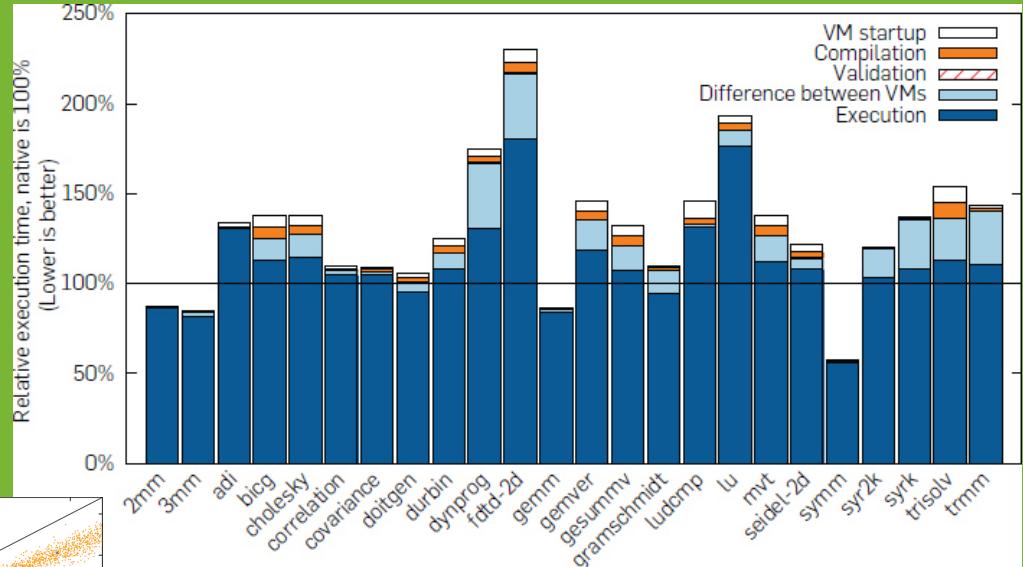
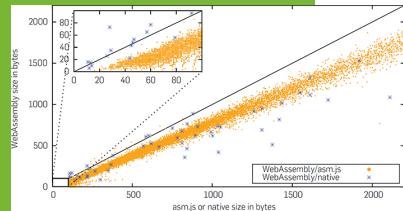


WASM

WebAssembly

John Feiner



<https://cacm.acm.org/magazines/2018/12/232881-bringing-the-web-up-to-speed-with-webassembly/fulltext>

Web Assembly (WASM)

Basic Idea:

High performance web applications

How to Improve Performance?

No code / Calculation on server HTML 1.0

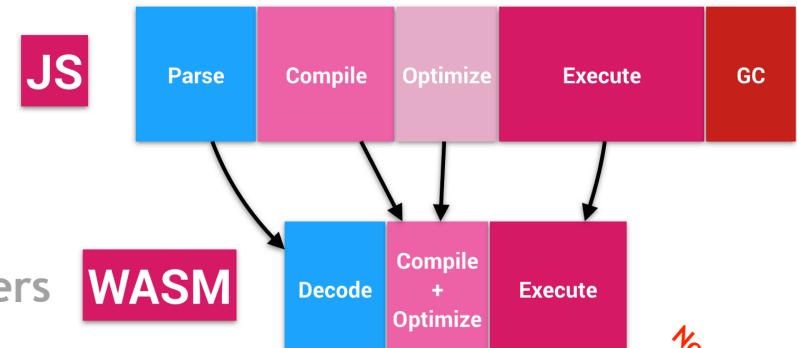
↳ Plugins ↳ Flash, Java Applets

Interpreter

- + better CPU, more RAM
- + caching: Cookies, IndexDB, Service Workers
- + multithreading: Workers (2012)
- + hardware support (GPU): CSS-Animation(?), WebGL, WebGPU

JIT

Compiler ↳ High Level language -> IR (optimisation, LLVM) -> Machine Code



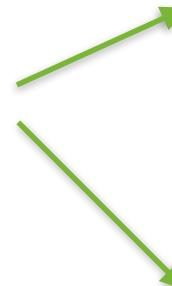
WebAssembly

– Selected Applications

Browser-based Crypto Mining

Find extensive link collection and more on
"Awesome WASM"

<https://github.com/mbasso/awesome-wasm#slides>



Digging into Browser-based Crypto Mining

Jan Rüth, Torsten Zimmermann, Konrad Wolsing, Oliver Hohlfeld
Communication and Distributed Systems, RWTH Aachen University, Germany
[lastname]@comsys.rwth-aachen.de

ABSTRACT

Mining is the foundation of blockchain-based cryptocurrencies such as Bitcoin rewarding the miner for finding blocks for new transactions. The Monero currency enables mining with standard hardware in contrast to special hardware (ASICs) as often used in Bitcoin, paving the way for in-browser mining as a new revenue model for website operators. In this work, we study the prevalence of this new phenomenon. We identify and classify mining websites in 138M domains and present a new fingerprinting method which finds up to a factor of 5 more miners than publicly available block lists. Our work identifies and dissects Coinhive as the major browser-mining stakeholder. Further, we present a new method to associate mined blocks in the Monero blockchain to mining pools and uncover that Coinhive currently contributes 1.18% of mined blocks having turned over 1293 Moneros in June 2018.

requires miners to solve a computationally expensive puzzle to cryptographically link a new block to the previous block in the blockchain. The difficulty to solve this puzzle depends on the combined computing power of all users—depending on the difficulty, an individual requires powerful machines to increase the probability of mining a block (e.g., GPUs, FPGAs, or even ASICs). To provide an incentive for contributing computational power, miners are awarded currency for every mined block. This monetary reward has rendered crypto mining a business—browser-based mining extends this business to monetize the web.

Not all cryptocurrencies are equally suited for browser-based mining. The hardware imbalance and the consequential high difficulty to mine Bitcoin renders its in-browser mining inefficient and motivates the use of, e.g., Monero as an alternative currency that can be efficiently mined on CPUs and thus browsers. Given its design, Monero has been adopted by websites (*e.g.*, The Pirate Bay or a video

Digging into Browser-based Crypto Mining

	Alexa Class.	Count	.org Class.	Count
1	coinhive	311	coinhive	711
2	skencituer	123	cryptoloot	183
3	cryptoloot	103	web.stati.bid	120
4	UnknownWSS	56	freecontent.date	108
5	notgiven688	46	notgiven688	92
Total	WebAssembly	796	WebAssembly	1491

Table 1: Top 5 (~80%) WebAssembly signatures. Most WebAssembly are miners (~96%), dominated by Coinhive.

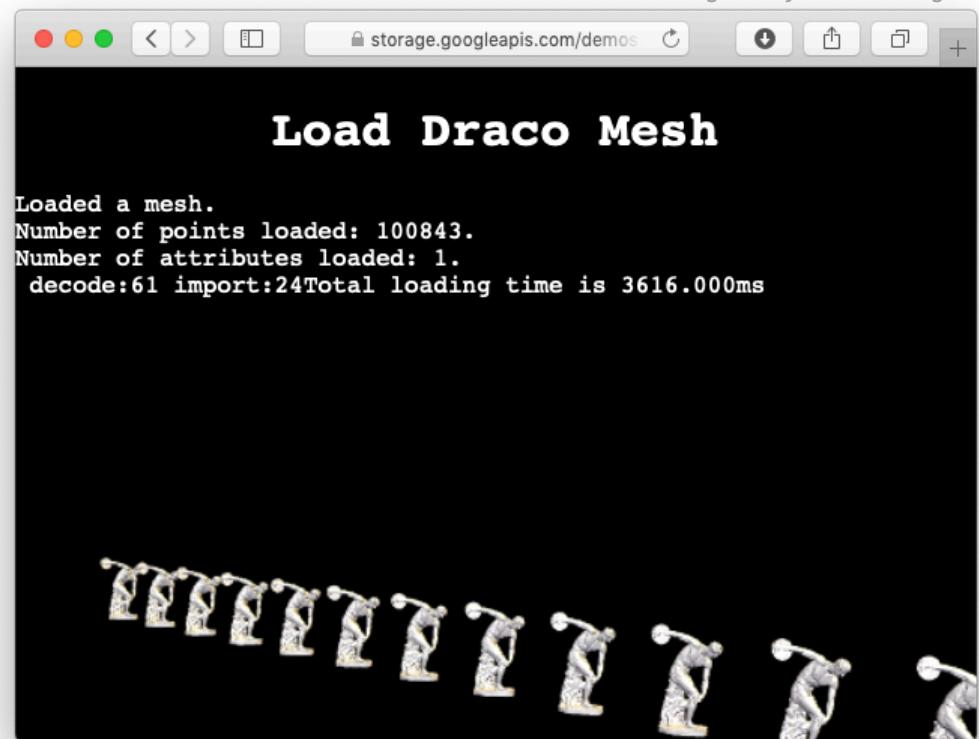
[https://doi.org/
10.1145/3278532.3278539](https://doi.org/10.1145/3278532.3278539)

Web Assembly

Examples:

Google "draco"
library: compress
3D point clouds

<https://github.com/google/draco>



[https://storage.googleapis.com/demos.webmproject.org/
draco/draco_loader_throw.html](https://storage.googleapis.com/demos.webmproject.org/draco/draco_loader_throw.html)

Review JavaScript

How JS works... and why it might be slow

John Feiner

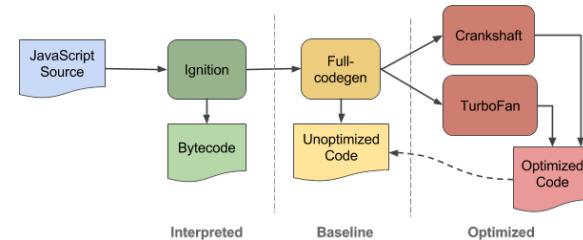
About JavaScript Engines

Blink/V8 (Chrome, announced 2018: Edge)

Gecko/IonMonkey (Firefox)

Trident/Chakra (Edge until late 2018)

WebKit/JSCore (Safari)



WAS: Interpreter

IS: Just-in-time compiler (JIT), inline caching (IC) reduce lookups

JavaScript is Assembly Language of the Web

Scott Hanselman 2011,
Principal Program Manager Lead at Microsoft

<https://www.hanselman.com/blog/>
[JavaScriptIsAssemblyLanguageForTheWebSematicMarkupIsDeadCleanVsMachinecodedHTML.aspx](#)

view minified sources

```
script type="text/javascript">var date = new Date();date.setMinutes(0);date.setSeconds(0);date.setMilliseconds(0);var cookieString = "ASP.NET_SessionId=" + date.toISOString();document.cookie = cookieString;function getCookieValue(cookieName){var cookieArr = document.cookie.split(" ");for(var i=0;i<cookieArr.length;i++){var cookie = cookieArr[i];if(cookie.substring(0, cookieName.length) == cookieName){return cookie.substring(cookieName.length+1)}}}function setCookie(cookieName,cookieValue){var date = new Date();date.setTime(date.getTime()+(1000*60*60*24*30));var cookieString = cookieName + "=" + cookieValue + "; expires=" + date.toUTCString();document.cookie = cookieString}function getLocalStorageValue(key){var val = localStorage.getItem(key);if(val != null){val = decodeURIComponent(val)}}function setLocalStorageValue(key,value){localStorage.setItem(key, encodeURIComponent(value))}function getLocalStorageLength(){return localStorage.length}function removeLocalStorageItem(key){localStorage.removeItem(key)}function clearLocalStorage(){localStorage.clear()}function getLocalStorageKeys(){return Object.keys(localStorage)}function getLocalStorageKeyIndex(key){var keys = getLocalStorageKeys();for(var i=0;i<keys.length;i++){if(keys[i] == key){return i}}}
```

```
function getLocalStorageValue(cookieName){var cookieArr = document.cookie.split(" ");for(var i=0;i<cookieArr.length;i++){var cookie = cookieArr[i];if(cookie.substring(0, cookieName.length) == cookieName){return cookie.substring(cookieName.length+1)}}}}
```

```
function setCookie(cookieName,cookieValue){var date = new Date();date.setTime(date.getTime()+(1000*60*60*24*30));var cookieString = cookieName + "=" + cookieValue + "; expires=" + date.toUTCString();document.cookie = cookieString}}
```

```
function getLocalStorageLength(){return localStorage.length}}
```

```
function removeLocalStorageItem(cookieName){localStorage.removeItem(cookieName)}
```

```
function clearLocalStorage(){localStorage.clear()}
```

```
function getLocalStorageKeys(){return Object.keys(localStorage)}
```

```
function getLocalStorageKeyIndex(cookieName){var keys = getLocalStorageKeys();for(var i=0;i<keys.length;i++){if(keys[i] == cookieName){return i}}}}
```

Review -- JIT

monitor code:

cold

warm

hot

just-in-time:

compile

optimise

(and deoptimise)

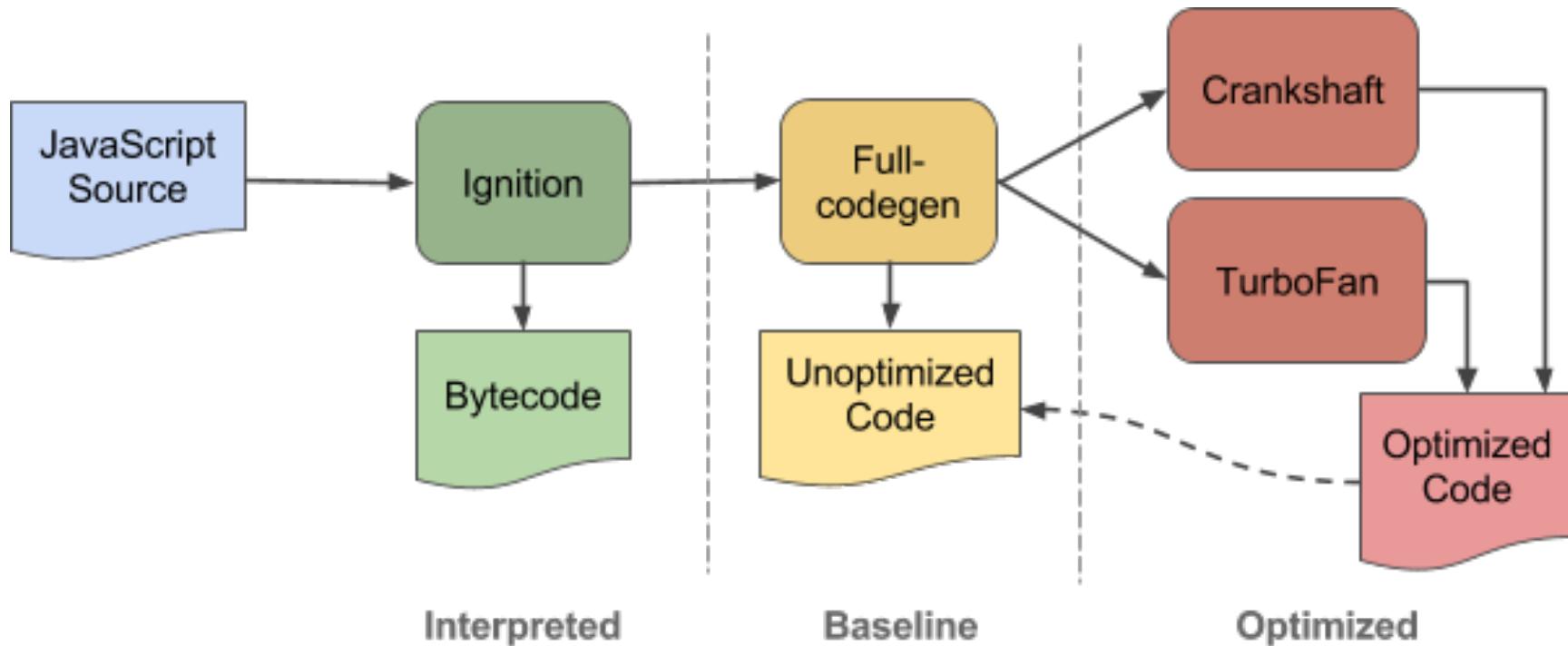


```
function arraySum(arr) {  
    var sum = 0;  
    for (var i = 0; i < arr.length; i++) {  
        sum += arr[i];  
    }  
}
```

arraySum	line 4	sum = int	
arraySum	line 2	sum = int	
arraySum	line 3	arr = array	

dynamic types
polymorphic code ↗





JavaScript Engines are fast, but

Explain inline caching
property lookup
monomorphic

```
const luke = {  
    firstname: "Luke",  
    lastname: "Skywalker"};  
...  
const people = [luke, ...]
```

✓ fast

```
const getName = (person) => person.lastname;  
console.time("engine");  
for(var i = 0; i < 1000 * 1000 * 1000; i++) {  
    getName(people[i & 7]);  
}  
console.timeEnd("engine");
```

JavaScript Engines are fast, but possibly slow

Explain inline caching
property lookup

monomorphic

polymorphic

megamorphic

See also: 2023-04 Bugzilla reports "Ion compiled polymorphic calls are 10x slower than V8 (InvokeFunction)" https://bugzilla.mozilla.org/show_bug.cgi?id=1829411

```
const luke = {  
    firstname: "Luke",  
    lastname: "Skywalker"};  
const yoda = {  
    lastname: "Yoda"};  
...  
const people = [luke, yoda...]  
  
const getName = (person) => person.lastname;  
console.time("engine");  
for(var i = 0; i < 1000 * 1000 * 1000; i++) {  
    getName(people[i & 7]);  
}  
console.timeEnd("engine");
```

JavaScript Engines faster with TypeScript

Explain inline caching
property lookup
monomorphic

=> use JS classes,
use TypeScript

```
const yoda = new Person({  
    lastname: 'Yoda' });  
...  
const people = [  
    yoda, ...
```

✓ fast

```
const getName = person => person.lastname;  
console.time("engine");  
for(var i = 0; i < 1000 * 1000 * 1000; i++) {  
    getName(people[i & 7]);  
}  
console.timeEnd("engine");
```

JS fast/slow

Fibonacci Algorithm: WASM vs. JS

Web Assembly

41 Result: 165580141 Runtime: 1174ms

Javascript

41 Result: 165580141 Runtime: 3115ms

See the source on Github - [Link](#)

Demo

← Firefox
Chrome →

Fibonacci Algorithm: WASM vs. JS

Web Assembly

41 Result: 165580141 Runtime: 1218.9199999993434ms

Javascript

41 Result: 165580141 Runtime: 1915.5099999989034ms

See the source on Github - [Link](#)

<https://fib.johnny.sh/>

Review – Another Alternative to JS

wasm.js

John Feiner

What is asm.js?

<https://caniuse.com/#feat=asmjs>

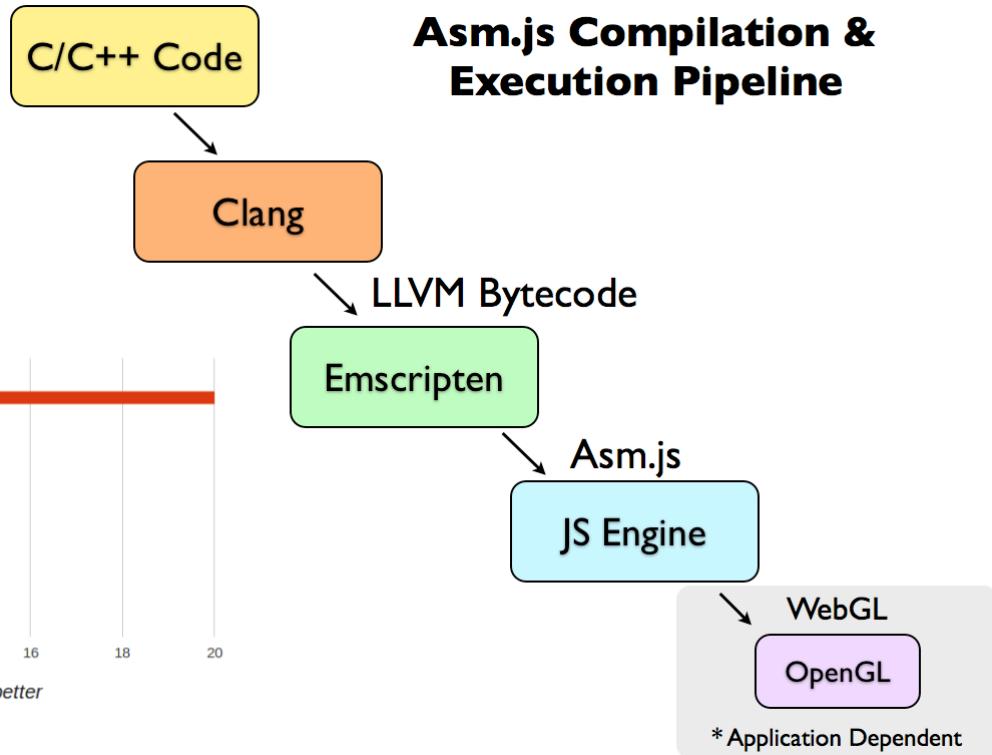
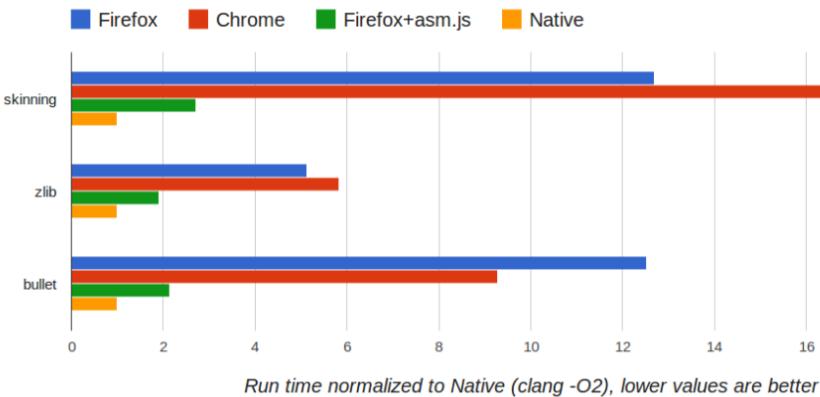
Subset of JavaScript (limited! "...*extraordinarily optimisable*...")

Write code in, for example, C, C++, or Rust then **translate**, output asm.js

i.e. write in a statically-typed language with manual memory management use Emscripten (input LLVM Bytecode) for **source-to-source compilation**, ahead-of-time optimisation, "virtually no garbage collection", no dynamic behaviour optionally: highly efficient native code

=> Runs in any browser, but runs in fast in Firefox (OdinMonkey with AoT-compilation)

Compare: asm.js



Example Speed Comparison: JS vs. asm.js

*"The ASM.js module of the Rust parser is in average **6 times** faster than the actual Javascript implementation. The median speedup is 6. That's far from the WebAssembly results, but **this is a fallback**, and in average, it is **6 times** faster, which is really great!"*

(Ian Enderling)

<https://mnt.io/2018/08/28/from-rust-to-beyond-the-asm-js-galaxy/>

asm.js with "emscripten" Compiler

The screenshot shows the official emscripten website at <https://emscripten.org>. The page features a dark header with the "emscripten" logo and navigation links for Documentation, Downloads, and Community. A "For me on GitHub" button is visible in the top right. The main content area has a white background. It includes a large "emscripten" logo, a brief introduction text, and three columns: "Porting", "APIs", and "Fast". Each column contains a short description of the feature.

Introducing Emscripten
Getting Started
Compiling and Running Projects
Porting
API Reference
Tools Reference
Optimizing Code
Optimizing WebGL
CyberDWARF Debugging
Building Emscripten from Source
Contributing to Emscripten
Profiling the Toolchain
About this site
Index

Documentation Downloads Community

For me on GitHub

emscripten

Emscripten is a toolchain for compiling to asm.js and WebAssembly, built using LLVM, that lets you run C and C++ on the web at near-native speed without plugins.

Porting **APIs** **Fast**

Compile your existing projects written in C or C++ and run them on all modern browsers.

Emscripten converts OpenGL into WebGL, and lets you use familiar APIs like SDL, or HTML5 directly.

Thanks to LLVM, Emscripten, asm.js and WebAssembly, code runs at near-native speed.

<https://emscripten.org>

The screenshot shows a GitHub wiki page titled "Porting Examples and Demos" for the repository "emscripten-core / emscripten". The page has a white background and a standard GitHub interface. It includes a "Porting Examples and Demos" heading, a "New Page" button, and a list of categories under "Games and Game Engines".

Code Issues 1,663 Pull requests 148 Projects 3 Wiki

Porting Examples and Demos

Edit New Page

Fabian Lauer edited this page 15 days ago - 68 revisions

- Games and Game Engines
- Graphics
- Emulators
- Application Frameworks
- Programming Languages
- Tutorials
- Utilities
- Other Examples

<https://github.com/emscripten-core/emscripten/wiki/Porting-Examples-and-Demos>

Step-by-step asm.js with Emscripten

brew install emscripten

1

```
#include <stdio.h>

int main() {
    printf("hello, world!\n");
    return 0;
}
```

hello_world.c (6 loc)

2

emcc hello_world.c

3

```
...
var Module = typeof Module !== 'undefined' ? Module : {};
...
var moduleOverrides = {};
var key;
for (key in Module) {
    if (Module.hasOwnProperty(key)) {
        moduleOverrides[key] = Module[key];
    }
}
Module['arguments'] = [];
Module['thisProgram'] = './this.program';
Module['quit'] = function(status, toThrow) {
    throw toThrow;
};
Module['preRun'] = [];
Module['postRun'] = [];
...
```

a.out.js (with about 2380 loc)

4

time node a.out.js
hello, world!

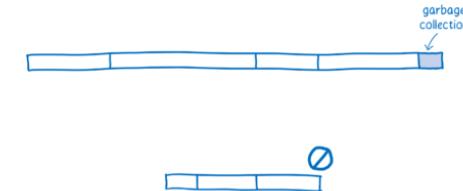
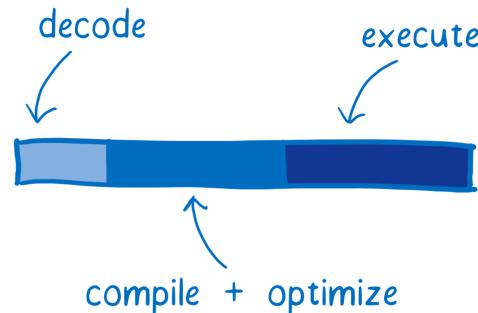
real 0m0.074s
user 0m0.090s
sys 0m0.020s

Full Speed = WASM

Web Assembly

John Feiner

Why is WASM so Fast?



Web Assembly (WASM)

History:

JavaScript 1995, V8 2008,
asm.js 2013, WASM 2015

Specification, 25 May 2020

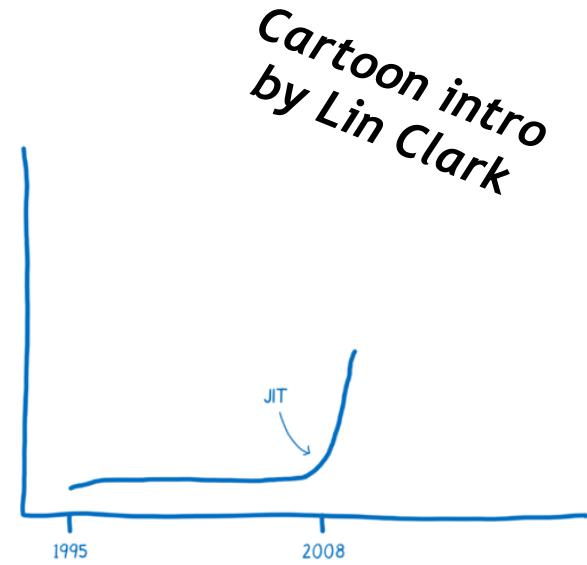
Technology: Stack machine, bridge to JS

Independent/Portable: Hardware, Language, Platform

Development: easy to inspect, debug

<https://webassembly.org/docs/high-level-goals/>

<https://cacm.acm.org/magazines/2018/12/232881-bringing-the-web-up-to-speed-with-webassembly/fulltext>



[https://hacks.mozilla.org/2017/02/
a-cartoon-intro-to-webassembly/](https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/)

Web Assembly (WASM)

Fast:

binary format (small size, load-time)

native execution speed

wasm
modules

Safe & secure:

memory-safe, sandboxed environment

permissions and same-origin policy of browser

Web Assembly (WASM)

Browser support:

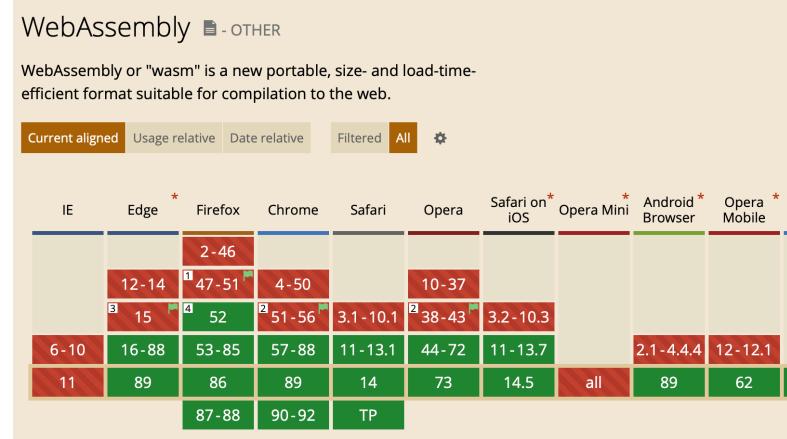
ok, see "can i use"

Compilation:

C++, ... WIP: Java, C#, Python, ...

Main supporters:

Firefox, Google, ...



WebAssembly - OTHER

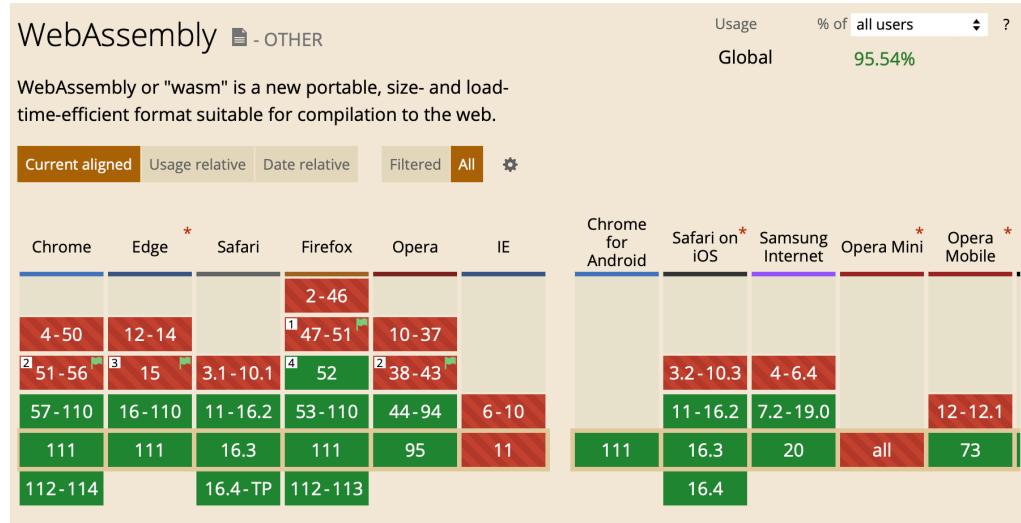
WebAssembly or "wasm" is a new portable, size- and load-time-efficient format suitable for compilation to the web.



2020-05-28

WebAssembly - OTHER

WebAssembly or "wasm" is a new portable, size- and load-time-efficient format suitable for compilation to the web.



2023-03-17

Web Assembly (WASM)

How does it work?

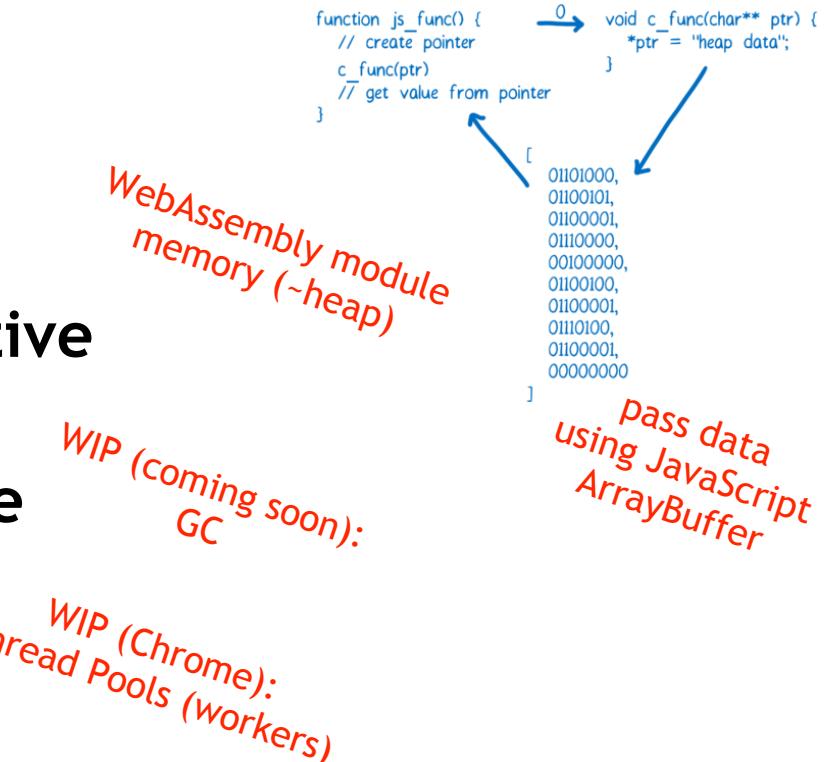
Pass data from browser to native code (and back)

Memory management: garbage collection?

Multithreading? stop tasks?

<https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/>

<https://developers.google.com/web/updates/2018/10/wasm-threads>



It is simple. E.g. Data Types



Introduction
 Structure
 Validation
 Execution
 Binary Format
 Text Format

- Conventions
- Lexical Format
- Values
- Types
- Instructions
- Modules

 Appendix
 Index of Types
 Index of Instructions
 Index of Semantic Rules

Index
 Download as PDF

Types

Number Types

```
numtype ::= 'i32'    ⇒ i32
          | 'i64'    ⇒ i64
          | 'f32'    ⇒ f32
          | 'f64'    ⇒ f64
```

Reference Types

```
reftype ::= 'funcref'   ⇒ funcref
          | 'externref'  ⇒ externref
heaptYPE ::= 'func'      ⇒ funcref
          | 'extern'     ⇒ externref
```

Value Types

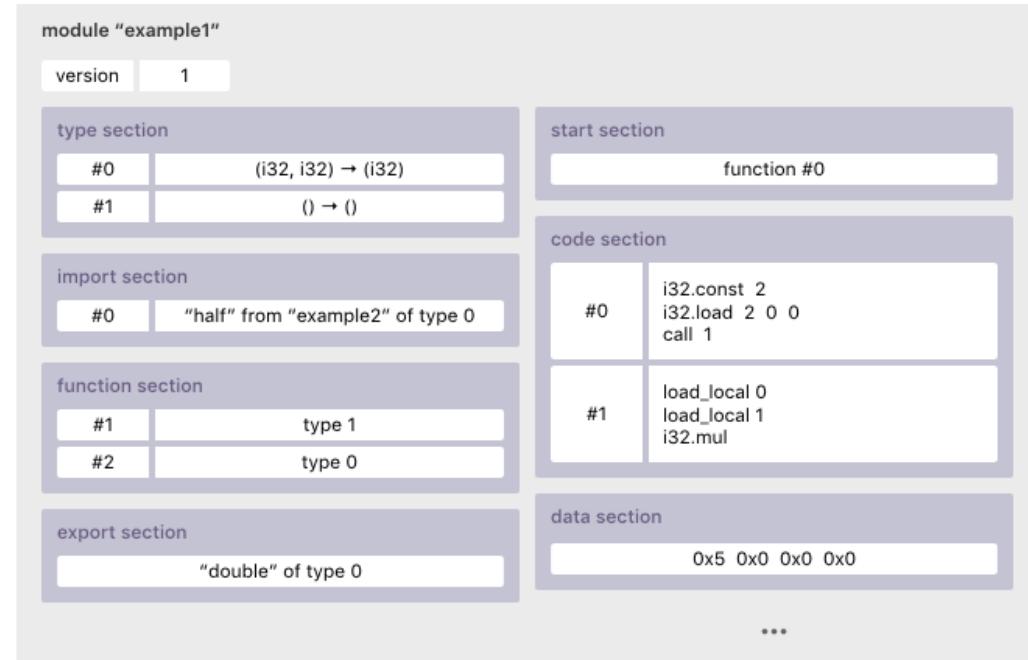
```
valtype ::= t:numtype  ⇒ t
          | t:reftype  ⇒ t
```

Function Types

```
functype ::= '(' 'func' t*: vec(param) t*: vec(result) ')' ⇒ [t*] → [t*]
param    ::= '(' 'param' id? t:valtype ')'                  ⇒ t
result   ::= '(' 'result' t:valtype ')'                      ⇒ t
```

WASM Modules

required sections:
type, function, code
optional sections:
export, import, start,
global, memory,
table, data, element

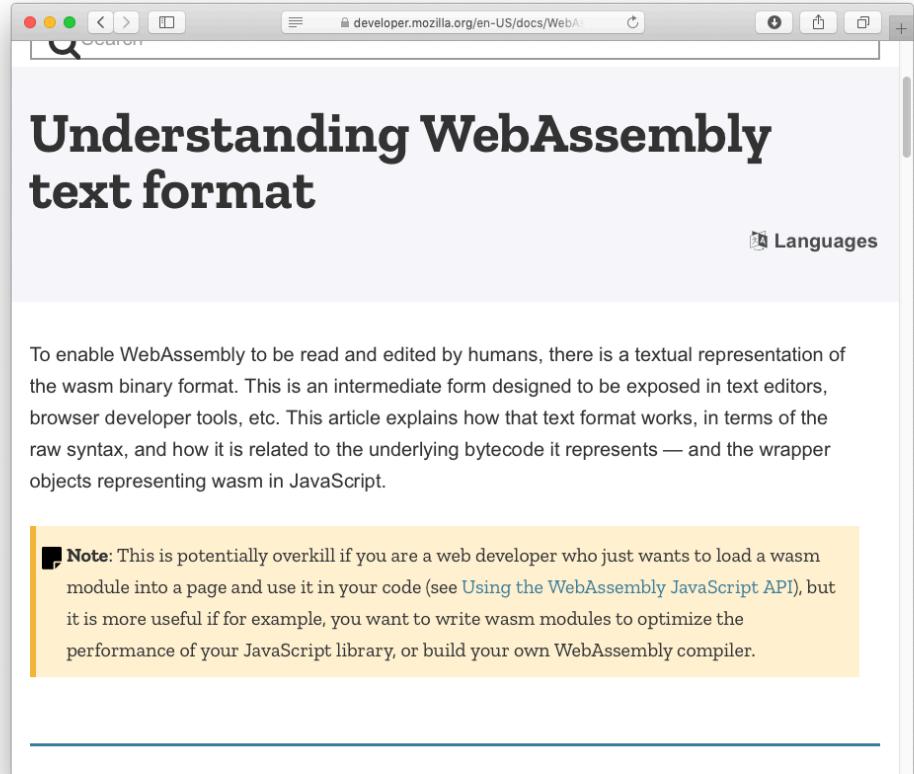


<https://rsms.me/wasm-intro>

S-Expressions

= "Text Format" (wat)

```
(module (memory 1) (func))  
  
(module  
  (table 0 anyfunc)  
  (memory $0 1)  
  (export "memory" (memory $0))  
  (export "main" (func $main))  
  (func $main (; 0 ;) (result i32)  
    (i32.const 42)  
  )  
)
```



The screenshot shows a web browser window displaying the Mozilla developer documentation titled "Understanding WebAssembly text format". The URL in the address bar is https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format. The page content explains the textual representation of the wasm binary format, designed for humans to read and edit. It includes a note about its use in web development and optimizing JavaScript libraries.

Understanding WebAssembly text format

To enable WebAssembly to be read and edited by humans, there is a textual representation of the wasm binary format. This is an intermediate form designed to be exposed in text editors, browser developer tools, etc. This article explains how that text format works, in terms of the raw syntax, and how it is related to the underlying bytecode it represents — and the wrapper objects representing wasm in JavaScript.

Note: This is potentially overkill if you are a web developer who just wants to load a wasm module into a page and use it in your code (see [Using the WebAssembly JavaScript API](#)), but it is more useful if for example, you want to write wasm modules to optimize the performance of your JavaScript library, or build your own WebAssembly compiler.

Try it out...

C-Code => S- Expression (*.wat)

The screenshot shows the WasmFiddle interface. On the left, there's a code editor with C code for a "Hello World" application. On the right, there's a terminal-like window showing the generated WebAssembly binary and the corresponding S-expression output.

```

WA https://wasdk.github.io/WasmFiddle/?wvzhb
Build Run JS
C
1 char * helloO {
2     return "Hello World";
3 }

1. function utf8ToString(h, p) {
2     let s = '';
3     for (let i = p; h[i]; i++) {
4         s += String.fromCharCode(h[i]);
5     }
6     return s;
7 }

8

9 let m = new WebAssembly.Instance(new
WebAssembly.Module(buffer));
10 let h = m.exports.hello;
11 let p = m.exports.hello;
12 logUtf8ToString(h, p);
13

```

```

#include <stdio.h>
#include <sys/uio.h>

#define WASM_EXPORT __attribute__((visibility("default")))

WASM_EXPORT
int main(void) {
    printf("Hello World\n");
}
...

```

```

(module
  (table 0 anyfunc)
  (env)
  (data (i32.const 16) "Hello World\00")
  (export "memory" (memory 30))
  (export "hello" (func $hello))
  (func $hello (i32.const 16)
    (i32.const 16)
  )
)

```

Try it on

[https://wasdk.github.io/
WasmFiddle/](https://wasdk.github.io/WasmFiddle/)

```

(module
  (type $t0 (func (param i32 i32 i32) (result i32)))
  (type $t1 (func (param i32)))
  (type $t2 (func (param i32 i32 i32 i32) (result i32)))
  (type $t3 (func (param i32 i32) (result i32)))
  (type $t4 (func (param i32 i32 i32 i32 i32 i32) (result i32)))
  (type $t5 (func))
  (type $t6 (func (result i32)))
  (type $t7 (func (param i32) (result i32)))
  (type $t8 (func (param i32 i64 i32) (result i64)))
  (import "env" "putc_js" (func $putc_js (type $t1)))
  (import "env" "__syscall3" (func $__syscall3 (type $t2)))
  (import "env" "__syscall1" (func $__syscall1 (type $t3)))
  (import "env" "__syscall5" (func $__syscall5 (type $t4)))
  (func $__wasm_call_ctors (type $t5))
  (func $main (export "main") (type $t6) (result i32)
    i32.const 1024
    call $puts
    drop
    i32.const 0
    (func $writev_c (export "writev_c") (type $t0) (param $p0 i32)
      (param $p1 i32) (param $p2 i32) (result i32)
      (local $l0 i32) (local $l1 i32) (local $l2 i32) (local $l3 i32) (local
      $l4 i32) (local $l5 i32)
      block $B0
      get_local $p2.....
    )
  )
)

```

Minimal Example JS+wasm

1) provide wasm

2) provide HTML/JS

3) run / inspect
in browser

Git clone: <https://github.com/mdn/webassembly-examples>

1a

```
cat simple.wat
(module
  (func $i (import "imports" "imported_func") (param i32))
  (func (export "exported_func")
    i32.const 42
    call $i))
```

1b

```
xxd simple.wasm
00000000: 0061 736d 0100 0000 0108 0260 017f 0060 .asm.....`...
00000010: 0000 0219 0107 696d 706f 7274 730d 696d .....imports.im
00000020: 706f 7274 6564 5f66 756e 6300 0003 0201 ported_func.....
00000030: 0107 1101 0d65 7870 6f72 7465 645f 6675 .....exported_fu
00000040: 6e63 0001 0a08 0106 0041 2a10 000b      nc.....A*...
```

Minimal Example JS+wasm

1) provide wasm

2) provide HTML/JS which
loads wasm
executes exported function

3) run / inspect in browser

2a

3a

3b

```
E html > E body > E script > T " var importObject = { ..."
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>WASM test</title>
  </head>
  <body>
    <script>
      var importObject = {
        imports: {
          imported_func: function(arg) {
            console.log(arg);
          }
        }
      };

      fetch('simple.wasm').then(response =>
        response.arrayBuffer()
      ).then(bytes =>
        WebAssembly.instantiate(bytes, importObject)
      ).then(result =>
        result.instance.exports.exported_func()
      );
    </script>
  </body>
</html>
```

ruby -run -ehttpd . -p8000



See also: https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API

Optional: Use libraries, frameworks, ...

For example "wasm-bindgen":
wasm modules and Rust

<https://github.com/rustwasm/wasm-bindgen>

For example "stdweb":

"...Expose a full suite of Web APIs as exposed by web browsers...."

<https://github.com/koute/stdweb>

Call JavaScript from WebAssembly

The screenshot shows a web browser window with the URL callahad.github.io/tccc20-wasm/slides/#/24. The page title is "WebAssembly can call JavaScript too!". Below the title is a code snippet in C:

```
// main.c
extern DLL_IMPORT void printInt(int);
int main() {
    printInt(42);
}
```

Text below the code explains imports:

where JS functions we want to call from `wasm` are *imported*:

```
// However DLL/DSO imports are defined in your compiler
#define DLL_IMPORT __attribute__ ((visibility ("default")))
```

Text below the imports explains compilation:

then compile to `wasm`:

```
clang -mwasm main.c -o main.wasm
```

At the bottom right of the slide is the handle `@callahad`.

[http://callahad.github.io/tccc20-wasm/
slides/#/24](http://callahad.github.io/tccc20-wasm/slides/#/24)

WebAssembly

– Reverse Engineering

[https://www.pnfsoftware.com/
reversing-wasm.pdf](https://www.pnfsoftware.com/reversing-wasm.pdf)

The screenshot shows a web browser window with the URL www.pnfsoftware.com/reversing-wasm.pdf in the address bar. The page title is "Reverse Engineering WebAssembly". Below it, author information and revision details are listed. The main content is an introduction to WebAssembly for reverse-engineers, followed by a detailed table of contents:

Introduction	2
WebAssembly Modules	3
Sections	3
Indexed Spaces	3
Primitive types	4
WebAssembly Instructions	4
Operator categories	5
Control flow	6
Step-by-step example	8
S-Expressions	9
Accessing the memory	10
WebAssembly Implementation Details	10
Implicit Memory Initialization	12
Implicit Table Initialization	12
Explicit Module Initialization	12
Stack pointer initialization	13
Function pointer table initialization	14
Indirect Function Invocation	15
Local variables and local buffers	17
System calls	18

WASM Tools

Online

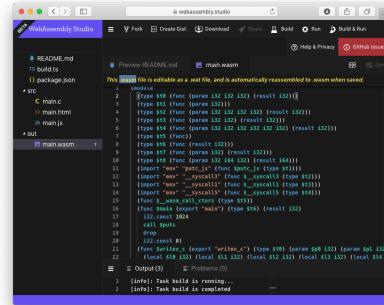
e.g. WebAssembly Studio

Command Line

e.g.: wasm decompiler to C

Toolsets

e.g.: binary toolkit



<https://webassembly.studio>

Simple Example

wasmdec will translate this WebAssembly binary:

```
(module
  (func $addTwo (param i32 i32) (result i32)
    (return
      i32.add (get_local 0) (get_local 1)
    )
  )
  (export "addTwo" $addTwo)
)
```

To the following pseudo-C code:

```
int fn_addTwo(int argc, int arg1) {
    return argc + arg1;
}
```

<https://github.com/wwwg/wasmdec>



<https://wwwg.github.io/web-wasmdec/>

WABT: The WebAssembly Binary Toolkit

WABT (we pronounce it "wabbit") is a suite of tools for WebAssembly, including:

- **wat2wasm:** translate from [WebAssembly text format](#) to the [WebAssembly binary format](#)
- **wasm2wat:** the inverse of wat2wasm, translate from the binary format back to the text format (also known as a .wat)
- **wasm-objdump:** print information about a wasm binary. Similar to objdump.
- **wasm-interp:** decode and run a WebAssembly binary file using a stack-based interpreter
- **wat-desugar:** parse wat text form as supported by the spec interpreter (s-expressions, flat syntax, or mixed) and print "canonical" flat format
- **wasm2c:** convert a WebAssembly binary file to a C source and header
- **wasm-strip:** remove sections of a WebAssembly binary file
- **wasm-validate:** validate a file in the WebAssembly binary format
- **wat2json:** convert a file in the wasm spec test format to a JSON file and associated wasm binary files
- **wasm-opcodecnt:** count opcode usage for instructions
- **spectest-interp:** read a Spectest JSON file, and run its tests in the interpreter

<https://github.com/WebAssembly/wabt>

Outlook

– Research Directions

For example, compile on server....

WATT : A novel web-based toolkit to generate WebAssembly-based libraries and applications

2018 IEEE International Conference on Consumer Electronics (ICCE)

WATT : a novel web-based toolkit to generate WebAssembly-based libraries and applications

Hunseop Jeong, Jinwoo Jeong, Sangyong Park and Kwanghyuk Kim
Samsung Electronics, Suwon, South Korea
{hs85.jeong, jw00.jeong, sy302.park, hyuki.kim}@samsung.com

Abstract—We present the WebAssembly Translation Toolkit (WATT), a novel WebAssembly (WASM) application authoring tool that allows developers to create a WASM-based library easily and to write WASM-based applications using the WASM library. Since WATT is a web server solution, developers can write web applications without installing any additional tools, and they can instantly run applications on a browser on a CE device or PC. Because WATT is designed to support automatic JavaScript API generation, many native libraries can be converted to WASM libraries without any knowledge of the compiler or the Emscripten tool.

I. INTRODUCTION

The performance of JavaScript is known to be the biggest factor that determines the performance of web contents [1]. WebAssembly (WASM) is the state-of-the-art technology to improve the performance of JavaScript [2]. WASM handles JavaScript in a binary format that is similar to assembly language. It improves the computation performance of JavaScript since it reduces code size and simplifies both parsing and compilation. Additionally, it reduces the impact of garbage collection (GC) by using programmed memory handling. The Emscripten tool, developed by Mozilla, enables the conversion of native code to web code [3]. Recently, WASM technology has been applied to this tool as well. Therefore, Emscripten is commonly used to facilitate the conversion of existing native applications to WASM-based applications. Unity and Unreal SDK are popular tools that use Emscripten to create web applications by converting native code in their own respective formats to JavaScript-based code. High-performance game applications can be converted to web applications by using these tools [4], [5]. However, existing Emscripten-based tools are focused on application conversion

basic model for converting native-based code to JavaScript-based code, as shown in Fig. 1. The Emscripten tool is typically used to convert the native code to JavaScript code. While Unity translates the scripting language into intermediate code and converts it into C++ code, Unity and the Unreal SDK follow the same conversion model when they export web applications by default. As examples for the publishing of JavaScript libraries using Emscripten, several projects convert the main native library to a JavaScript library manually. ammo.js is a JavaScript 3D physics engine library that converts bullet [7], and box2d.js is converted from the Box2D native library [8]. In general, the approaches applied for these libraries require knowledge about the compiler and the Emscripten tool; therefore, it is very cumbersome in most cases.



Fig. 1. JavaScript conversion model using Emscripten

III. WATT ARCHITECTURE

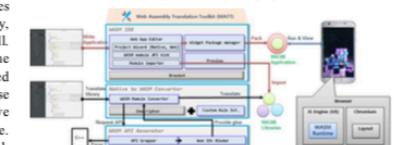


Fig. 2. Architecture view of WATT

Web Assembly (WASM)

Specifications

<https://webassembly.github.io/spec/core/bikeshed/index.html>



Core

JS-Embedding
defines JS objects + methods
validate, compile, instantiate
import/export JS-objects
js-api

<https://webassembly.github.io/spec/js-api/index.html>

WebAssembly JavaScript Interface
Editor's Draft, 25 May 2020



This version:
<https://webassembly.github.io/spec/js-api/>

<https://webassembly.github.io/spec/web-api/index.html>

WebAssembly Web API
Editor's Draft, 25 May 2020



This version:
<https://webassembly.github.io/spec/web-api/>

Web Embedding:
extensions to JS API
web-api

Web Assembly (WASM)

Core Spec

Goals, Conventions

Modules, Types

Execution

Binary Format

<https://www.w3.org/TR/wasm-core-1/>

<https://webassembly.github.io/spec/core/bikeshed/index.html>

§ 2.2.4. Names

Names are sequences of *characters*, which are *scalar values* as defined by [\[UNICODE\]](#) (Section 2.4).

```
name ::= char*           (if |utf8(char*)| < 232)
char ::= U+00 | ... | U+D7FF | U+E000 | ... | U+10FFFF
```

Due to the limitations of the [binary format](#), the length of a name is bounded by the length of its [UTF-8](#) encoding.

§ 2.4.5. Control Instructions

Instructions in this group affect the flow of control.

```
instr ::= ...
| nop
| unreachable
| block resulttype instr* end
| loop resulttype instr* end
| if resulttype instr* else instr* end
| br labelidx
| br_if labelidx
| br_table vec(labelidx) labelidx
| return
| call funcidx
| call_indirect typeidx
```

The `nop` instruction does nothing.

§ 2.5. Modules

WebAssembly programs are organized into *modules*, which are the unit of deployment, loading, and compilation. A module collects definitions for [types](#), [functions](#), [tables](#), [memories](#), and [globals](#). In addition, it can declare [imports](#) and [exports](#) and provide initialization logic in the form of [data](#) and [element](#) segments or a [start](#) function.

```
module ::= { types vec(functype),
            funcs vec(func),
            tables vec(table),
            mems vec(mem),
            globals vec(global),
            elem vec(elem),
            data vec(data),
            start start7,
            imports vec(import),
            exports vec(export) }
```

Each of the vectors – and thus the entire module – may be empty.

Web Assembly (WASM)

JS Embedding

WebAssembly Store

WebAssembly JS Object
Caches

Namespaces, exported
functions, error objects, ...

For example:
memory object cache
table object cache
exported function cache
global object cache

`demo.wat` (encoded to `demo.wasm`):

```
(module
  (import "js" "import1" (func $i1))
  (import "js" "import2" (func $i2))
  (func $main (call $i1))
  (start $main)
  (func (export "f") (call $i2))
)
```

JavaScript, run in a browser

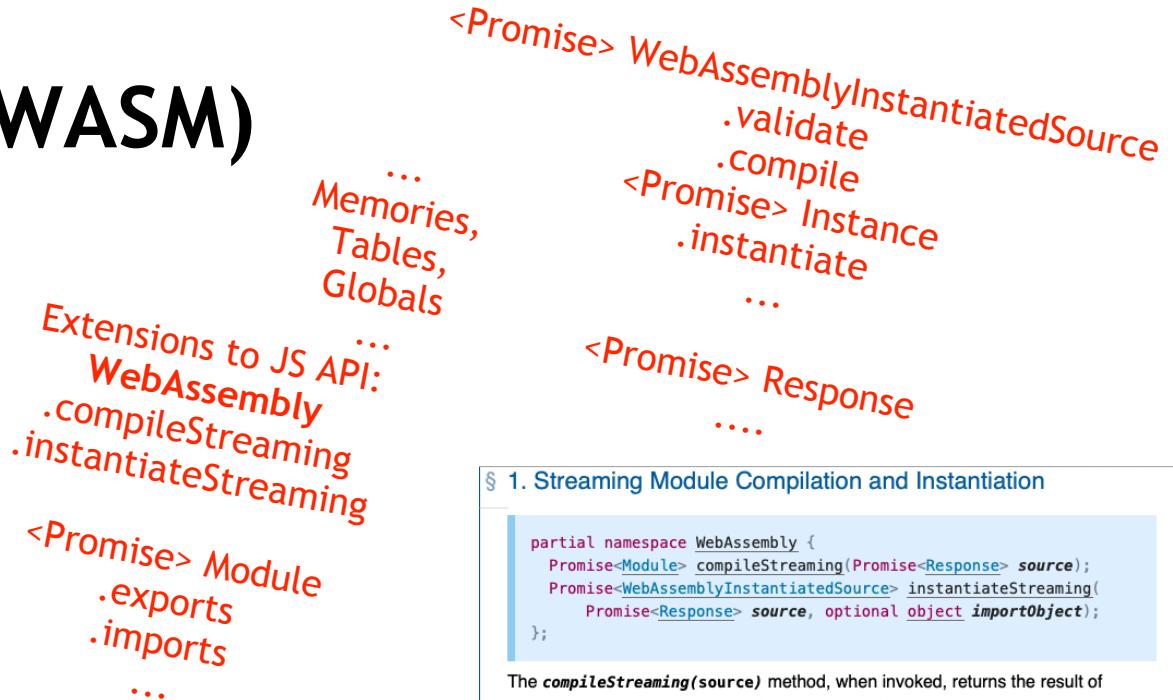
```
var importObj = {js: {
  import1: () => console.log("hello,"),
  import2: () => console.log("world!")
}};
fetch('demo.wasm').then(response =>
  response.arrayBuffer()
).then(buffer =>
  WebAssembly.instantiate(buffer, importObj)
).then(({module, instance}) =>
  instance.exports.f()
);
```

<https://www.w3.org/TR/wasm-js-api-1/>

<https://webassembly.github.io/spec/js-api/index.html>

Web Assembly (WASM)

Web Embedding
 Streaming module
 + compile
 + instantiate
 Serialisation



<https://www.w3.org/TR/wasm-web-api-1/>

<https://webassembly.github.io/spec/web-api/index.html>

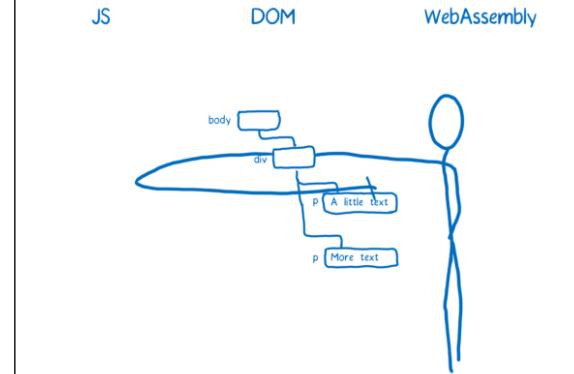
Web Assembly Future

Work-in-progress (WIP):
threads, source maps,
memory inspection,
garbage collection

Working directly with the DOM

Currently, there's no way to interact with the DOM. This means you can't do something like `element.innerHTML` to update a node from WebAssembly.

Instead, you have to go through JS to set the value. This can mean passing a value back to the JavaScript caller. On the other hand, it can mean calling a JavaScript function from within WebAssembly—both JavaScript and WebAssembly functions can be used as imports in a WebAssembly module.



[https://hacks.mozilla.org/2017/02/
where-is-webassembly-now-and-
whats-next/](https://hacks.mozilla.org/2017/02/where-is-webassembly-now-and-whats-next/)

WASM & Security

– very good, but not perfect

The Problems and Promise of WebAssembly

Posted by Natalie Silvanovich, Project Zero

WebAssembly is a format that allows code written in assembly-like instructions to be run from JavaScript. It has recently been implemented in all four major browsers. We reviewed each browser's WebAssembly implementation and found three vulnerabilities. This blog post gives an overview of the features and attack surface of WebAssembly, as well as the vulnerabilities we found.

[https://
googleprojectzero.blogspot.com/
2018/08/the-problems-and-promise-
of-webassembly.html](https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html)

WebAssembly Changes Could Ruin Meltdown and Spectre Browser Patches

Tara Seals • June 27, 2018 2:26 pm

NCC Group Whitepaper

Security Chasms of WASM

August 3, 2018 – Version 1.0

Prepared by

Brian McFadden
Tyler Lukasiewicz
Jeff Dileo
Justin Engler

Abstract

WebAssembly is a new technology that allows web developers to run native C/C++ on a webpage with near-native performance. This paper provides a basic introduction to WebAssembly and examines the security risks that a developer may take on by using it. We cover several examples exploring the theoretical security implications of WebAssembly. We also cover Emscripten, which is currently the most popular WebAssembly compiler toolchain. Our assessment of Emscripten includes its implementation of compiler-and-linker-level exploit mitigations as well as the internal hardening of its libc implementation, and how its augmentation of WASM introduces new attack vectors and methods of exploitation. We also provide examples of memory corruption exploits in the Wasm environment. Under certain circumstances, these exploits could lead to hijacking control flow or even executing arbitrary JavaScript within the context of the web page. Finally, we provide a basic outline of best practices and security considerations for developers wishing to integrate WebAssembly into their product.

[https://i.blackhat.com/us-18/Thu-August-9/
us-18-Lukasiewicz-WebAssembly-A-New-World-
of-Native_Exploits-On-The-Web-wp.pdf](https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf)

[https://threatpost.com/webassembly-changes-could-ruin-
meltdown-and-spectre-browser-patches/133657/](https://threatpost.com/webassembly-changes-could-ruin-meltdown-and-spectre-browser-patches/133657/)

*Malware, Scams,
Keyloggers, ...*

The dark side of WebAssembly

Aishwarya Lonkar & Siddhesh Chandrayan

Symantec, India

Copyright © 2018 Virus Bulletin

Abstract

The WebAssembly (Wasm) format is a way to run code, compiled in native languages such as C/C++, on web browsers. WebAssembly has better performance when running native code than other variations of compiled JavaScript such as asm.js (Assembly JS). WebAssembly is often used in developing web games. Recent versions of all popular browsers including Chrome, Firefox and Microsoft Edge support WebAssembly execution.

Though Wasm has been around for a few years, it rose to prominence more recently when it was used for cryptocurrency mining in browsers. This opened a Pandora's box of potential malicious uses of Wasm.

In this paper we will walk through some of the instances in which Wasm can be used maliciously, such as:

[https://www.virusbulletin.com/
virusbulletin/2018/10/dark-side-
webassembly/](https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly/)

WASM Summary

- (+) efficient (compact, binary), fast (streaming, pre-compiled assembly), secure (typesafe, memory safe), many supported source languages, (mobile) browser support
- (-) Lacks JS APIs (integration), no garbage collector (no support for dynamic languages), low level software engineering toolchain/workflow (debugging), multithreading

<https://cacm.acm.org/magazines/2018/12/232881-bringing-the-web-up-to-speed-with-webassembly/fulltext>