
[19] It seems that non-leaf methods always allocate a small 16-byte stub on the stack. This may be a failsafe mechanism to prevent infinite recursion - and trigger stack overflow errors when one occurs.
[20] A malicious cryptocurrency compiled as a wasm binary module has been found in the wild already.

For those reasons, reverse engineers will need to have a deep understanding of the binary format, virtual machine and instruction set, execution environment and implementation details, as well as have powerful tools at their disposal to quickly navigate and analyze a closed-source piece of code.

Annex: Reversing Wasm Files with JEB

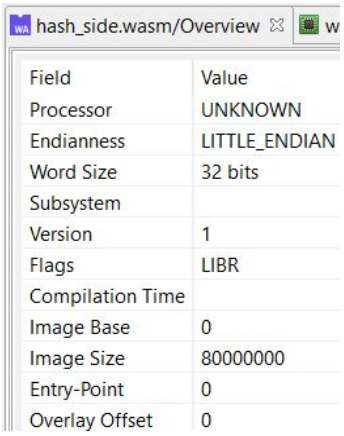
JEB ships with three modules to handle WebAssembly: a wasm binary parser, a disassembler extension, and a decompiler extension.



After opening a wasm binary in JEB, it will be recognized and processed as such by the WebAssembly plugin:

- The top-level unit node, under the blue-dot artifact node, represents the module itself; it has the official WebAssembly icon.
- Its first child node is named “wasmbc image”, and represents a memory view of the entire WebAssembly module, with code and data.

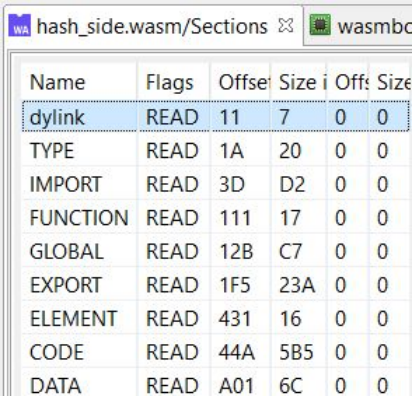
The module unit



Field	Value
Processor	UNKNOWN
Endianness	LITTLE_ENDIAN
Word Size	32 bits
Subsystem	
Version	1
Flags	LIBR
Compilation Time	
Image Base	0
Image Size	80000000
Entry-Point	0
Overlay Offset	0

The *Overview* fragment displays standard information. Note the following interesting bits:

- The word-size is set to 32-bit, as the plugin assumes a wasm32 environment.
- The endianness is little endian.
- The start of image is set to 0, and the virtual image size to 2Gb: the entire module will be sparsely mapped within this memory range.
- Wasm binaries do not embed standard metadata that would hold compilation timestamps (they could be added to a custom section though).



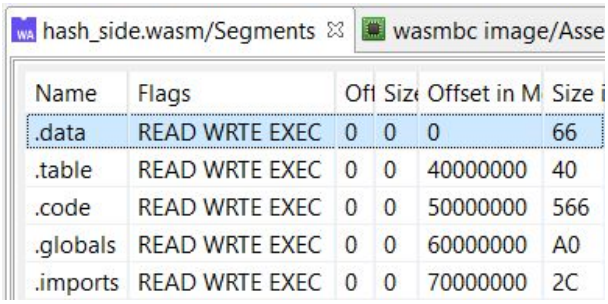
Name	Flags	Offse	Size i	Offs	Size
dylink	READ	11	7	0	0
TYPE	READ	1A	20	0	0
IMPORT	READ	3D	D2	0	0
FUNCTION	READ	111	17	0	0
GLOBAL	READ	12B	C7	0	0
EXPORT	READ	1F5	23A	0	0
ELEMENT	READ	431	16	0	0
CODE	READ	44A	5B5	0	0
DATA	READ	A01	6C	0	0

The *Sections* fragment represents an unmodified list of wasm sections. The in-memory size and offset are irrelevant and set to zero. Note that the example on the left side shows a custom *dylink* section, most likely[21] indicating a binary compiled with the emcc flag SIDE_MODULE=1.

The *Segments* and *Symbols* tabs detail how the wasm plugin laid out and transformed the elements of the wasm module in order to allow the underlying code plugin to process it.

[21] Custom sections are non-critical pieces of metadata, and as such, should always be taken with a grain of salt. The information they contain may be forged and does not impact program execution.

Pseudo segments



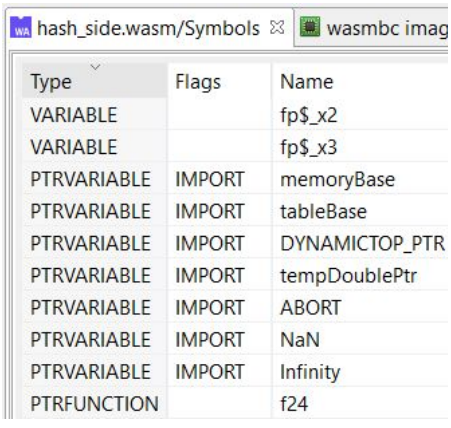
Name	Flags	Off	Size	Offset in M	Size i
.data	READ WRTE EXEC	0	0	0	66
.table	READ WRTE EXEC	0	0	40000000	40
.code	READ WRTE EXEC	0	0	50000000	566
.globals	READ WRTE EXEC	0	0	60000000	A0
.imports	READ WRTE EXEC	0	0	70000000	2C

The concept of mappable segments does not exist per-say in WebAssembly.[22] However, in order to allow JEB code plugins (disassemblers, decompilers, and above) to perform their work, the wasm plugin creates the following pseudo sections:

- `.data` starts at address 0 and maps the single Memory section (along with the Data elements that initialize it)
- `.table` starts at address 0x4000_000 and holds a table of pointers to the functions referenced in the Table section (initialized by the Element section)
- `.code` starts at address 0x5000_0000 and contains the bytecode of all internal functions, in order of index: the first function body of size S0 is at address 0x5000_0000, the second at address 0x5000_0000+S0, the third at 0x5000_0000+S0+S1, etc.
- `.globals` starts at address 0x6000_0000 and maps the internal globals as standard global variables; practically, wasm' globals (accessible by {SET,GET}_GLOBAL operators) and wasm' memory bytes (accessible by load/store operators) are being treated as equals by JEB.
- `.imports` starts at address 0x7000_0000 contains pointer references to the imported (external) function section entries and global section entries

The image is set to be mapped in the [0, 0x8000_0000) range. Sections are sparsely mapped; only used bytes are allocated. The start addresses are flexible and can be adjusted if needed.

Symbols



Type	Flags	Name
VARIABLE		fp$_x2
VARIABLE		fp$_x3
PTRVARIABLE	IMPORT	memoryBase
PTRVARIABLE	IMPORT	tableBase
PTRVARIABLE	IMPORT	DYNAMICTOP_PTR
PTRVARIABLE	IMPORT	tempDoublePtr
PTRVARIABLE	IMPORT	ABORT
PTRVARIABLE	IMPORT	NaN
PTRVARIABLE	IMPORT	Infinity
PTRFUNCTION		f24

Symbols generated by the wasm plugin are of four types, and can have a variety of attributes:

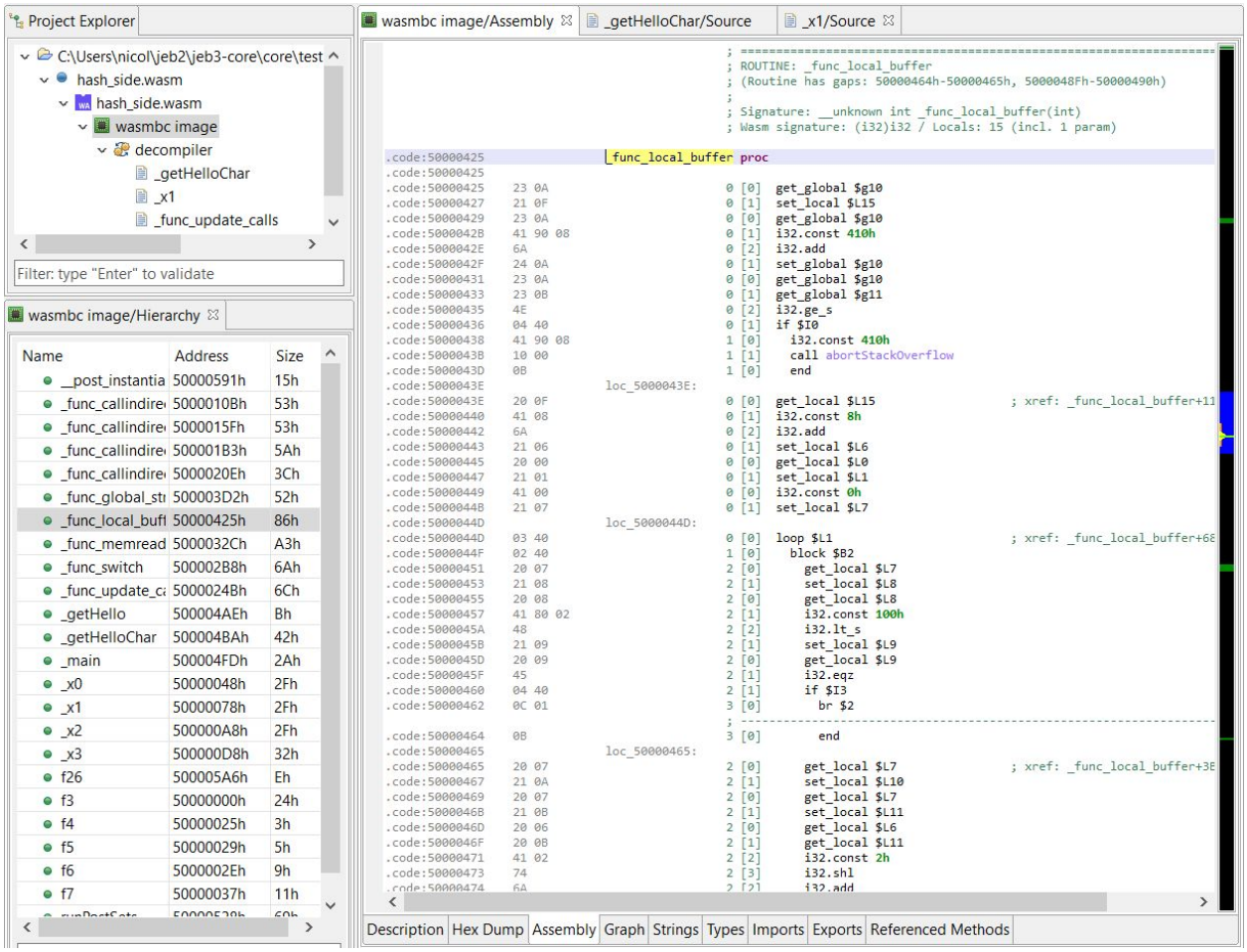
- FUNCTION, for internal functions
- PTRUNCTION, for imported functions and referenced internal functions
- VARIABLE for globals
- PTRVARIABLE for imported globals

[22] As is the case with all bytecode binary formats.

Code View

The interactive disassembly window shows the pseudo virtual memory representing the entire WebAssembly module, as explained in the above section.

The screenshot below shows the disassembly area of an internal function. Note that JEB's representation of a function's bytecode is linear and matches the underlying binary code. The green columns prefixing the instruction indicate the current block depth and current operand stack height, pre-execution.



The snapshot below shows the beginning of the `.data` virtual segment, representing the WebAssembly `Data` section.

The Code view is interactive. However, code modification (ie, tampering with routine bodies) is forbidden since they would introduce inconsistencies in the function index space.

```
; Code Disassembly

; ..........................................................................................
;
; Segment:".data" Size:66h Permissions:READ,WRITE,EXECUTE
;
; ..........................................................................................

.data:00000000    00 00 00 00  gvar_0          dd 0h
.data:00000004    00 00 00 00  gvar_4          dd 0h
.data:00000008    00 00 00 00+                 db 0, 0, 0, 0, 0, 0, 0, 0
.data:00000010    00 00 00 00+                 db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:00000020    11 11 00 00+                 db 11h, 11h, 0, 0, """", 0, 0
.data:00000028    54 68 69 73+ aThis_is_a_loooo db "This is a looooooooooooooooog non-static string!"
.data:00000058    48 65 6C 6C+ aHello__world__ db "Hello, world!\n"
SLACK:00000067    00 00 00 00+                 db 0, 0, 0, 0, 0, 0, 0, 0, 0
SLACK:00000070    00 00 00 00+                 db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Decompilation

The WebAssembly decompiler plugin for JEB uses JEB's decompilation pipeline to produce pseudo C code. As such, the plugin consists of a wasm-to-IR converter and additional analyzer extensions. Slots on the operand stack are converted to standard, routine context IR variables.[23] As of the time of writing, the wasm decompiler has the following limitations:

- The global, advanced analysis normally provided the decompiler is partially disabled (the advanced analysis is responsible, for instance, to discover register values and callsite targets during a fast static analysis phase, and subsequently annotate the assembly listing).
- There is no support for floating point operation conversion.
- Memory stack frames, due their dynamic (and optional) nature in WebAssembly are currently not accessible and customizable.

Those limitations will be addressed as the decompiler plugin matures.

```
1 [0]    block $B2
2 [0]      get_local $L0
2 [1]      set_local $L22
2 [0]      get_local $L22
2 [1]      i32.const 0h
2 [2]      i32.ne
2 [1]      set_local $L23
2 [0]      get_local $L23
2 [1]      i32.eqz
2 [1]      if $I3
3 [0]        br $2
;  ------------------------------------------------------
3 [0]        end

2 [0]      get_local $L12              ; xref: _crc32_i
2 [1]      set_local $L24
2 [0]      get_local $L24
2 [1]      i32.const 1h
2 [2]      i32.shr_u
2 [1]      set_local $L25
2 [0]      get_local $L12
2 [1]      set_local $L26
2 [0]      get_local $L26
2 [1]      i32.const 1h
```

```
void _crc32_init() {
    unsigned int r30 = g10;

    if(g10 + 16 >= g11) {
        abortStackOverflow(16);
    }

    unsigned int r12 = 1;
    *(memoryBase + 0x500090) = 0;
    unsigned int r0;

    for(r0 = 128; r0 != 0; r0 >>>= 1) {
        unsigned int r25 = r12 >>> 1;
        unsigned int var32 = (r12 & 1) != 0 ? 0xEDB88320: 0;
        r12 = r25 ^ var32;
        unsigned int r1;

        for(r1 = 0; r1 < 256; r1 += r0 * 2) {
            *(memoryBase + 0x500090 + (r0 + r1) * 4) = *(memoryBase + 0x500090 + r1 * 4) ^ r12;
        }
    }

    g10 = r30;
}
```

[23] Design and implementation choices of the wasm decompiler may be the object of a separate article.