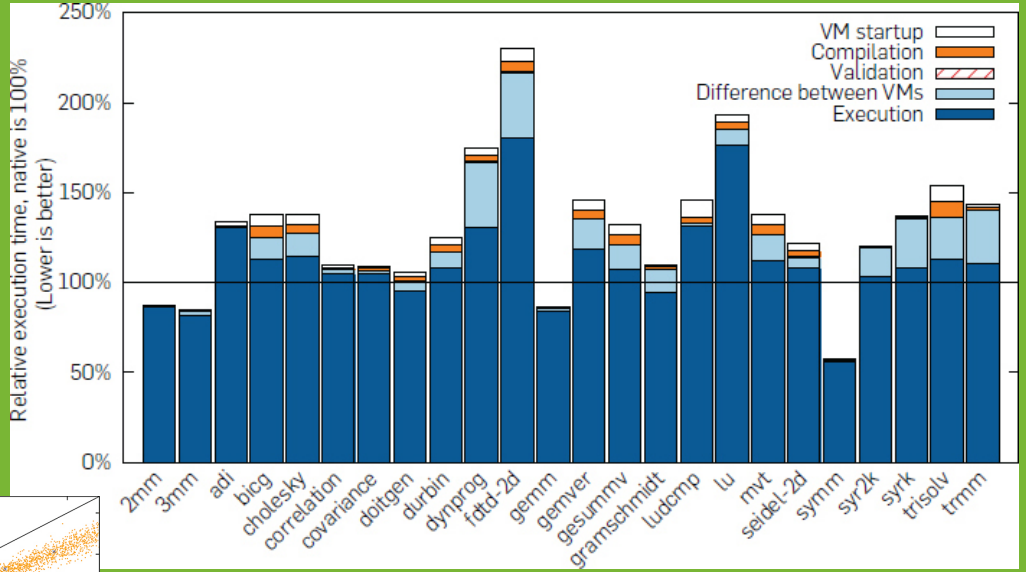
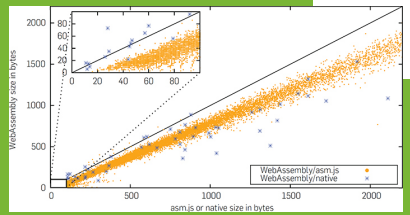
# WASM

## 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

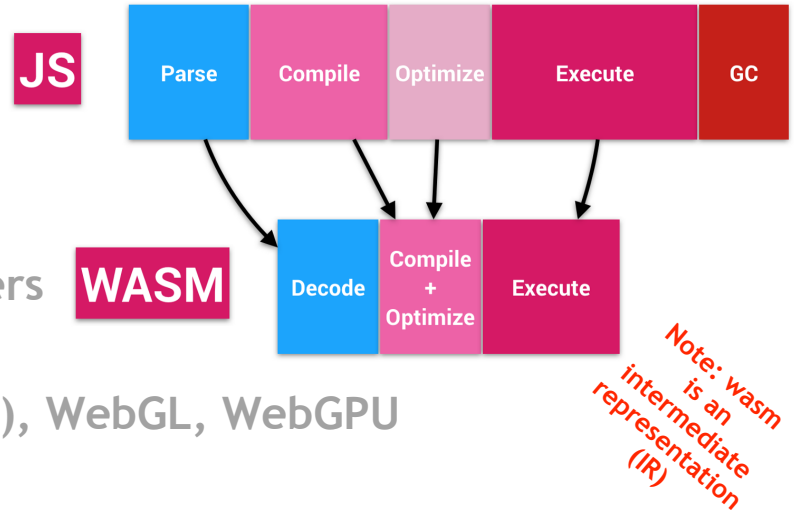
↯↯ Plugins ↯↯ Flash, Java Applets

Interpreter

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

JIT

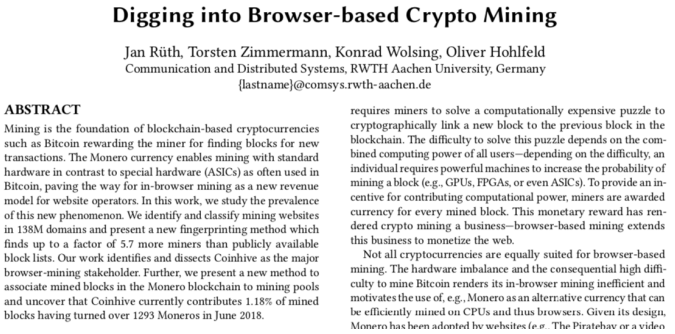
Compiler↯ High Level language -> IR (optimisation, LLVM) -> Machine Code




https://hackernoon.com/webassembly-the-journey-what-is-wasm-caf9871108aa

# WebAssembly
## — Selected Applications

## Browser-based Crypto Mining



Digging into Browser-based Crypto Mining

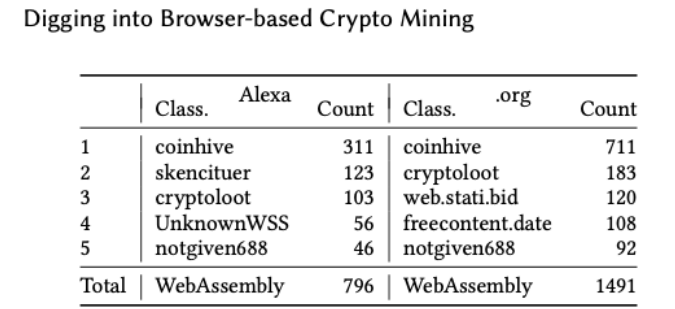
Jan Rüth, Torsten Zimmermann, Konrad Wolsing, Oliver Hohlfeld
Communication and Distributed Systems, RWTH Aachen University, Germany
{lastname}@comsys.rwth-aachen.de

**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 5.7 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 Piratebay 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

Find extensive link collection and more on "Awesome WASM"

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

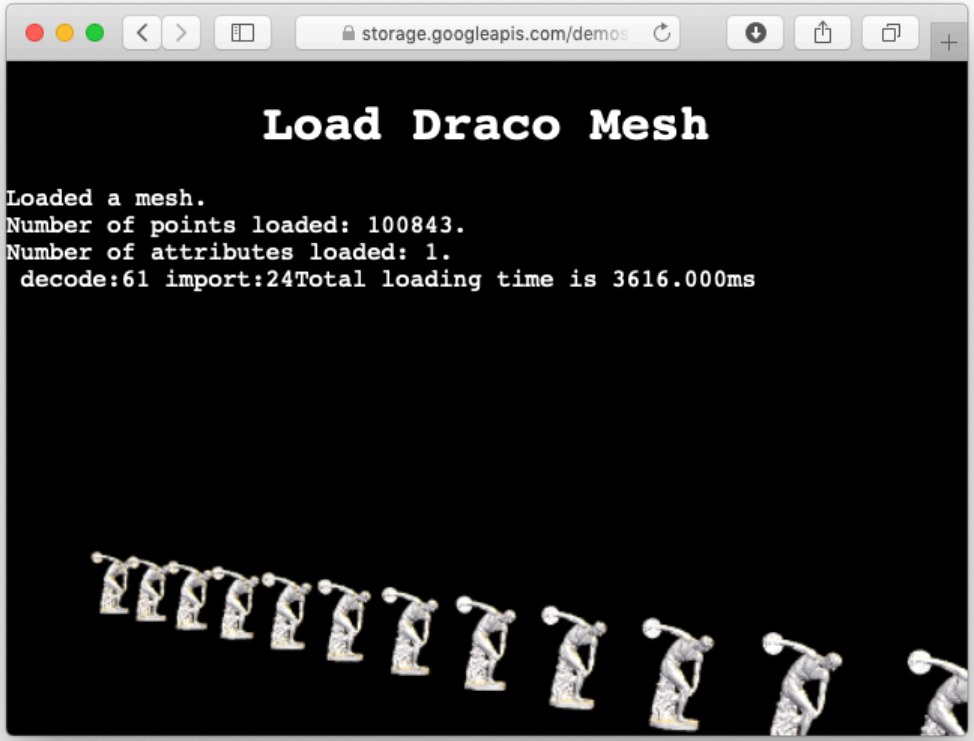
# Web Assembly

Examples:

Google "draco" library: compress 3D point clouds

https://github.com/google/draco

Using three.js for rendering...




https://storage.googleapis.com/demos.webmproject.org/draco/draco_loader_throw.html

FH | JOANNEUM
University of Applied Sciences

# Review JavaScript

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

John Feiner

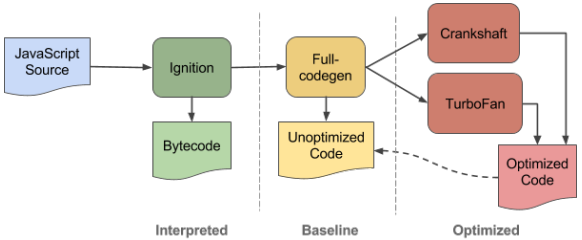
# About JavaScript Engines

**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

https://medium.freecodecamp.org/javascript-essentials-why-you-should-know-how-the-engine-works-c2cc0d321553

view minified sources

# *JavaScript is Assembly Language of the Web*

*Scott Hanselman 2011,*
*Principal Program Manager Lead at Microsoft*

https://www.hanselman.com/blog/JavaScriptIsAssemblyLanguageForTheWebSematicMarkupIsDeadCleanVsMachinecodedHTML.aspx

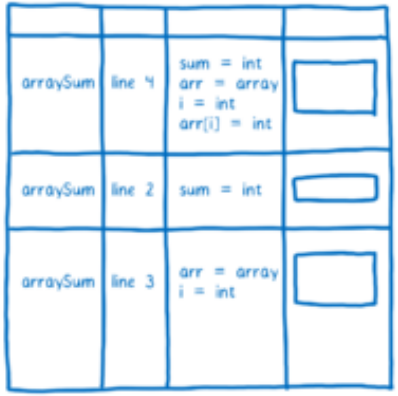
# Review -- JIT

monitor code:
- cold
- warm
- hot

just-in-time:
- compile
- optimise
  (and deoptimise)

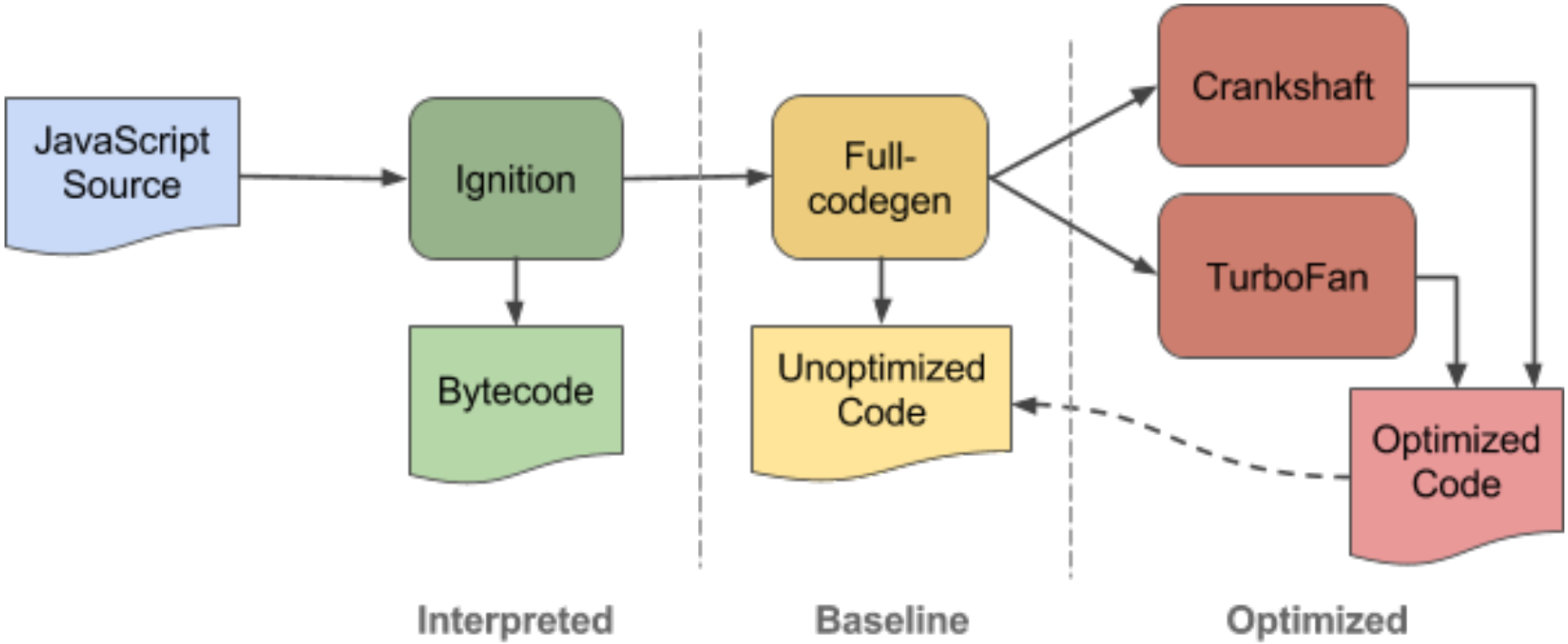
```
function arraySum(arr) {
  var sum = 0;
  for (var i = 0; i < arr.length; i++) {
    sum += arr[i];
  }
}
```

↯dynamic types
polymorphic code↯



https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/




https://www.zeolearn.com/magazine/the-engines-explore-javascript-engines

# 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");
```

https://medium.freecodecamp.org/javascript-essentials-why-you-should-know-how-the-engine-works-c2cc0d321553

# JavaScript Engines are fast, but possibly slow

## Explain inline caching

### property lookup

*monomorphic*

*polymorphic*

***megamorphic***

↯ slow

```
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");
```

*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*

https://medium.freecodecamp.org/javascript-essentials-why-you-should-know-how-the-engine-works-c2cc0d321553

# JavaScript Engines faster with TypeScript

Explain inline caching

property lookup

*monomorphic*

=> use JS classes,

use TypeScript

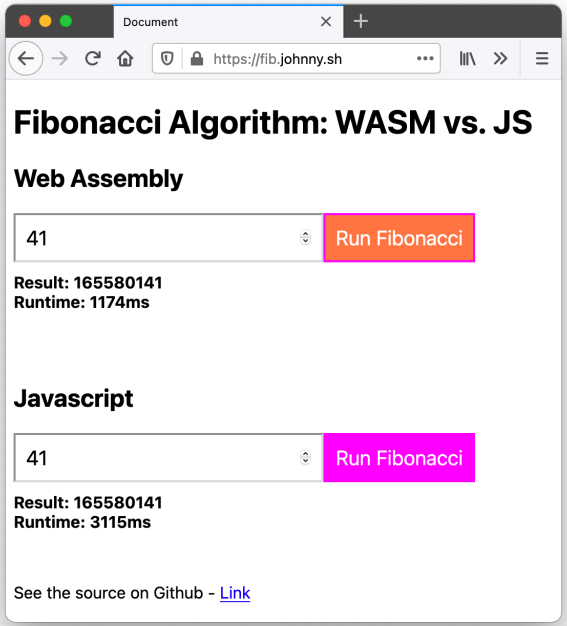
```
const yoda = new Person({
  lastname: 'Yoda' });
...
const people = [
    yoda,...
```

✓ 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");
```

https://medium.freecodecamp.org/javascript-essentials-why-you-should-know-how-the-engine-works-c2cc0d321553

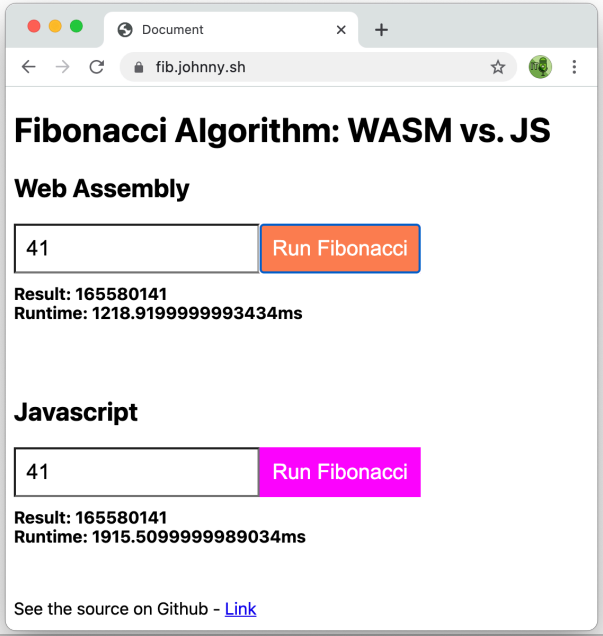
# JS fast/slow




Demo

← Firefox

Chrome →




https://fib.johnny.sh/

# Review — Another Alternative to JS

**wasm.js**

John Feiner

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

# What is asm.js?

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

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

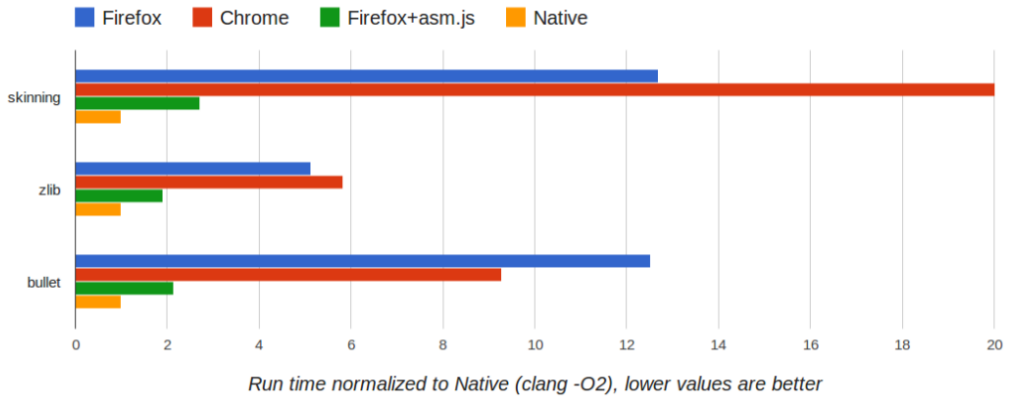
i.e. write in a statically-typed language with manual memory management use Emscripten (input LLVM Bytecode) for **source-to-source compilation**, ahead-of-time optimisation, "virtually no garbage collection", no dynamic behaviour

optionally: highly efficient native code

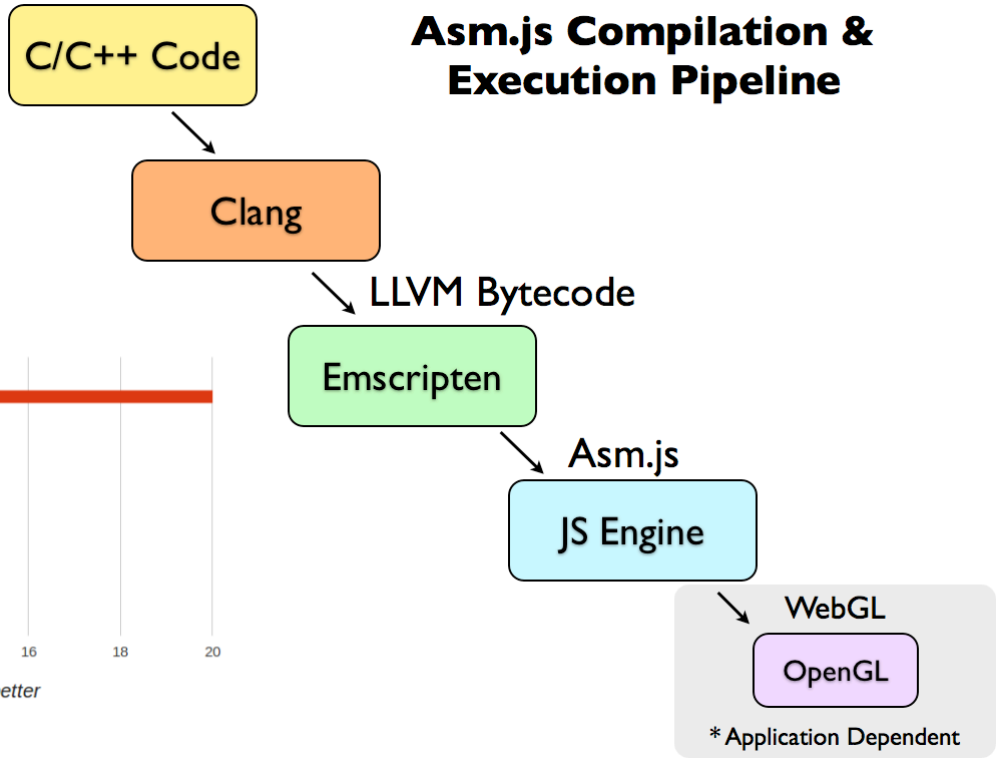
=> Runs in any browser, but runs in fast in Firefox (OdinMonkey with AoT-compilation)

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

# Compare: asm.js




*Run time normalized to Native (clang -O2), lower values are better*




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

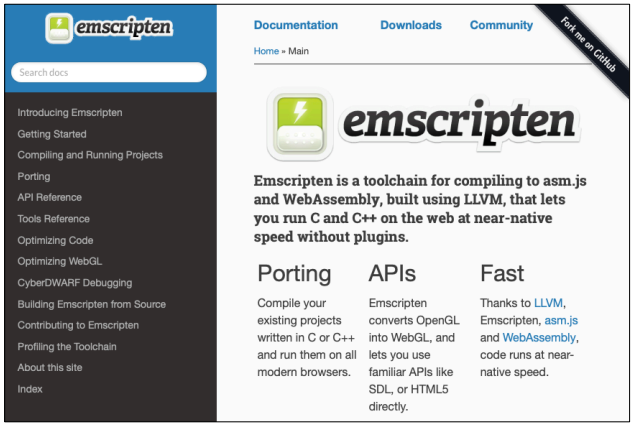
# Example Speed Comparison: JS vs. asm.js

*"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!"*

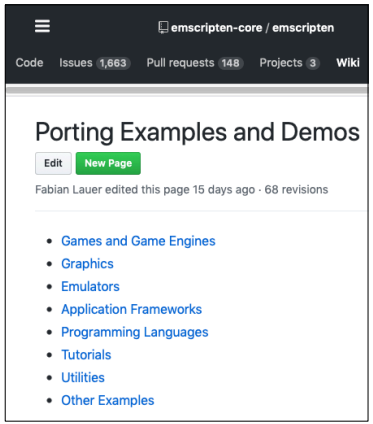
*(Ian Enderling)*

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

# asm.js with "emscripten" Compiler




https://emscripten.org




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!

real0m0.074s
user0m0.090s
sys 0m0.020s
```

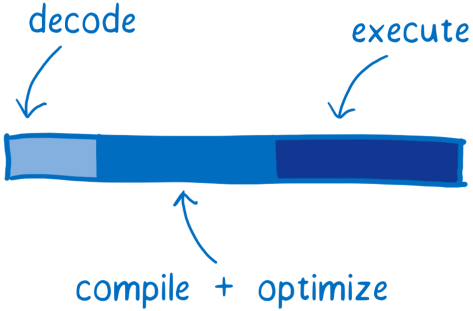
https://emscripten.org/docs/getting_started/Tutorial.html

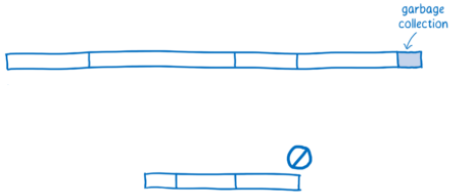
# Full Speed = WASM

**Web Assembly**

John Feiner

# Why is WASM so Fast?



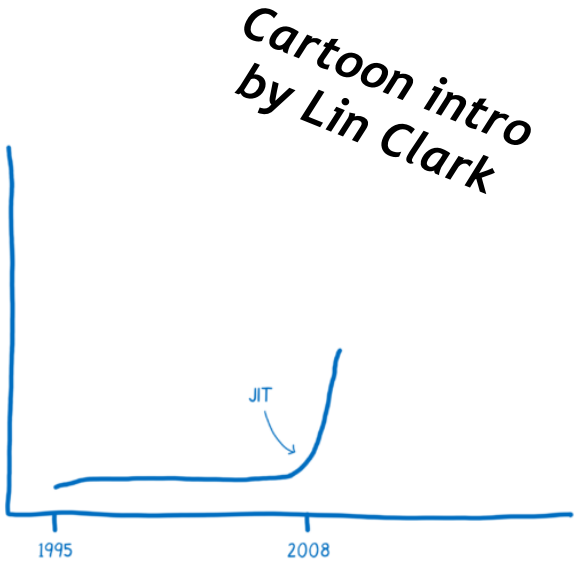

https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/

# 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://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/

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

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

# Web Assembly (WASM)


Fast:

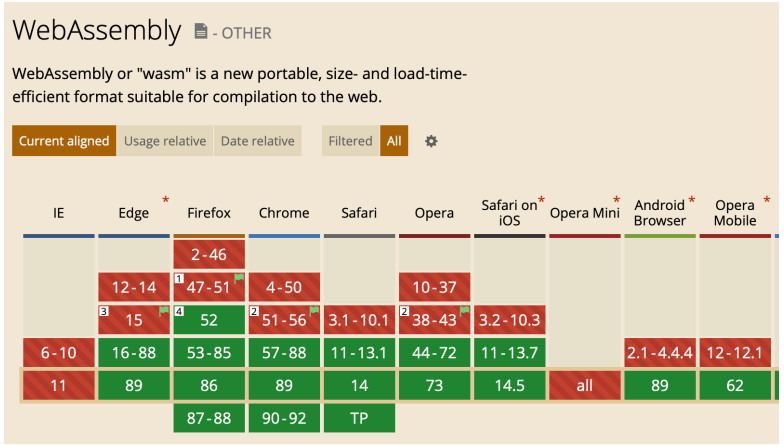
- binary format (small size, load-time)
- **native execution speed**

Safe & **secure**:

- memory-safe, sandboxed environment
- permissions and same-origin policy of browser

https://webassembly.org/docs/portability/#assumptions-for-efficient-execution

# Web Assembly (WASM)




Browser support:

ok, see "can i use"

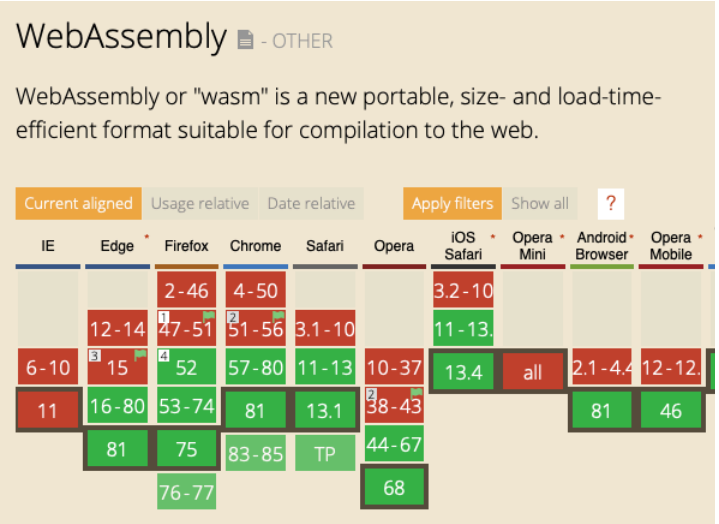
Compilation:

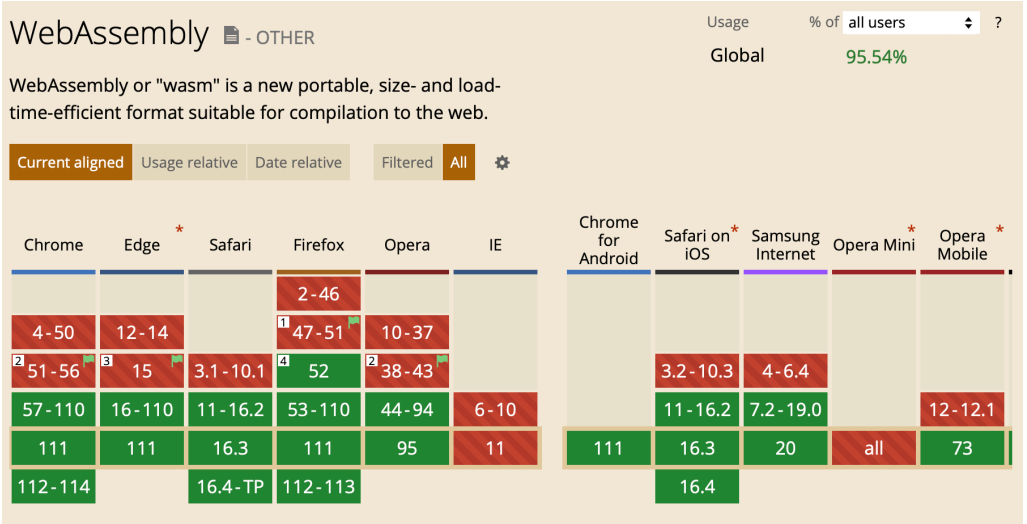
C++, ... WIP: Java, C#, Python, ...

Main supporters:

Firefox, Google, ...

https://caniuse.com/#search=web%20assembly



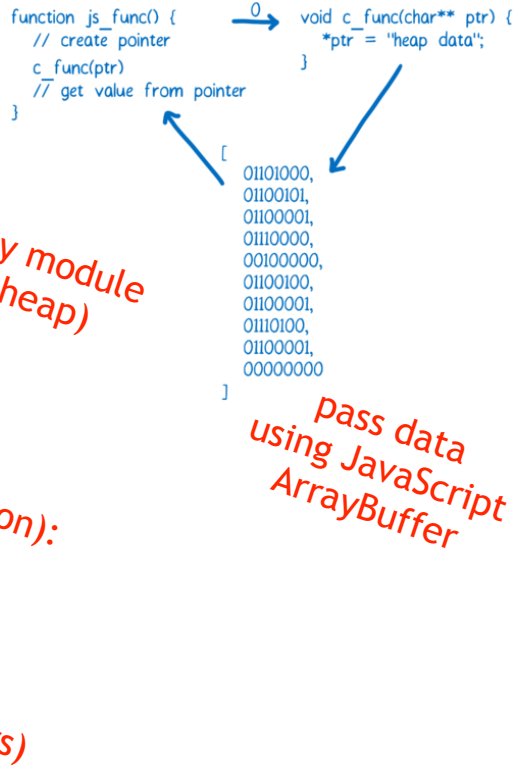




***2020-05-28***

***2023-03-17***

https://caniuse.com/#search=web%20assembly

# 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




## 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' t1*:vec(param) t2*:vec(result) ')' ⇒ [t1*] → [t2*]
param    ::= '(' 'param' id? t:valtype ')'                 ⇒ t
result   ::= '(' 'result' t:valtype ')'                    ⇒ t
```

https://webassembly.github.io/spec/core/text/types.html#result-types

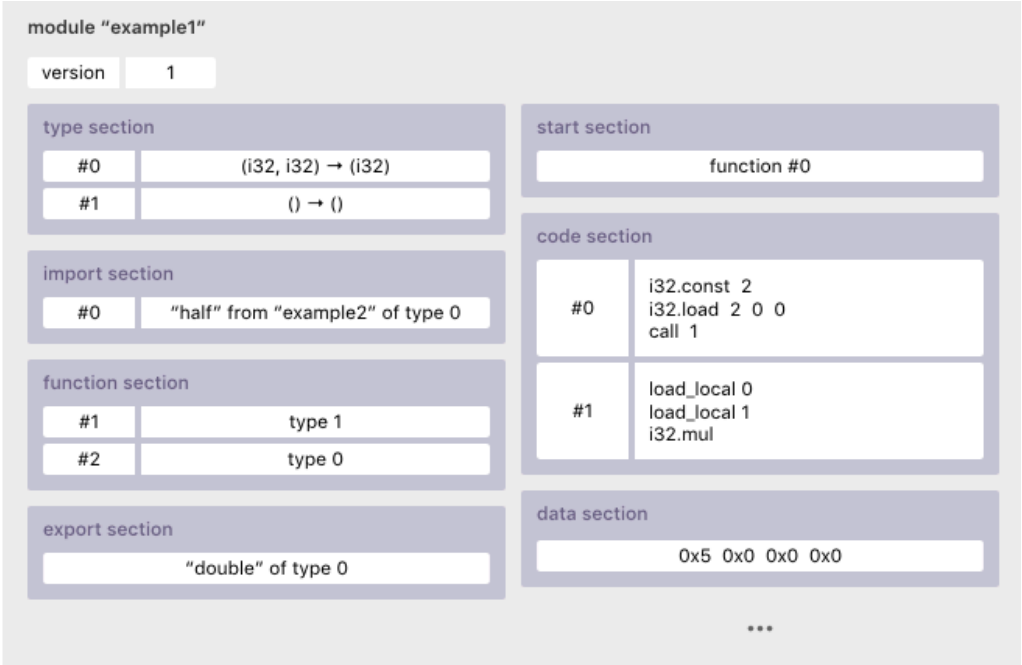
# WASM Modules

required sections:

type, function, code

optional sections:

export, import, start, global, memory, table, data, element




https://rsms.me/wasm-intro

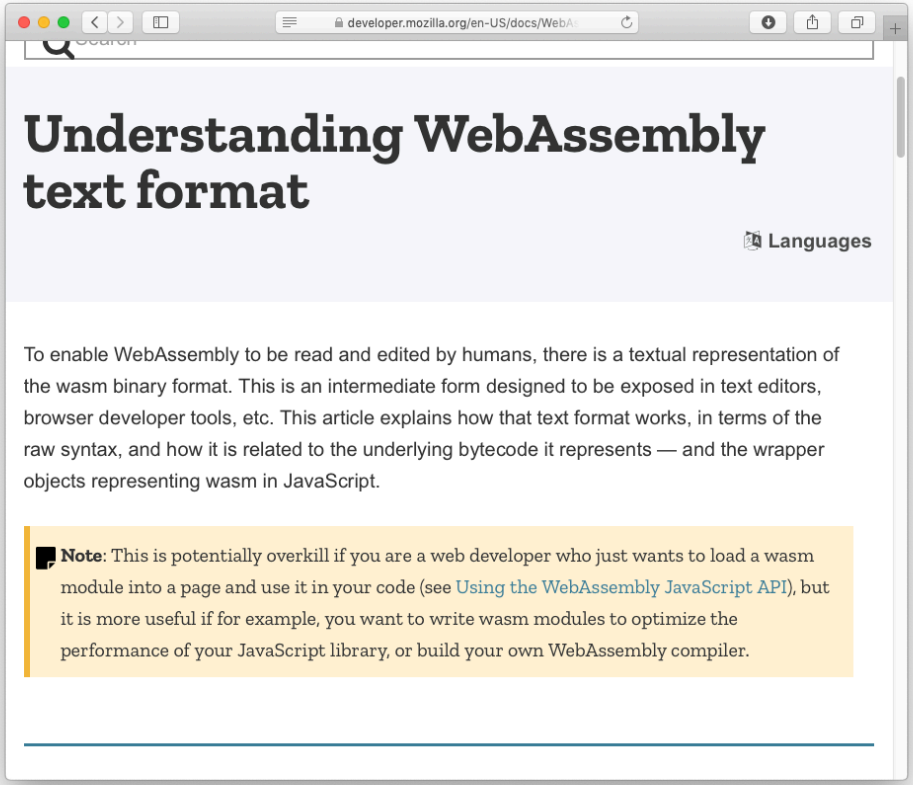
https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/

# S-Expressions

## = "Text Format" (wat)

```
(module (memory 1) (func))
```

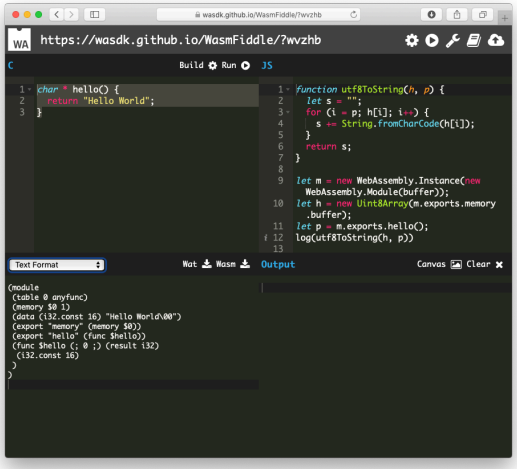
```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "main" (func $main))
 (func $main (; 0 ;) (result i32)
  (i32.const 42)
 )
)
```




https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

# Try it out...

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




Try it on
https://wasdk.github.io/WasmFiddle/

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

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

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

```
(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

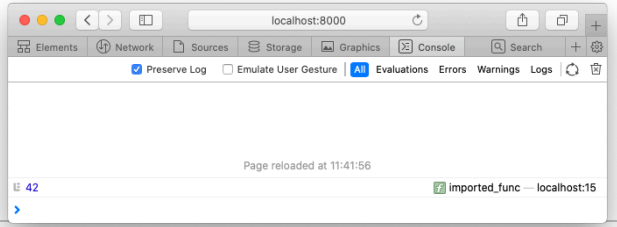
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>WASM test</title>
  </head>
  <body>
    <script>
      var importObject = {
        imports: {
          imported_func: function(arg) {
            console.log(arg);
          }
        }
      };

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

3a

```
ruby -run -ehttpd . -p8000
```

3b




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

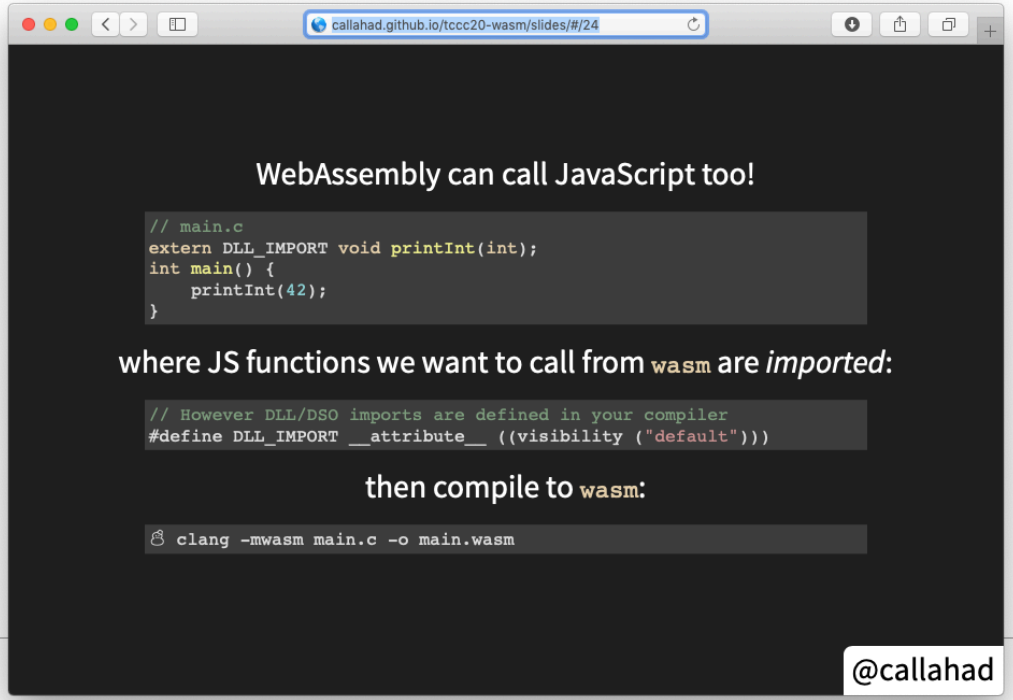
https://github.com/rustwasm/wasm-bindgen

## For example "stdweb":

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

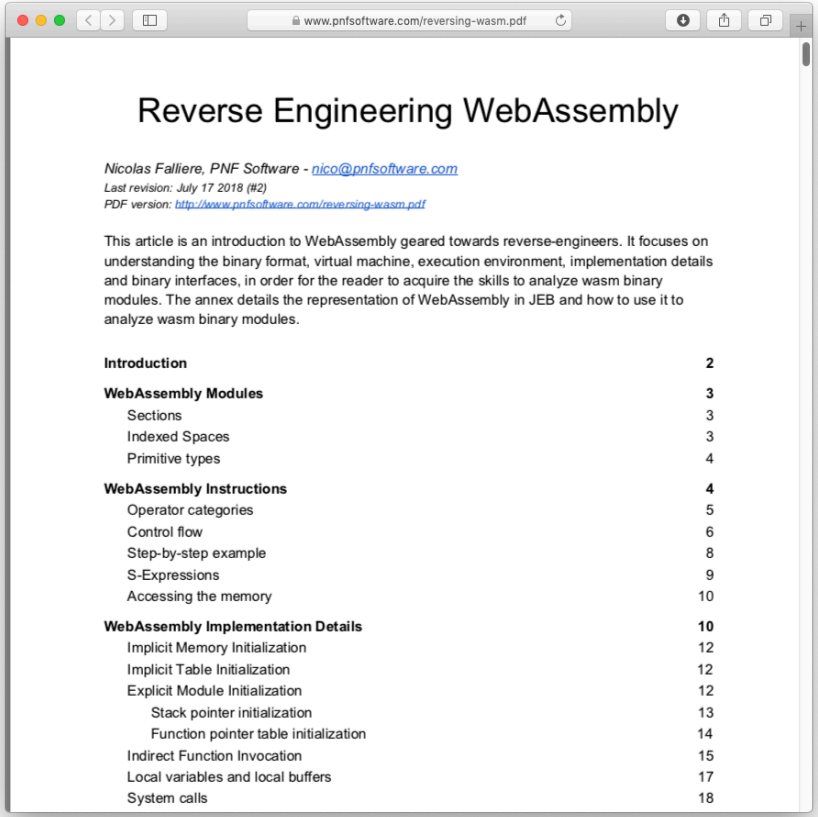
https://github.com/koute/stdweb

# Call JavaScript from WebAssembly




http://callahad.github.io/tccc20-wasm/slides/#/24

# WebAssembly — Reverse Engineering

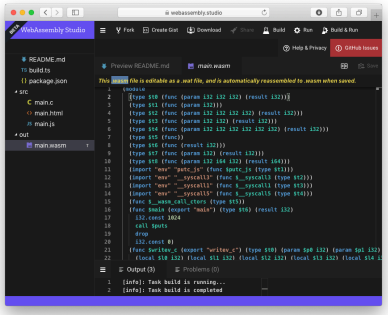



https://www.pnfsoftware.com/reversing-wasm.pdf

# WASM Tools

## Online

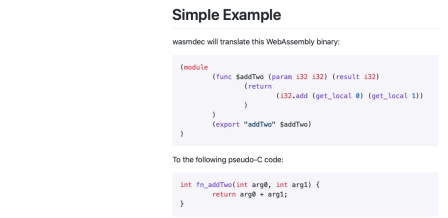
e.g. WebAssembly Studio



https://webassembly.studio

## Command Line

e.g.: wasm decompiler to C




https://github.com/wwwg/wasmdec


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

## Toolsets

e.g.: binary toolkit


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

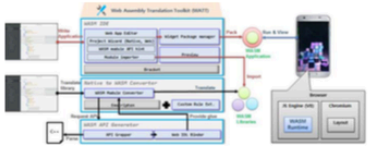


Fig. 2. Architecture view of WATT

https://doi.org/10.1109/ICCE.2018.8326230

# Web Assembly (WASM)

## Specifications

Core

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


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

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


Web Embedding:
extensions to JS API
**web-api**

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

# Web Assembly (WASM)

## Core Spec

- Goals, Conventions
- Modules, Types
- Execution
- Binary Format

### § 2.2.4. Names

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

$$\begin{array}{lcll} name & ::= & char^* & (\text{if } |\text{utf8}(char^*)| < 2^{32}) \\ char & ::= & \text{U+00} \mid \ldots \mid \text{U+D7FF} \mid \text{U+E000} \mid \ldots \mid \text{U+10FFFF} \end{array}$$

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.

$$\begin{array}{lcl} instr & ::= & \ldots \\ & \mid & \text{nop} \\ & \mid & \text{unreachable} \\ & \mid & \text{block } resulttype\ instr^*\ \text{end} \\ & \mid & \text{loop } resulttype\ instr^*\ \text{end} \\ & \mid & \text{if } resulttype\ instr^*\ \text{else } instr^*\ \text{end} \\ & \mid & \text{br } labelidx \\ & \mid & \text{br\_if } labelidx \\ & \mid & \text{br\_table } vec(labelidx)\ labelidx \\ & \mid & \text{return} \\ & \mid & \text{call } funcidx \\ & \mid & \text{call\_indirect } typeidx \end{array}$$

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.

$$\begin{array}{lcll} module & ::= & \{ & \text{types } vec(functype), \\ & & & \text{funcs } vec(func), \\ & & & \text{tables } vec(table), \\ & & & \text{mems } vec(mem), \\ & & & \text{globals } vec(global), \\ & & & \text{elem } vec(elem), \\ & & & \text{data } vec(data), \\ & & & \text{start } start^?, \\ & & & \text{imports } vec(import), \\ & & & \text{exports } vec(export) \quad \} \end{array}$$

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

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

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

# Web Assembly (WASM)

## JS Embedding

- WebAssembly Store
- WebAssembly JS Object Caches
- Namespaces, exported functions, error objects, ...

For exampple:
memory object cache
table object cache
exported funciton 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

<Promise> WebAssemblyInstantiatedSource
.validate
.compile
<Promise> Instance
.instantiate
…
<Promise> Response
….

…
Memories,
Tables,
Globals
…

Extensions to JS API:
**WebAssembly**
.compileStreaming
.instantiateStreaming

<Promise> Module
.exports
.imports
…

§ 1. Streaming Module Compilation and Instantiation

```
partial namespace WebAssembly {
  Promise<Module> compileStreaming(Promise<Response> source);
  Promise<WebAssemblyInstantiatedSource> instantiateStreaming(
      Promise<Response> source, optional object importObject);
};
```

The ***compileStreaming(source)*** method, when invoked, returns the result of compiling a potential WebAssembly response with *source*.
The ***instantiateStreaming(source, importObject)*** method, when invoked, performs the following steps:

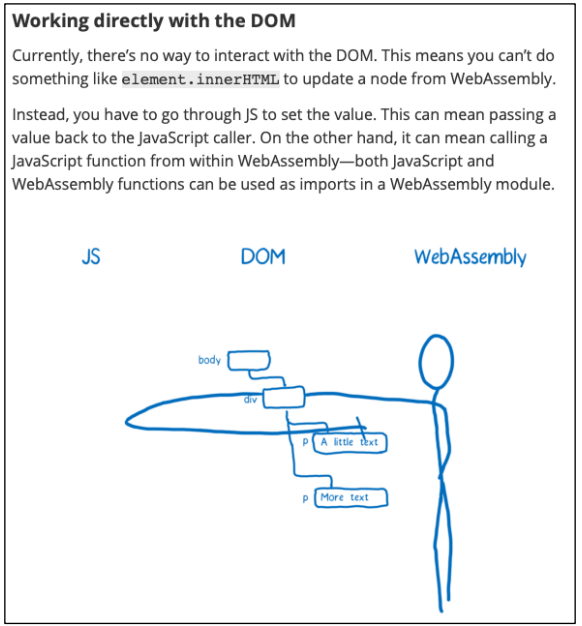
https://www.w3.org/TR/wasm-web-api-1/

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

# Web Assembly Future

Work-in-progress (WIP):

threads, source maps,
memory inspection,
garbage collection




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

https://webassembly.org/docs/security/



# WASM & Security — very good, but not perfect

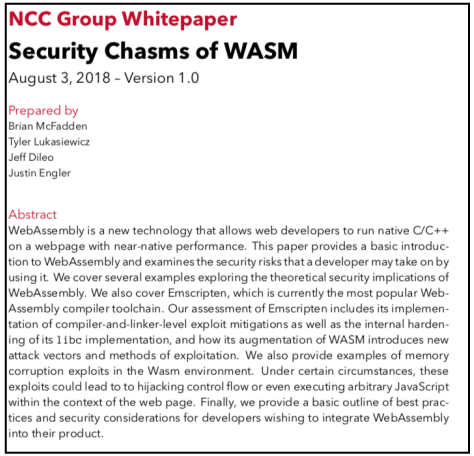
*Malware, Scams, Keyloggers, ...*

**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



**NCC Group Whitepaper**

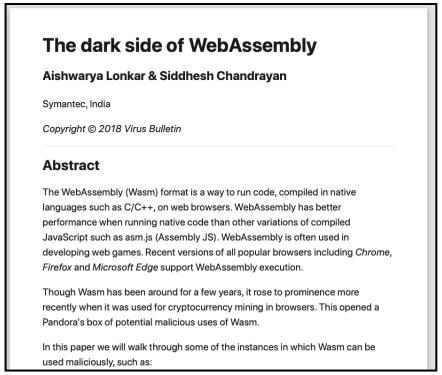
**Security Chasms of WASM**

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 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.



**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/

**WebAssembly Changes Could Ruin Meltdown and Spectre Browser Patches**

Tara Seals • June 27, 2018 2:26 pm

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/

# 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