Flare-On 5: Challenge 5 Solution – web2point0

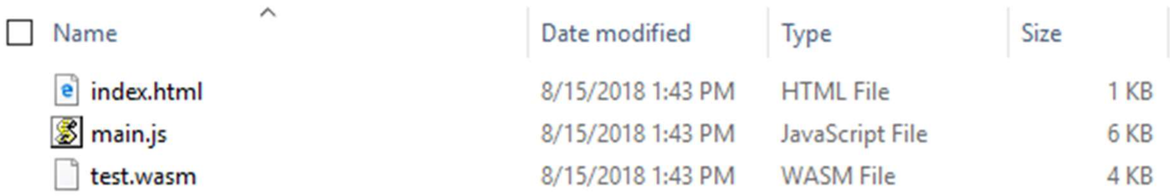
Challenge Author: William Ballenthin (@williballenthin)

Challenge five ("web2point0") tests our ability to reverse engineer WebAssembly modules that execute in a web browser. The primary hurdles are: finding appropriate analysis tools, learning the WebAssembly architecture, and reasoning about the logic of the program. Fortunately, WebAssembly is developed in the open and documentation is easy to find; however, not all the tools we'd want exist yet. Hopefully this challenge highlights pain points and leads to better tooling for reverse engineers.

TABLE OF CONTENTS

Initial triage

To begin, we open the challenge archive and find that it contains three files: a HTML document, a JavaScript resource, and a WebAssembly module. Figure 1 shows the contents in Windows Explorer.



Name	Date modified	Type	Size
index.html	8/15/2018 1:43 PM	HTML File	1 KB
main.js	8/15/2018 1:43 PM	JavaScript File	6 KB
test.wasm	8/15/2018 1:43 PM	WASM File	4 KB

Figure 1- Contents of web2point0.zip

We assume that the entry point into the challenge is the HTML document, so we load it a web browser. Figure 2 shows `index.html` rendered using the Firefox browser. It displays a single Emoji character: 💩.

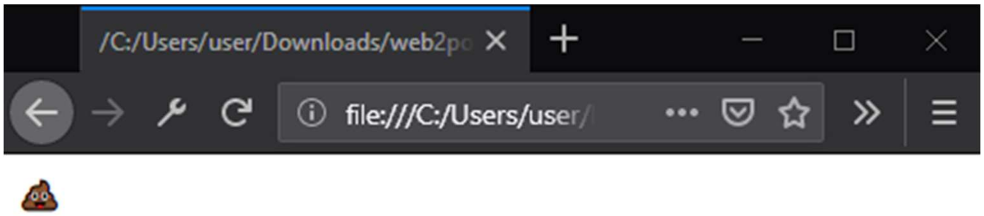


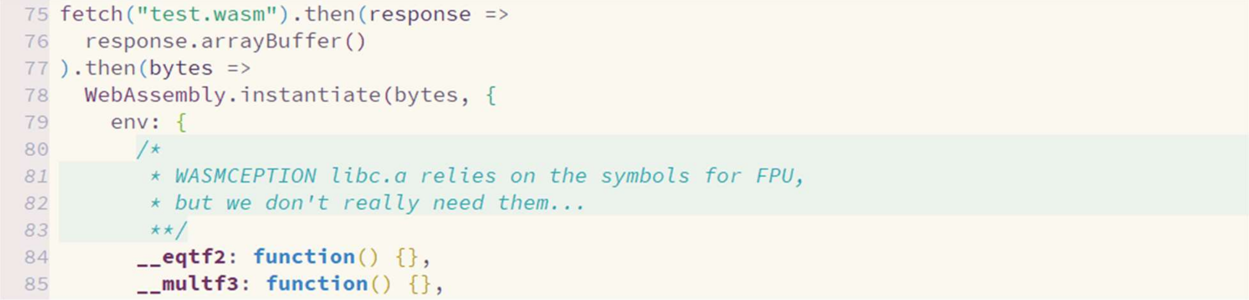
Figure 2 - index.html rendered in the Firefox web browser

Since there are no obvious inputs or controls, we review the contents of the HTML source code to see how the page operates. Figure 3 lists the contents of the HTML document. The HTML document is very simple: it loads the JavaScript resource named `main.js`. We'll have to focus our attention there.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <style>
  </style>
</head>
<body>
  <span id="container"></span>
  <script src="./main.js"></script>
</body>
</html>
```

Figure 3 - Contents of index.html

Figure 4 lists the entry point of the JavaScript code in `main.js`. On line 75, the program asynchronously loads the contents of the file `test.wasm` and creates a WebAssembly module from the binary data. Once the module is created, lines 119 through 143 interact with the WebAssembly module to check a key provided as the HTTP query string parameter named q. If the client provides the correct key, the webpage will render 🎉, otherwise, 💩. Figure 5 shows Firefox rendering `index.html` once the correct key has been provided. This is our goal!



```
fetch("test.wasm").then(response =>
  response.arrayBuffer()
).then(bytes =>
  WebAssembly.instantiate(bytes, {
    env: {
      /*
       * WASMCEPTION libc.a relies on the symbols for FPU,
       * but we don't really need them...
       **/
      __eqtf2: function() {},
      __multf3: function() {},
```

```
).then(results => {
    instance = results.instance;

    let a = new Uint8Array([
        0xE4, 0x47, 0x30, 0x10, 0x61, 0x24, 0x52, 0x21, 0x86, 0x40, 0xAD, 0xC1, 0xA0, 0xB4,
75, 0x32, 0x48, 0x24, 0x86, 0xE3, 0x48, 0xA1, 0x85, 0x36, 0x6D, 0xCC, 0x33, 0x7B, 0x6E, 0x9
0xA0, 0xF6, 0x86, 0xEA, 0x55, 0x48, 0x2A, 0xB3, 0xFF, 0x6F, 0x91, 0x90, 0xA1, 0x93, 0x70, 0x
, 0x66, 0x64, 0xCA, 0x94, 0x20, 0x4C, 0x10, 0x61, 0x53, 0x77, 0x72, 0x42, 0xE9, 0x8C, 0x30,
6F, 0xB1, 0x91, 0x65, 0x24, 0x0A, 0x14, 0x21, 0x42, 0xA3, 0xEF, 0x6F, 0x55, 0x97, 0xD6

        //0xB6, 0xFF, 0x65, 0xC3, 0xED, 0x7E, 0xA4, 0x00,
        //                      0x61, 0xD3, 0xFF, 0x72, 0x36, 0x02, 0x67, 0x91,
        //0xD2, 0xD5, 0xC8, 0xA7, 0xE0, 0x6E
    ]);

    let b = new Uint8Array(new TextEncoder().encode(getParameterByName("q")));

    let pa = wasm_alloc(instance, 0x200);
    wasm_write(instance, pa, a);

    let pb = wasm_alloc(instance, 0x200);
    wasm_write(instance, pb, b);

    if (instance.exports.Match(pa, a.byteLength, pb, b.byteLength) == 1) {
        // PARTY POPPER
        document.getElementById("container").innerText = "🎉";
    } else {
        // PILE OF POO
        document.getElementById("container").innerText = "💩";
    }
});
```

Figure 4 - JavaScript entry point

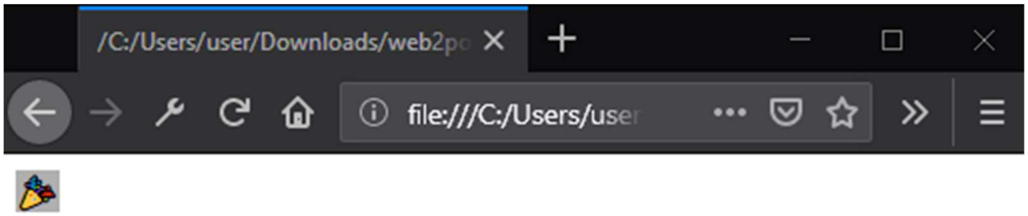


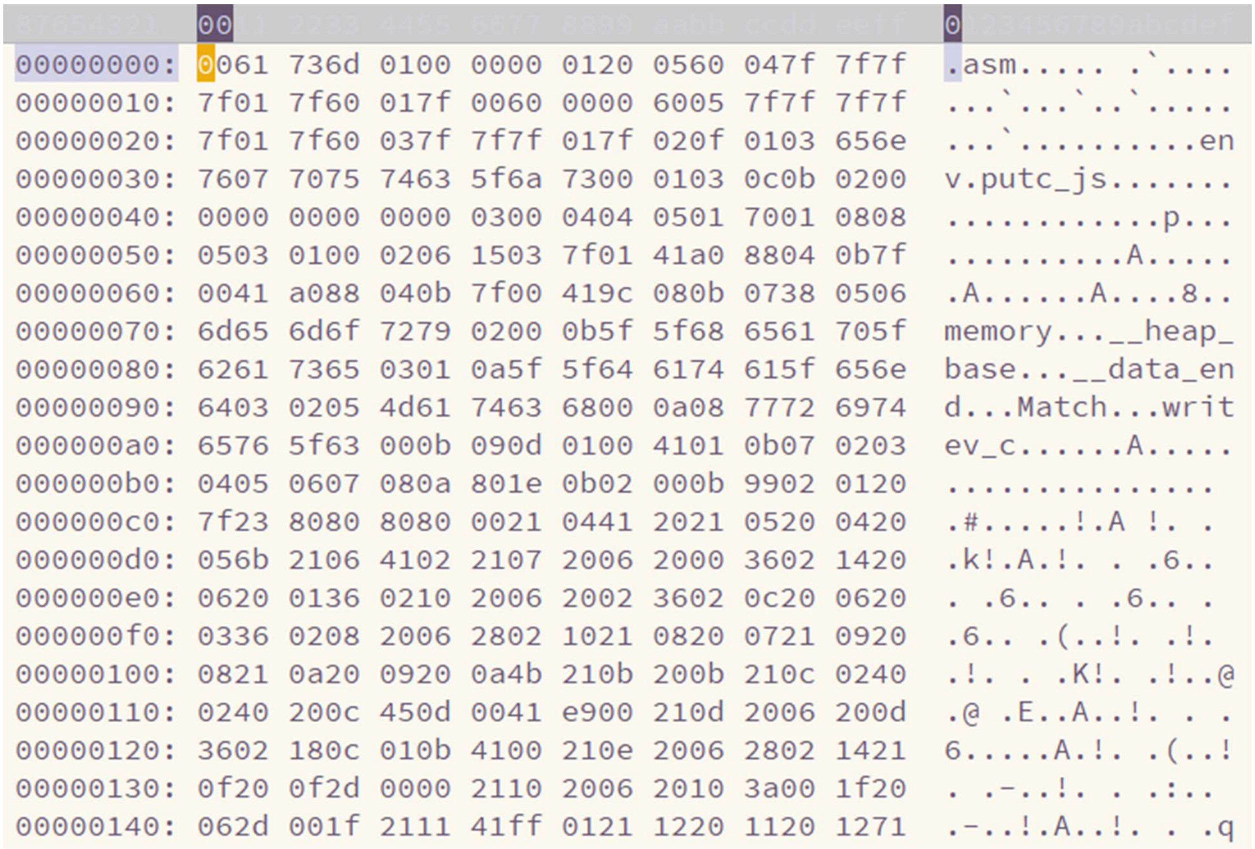
Figure 5 - Rendered index.html with the correct key

To figure out how to decode the byte array initialized on line 122, we'll have to extract and analyze the logic in the WebAssembly module `test.wasm`. But first, what is WebAssembly, and how do we analyze a `.wasm` file?

> *WebAssembly is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.*
>
> *The Wasm stack machine is designed to be encoded in a size- and load-time-efficient binary format. WebAssembly aims to execute at native speed by taking advantage of common hardware capabilities available on a wide range of platforms*
>
> *via: https://webassembly.org/*

So, WebAssembly is both a file format and architecture that augments the JavaScript runtime available on modern web browsers. Its authors designed the format to easily map onto common CPU architectures, such as x86, during JIT compilation. Therefore, we should expect to recognize the semantics of many of the common WebAssembly instruction, e.g. `i32.sub`, `i32.gt_u`, and `i32.load8_u`. The best resources for learning WebAssembly representations are available on project’s website here: https://webassembly.org/docs/binary-encoding/.

We can use basic static analysis techniques to triage the WebAssembly file. Figure 6 shows the file’s header in a hex editor, while Figure 7 enumerates the human-readable ASCII strings (there are no UTF-16 strings). While there is some noticeable structure in the first 0xA0 bytes (magic header `|00 61 73 6D|` (“`asm`”) at offset 0x0 and function names in UTF-8), most of the file contains binary data that is difficult to parse by hand.



```
87654321  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789abcdef
00000000: 0061 736d 0100 0000 0120 0560 047f 7f7f  .asm..... .`....
00000010: 7f01 7f60 017f 0060 0000 6005 7f7f 7f7f  ...`...`..`.....
00000020: 7f01 7f60 037f 7f7f 017f 020f 0103 656e  ...`..........en
00000030: 7607 7075 7463 5f6a 7300 0103 0c0b 0200  v.putc_js.......
00000040: 0000 0000 0000 0300 0404 0501 7001 0808  ............p...
00000050: 0503 0100 0206 1503 7f01 41a0 8804 0b7f  ..........A.....
00000060: 0041 a088 040b 7f00 419c 080b 0738 0506  .A......A....8..
00000070: 6d65 6d6f 7279 0200 0b5f 5f68 6561 705f  memory...__heap_
00000080: 6261 7365 0301 0a5f 5f64 6174 615f 656e  base...__data_en
00000090: 6403 0205 4d61 7463 6800 0a08 7772 6974  d...Match...writ
000000a0: 6576 5f63 000b 090d 0100 4101 0b07 0203  ev_c......A.....
000000b0: 0405 0607 080a 801e 0b02 000b 9902 0120  ...............
000000c0: 7f23 8080 8080 0021 0441 2021 0520 0420  .#.....!.A !. .
000000d0: 056b 2106 4102 2107 2006 2000 3602 1420  .k!.A.!. . .6..
000000e0: 0620 0136 0210 2006 2002 3602 0c20 0620  . .6.. . .6.. .
000000f0: 0336 0208 2006 2802 1021 0820 0721 0920  .6.. .(..!. .!.
00000100: 0821 0a20 0920 0a4b 210b 200b 210c 0240  .!. . .K!. .!..@
00000110: 0240 200c 450d 0041 e900 210d 2006 200d  .@ .E..A..!. . .
00000120: 3602 180c 010b 4100 210e 2006 2802 1421  6.....A.!. .(..!
00000130: 0f20 0f2d 0000 2110 2006 2010 3a00 1f20  . .-..!. . .:..
00000140: 062d 001f 2111 41ff 0121 1220 1120 1271  .-..!.A..!. . .q
```

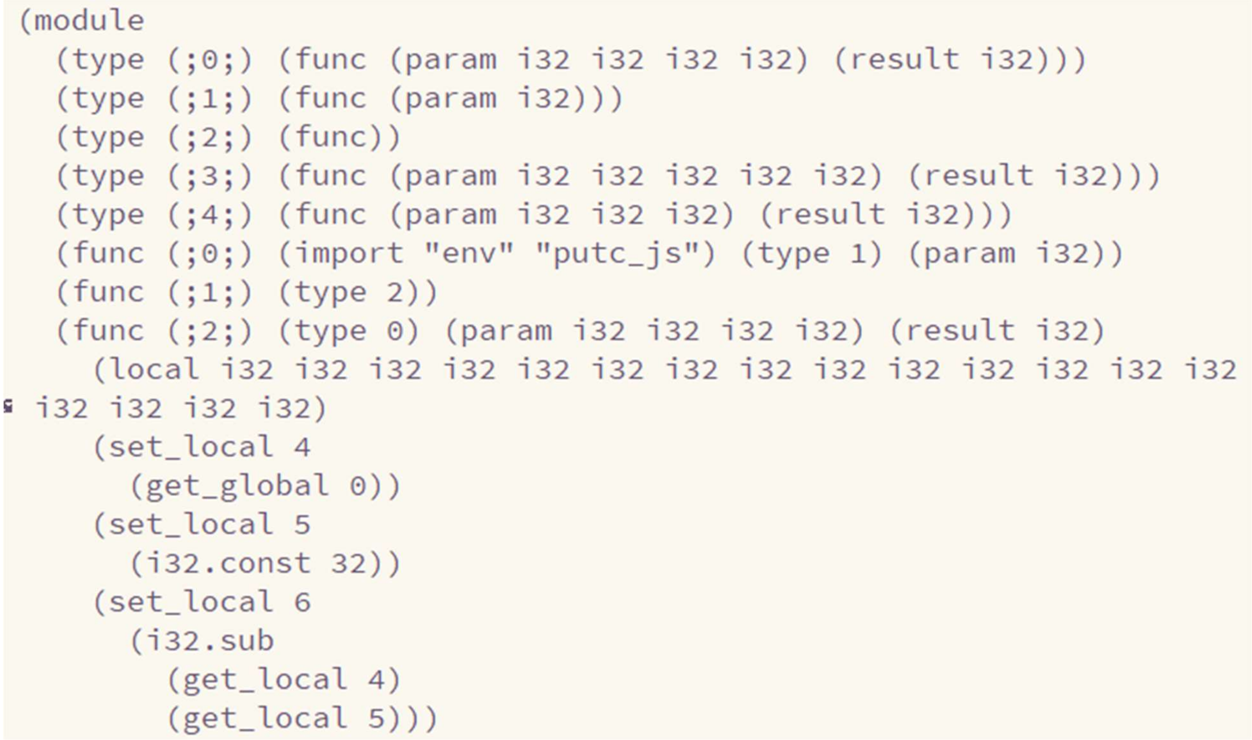
Figure 6 - Hex dump of test.wasm

```
user@hostname:/mnt/c/Users/user/Downloads/web2point0$ strings test.wasm
putc_js
memory
__heap_base
__data_end
Match
writev_c
!! !  :
!" "
!# #
!!   !q!"A
!# " #s!$
!% % $:
!& &
!' '
!!   !q!"
!# #-
!% $ %q!& " &s!'
!( ( ':
```

Figure 7 - ASCII strings extracted from test.wasm

WABT

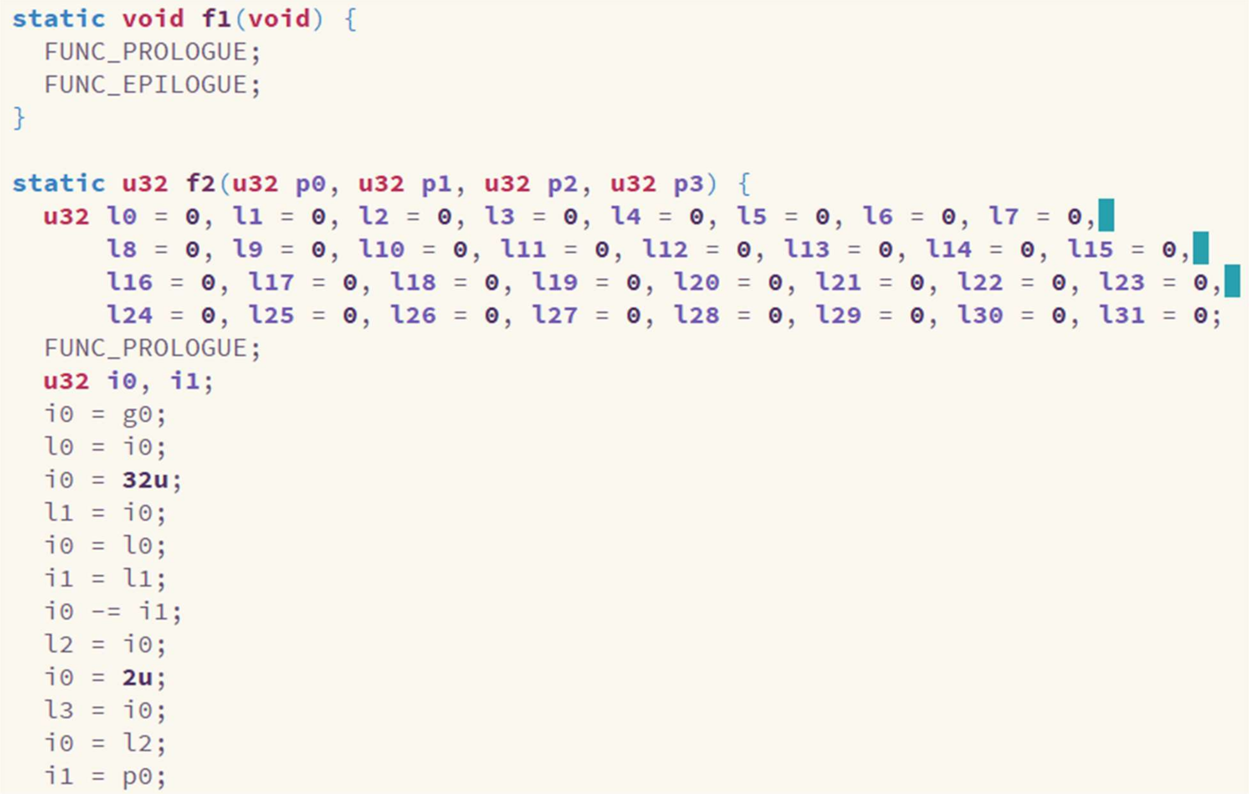
Fortunately, there exist utilities to inspect the binary file format. The WebAssembly Binary Toolkit (https://github.com/WebAssembly/wabt) provides the command line tool `wasm2wat` that translates a `.wasm` file into the human-readable `.wat` format, as well as a basic decompiler called `wasm2c`. For example, Figure 8 lists a portion of the `.wat` file produced from `test.wasm`. The `.wat` format can be much nicer to review than the raw disassembly, because it collapses expressions that manipulate the stack and uses whitespace to delimit blocks.



```
(module
  (type (;0;) (func (param i32 i32 i32 i32) (result i32)))
  (type (;1;) (func (param i32)))
  (type (;2;) (func))
  (type (;3;) (func (param i32 i32 i32 i32 i32) (result i32)))
  (type (;4;) (func (param i32 i32 i32) (result i32)))
  (func (;0;) (import "env" "putc_js") (type 1) (param i32))
  (func (;1;) (type 2))
  (func (;2;) (type 0) (param i32 i32 i32 i32) (result i32)
    (local i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32
 i32 i32 i32 i32)
    (set_local 4
      (get_global 0))
    (set_local 5
      (i32.const 32))
    (set_local 6
      (i32.sub
        (get_local 4)
        (get_local 5)))
```

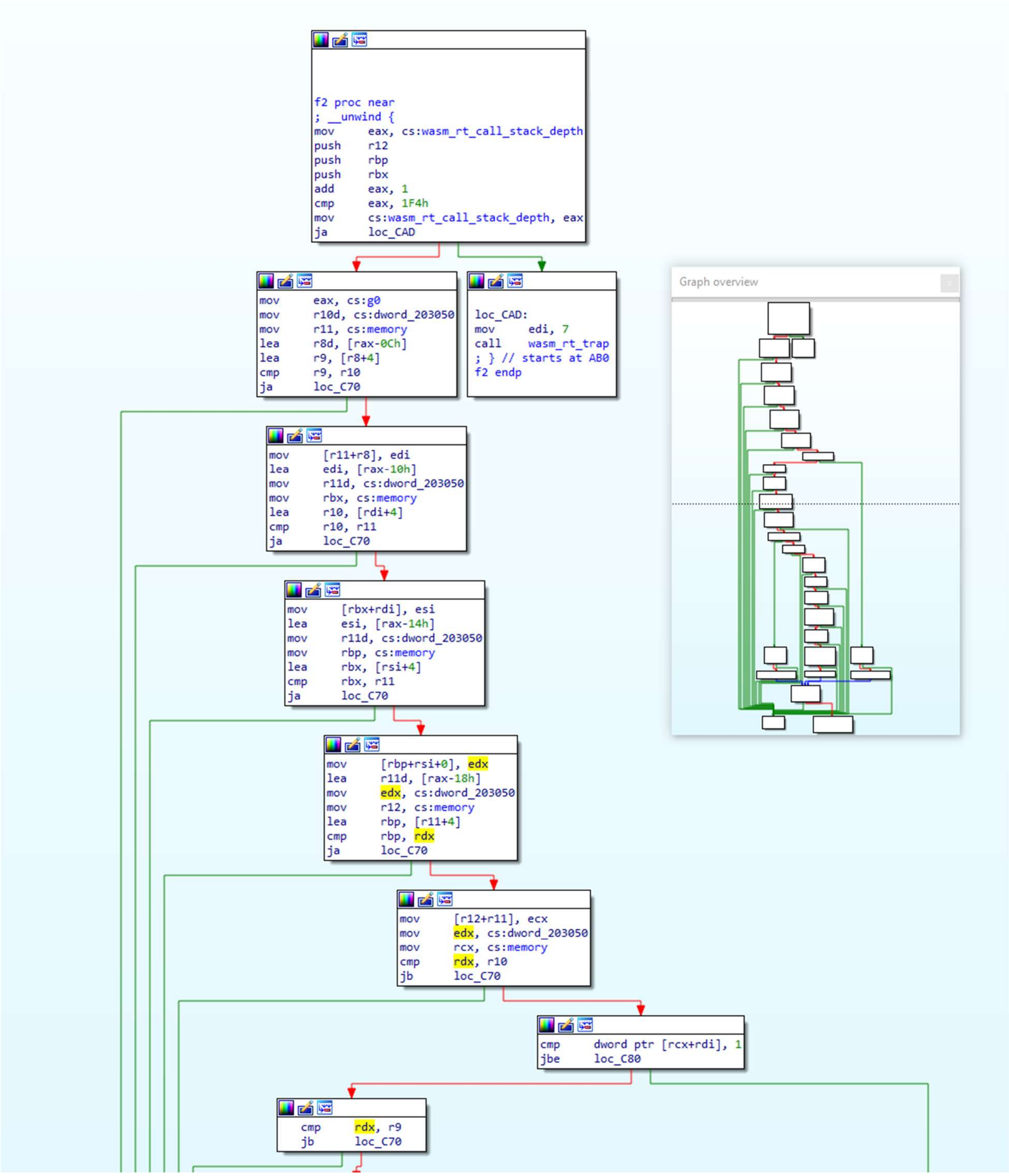
Figure 8 - Human-readable .wat representation of test.wasm

We can use the `wasm2c` utility to decompile the module into C, which results in the listing shown in Figure 9. While the format may be more familiar to us, it is clear that the source code has been mechanically generated with few optimizations applied. Though not very useful in its raw form, we could potentially compile the C source code into a native binary and analyze the logic using existing tools like IDA Pro. Figure 10 shows a portion of `$func2` from a binary compiled with `gcc -O3` on the decompilation output.



```
static void f1(void) {
  FUNC_PROLOGUE;
  FUNC_EPILOGUE;
}

static u32 f2(u32 p0, u32 p1, u32 p2, u32 p3) {
  u32 l0 = 0, l1 = 0, l2 = 0, l3 = 0, l4 = 0, l5 = 0, l6 = 0, l7 = 0,
      l8 = 0, l9 = 0, l10 = 0, l11 = 0, l12 = 0, l13 = 0, l14 = 0, l15 = 0,
      l16 = 0, l17 = 0, l18 = 0, l19 = 0, l20 = 0, l21 = 0, l22 = 0, l23 = 0,
      l24 = 0, l25 = 0, l26 = 0, l27 = 0, l28 = 0, l29 = 0, l30 = 0, l31 = 0;
  FUNC_PROLOGUE;
  u32 i0, i1;
  i0 = g0;
  l0 = i0;
  i0 = 32u;
  l1 = i0;
  i0 = l0;
  i1 = l1;
  i0 -= i1;
  l2 = i0;
  i0 = 2u;
  l3 = i0;
  i0 = l2;
  i1 = p0;
```

Figure 9 - Output of test.wasm decompiled with wasm2c



```
f2 proc near
; __unwind {
mov     eax, cs:wasm_rt_call_stack_depth
push    r12
push    rbp
push    rbx
add     eax, 1
cmp     eax, 1F4h
mov     cs:wasm_rt_call_stack_depth, eax
ja      loc_CAD
```

```
mov     eax, cs:g0
mov     r10d, cs:dword_203050
mov     r11, cs:memory
lea     r8d, [rax-0Ch]
lea     r9, [r8+4]
cmp     r9, r10
ja      loc_C70
```

```
loc_CAD:
mov     edi, 7
call    wasm_rt_trap
; } // starts at AB0
f2 endp
```

```
mov     [r11+r8], edi
lea     edi, [rax-10h]
mov     r11d, cs:dword_203050
mov     rbx, cs:memory
lea     r10, [rdi+4]
cmp     r10, r11
ja      loc_C70
```

```
mov     [rbx+rdi], esi
lea     esi, [rax-14h]
mov     r11d, cs:dword_203050
mov     rbp, cs:memory
lea     rbx, [rsi+4]
cmp     rbx, r11
ja      loc_C70
```

```
mov     [rbp+rsi+0], edx
lea     r11d, [rax-18h]
mov     edx, cs:dword_203050
mov     r12, cs:memory
lea     rbp, [r11+4]
cmp     rbp, rdx
ja      loc_C70
```

```
mov     [r12+r11], ecx
mov     edx, cs:dword_203050
mov     rcx, cs:memory
cmp     rdx, r10
jb      loc_C70
```

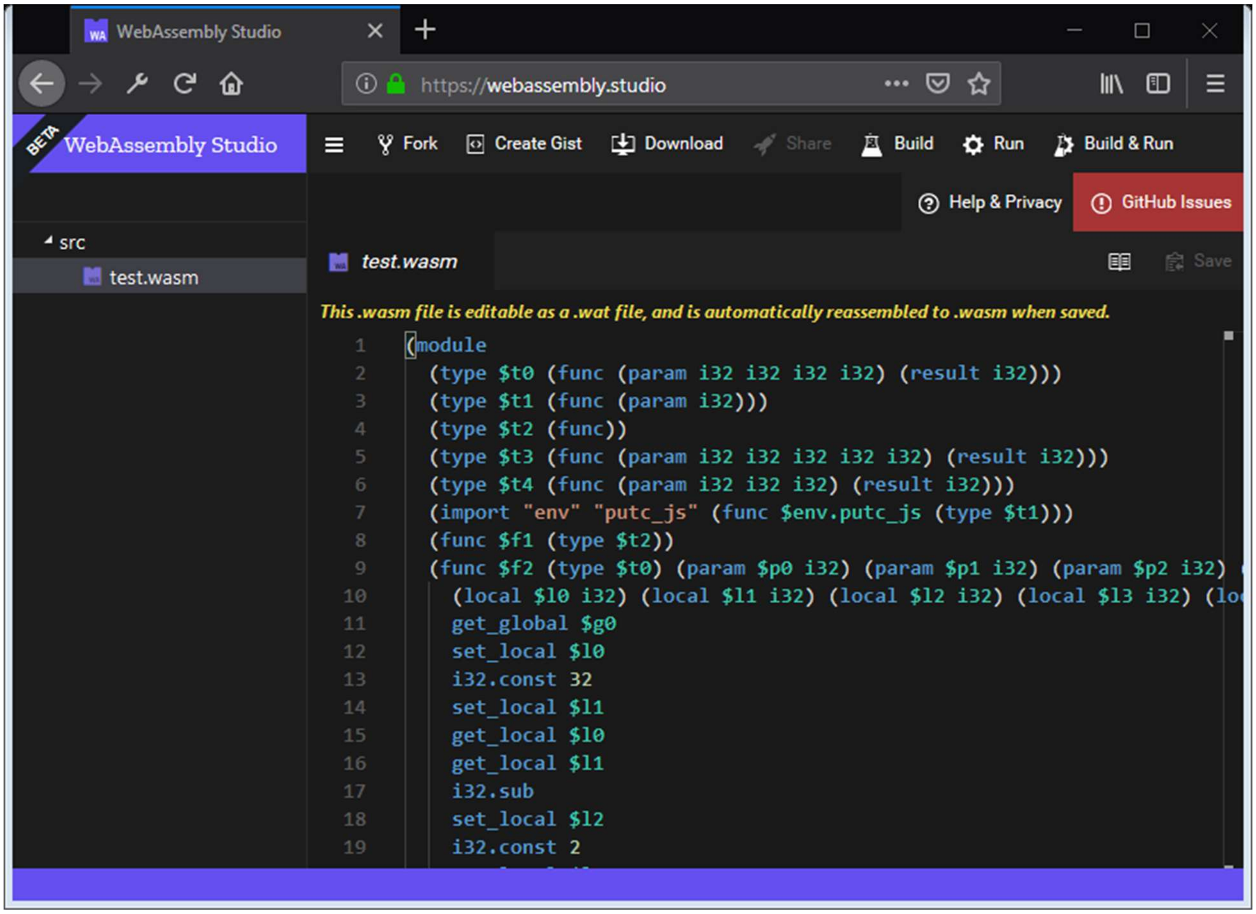
```
cmp     dword ptr [rcx+rdi], 1
jbe     loc_C80
```

```
cmp     rdx, r9
jb      loc_C70
```

Figure 10 - Disassembly of program compiled from the decompilation of test.wasm

WebAssembly Studio

Alternatively, we could use the web-based WebAssembly Studio IDE available at https://webassembly.studio to inspect `test.wasm`. While this tool was primarily developed for writing high-level code that compiles into WebAssembly modules, it also exposes features for extracting `.wasm` files into interesting formats. For example, Figure 11 shows how the WebAssembly Studio has extracted `test.wasm` into the human-readable `.wat` format with syntax highlighting.



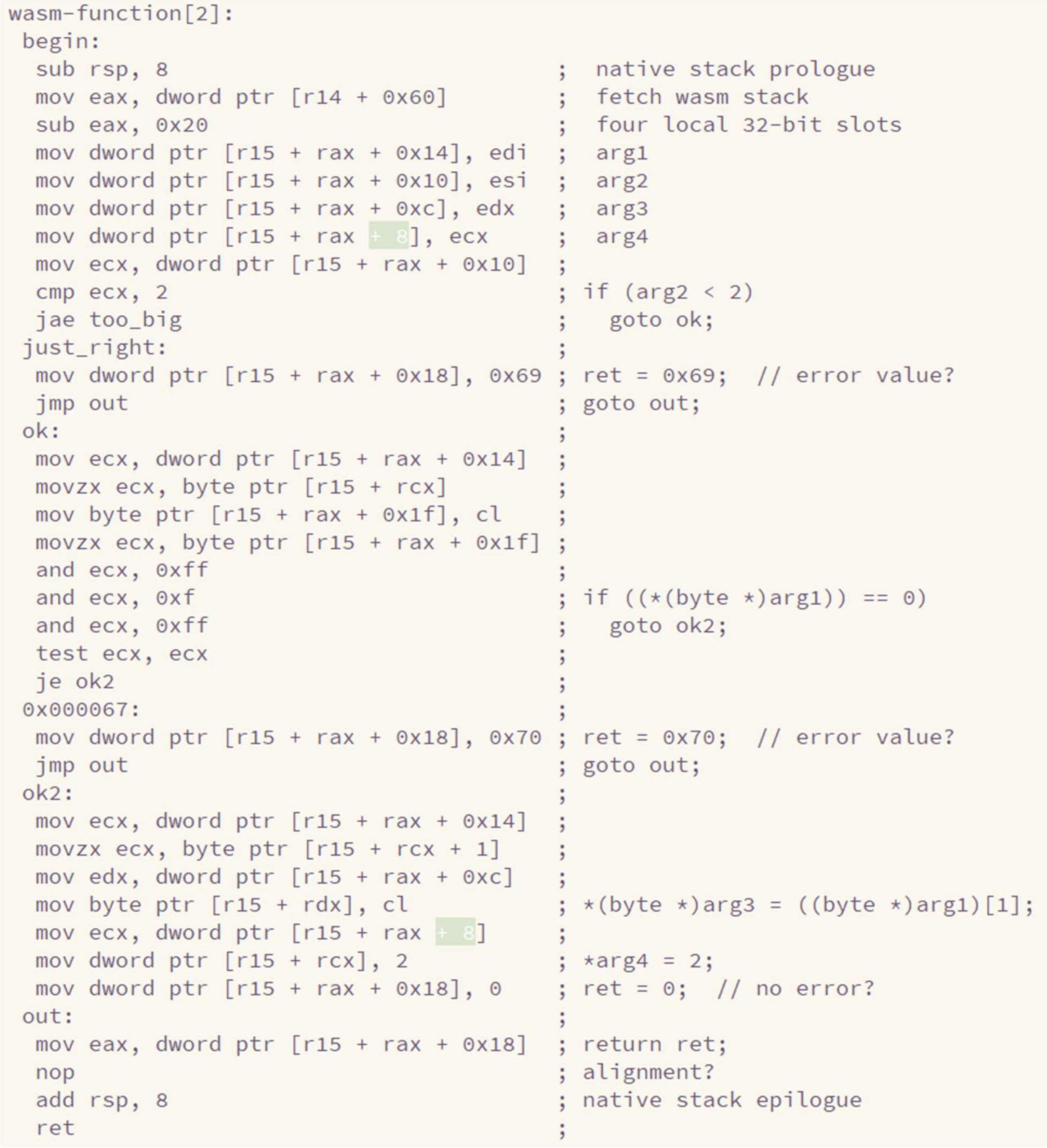
This .wasm file is editable as a .wat file, and is automatically reassembled to .wasm when saved.

```
(module
  (type $t0 (func (param i32 i32 i32 i32) (result i32)))
  (type $t1 (func (param i32)))
  (type $t2 (func))
  (type $t3 (func (param i32 i32 i32 i32 i32) (result i32)))
  (type $t4 (func (param i32 i32 i32) (result i32)))
  (import "env" "putc_js" (func $env.putc_js (type $t1)))
  (func $f1 (type $t2))
  (func $f2 (type $t0) (param $p0 i32) (param $p1 i32) (param $p2 i32)
    (local $l0 i32) (local $l1 i32) (local $l2 i32) (local $l3 i32) (lo
    get_global $g0
    set_local $l0
    i32.const 32
    set_local $l1
    get_local $l0
    get_local $l1
    i32.sub
    set_local $l2
    i32.const 2
```

Figure 11 - test.wasm viewed with webassembly.studio

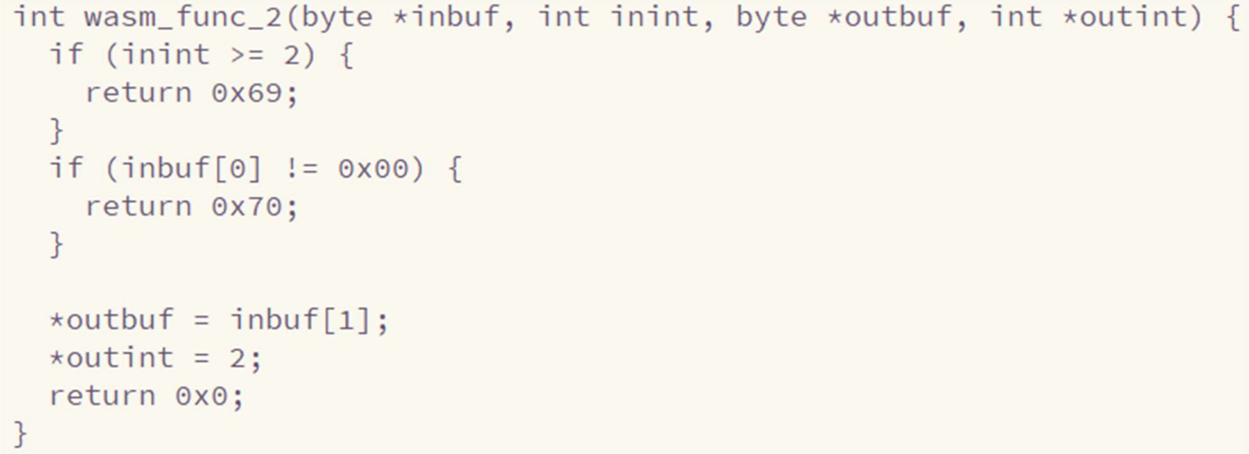
One neat feature of the WebAssembly Studio is that we can invoke Firefox's SpiderMonkey JIT compiler on a WebAssembly module and extract x86-64 instructions. This allows us to see how Firefox would execute `test.wasm` on a host CPU. For those of us most familiar with x86 assembly, this representation might be easier to digest.

For example, Figure 12 shows an annotated listing of the x86 instructions generated by SpiderMonkey for `$func2` in `test.wasm`. In the left column is the raw disassembly, while in the right column are (human-supplied) notes that indicate an interpretation of the code. If we were to translate this interpretation into C, we might end up with the function definition listed in Figure 13. This function of four parameters validates its inputs and then copies a byte of input into an output buffer. We’re not sure why, yet.



```
wasm-function[2]:
 begin:
  sub rsp, 8                              ;  native stack prologue
  mov eax, dword ptr [r14 + 0x60]         ;  fetch wasm stack
  sub eax, 0x20                           ;  four local 32-bit slots
  mov dword ptr [r15 + rax + 0x14], edi   ;  arg1
  mov dword ptr [r15 + rax + 0x10], esi   ;  arg2
  mov dword ptr [r15 + rax + 0xc], edx    ;  arg3
  mov dword ptr [r15 + rax + 8], ecx      ;  arg4
  mov ecx, dword ptr [r15 + rax + 0x10]   ;
  cmp ecx, 2                              ; if (arg2 < 2)
  jae too_big                             ;   goto ok;
 just_right:                              ;
  mov dword ptr [r15 + rax + 0x18], 0x69 ; ret = 0x69;  // error value?
  jmp out                                 ; goto out;
 ok:                                      ;
  mov ecx, dword ptr [r15 + rax + 0x14]   ;
  movzx ecx, byte ptr [r15 + rcx]         ;
  mov byte ptr [r15 + rax + 0x1f], cl     ;
  movzx ecx, byte ptr [r15 + rax + 0x1f] ;
  and ecx, 0xff                           ;
  and ecx, 0xf                            ; if ((*(byte *)arg1)) == 0)
  and ecx, 0xff                           ;   goto ok2;
  test ecx, ecx                           ;
  je ok2                                  ;
 0x000067:                                ;
  mov dword ptr [r15 + rax + 0x18], 0x70 ; ret = 0x70;  // error value?
  jmp out                                 ; goto out;
 ok2:                                     ;
  mov ecx, dword ptr [r15 + rax + 0x14]   ;
  movzx ecx, byte ptr [r15 + rcx + 1]     ;
  mov edx, dword ptr [r15 + rax + 0xc]    ;
  mov byte ptr [r15 + rdx], cl            ; *(byte *)arg3 = ((byte *)arg1)[1];
  mov ecx, dword ptr [r15 + rax + 8]      ;
  mov dword ptr [r15 + rcx], 2            ; *arg4 = 2;
  mov dword ptr [r15 + rax + 0x18], 0     ; ret = 0;  // no error?
 out:                                     ;
  mov eax, dword ptr [r15 + rax + 0x18]   ; return ret;
  nop                                     ; alignment?
  add rsp, 8                              ; native stack epilogue
  ret                                     ;
```

Figure 12 - JIT-compiled x86 from test.wasm



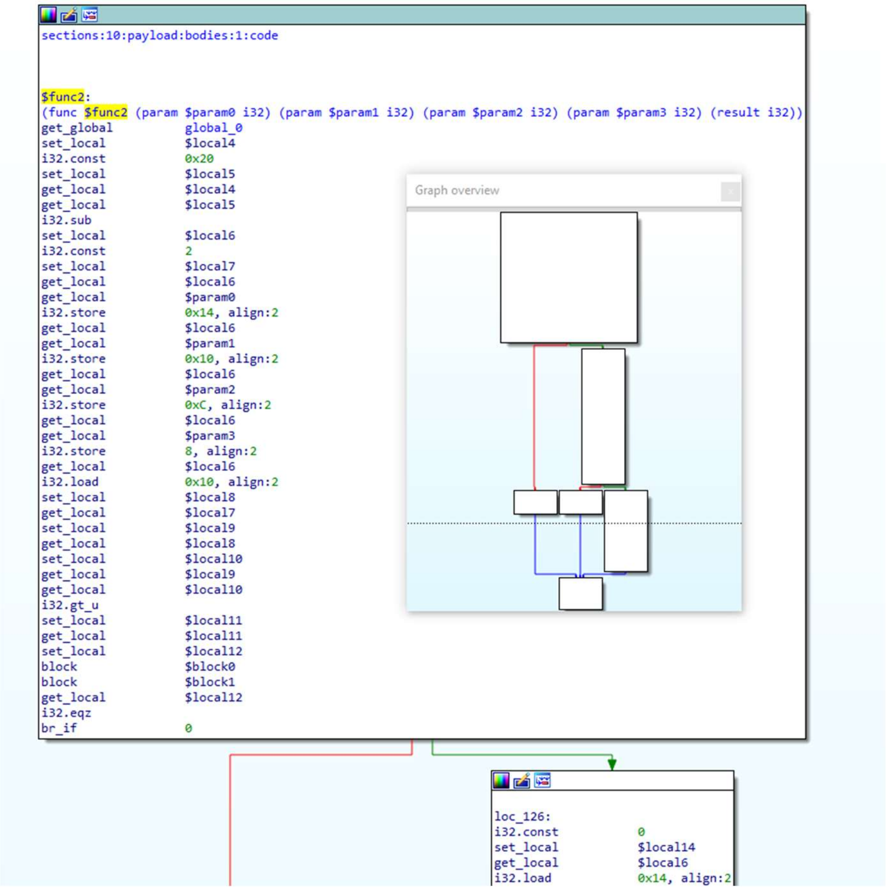
```
int wasm_func_2(byte *inbuf, int inint, byte *outbuf, int *outint) {
  if (inint >= 2) {
    return 0x69;
  }
  if (inbuf[0] != 0x00) {
    return 0x70;
  }

  *outbuf = inbuf[1];
  *outint = 2;
  return 0x0;
}
```

Figure 13 - Function source reconstructed from x86 instructions

IDA Pro plugin for WebAssembly

Finally, we could develop our own tools for inspecting the WebAssembly module. For example, while IDA Pro does not natively support the `.wasm` file format, we can extend the tool with Python. As the FLARE team drafted this solution, they developed an IDA Pro loader and processor plugin that enables support for WebAssembly. This lets us review the logic of `test.wasm` in a familiar graph mode without any forward- or backward-compilation shenanigans. Figure 14 displays a portion of `$func2` in IDA Pro with the idawasm plugin enabled. You can download and install the idawasm IDA Pro plugin from here: https://github.com/fireeye/idawasm.

This graph mode clearly indicates control flow structures such as `if` and `while` loops; however, the plugin is not yet able to collapse stack manipulations. Fortunately, we can add comments and rename functions, variables, and globals to remind us of functionality. For many of us, IDA Pro may be the most comfortable interface.



```
sections:10:payload:bodies:1:code


$func2:
(func $func2 (param $param0 i32) (param $param1 i32) (param $param2 i32) (param $param3 i32) (result i32))
get_global          global_0
set_local           $local4
i32.const           0x20
set_local           $local5
get_local           $local4
get_local           $local5
i32.sub
set_local           $local6
i32.const           2
set_local           $local7
get_local           $local6
get_local           $param0
i32.store           0x14, align:2
get_local           $local6
get_local           $param1
i32.store           0x10, align:2
get_local           $local6
get_local           $param2
i32.store           0xC, align:2
get_local           $local6
get_local           $param3
i32.store           8, align:2
get_local           $local6
i32.load            0x10, align:2
set_local           $local8
get_local           $local7
set_local           $local9
get_local           $local8
set_local           $local10
get_local           $local9
get_local           $local10
i32.gt_u
set_local           $local11
get_local           $local11
set_local           $local12
block               $block0
block               $block1
get_local           $local12
i32.eqz
br_if               0
```

```
loc_126:
i32.const           0
set_local           $local14
get_local           $local6
i32.load            0x14, align:2
```

Figure 14 - test.wasm disassembled with IDA Pro and idawasm

Now that we have the means to inspect the WebAssembly module, it's time to figure out what the logic does.

test.wasm logic analysis

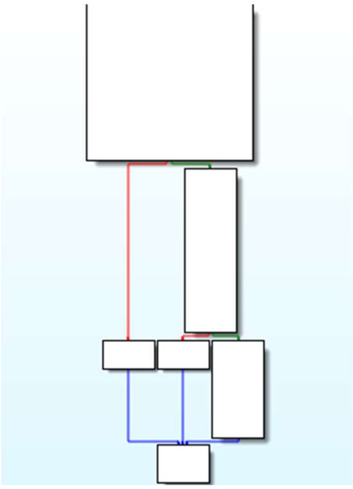
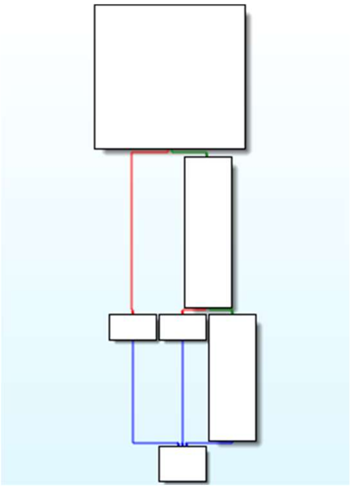
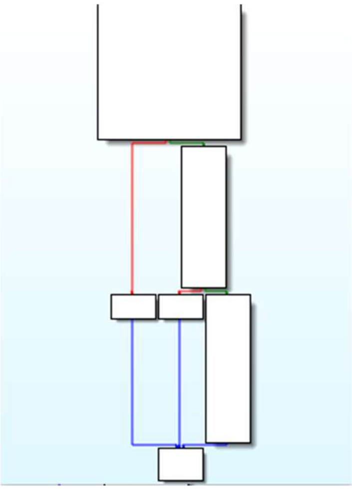
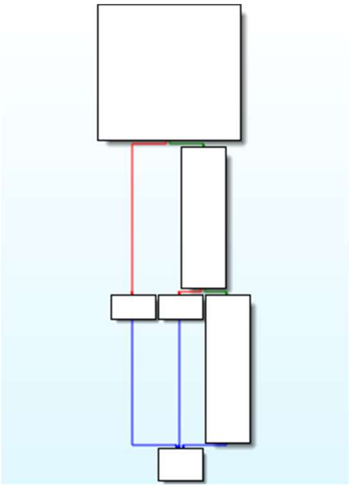
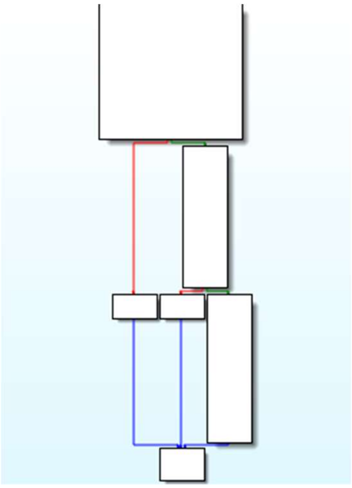
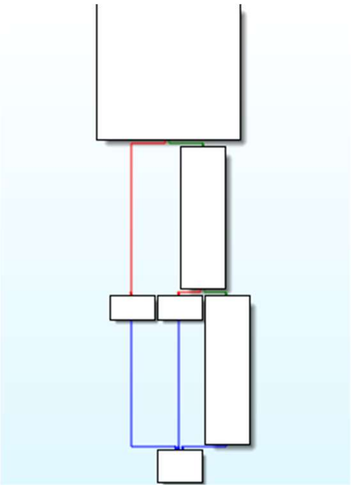
First, let's survey the high-level features. There are 11 functions:

- `$func1`
- `$func2`
- `$func3`

- `$func4`
- `$func5`
- `$func6`
- `$func7`
- `$func8`
- `$func9`
- `Match`
- `writev_c`

While `$func1` doesn't do anything, `Match` is the exported routine invoked by `main.js`. We'll return our attention here in a moment. `writev_c` is a function exported to `main.js` and used to implement the `writev` system call handler within WebAssembly; this function is likely part of the runtime framework and probably not yet worth any effort.

As we review the remaining functions, a pattern emerges. The control flow structure of functions `$func2` through `$func8` are identical! Figure 15 through Figure 21 compare the control flow graph overview exported from IDA Pro; notice that they all have essentially the same dimensions and layout. (If you don't have access to the idawasm plugin, you can replicate this result by comparing the textual representation of these functions using the human-readable `.wat` file). This pattern indicates that the functions probably have the same logic but may differ in key instructions.

 *Figure 15 - $func2 control flow*	 *Figure 16 - $func3 control flow*	 Figure 17 - $func4 control flow
 *Figure 18 - $func5 control flow*	 *Figure 19 - $func6 control flow*	 *Figure 20 - $func7 control flow*

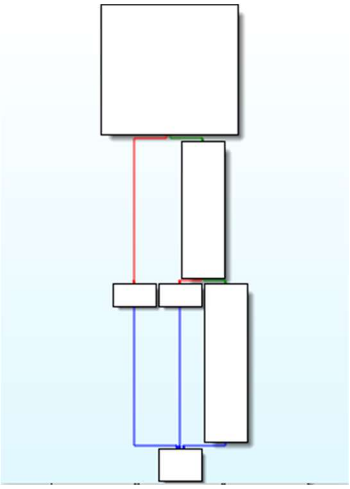
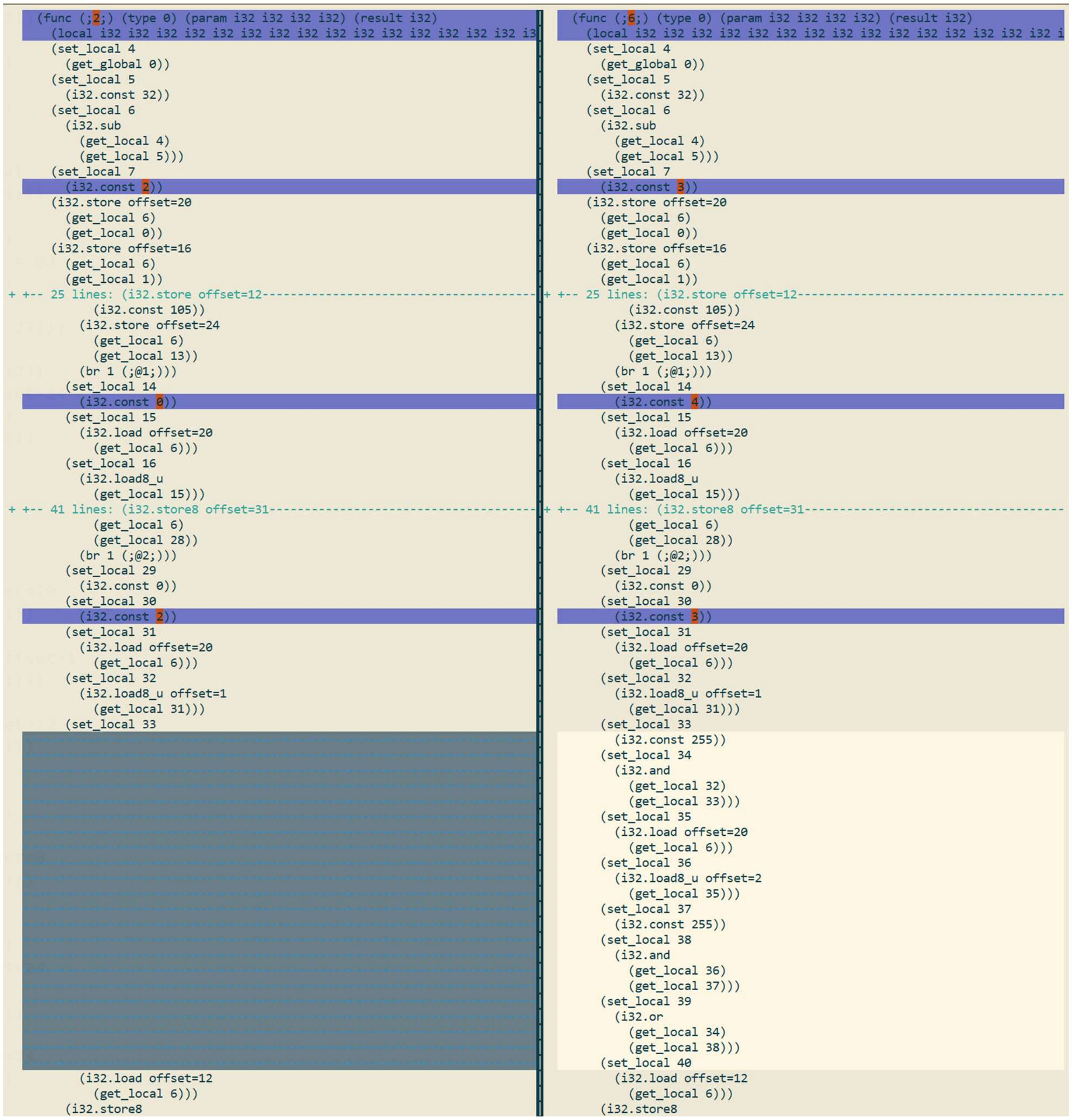


Figure 21 - $func8 control flow

By diffing the instructions among these functions, we quickly realize that the functions have changes in four places: in three immediate constants and in a small region of instructions. For example, Figure 22 illustrates the differences between `$func2` and `$func6`. We can see that `$local7`, `$local14`, and `$local30` are initialized with different immediate constants, and that `$func6` has additional code to initialize `$local33`. Table 1 summarizes the differences among all the related functions.



```
  (func (;2;) (type 0) (param i32 i32 i32 i32) (result i32)
    (local i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i3
    (set_local 4
      (get_global 0))
    (set_local 5
      (i32.const 32))
    (set_local 6
      (i32.sub
        (get_local 4)
        (get_local 5)))
    (set_local 7
      (i32.const 2))
    (i32.store offset=20
      (get_local 6)
      (get_local 0))
    (i32.store offset=16
      (get_local 6)
      (get_local 1))
+ +-- 25 lines: (i32.store offset=12--------------------------------------
          (i32.const 105))
        (i32.store offset=24
          (get_local 6)
          (get_local 13))
        (br 1 (;@1;)))
      (set_local 14
        (i32.const 0))
      (set_local 15
        (i32.load offset=20
          (get_local 6)))
      (set_local 16
        (i32.load8_u
          (get_local 15)))
+ +-- 41 lines: (i32.store8 offset=31-------------------------------------
          (get_local 6)
          (get_local 28))
        (br 1 (;@2;)))
      (set_local 29
        (i32.const 0))
      (set_local 30
        (i32.const 2))
      (set_local 31
        (i32.load offset=20
          (get_local 6)))
      (set_local 32
        (i32.load8_u offset=1
          (get_local 31)))
      (set_local 33
        (i32.load offset=12
          (get_local 6)))
      (i32.store8
```

```
  (func (;6;) (type 0) (param i32 i32 i32 i32) (result i32)
    (local i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i
    (set_local 4
      (get_global 0))
    (set_local 5
      (i32.const 32))
    (set_local 6
      (i32.sub
        (get_local 4)
        (get_local 5)))
    (set_local 7
      (i32.const 3))
    (i32.store offset=20
      (get_local 6)
      (get_local 0))
    (i32.store offset=16
      (get_local 6)
      (get_local 1))
+ +-- 25 lines: (i32.store offset=12--------------------------------------
          (i32.const 105))
        (i32.store offset=24
          (get_local 6)
          (get_local 13))
        (br 1 (;@1;)))
      (set_local 14
        (i32.const 4))
      (set_local 15
        (i32.load offset=20
          (get_local 6)))
      (set_local 16
        (i32.load8_u
          (get_local 15)))
+ +-- 41 lines: (i32.store8 offset=31-------------------------------------
          (get_local 6)
          (get_local 28))
        (br 1 (;@2;)))
      (set_local 29
        (i32.const 0))
      (set_local 30
        (i32.const 3))
      (set_local 31
        (i32.load offset=20
          (get_local 6)))
      (set_local 32
        (i32.load8_u offset=1
          (get_local 31)))
      (set_local 33
        (i32.const 255))
      (set_local 34
        (i32.and
          (get_local 32)
          (get_local 33)))
      (set_local 35
        (i32.load offset=20
          (get_local 6)))
      (set_local 36
        (i32.load8_u offset=2
          (get_local 35)))
      (set_local 37
        (i32.const 255))
      (set_local 38
        (i32.and
          (get_local 36)
          (get_local 37)))
      (set_local 39
        (i32.or
          (get_local 34)
          (get_local 38)))
      (set_local 40
        (i32.load offset=12
          (get_local 6)))
      (i32.store8
```

Figure 22 - Differences between $func2 and $func6

FUNCTION NAME	$LOCAL7	$LOCAL14	$LOCAL30	$LOCAL33	NOTABLE INSTRUCTIONS
$FUNC2	2	0	2		
$FUNC3	2	1	2		i32.xor
$FUNC4	3	2	3		i32.xor
$FUNC5	3	3	3		i32.and
$FUNC6	3	4	3		i32.or
$FUNC7	3	5	3		i32.add
$FUNC8	3	6	3		i32.sub

Table 1 - Comparison of function features

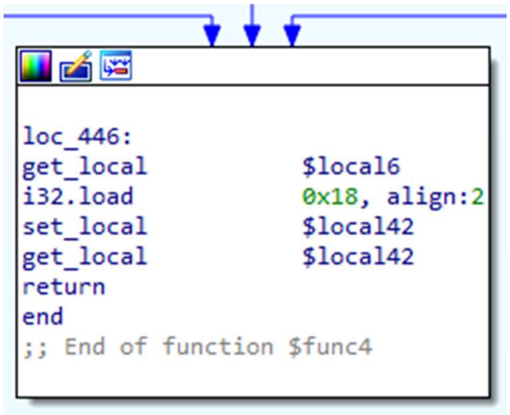
This table hints at the primary purpose of each function; however, to really understand what's happening, a closer inspection of the logic is required. Let's break down `$func4` in detail over the next few sections, start with how the WebAssembly calling conventions works.

Calling convention

In WebAssembly, function arguments are pushed onto the operand stack from left to right. Once control enters a function, arguments can be referenced directly using the `get_local` instruction (the runtime uses the function declaration to map *N* declared parameters into the first *N* local variable slots).

A function returns values by pushing them onto the operand stack prior to invoking the `return` instruction. While WebAssembly will eventually support multiple return values, the current specification supports a single return value.

With this in mind, we can see in Figure 23 that `$func4` returns a single value that comes from local variable `$local42`. The instruction `get_local $local42` pushes the current value of local variable `$local42` onto the top of the stack, and the `return` instruction returns it.



```
loc_446:
get_local               $local6
i32.load                0x18, align:2
set_local               $local42
get_local               $local42
return
end
;; End of function $func4
```

Figure 23 - Final basic block of $func4

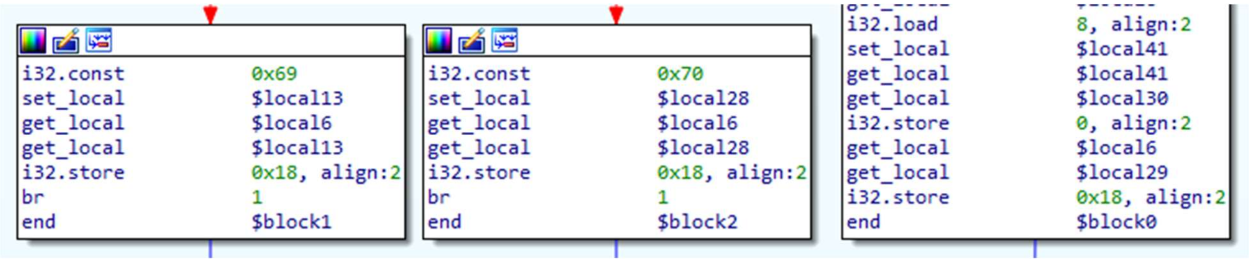
Memory references

As a stack-based machine, WebAssembly instructions cannot access arbitrary entries of the operand stack – only the top few. For sequences of contiguous data, WebAssembly exposes memory regions (well, in the current version, a single region) that can be indexed by a base and offset. Instructions such as `i32.load` fetch an element from a memory region and push it onto the top of the operand stack, where it can be manipulated by subsequent instructions.

As a specific example, the instruction `i32.load` fetches 32-bit unsigned, little-endian integer from a memory region. To compute the memory offset, the instruction pops from the operand stack a base offset value and adds to it the immediate constant offset. While the immediate constant offset is fixed, the base offset is a result of prior instructions, which enables pointer arithmetic in WebAssembly. For further information, refer to the second appendix to this solution document that summarizes the semantics of several other common WebAssembly instructions.

Referring back to Figure 23, we can be even more specific about the return value from `$func4`. The 32-bit value is read from the memory index `$local6 + 0x18`, assigned to local variable `$local42`, and then returned.

As we scan backwards through the function, it's easy to find the three basic blocks in which memory index `$local6 + 0x18` is written (The `i32.store` instruction works just like `i32.load`, except it reads an additional value from the operand stack and writes it into the computed memory index). Figure 24 shows these basic blocks. These cases correspond to the return values 0x69, 0x70, and 0x0, respectively. It's reasonable to assume these constants are error codes that indicate the success or failure of the function.



```
i32.const          0x69
set_local          $local13
get_local          $local6
get_local          $local13
i32.store          0x18, align:2
br                 1
end                $block1
```

```
i32.const          0x70
set_local          $local28
get_local          $local6
get_local          $local28
i32.store          0x18, align:2
br                 1
end                $block2
```

```
i32.load           8, align:2
set_local          $local41
get_local          $local41
get_local          $local30
i32.store          0, align:2
get_local          $local6
get_local          $local29
i32.store          0x18, align:2
end                $block0
```

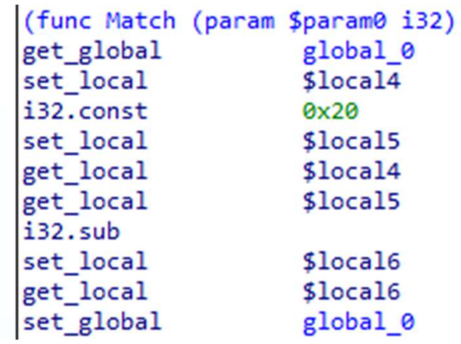
Figure 24 - Return value set in $func4

Frame pointer

In the example above, the base offsets come from local variable `$local6`, which we can interpret as the function frame pointer. The function frame is a construct specific to this compiler (LLVM) and not part of the WebAssembly specification. The compiler uses the frame pointer to prepare a region of function-local memory that can be arbitrarily indexed (remember, the operand stack cannot be indexed). The setup and teardown of the frame pointer happens in the function prologue and epilogue, just like in the Microsoft x64 ABI.

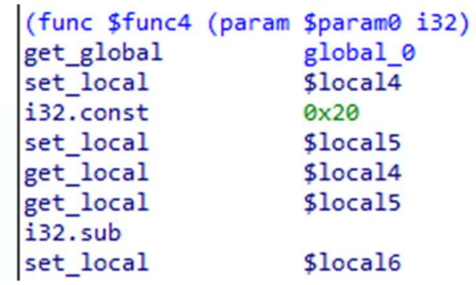
Figure 25 and Figure 26 show the prologues of a non-leaf and a leaf function, respectively. Notice that both fetch the current value of the global variable `$global0` (the "top of frame stack" pointer), allocate a region of 0x20 bytes, and store a pointer to this region in local variable `$local6`. The non-leaf function then updates the global variable `$global0`.

Figure 27 shows the epilogue of a non-leaf function, which de-allocates the function frame and updates the "top of frame stack" pointer `$global0`.



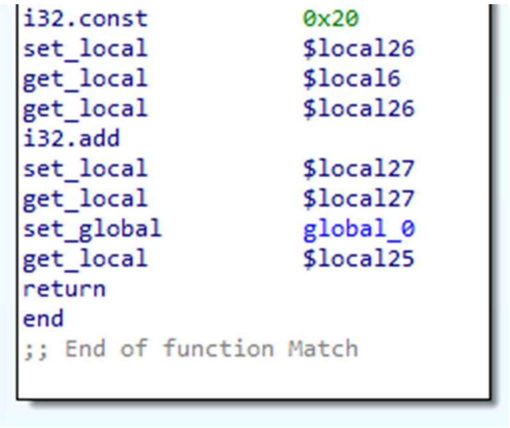
```
(func Match (param $param0 i32)
get_global          global_0
set_local           $local4
i32.const           0x20
set_local           $local5
get_local           $local4
get_local           $local5
i32.sub
set_local           $local6
get_local           $local6
set_global          global_0
```

Figure 25 - Function prologue of non-leaf function



```
(func $func4 (param $param0 i32)
get_global          global_0
set_local           $local4
i32.const           0x20
set_local           $local5
get_local           $local4
get_local           $local5
i32.sub
set_local           $local6
```

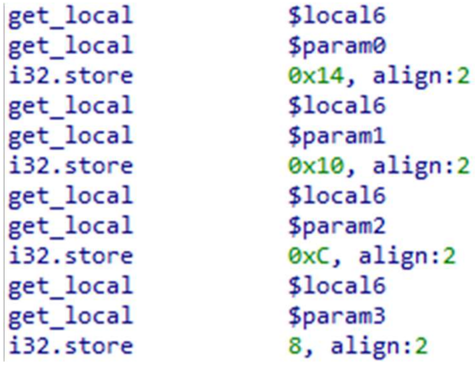
Figure 26 - Function prologue of leaf function



```
i32.const           0x20
set_local           $local26
get_local           $local6
get_local           $local26
i32.add
set_local           $local27
get_local           $local27
set_global          global_0
get_local           $local25
return
end
;; End of function Match
```

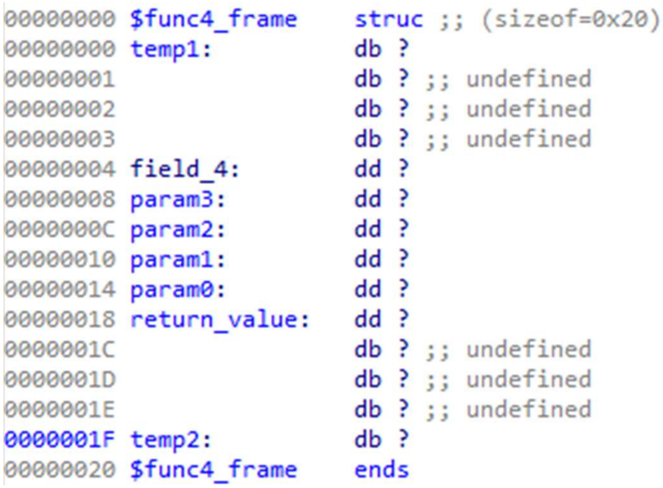
Figure 27 - Function epilogue of a non-leaf function

With this new understanding, we can scan through `$func4`, find memory load and store instructions that reference offsets relative to the frame pointer `$local6`, and map out the frame's layout. For example, in Figure 28, we can see that the four function parameters are saved off into frame offsets 0x14, 0x10, 0xC, and 0x8. Mapping out the other references leaves us with the layout shown in Figure 29.



```
get_local           $local6
get_local           $param0
i32.store           0x14, align:2
get_local           $local6
get_local           $param1
i32.store           0x10, align:2
get_local           $local6
get_local           $param2
i32.store           0xC, align:2
get_local           $local6
get_local           $param3
i32.store           8, align:2
```

Figure 28 - Copying of function parameters into the function frame



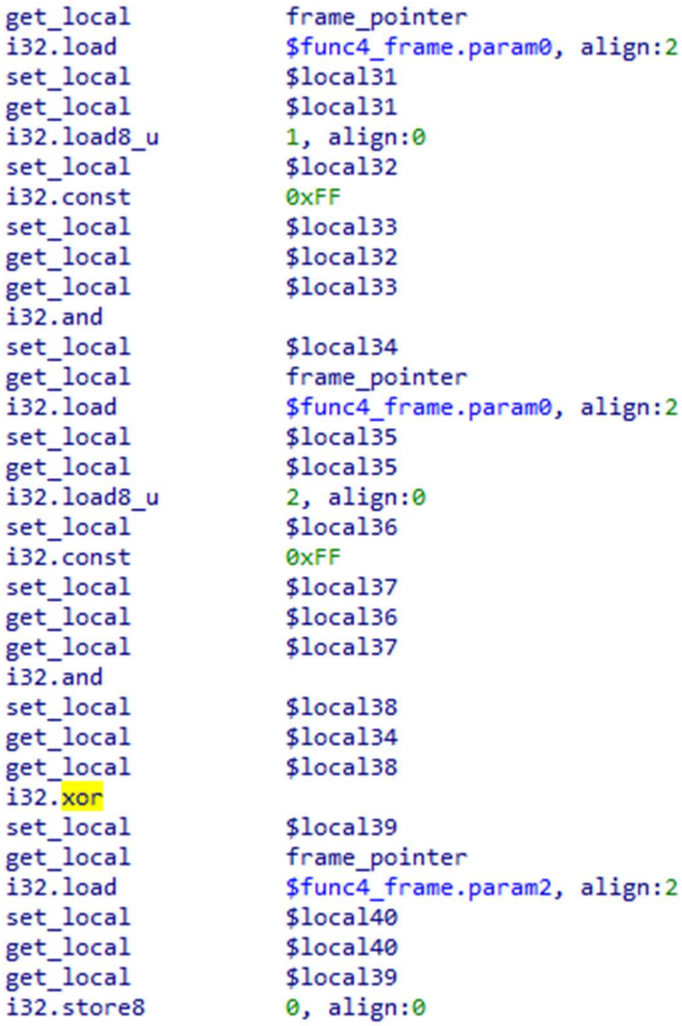
```
00000000 $func4_frame    struc ;; (sizeof=0x20)
00000000 temp1:          db ?
00000001                 db ? ;; undefined
00000002                 db ? ;; undefined
00000003                 db ? ;; undefined
00000004 field_4:        dd ?
00000008 param3:         dd ?
0000000C param2:         dd ?
00000010 param1:         dd ?
00000014 param0:         dd ?
00000018 return_value:   dd ?
0000001C                 db ? ;; undefined
0000001D                 db ? ;; undefined
0000001E                 db ? ;; undefined
0000001F temp2:          db ?
00000020 $func4_frame    ends
```

Figure 29 - Function frame layout for $func4

Finally, we can trace how the parameters to `$func4` are accessed. The key instructions are listed in Figure 30. Starting from the top, `$param0` is loaded from the function frame and used as a memory base offset to read a byte. This is a pointer dereference! Therefore, we can infer that the first argument is a byte array.

The code reads the byte at index 1 from our input buffer, uses a bitmask to ensure we're dealing with an eight-bit integer, and stores the value in `$local34`. Next, the code repeats itself, but this time reads the byte at index 2, and stores the result in `$local38`.

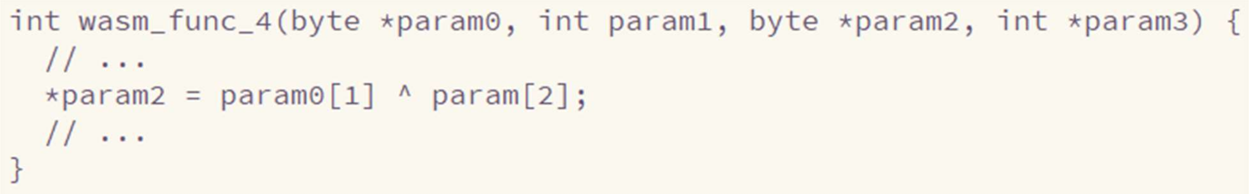
With `$local34` and `$local38`, the code XORs the byte values, storing the resulting in `$local39`. Finally, the code writes `$local39` into the memory index read from `$param2`.



```
get_local           frame_pointer
i32.load            $func4_frame.param0, align:2
set_local           $local31
get_local           $local31
i32.load8_u         1, align:0
set_local           $local32
i32.const           0xFF
set_local           $local33
get_local           $local32
get_local           $local33
i32.and
set_local           $local34
get_local           frame_pointer
i32.load            $func4_frame.param0, align:2
set_local           $local35
get_local           $local35
i32.load8_u         2, align:0
set_local           $local36
i32.const           0xFF
set_local           $local37
get_local           $local36
get_local           $local37
i32.and
set_local           $local38
get_local           $local34
get_local           $local38
i32.xor
set_local           $local39
get_local           frame_pointer
i32.load            $func4_frame.param2, align:2
set_local           $local40
get_local           $local40
get_local           $local39
i32.store8          0, align:0
```

Figure 30 - Key instructions in $func4

We might summarize this portion of the function with the C source code listed in Figure 31: the code computes the XOR of two bytes and writes the result into an output buffer.



```
int wasm_func_4(byte *param0, int param1, byte *param2, int *param3) {
  // ...
  *param2 = param0[1] ^ param[2];
  // ...
}
```

Figure 31 - Main functionality of $func4

After going back to repeat this style of analysis on the other similar functions, we can recover the following summaries:

FUNCTION NAME	PURPOSE
$FUNC2	compute `param0[1]`
$FUNC3	compute `~(param0[1])`
$FUNC4	compute `param0[1] ^ param0[2]`
$FUNC5	compute `param0[1] & param0[2]`
$FUNC6	compute `param0[1] | param0[2]`
$FUNC7	compute `param0[1] + param0[2]`
$FUNC8	compute `param0[1] – param0[2]`

Indirect calls

This is great progress! But, how are these functions used? When we search for `call` instructions, none reference any of these handler functions. Are they even used?

Fortunately, our effort is not wasted, as we notice an unusual `call_indirect` instruction within

`$func9`. WebAssembly does not allow a compiler to mix code and data, but provides the `call_indirect` instruction to support dynamic dispatch. This instruction pops a value off the top of the stack and uses it to index into the `elements` table. The `elements` table contains the indices of functions that may be invoked indirectly.

For example, consider the `elements` table `[0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]` found in `test.wasm` and a `call_indirect` instruction with the value `0x3` on the top of the stack. The runtime uses `0x3` to index into the table, fetching value `0x5`. This value `0x5` refers to the fifth function, so the runtime invokes `$func5`.

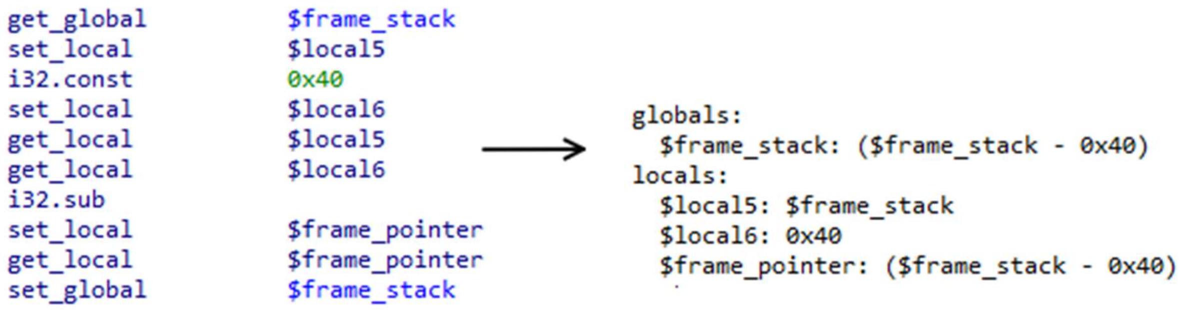
Resolving our elements table into a table of function names, we're left with `[$func2, $func3, $func4, $func5, $func6, $func7, $func8]`. These are the handlers we identified above! If we can figure out which index is placed onto the stack, and where the second parameter comes from, perhaps we can figure out how the flag is encoded.

The value of local variable `$local60` determines the index into the elements table. With enough patience, we can trace the flow data through other local variables and into `$local60`. To understand `$func9`, the manual approach is probably sufficient. However, if we plan to analyze other WebAssembly modules, we should consider developing additional tools to simplify our work.

By convention, WebAssembly compilers emit instructions that reference variables in Single Static Assignment (SSA) form. This is a handout to browser engines, because code that is already in SSA form is easier to import into analysis systems such as optimizers and JIT engines. Its SSA form that explains why we see dozens or hundreds of local variables in even the simplest functions.

But as a human, SSA form is tedious to analyze since we must trace operations across many separate local variables. To help us out, we might develop a WebAssembly emulator that can track instructions at a symbolic level. This would enable us to collapse a sequence of simple-but-related instructions into a single complex expression.

The idawasm project includes a WebAssembly code emulator that does just this! To use it, we select a region of instructions and run the `wasm_emu.py` script. The script emulates the instructions, simplifies their effects, and renders the effect to global variables, locals, memory, and the stack. Figure 32 shows how a function's prologue is simplified to a single global variable update.



```
get_global          $frame_stack
set_local           $local5
i32.const           0x40
set_local           $local6
get_local           $local5
get_local           $local6
i32.sub
set_local           $frame_pointer
get_local           $frame_pointer
set_global          $frame_stack
```

```
globals:
  $frame_stack: ($frame_stack - 0x40)
locals:
  $local5: $frame_stack
  $local6: 0x40
  $frame_pointer: ($frame_stack - 0x40)
```

Figure 32 - wasm-emu translation

With an emulator like `wasm_emu.py`, it's easier to understand the effects of a basic block of WebAssembly instructions. If we apply the emulator to the basic blocks of `$func9`, then we can quickly infer the following:

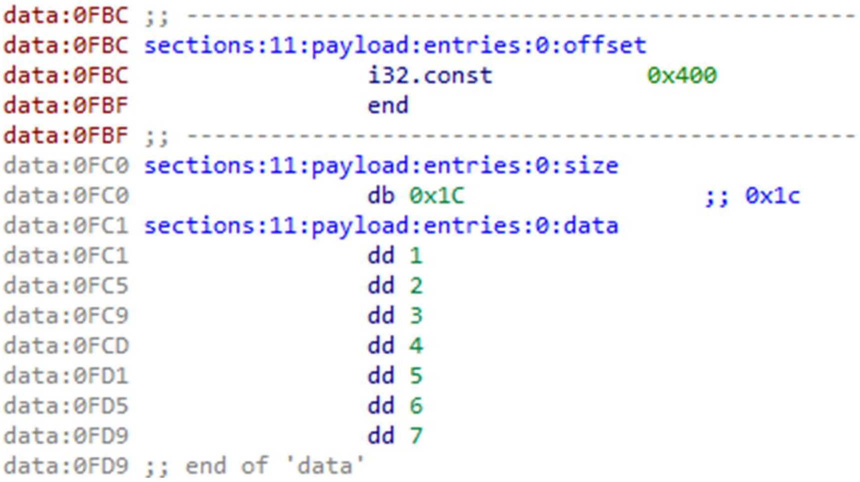
- `$local60` is the handler index that specifies one of `$func2` … `$func8`
- `$local60` contains the value from `$frame.field_10`, and
- `$frame.field_10` contains the value from `memory[memory[($frame.field_17<<0x2)+0x400]]`

Translating this expression into pseudo-C, we'd have something like:

```
int index = ((int *)400)[$frame.field_17 * 4]
```

Or, in other words, the 8-bit function frame member at offset `0x17` is used as an index into an array of 32-bit integers located at memory address `0x400`.

What's at address 0x400? The memory at `0x400` is not written by any instructions in `test.wasm`; however, the WebAssembly runtime uses the `data` section to initialize `0x1C` bytes starting at 0x400, as seen in Figure 33. Looks like this is yet another translation table to assist with dynamic dispatch. In summary, `$frame.field_17` indexes into table at `0x400`, which indexes into the `elements` table, which resolves to our handler routines. Neat!



```
data:0FBC ;; ---------------------------------------------
data:0FBC sections:11:payload:entries:0:offset
data:0FBC                 i32.const           0x400
data:0FBF                 end
data:0FBF ;; ---------------------------------------------
data:0FC0 sections:11:payload:entries:0:size
data:0FC0                 db 0x1C                    ;; 0x1c
data:0FC1 sections:11:payload:entries:0:data
data:0FC1                 dd 1
data:0FC5                 dd 2
data:0FC9                 dd 3
data:0FCD                 dd 4
data:0FD1                 dd 5
data:0FD5                 dd 6
data:0FD9                 dd 7
data:0FD9 ;; end of 'data'
```

Figure 33 - Memory initialization from data section

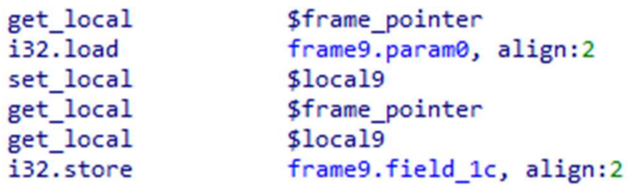
To find the contents of `$frame.field_17`, we trace back a bit further, finding:

- `$frame.field_17` contains `$frame.field_3f & 0xF`, and
- `$frame.field_3f` contains `memory[$frame.field_1c]`

So, the function index comes from the lower nibble of `$frame.field_3f`, which is the value dereferenced from `$frame.field_1c`. In pseudo-C:

```
byte index = ((byte *)$frame.field_1c)[0] & 0xF.
```

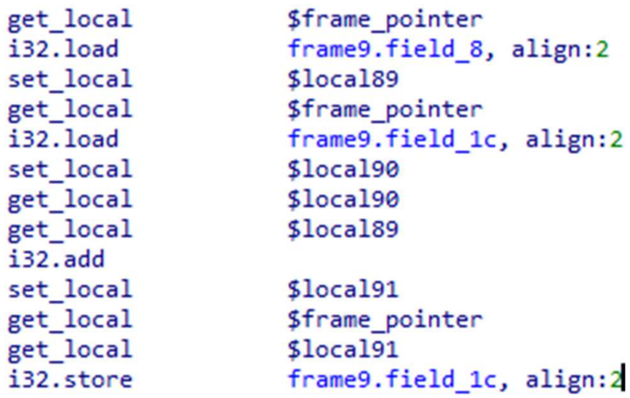
And what is `$frame.field_1C`? Figure 34 shows that it's initialized `to $param0`, and Figure 35 shows that it's incremented with each pass through the loop.



```
get_local           $frame_pointer
i32.load            frame9.param0, align:2
set_local           $local9
get_local           $frame_pointer
get_local           $local9
i32.store           frame9.field_1c, align:2
```

Figure 34 - Initialization of frame member at offset 0x1C

In pseudo-C: `$frame.field_1c = $frame.param0;`



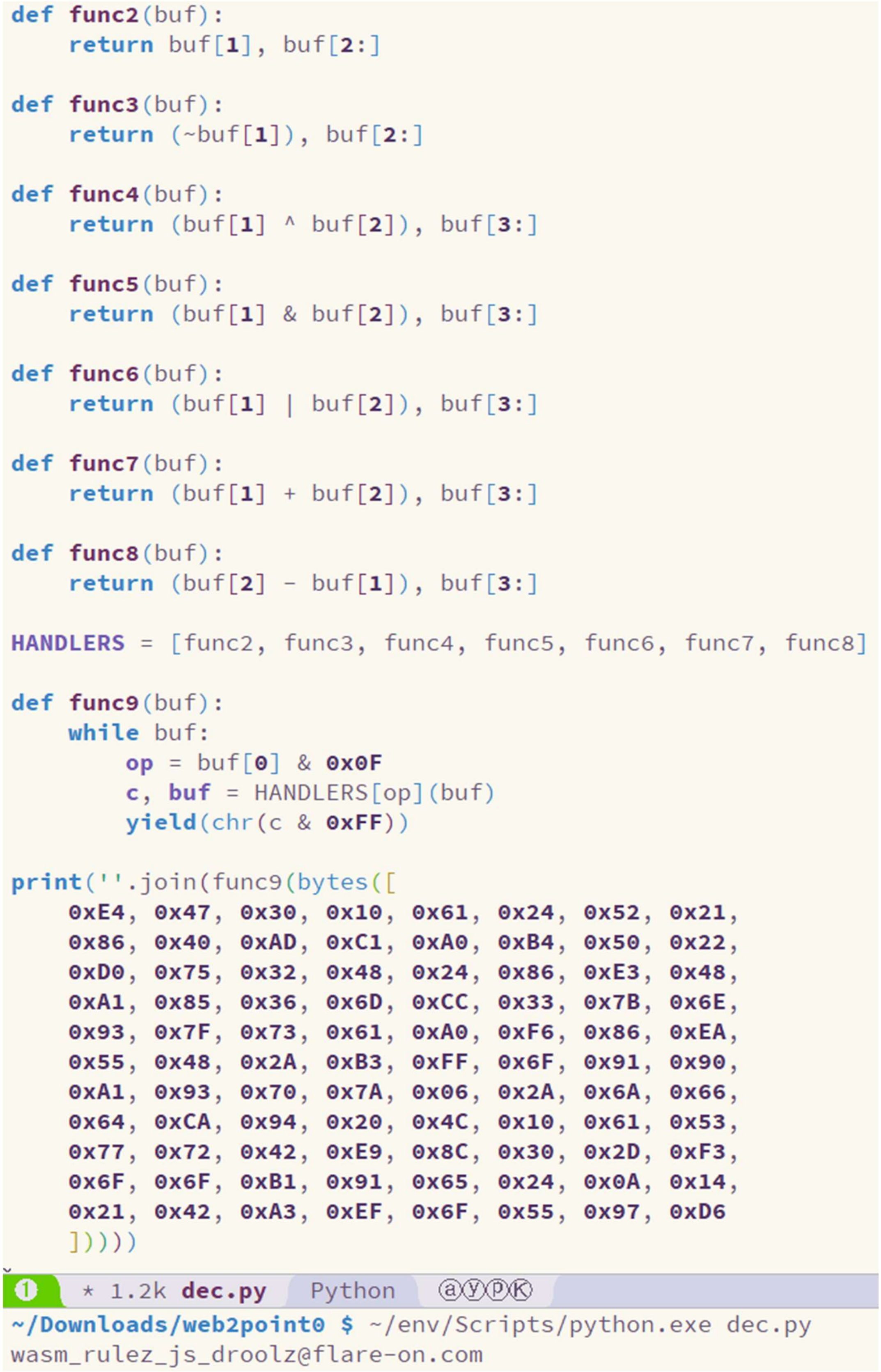
```
get_local           $frame_pointer
i32.load            frame9.field_8, align:2
set_local           $local89
get_local           $frame_pointer
i32.load            frame9.field_1c, align:2
set_local           $local90
get_local           $local90
get_local           $local89
i32.add
set_local           $local91
get_local           $frame_pointer
get_local           $local91
i32.store           frame9.field_1c, align:2
```

Figure 35 - Update of frame member at offset 0x1C

In pseudo-C: `$frame.field_1c += $frame.field_8;`

After chasing down a few details (that are left as an exercise for the reader), we can infer that `$frame.field_1C` is a pointer to the encrypted blob. The lower nibble of the byte it points to specifies the handler to invoke, and the handler manipulates subsequent bytes (e.g. ADD, XOR, NOT, etc.). Then, `$frame.field_1C` increments by the number of bytes consumed. The key comparison routine ensures that characters from the user-provided key match the data decrypted from the blob.

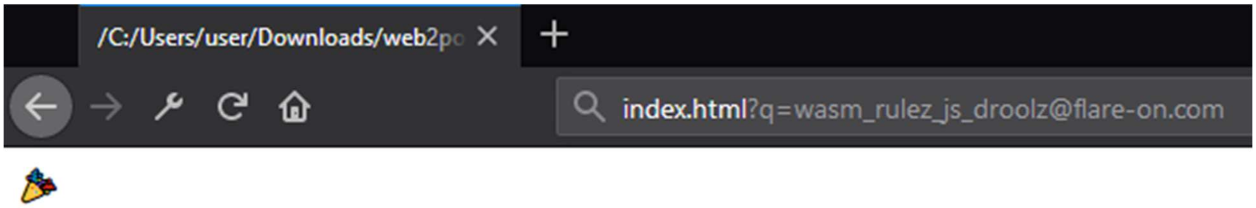
With this in mind, we can develop the script to dump the decrypted key shown in Figure 36; Figure 37 shows us solving challenge five successfully!



```python
def func2(buf):
    return buf[1], buf[2:]

def func3(buf):
    return (~buf[1]), buf[2:]

def func4(buf):
    return (buf[1] ^ buf[2]), buf[3:]

def func5(buf):
    return (buf[1] & buf[2]), buf[3:]

def func6(buf):
    return (buf[1] | buf[2]), buf[3:]

def func7(buf):
    return (buf[1] + buf[2]), buf[3:]

def func8(buf):
    return (buf[2] - buf[1]), buf[3:]

HANDLERS = [func2, func3, func4, func5, func6, func7, func8]

def func9(buf):
    while buf:
        op = buf[0] & 0x0F
        c, buf = HANDLERS[op](buf)
        yield(chr(c & 0xFF))

print(''.join(func9(bytes([
    0xE4, 0x47, 0x30, 0x10, 0x61, 0x24, 0x52, 0x21,
    0x86, 0x40, 0xAD, 0xC1, 0xA0, 0xB4, 0x50, 0x22,
    0xD0, 0x75, 0x32, 0x48, 0x24, 0x86, 0xE3, 0x48,
    0xA1, 0x85, 0x36, 0x6D, 0xCC, 0x33, 0x7B, 0x6E,
    0x93, 0x7F, 0x73, 0x61, 0xA0, 0xF6, 0x86, 0xEA,
    0x55, 0x48, 0x2A, 0xB3, 0xFF, 0x6F, 0x91, 0x90,
    0xA1, 0x93, 0x70, 0x7A, 0x06, 0x2A, 0x6A, 0x66,
    0x64, 0xCA, 0x94, 0x20, 0x4C, 0x10, 0x61, 0x53,
    0x77, 0x72, 0x42, 0xE9, 0x8C, 0x30, 0x2D, 0xF3,
    0x6F, 0x6F, 0xB1, 0x91, 0x65, 0x24, 0x0A, 0x14,
    0x21, 0x42, 0xA3, 0xEF, 0x6F, 0x55, 0x97, 0xD6
    ]))))
```

```
~/Downloads/web2point0 $ ~/env/Scripts/python.exe dec.py
wasm_rulez_js_droolz@flare-on.com
```

Figure 36 - Decoder script and decrypted key



/C:/Users/user/Downloads/web2po

index.html?q=wasm_rulez_js_droolz@flare-on.com

Figure 37 - Successfully recovering the key

Appendix: Further resources

- Project homepage: https://webassembly.org/
- Design documents: https://github.com/WebAssembly/design
- The WebAssembly Binary Toolkit: https://github.com/WebAssembly/wabt
- The WebAssembly Studio: https://webassembly.studio/
- IDA Pro loader and processor module: https://github.com/fireeye/idawasm
- Python parser and disassembler: https://github.com/athre0z/wasm
- Radare2 support: https://github.com/radare/radare2/tree/master/libr/asm/arch/wasm
- Analysis techniques: https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/understanding-web-assembly.pdf

Appendix: Common instruction reference

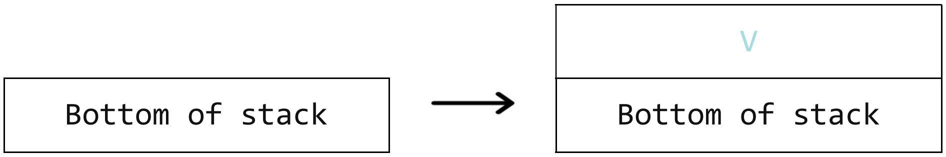
WebAssembly is a stack-based architecture. Instructions may have up to one immediate operand, and push and/or pop additional operands from the stack. The following section enumerates the semantics of instructions commonly encountered when reverse engineering WebAssembly modules.

i32.const

i32.const pushes the immediate constant value onto the top of the stack.

Example:

```
i32.const V
```

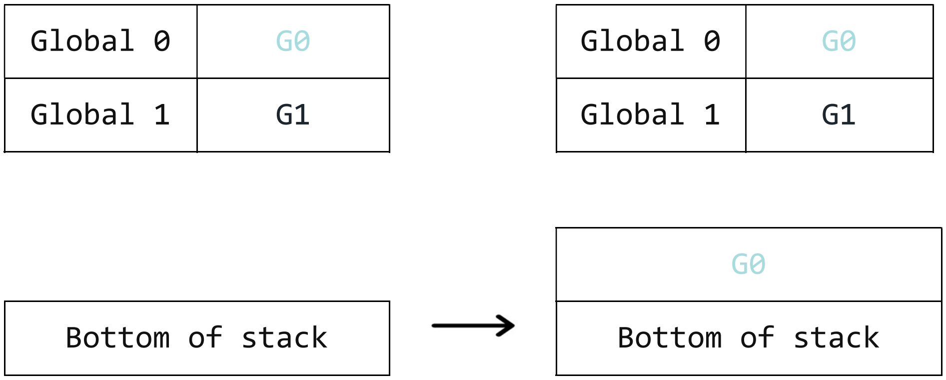


get_global

`get_global` pushes the current value of the global variable identified by the immediate onto the top of the stack.

Example:

```
get_global 0
```

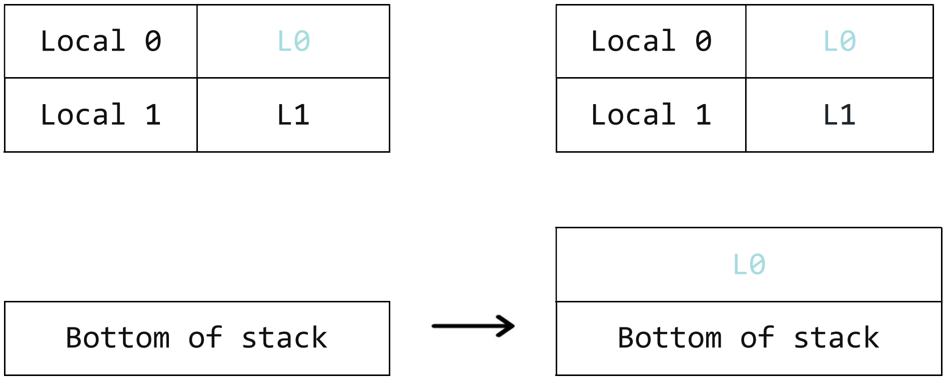


get_local

`get_local` pushes the current value of the local variable identified by the immediate onto the top of the stack.

Example:

`get_local 0`

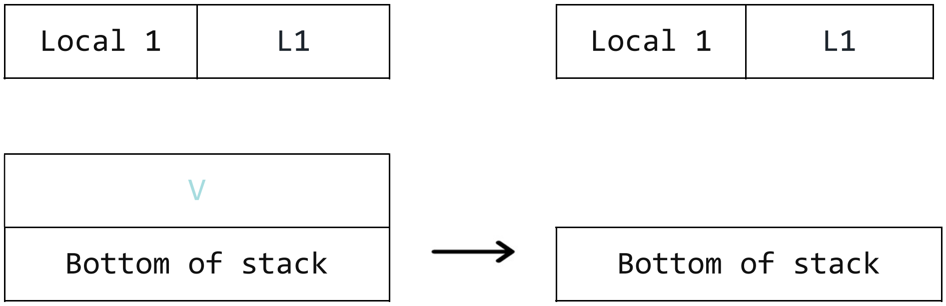


set_local

`set_local` pops the value off the top of the stack and assigns it to the local variable identified by the immediate.

Example:

`set_local 0`

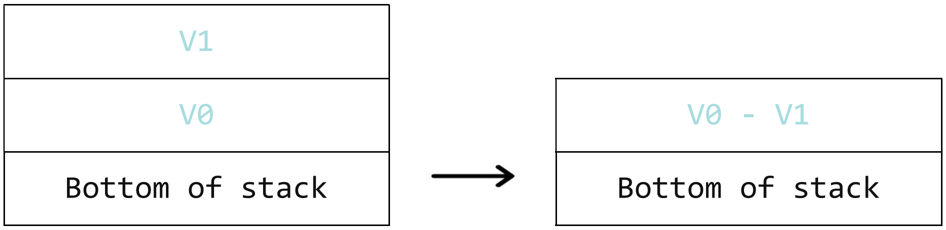


i32.sub

`i32.sub` pops two values off the top of the stack, subtracts one from the other, and pushes the result onto the top of the stack.

Example:

```
i32.sub
```

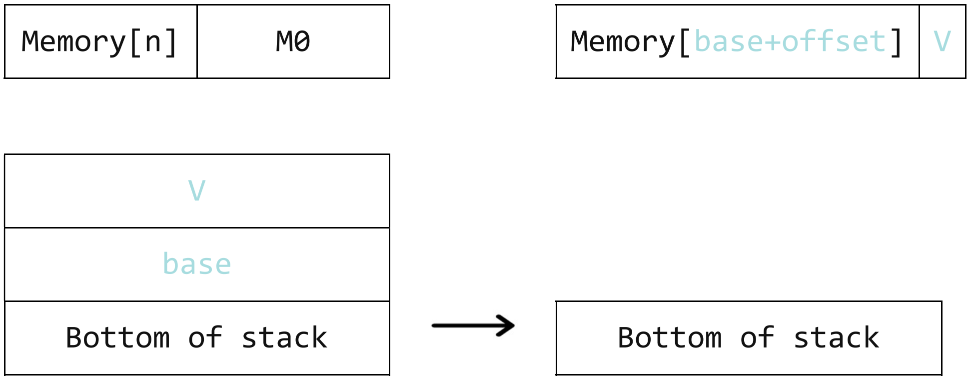


i32.store

`i32.store` operates on three values: two from the stack, and one immediate value. It pops the two values (base and V) off the top of the stack, and stores V at the memory cell identified by the sum of the base value and the immediate operand value (offset).

Example:

`i32.store offset`



i32.load

`i32.load` operates on two values: one from the stack, and one immediate value. It pops the base value from the top of the stack, and pushes onto the top of the stack the 32-bit integer value from the memory cell identified by the sum of the base value and the immediate operand value (offset).

Example:

`i32.load offset`