Security of WebAssembly Applications

Quentin Stiévenart, Vrije Universiteit Brussel
SecAppDev 2022

@acieroid

quentin.stievenart@vub.be

WebAssembly: Context

- History of native web technologies
- What is WebAssembly
- Existing uses for WebAssembly
- WebAssembly in practice

History of Native Web Technologies

History of Native Web Technologies

Why do we want native web technologies?

- Optimize compute-intensive parts of an application
- Enable ahead-of-time compilation
- Hardware accelerated graphics
- Compile from a variety of programming languages

Java Applets (1995-2017)

First released with Java 1 (1995)

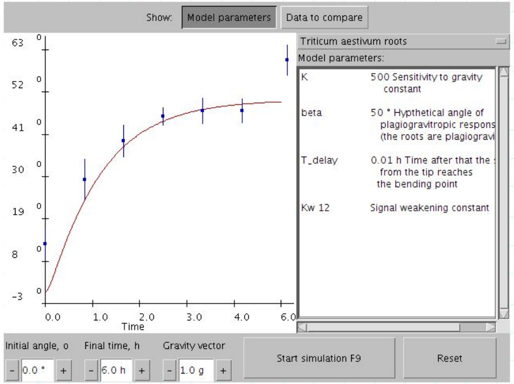
Requires local installation of Java + browser plugin

JVM running the applet is separated from browser at OS-level

Only supports Java

Phased out from 2013, fully removed in Java 9 (2017)

```
<applet code="First.class" width="300" height="300">
```



ActiveX (1996-2015)

Introduced by Microsoft in 1996, as a competitor to Java Applets

Supports more programming languages, supposedly faster

Designed to be cross-platform and not tied to Windows (but not really in practice)

Many criticisms: security issues (no sandboxing), lack of portability

Dropped support in Microsoft Edge (2015)

Flash (1996-2017)

Released by Macromedia in 1996

Closed-source implementation

Many security issues: 1078 CVE entries, 842 leading to arbitrary code execution

Deprecated by Adobe in 2017, EOL in 2020

Google Native Client & PNaCl (2011-2017)

Introduced by Google in 2011

Sandboxing technology to run native code

OS and architecture independent

Deprecation announced in 2017 in favor of WebAssembly

asm.js (2013-...)

Strict subset of JavaScript: can be run on any JS engine

Rely on annotations to compile AOT:

- More efficient representations (e.g., unboxed ints)
- If validation fails, falls back to regular JIT
- Manual memory management, no GC

Still supported, mostly as fallback

Emscripten introduced to compile C and C++ to asm.js

```
function isPair(x) {
    x = x | 0;
    return ((x & 1 ? MEM8[x & 31] | 0 : (MEM32[x >> 2] | 0) >>> 2 & 63 | 0) | 0) == 0 | 0;
}
```

WebAssembly: What It Is, Its Goals, and Its Usage

WebAssembly

"WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications."

– https://webassembly.org/

A Brief History of WebAssembly

2015 — **First public announcement**

"Unlike asm.js, which was just a pure optimization - i.e. it did not require standardization - WebAssembly is being developed as a standard."

Public Announcement Pre-release

jfbastien released this Jun 18, 2015 public-annou... 9974cde

Compare

As discussed in #150, this is the state of the repository on the initial public announcement of WebAssembly.

2016 — **First demo**

"AngryBots" game developed in Unity, compiled to WebAssembly

2017 — **End of preview phase**

2018 — **First working draft**

WA

WebAssembly Specification

2019 — **W3C recommendation**

For WebAssembly Core 1.0

WebAssembly Core Specification

W3C Recommendation, 5 December 2019

W3C

This version:

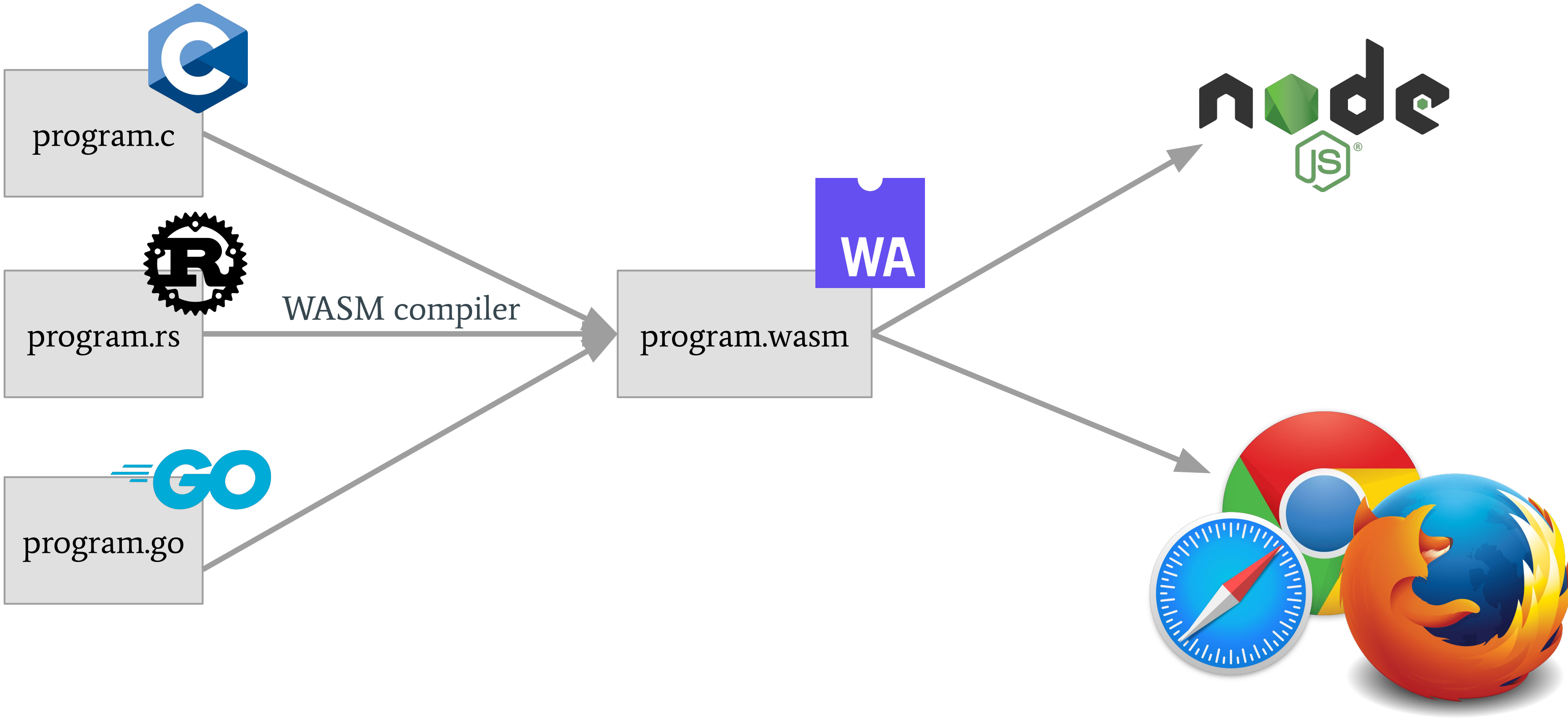
https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/

WebAssembly Goals

Major goals:

- Provide an execution environment for client- and server-side applications
- Enable almost-native performance
- Isolate executed code from the rest of the browser/OS
- Open standard

WebAssembly Usage in a Nutshell



Today's Use of WebAssembly: Web Applications

earth.google.com

Today's Use of WebAssembly: IoT

Wasmachine: Bring IoT up to Speed with A WebAssembly OS

Elliott Wen
The University of Auckland
jwen929@aucklanduni.ac.nz

Gerald Weber
The University of Auckland
g.weber@aucklanduni.ac.nz

***Abstract*—WebAssembly is a new-generation low-level bytecode format and gaining wide adoption in browser-centric applications. Nevertheless, WebAssembly is originally designed as a general approach for running binaries on any runtime environments more than the web. This paper presents Wasmachine, an OS aiming to efficiently and securely execute WebAssembly applications in IoT and Fog devices with constrained resources. Wasmachine achieves more efficient execution than conventional OSs by compiling WebAssembly ahead of time to native binary and executing it in kernel mode for zero-cost system calls. Wasmachine maintains high security by not only exploiting many sandboxing features of WebAssembly but also implementing the OS kernel in Rust to ensure memory safety. We benchmark commonly-used IoT and fog applications and the results show that Wasmachine is up to 11% faster than Linux.**

I. Introduction

A conventional WebAssembly runtime, as shown in Fig I (a), is a program that translates WebAssembly binary instructions to native CPU machine codes before execution. The translation is most achieved in a just-in-time (JIT) fashion; when a WebAssembly application starts, it will be first interpreted, and after a while, methods frequently executed will be compiled to native codes to improve execution efficiency. JIT enables fast start up time but less efficient codes due to limited time that can be spent on code optimization. Using JIT is reasonable in the context of web browsing, where startup time may significantly affect user experience. However, it is suboptimal for IoT or fog computing, where code efficiency is preferred.

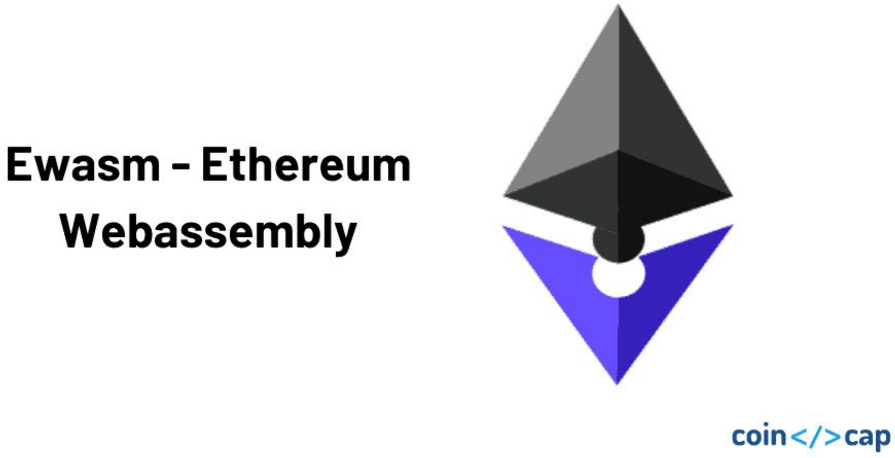
A runtime also assists a WebAssembly program with system call operations (e.g., networking or file access). Specifi-

Wen and Weber, PerCom 2020

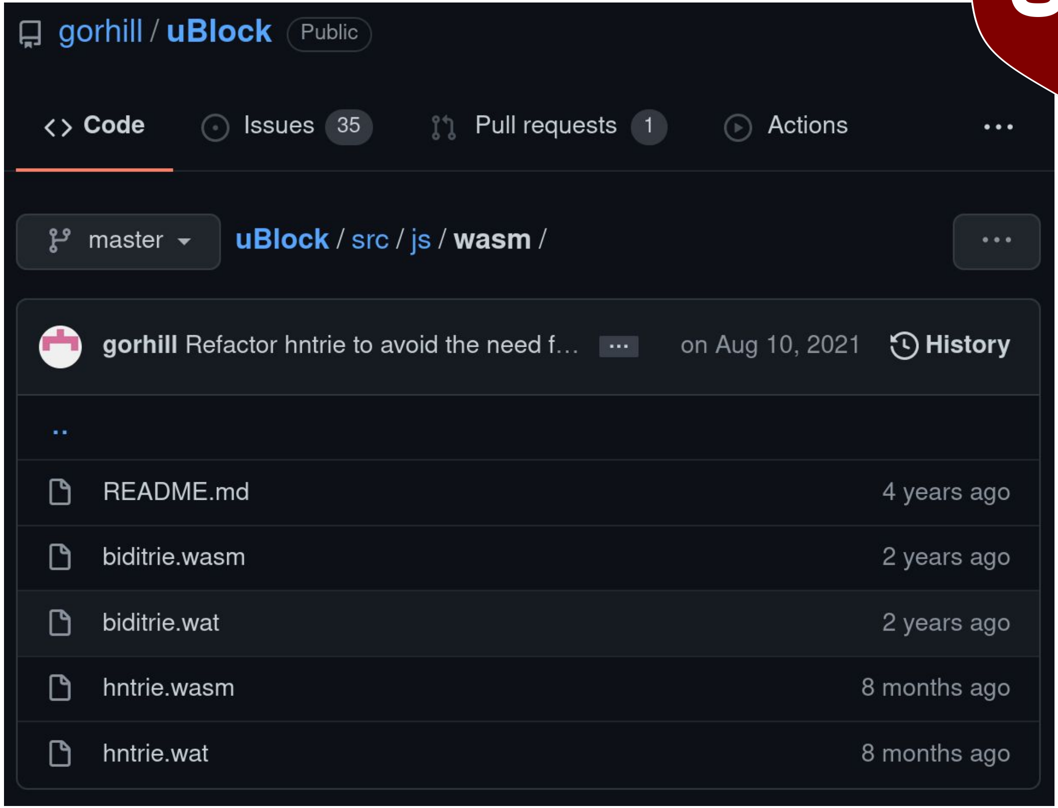
Today’s Use of WebAssembly: Embedded Systems

Gurdeep Singh and Scholliers, MPLR’19

Today’s Use of WebAssembly: Smart Contract Platforms



Today's Use of WebAssembly: Browser Add-Ons



Today’s Use of WebAssembly: Edge Computing

Compute@Edge

The Compute@Edge platform helps you compile your custom code to WebAssembly and runs it at the Fastly edge using the WebAssembly System Interface for each compute request. Per-request isolation and lightweight sandboxing create an environment focused on performance and security.

Serverless isolation technology

Compute@Edge runs WebAssembly (Wasm). When a Compute request is received by Fastly, an instance is created and the serverless function is run, allowing developers to apply custom business logic on demand.

WebAssembly in Practice: Two Formats

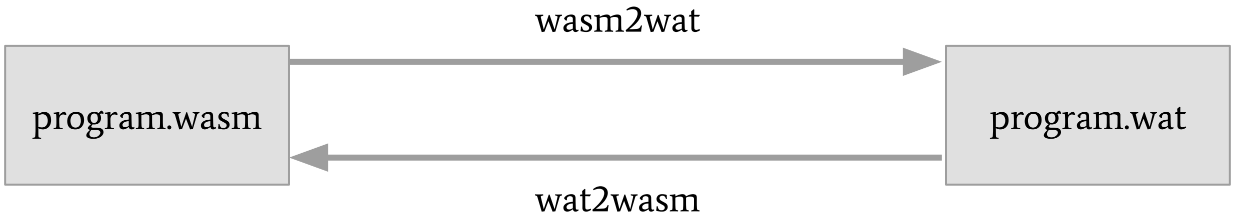
```
0061 736d 0100 0000 0138 0a60 027f 7f01
7f60 037f 7f7f 017f 6002 7f7f 0060 017f
0060 037f 7f7f 0060 017f 017f 6004 7f7f
7f7f 017f 6001 7f01 7e60 0000 6002 7e7f
017f 0230 010f 2e2f 6865 6c6c 5f5f 7761
736d 2e6a 731c 5f5f 7762 675f 616c 6572
745f 3238 3437 6336 3164 6632 3737 3436
3563 0002 0331 3005 0101 0300 0102 0002
0902 0002 0203 0402 0202 0303 0400 0600
0104 0105 0102 0103 0300 0506 0000 0000
```

program.wasm

```
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (type (;1;) (func (param i32 i32 i32) (result i32)))
  (type (;2;) (func (param i32 i32)))
  (type (;3;) (func (param i32)))
  (type (;4;) (func (param i32 i32 i32)))
  (type (;5;) (func (param i32) (result i32)))
  (import "./hell__wasm.js" "alert" (func (;0;) (type 2)))
  (func (;1;) (type 5) (param i32) (result i32)
    (local i32 i32 i32 i32 i32 i32 i32 i32 i64)
    block  ;; label = @1
      block  ;; label = @2
        block  ;; label = @3
          local.get 0
          i32.const 245
          i32.ge_u
          if  ;; label = @4
            local.get 0
            i32.const -65587
            i32.ge_u
            …
```

program.wat

WebAssembly in Practice: Two Formats



WebAssembly in Practice: Compiling to WebAssembly

Many existing compilers rely on LLVM support

Emscripten generates a .wasm and all necessary glue code

```
$ emcc hello.c -o hello.html
$ clang --target=wasm32 … hello.wasm hello.c
```

WebAssembly in Practice: Interfacing with JavaScript

`WebAssembly` object provides way of interacting with WebAssembly

```
WebAssembly.instantiateStreaming(fetch('myModule.wasm'), importObject).then(obj => {
  obj.instance.exports.exported_func();
  var i32 = new Uint32Array(obj.instance.exports.memory.buffer);
  var table = obj.instance.exports.table;
  console.log(table.get(0)());
});
```

WebAssembly in Practice: Interfacing with JavaScript

```
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (type (;1;) (func (param i32 i32 i32) (result i32)))
  (type (;2;) (func (param i32 i32)))
  (import "./module.js" "add" (func (;0;) (type 0)))
  (func (;1;) (type 0) (param i32 i32) (result i32)
    i32.const 1
    i32.const 2
    call 0)
  …)
```

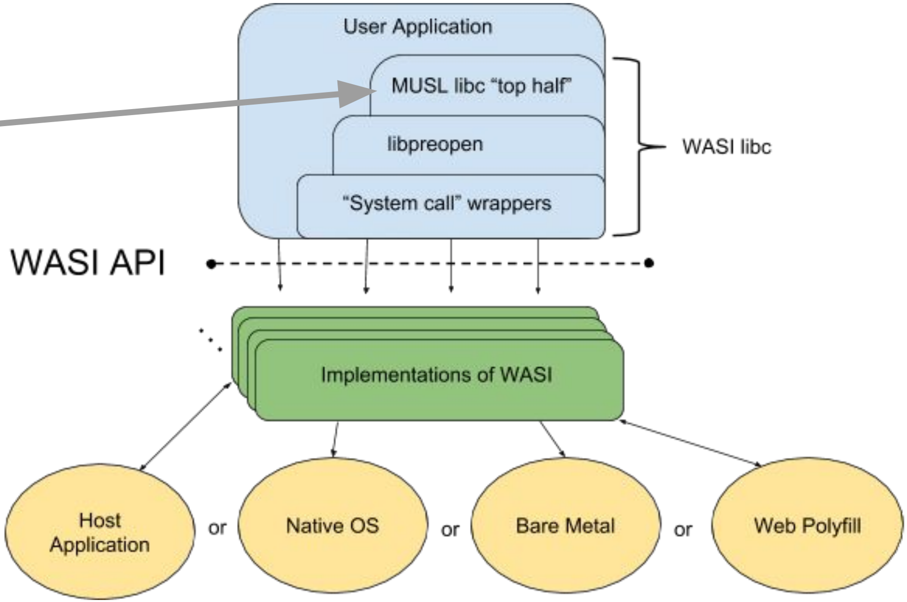
```
var importObject = {
  imports: { add: (x, y) => { return x + y; } }
};
```

WebAssembly in Practice: WASI

For stand-alone applications, it is necessary to interface with the operating system

WASI is currently experimental

```c
int main() {
  printf("Hello, world!\n");
}
```



Can I Use WebAssembly Today?

Yes!

https://caniuse.com/wasm

WebAssembly - OTHER

Usage % of all users ?

Global 93.06%

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

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
		2-46														
	12-14	[1] 47-51	4-50		10-37											
	[3] 15	[4] 52	[2] 51-56	3.1-10.1	[2] 38-43	3.2-10.3							4-6.4			
6-10	16-101	53-100	57-101	11-15.4	44-85	11-15.4		2.1-4.4.4	12-12.1				7.2-15.0			
11	102	101	102	15.5	86	15.5	all	101	64	102	101	12.12	16.0	10.4	7.12	[1] 2.5
		102-103	103-105	16.0-TP	87	16.0										

Can I Use WebAssembly Today?

Many projects are starting to target WebAssembly

For a full list of resources: https://github.com/mbasso/awesome-wasm

Blazor

Build client web apps with C#

Yew

Rust / Wasm client web app framework

WebAssembly's Stack-Based Execution Model

```
(module
  (func (type 0)
    (param i32) (result i32) …)
  (func (type 1)
    (param i32 i32) (result)
    local.get 0 ;; -> i32
    if ;; i32 ->
      local.get 0 ;; -> i32
      local.get 1 ;; -> i32
      i32.add ;; i32 i32 -> i32
      call 0 ;; i32 -> i32
      drop ;; i32 ->
    end))
```

i32
i32

WebAssembly Advantages

- Simplicity
- Secure design
- WASI security
- Performance
- Energy usage
- Openness

Simplicity of WebAssembly: Size of the Specification

WebAssembly core is a small, well-defined standard

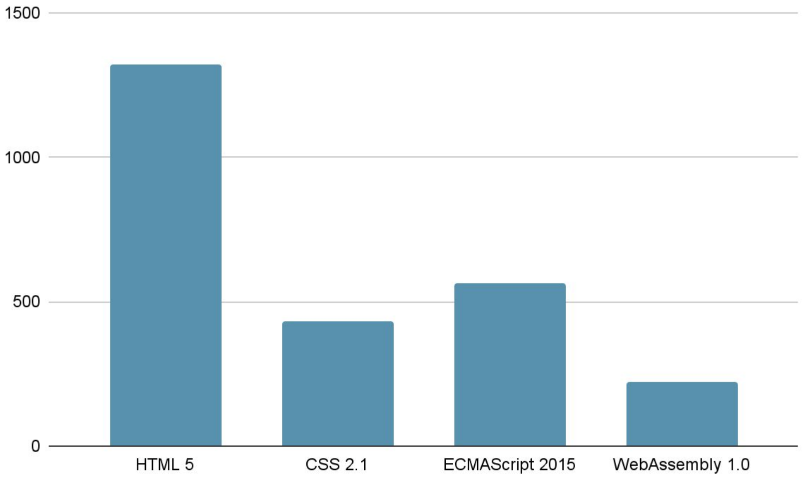
Semantics defined formally, along with a reference implementation

local.get x

1. Let F be the *current frame*.
2. Assert: due to *validation*, F.locals[x] exists.
3. Let val be the value F.locals[x].
4. Push the value val to the stack.

$$F; (\mathsf{local.get}\ x) \quad \hookrightarrow \quad F; val \qquad (\text{if } F.\mathsf{locals}[x] = val)$$

```
let rec step (c : config) : config =
  let {frame; code = vs, es; _} = c in
  let e = List.hd es in
  let vs', es' =
    match e.it, vs with
    | Plain e', vs ->
      (match e', vs with
      ...
      | LocalGet x, vs ->
        !(local frame x) :: vs, []
```



Specification size (number of pages)

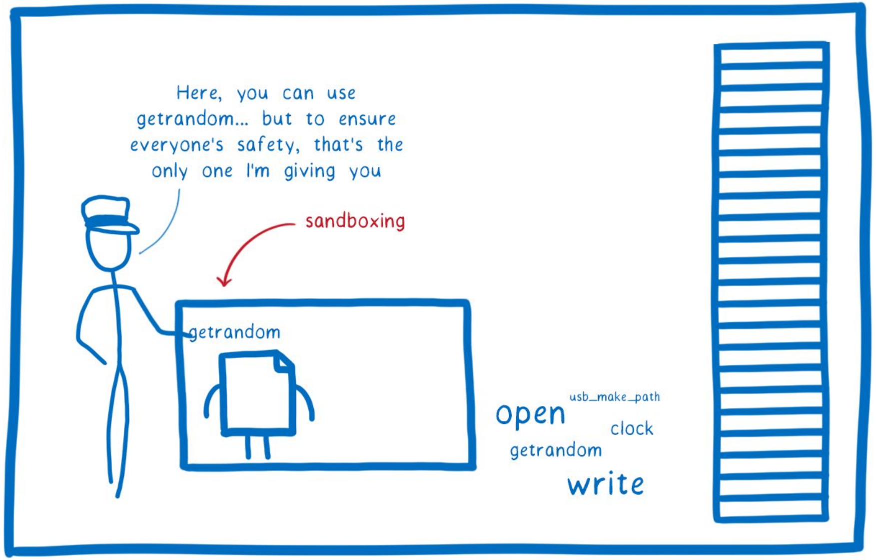
Simplicity of WebAssembly

```
(module
  …
  ;; Function with two parameters, one return value
  (func (type 1) (param i32 i32) (result i32)
    local.get 0 ; stack: [arg0]
    local.get 1 ; stack: [arg1, arg0]
    i32.add)    ; stack: [arg0+arg1]
  ;; Function with one parameter, no return value
  (func (type 0) (param i32) (result)
    …)
  …)
```

Secure Design of WebAssembly: Sandboxing

Applications are sandboxed

- Can’t escape expect through appropriate APIs
- Isolated from each other



Clark, Lin. "Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly" (2019)

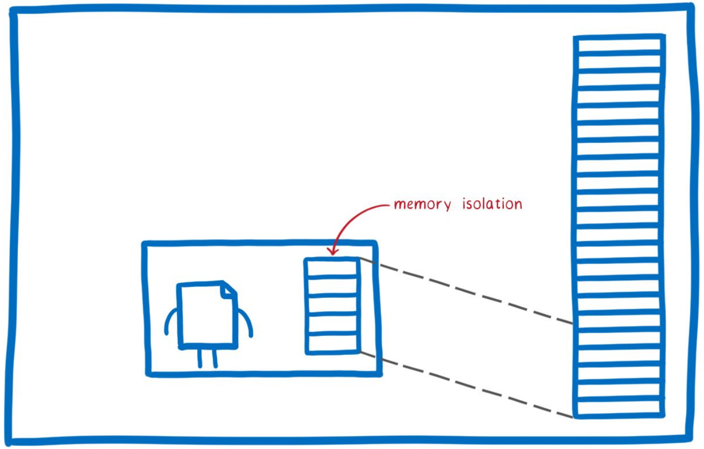
Secure Design of WebAssembly: Memory Model

WebAssembly programs have a single “linear memory”, isolated from the rest

Pointer arithmetic etc. are still doable, but potential damages are lessened

Linear memory is initialized to 0

```
(func (;memory-usage;) (type 0)
  (param i32) (result i32)
  global.get 0 ;; [global]
  local.get 0  ;; [arg0, global]
  i32.store    ;; [] binds @global to arg0 in memory
  global.get 0 ;; [global]
  i32.load     ;; [arg0] loads @global from memory
)
```



Clark, Lin. "Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly" (2019)

Secure Design of WebAssembly: Memory Safety

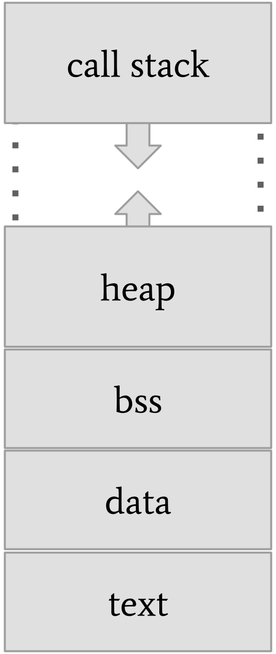
The linear memory is separated from:

- Local and global variables (~registers)
- Call stack

Linear memory is not executable: defeats some code injection attacks

```
i32.const 1
local.set 0
i32.const 2
global.set 0
i32.const 0
call 0
```

Everything lives in a different region.
Out of bounds accesses are caught.



x86 memory

Secure Design of WebAssembly: Control-Flow Integrity

Four control-flow mechanisms that need to be protected:

1. Local jumps (if, br, ...)
2. Direct function calls
3. Function returns
4. Indirect function calls

Secure Design of WebAssembly: Structured Control Flow

WebAssembly has no instruction for arbitrary jumps

Local control-flow instructions:

- Scopes: `block`, `loop`, `if`
- Jumps: `br`, `br_if`, `br_table`

```
block
  i32.const 1
  if
    br 0
  else
    br 1
  end
end
```

Secure Design of WebAssembly: Control-Flow Integrity

Four control-flow mechanisms that need to be protected:

1. Local jumps (if, br, …)
2. Direct function calls
3. Function returns
4. Indirect function calls

Secure Design of WebAssembly: Direct Function Calls

```
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (func (;0;) (type 0) (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add)
  (func (;1;) (type 0) (param i32 i32) (result i32)
    i32.const 1
    i32.const 2
    call 0))
```

Call implicitly manages the call stack. The program has no way of accessing it through other means.

Secure Design of WebAssembly: Control-Flow Integrity

Four control-flow mechanisms that need to be protected:

1. Local jumps (if, br, ...)
2. Direct function calls
3. Function returns
4. Indirect function calls

In x86, the return address is stored on the stack, and can be overwritten by an attacker in a vulnerable program

Secure Design of WebAssembly: Indirect Function Calls

```
(func (;0;) (type 0) (param i32) (result i32)
  local.get 0
  call_indirect (type 0))
(func (;1;) (type 0) (param i32) (result i32) ...)
(func (;2;) (type 0) (param i32) (result i32) ...)
(func (;3;) (type 1) (param i32 i32) (result i32) ...)
(table (;0;) 4 4 funcref)
(elem (;0;) (i32.const 1) 1 2 3)
```

Call target must have the right type

Possible targets of indirect calls, but can be mutated by host environment

Secure Design of WebAssembly: Control-Flow Integrity

Four control-flow mechanisms that need to be protected:

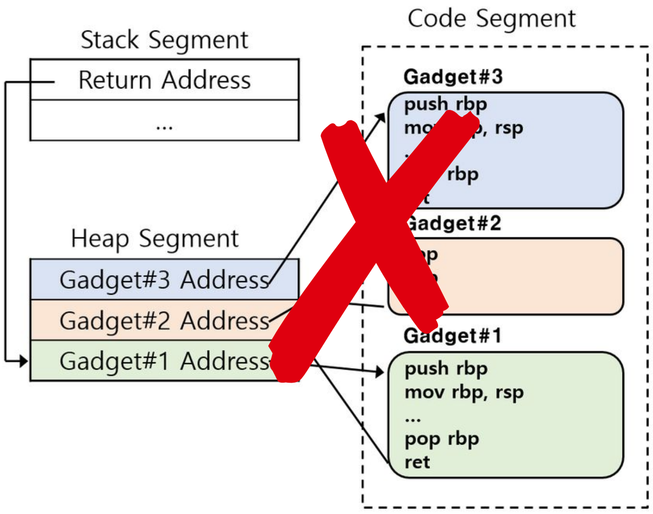
1. Local jumps (if, br, ...)
2. Direct function calls
3. Function returns
4. Indirect function calls

Secure Design of WebAssembly: Lack or Arbitrary Jumps

No arbitrary jumps:

- Prevents ROP
- Limit code-reuse attacks

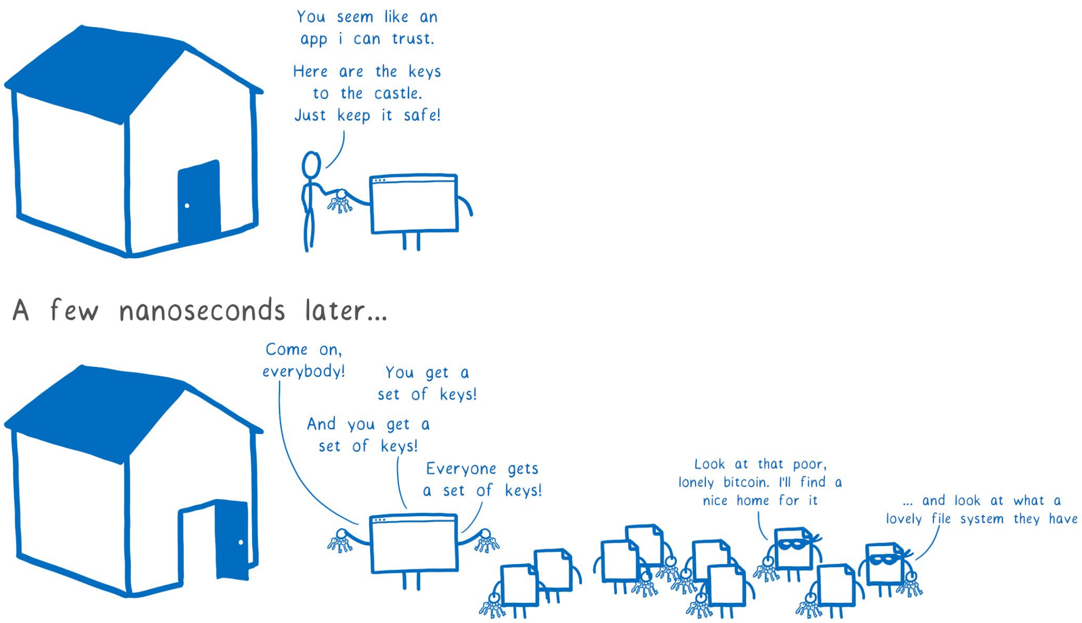
```
(func (;0;) (type 0) (param i32) (result i32)
  local.get 0
  call_indirect (type 0))
(func (;1;) (type 0) (param i32) (result i32) ...)
(func (;2;) (type 0) (param i32) (result i32) ...)
(func (;3;) (type 1) (param i32 i32) (result i32) ...)
(table (;0;) 4 4 funcref)
(elem (;0;) (i32.const 1) 1 2 3)
```

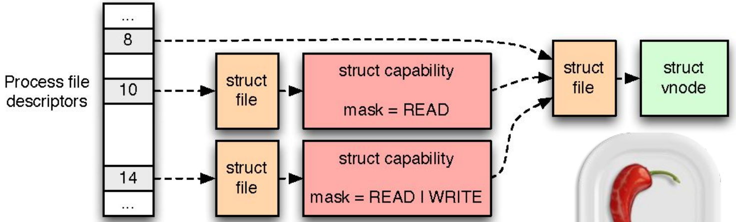


Yun, J., Park, K. W., Koo, D., & Shin, Y. (2020). Lightweight and seamless memory randomization for mission-critical services in a cloud platform. Energies, 13(6), 1332.

Secure Design of WebAssembly: WASI

WASI relies on capability-based security



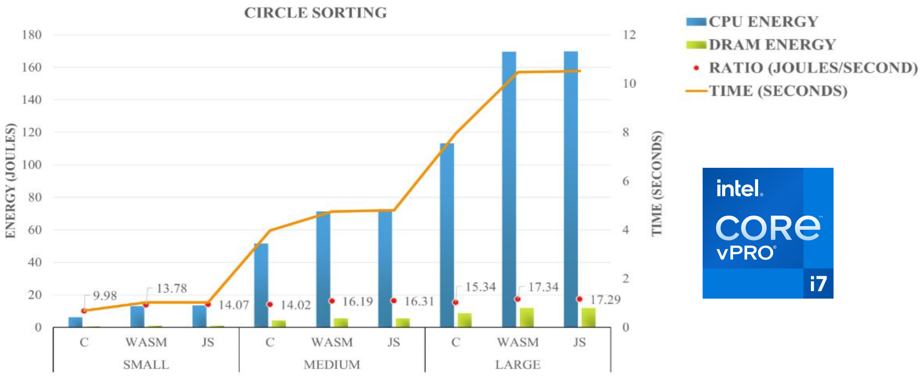


Capsicum

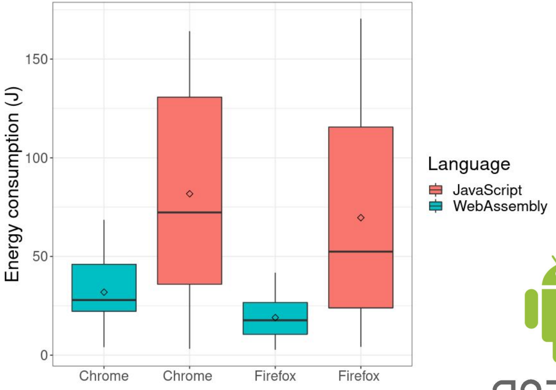
Watson, Robert NM, et al. "Capsicum: Practical Capabilities for UNIX." 19th USENIX Security Symposium (USENIX Security 10). 2010.

Energy Usage

WebAssembly is still in its early years, with lots of room to grow



De Macedo, João, et al. "On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet?." 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). IEEE, 2021.



van Hasselt, Max, et al. "Comparing the Energy Efficiency of WebAssembly and JavaScript in Web Applications on Android Mobile Devices." (2022).

Performance

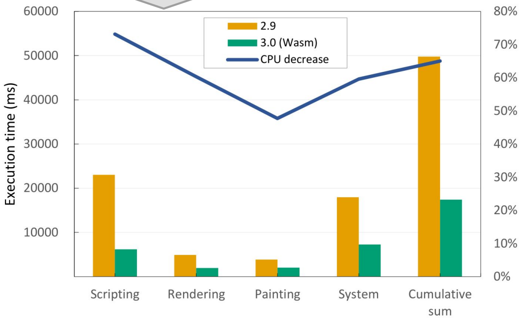
Again: plenty of room for improvements, while JS engines have been heavily optimized

As input size increases, JS becomes faster (JIT)

Input Size	SD #[1]	SD gmean[2]	SU #[3]	SU gmean[4]	All gmean[5]
Extra-small	0	0x ↓	30	35.30x ↑	35.30x ↑
Small	1	1.53x ↓	29	8.35x ↑	7.67x ↑
Medium	17	1.53x ↓	13	3.68x ↑	1.38x ↑
Large	15	1.67x ↓	15	1.16x ↑	0.83x ↑
Extra-large	17	1.22x ↓	13	1.08x ↑	0.92x ↑

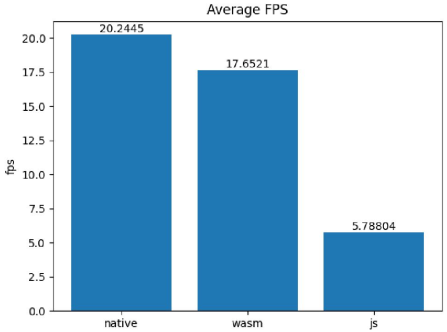
Wang, Weihang. "Empowering Web Applications with WebAssembly: Are We There Yet?." 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021.

On a real-world application (the Micrio storytelling platform)



Ketonen, T. (2022). Examining performance benefits of real-world WebAssembly applications: a quantitative multiple-case study.

On a raytracer



Johansson, L. (2022). Ray tracing in WebAssembly, a comparative benchmark.

Language Support for WebAssembly

https://github.com/appcypher/awesome-wasm-langs

- .Net
- AssemblyScript
- ~~Astro~~ `Unmaintained`
- Brainfuck
- C
- C#
- C++
- Clean
- Co
- COBOL
- D
- Eel
- Elixir
- F#
- Faust
- Forest
- Forth
- Go
- Grain
- Haskell
- Java
- JavaScript
- Julia
- ~~Idris~~ `Unmaintained`
- Kotlin/Native
- Kou
- Lisp
- Lobster
- Lua
- Lys
- Never
- Nim
- Ocaml
- Pascal
- Perl
- PHP
- Plorth
- Poetry
- Python
- Prolog
- Ruby
- Rust
- Scheme
- Scopes
- ~~Speedy.js~~ `Unmaintained`
- Swift
- ~~Turboscript~~ `Unmaintained`
- TypeScript
- ~~Wah~~ `Unmaintained`
- ~~Walt~~ `Unmaintained`
- ~~Wam~~ `Unmaintained`
- Wase
- WebAssembly
- ~~Wracket~~ `Unmaintained`
- Zig

Enterprise-Backed Standard

“The Bytecode Alliance is committed to establishing a capable, secure platform that allows application developers and service providers to confidently run untrusted code, on any infrastructure, for any operating system or device, leveraging decades of experience doing so inside web browsers.”

– bytecodealliance.org/

Enterprise-Backed Standard

BYTECODE ALLIANCE

amazon
arm
WASM, it's now AsmNext
igalia
intel
Microsoft
cosmonic
EDJX
midokura
A Sony Group Company
moz://a
Profian
embark
fastly
FERMYON
Rackner
shopify
SIEMENS
FUTUREWEI Technologies
Google
InfinyOn
SingleStore
StackBlitz
SUBORBITAL
tangram
UC San Diego

Open Standard

The goal is to have a standard, not a specific implementation

Standard can be extended through a proposal system, open to anyone

Features can only be standardized after being implemented in 2+ VMs

WebAssembly Security Concerns

- Malwares
- Vulnerabilities and their exploits
- Execution differences
- Compiler bugs
- Runtime bugs

WebAssembly Malwares

If a malicious program runs in a sandbox, can it still cause harm? Yes!

Category	# of unique samples		# of websites		Malicious
Custom	17	(11.3 %)	14	(0.9 %)	
Game	44	(29.3 %)	58	(3.5 %)	
Library	25	(16.7 %)	636	(38.8 %)	
Mining	48	(32.0 %)	913	(55.7 %)	✗
Obfuscation	10	(6.7 %)	4	(0.2 %)	✗
Test	2	(1.3 %)	244	(14.9 %)	
Unknown	4	(2.7 %)	5	(0.3 %)	
Total	150	(100.0 %)	1,639	(100.0 %)	

Musch, Marius, et al. "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild." International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Cham, 2019.

WebAssembly Malwares: Rise and Fall of Cryptojacking

In 2017: Coinhive mining scripts get misused on purpose

In 2019: Coinhive shuts down

Since then, 1% of sites that used Coinhive still do cryptojacking

“We concluded that cryptojacking is not dead after the Coinhive shutdown. It is still alive, but not as attractive as it used to be.”

Varlioglu, Said, et al. "Is cryptojacking dead after coinhive shutdown?." 2020 3rd International Conference on Information and Computer Technologies (ICICT). IEEE, 2020.

WebAssembly Malwares: A More Recent Study

Finds a similar number: 1% of binaries found on the web are doing cryptomining

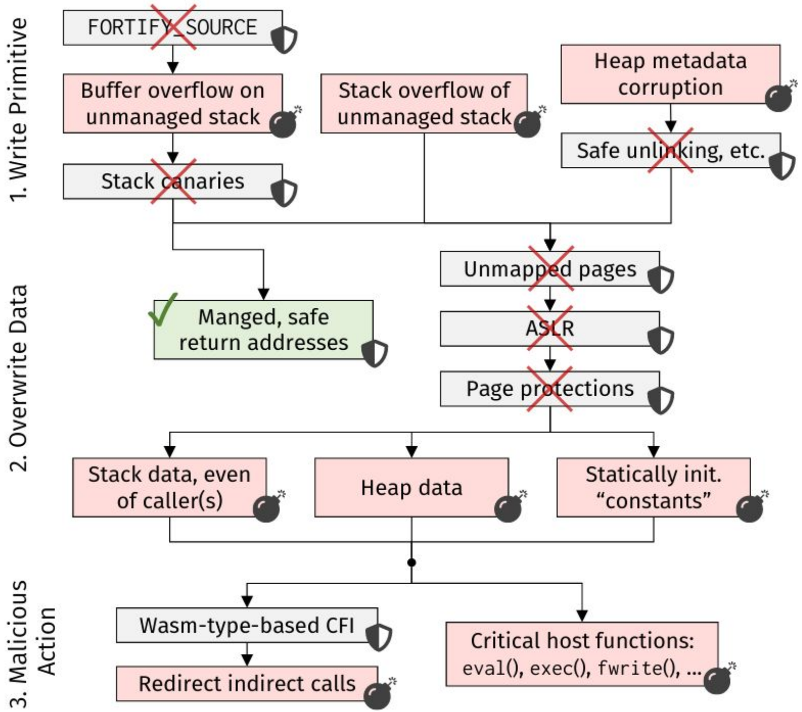
“We find WebAssembly-based cryptominers to have significantly dropped in importance compared to the results of an earlier study. This finding motivates security research to shift the focus from malicious WebAssembly to vulnerabilities in WebAssembly binaries”

Hilbig, Aaron, Daniel Lehmann, and Michael Pradel. "An empirical study of real-world webassembly binaries: Security, languages, use cases." Proceedings of the Web Conference 2021. 2021.

Vulnerabilities

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

How can we attack a WebAssembly binary?

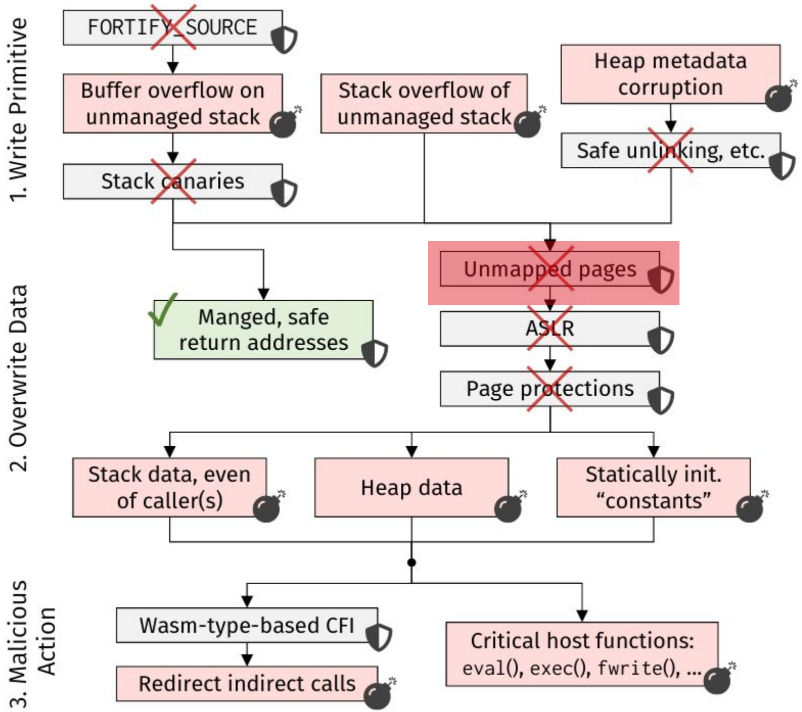


Missing Protection: Unmapped Pages

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

In native code: accessing unmapped pages trigger segfaults

In WebAssembly: all access to linear memory are allowed



Missing Protection: ASLR

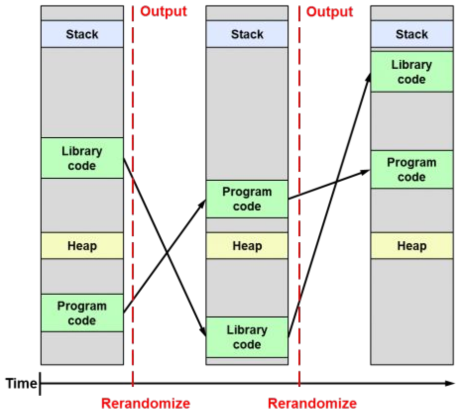
Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

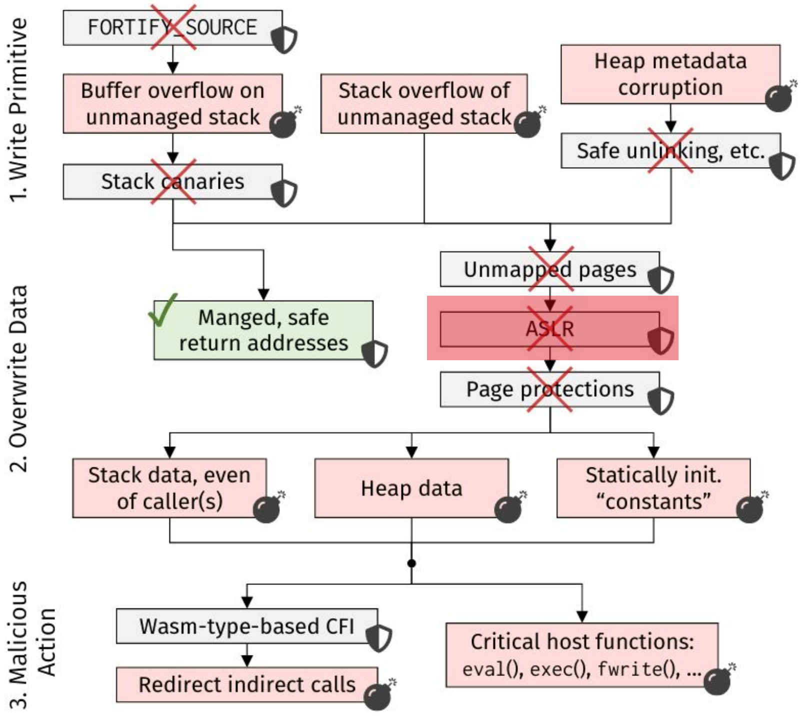
In native code: layout can be randomized

Render attacks on 64 bits much more difficult

In WebAssembly: no randomization

Even if added, 32 bit makes it easy to defeat



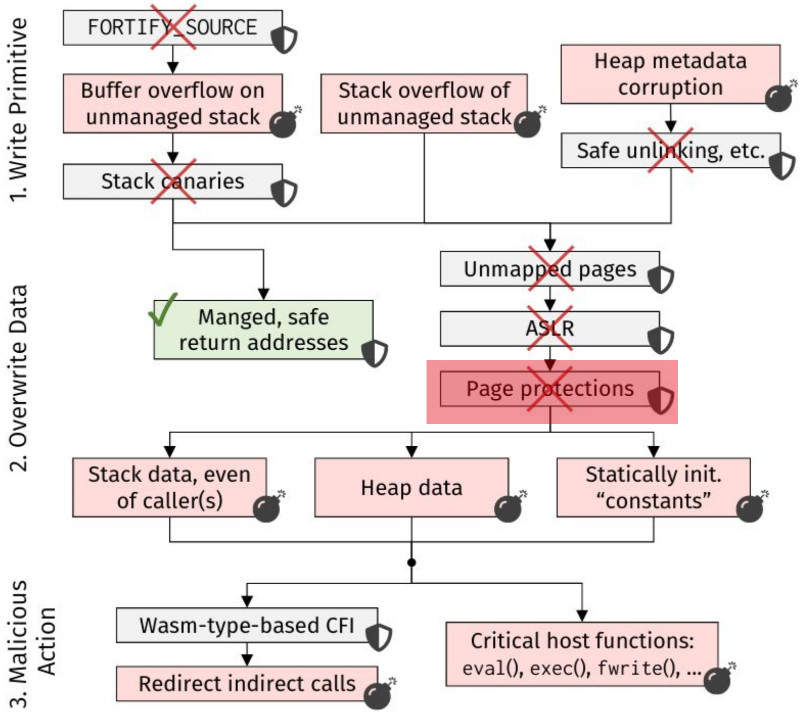


Missing Protection: Page Protection

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

In native code: pages have “protection flags”: readable, writable, executable

In WebAssembly: linear memory is r, w, but not x



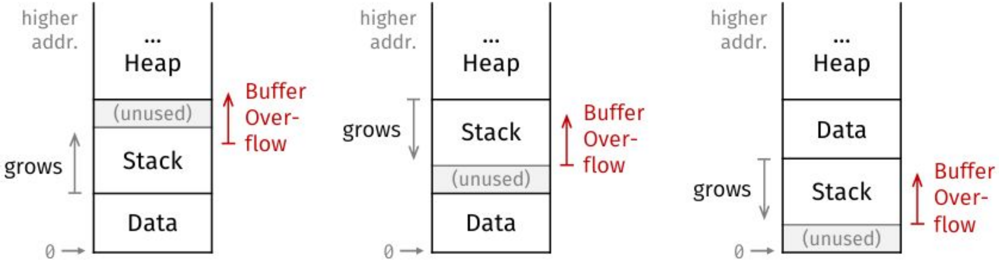
Managed vs. Unmanaged Data

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

Locals, globals, call stack, etc. are isolated in “managed data”

However, compilers need to use linear memory for unmanaged data: strings, arrays, …

→ Even though the WebAssembly call stack is isolated, the C call stack may not be!



(a) emcc 1.39.7 (*fastcomp* backend, deprecated).

(b) emcc 1.39.7 (*upstream* backend), clang 9 (WASI).

(c) clang 9 (WASI with stack-first), rustc 1.41 (WASI).

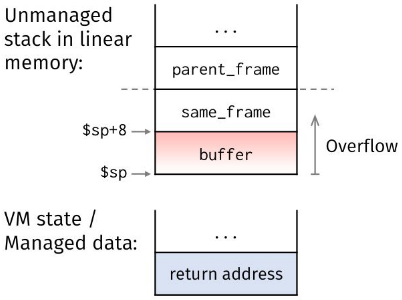
Attack: Stack-Based Buffer Overflow

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

WebAssembly is supposedly protected against stack smashing attacks

... but due to unmanaged stack when compiling from C, attacks are still possible

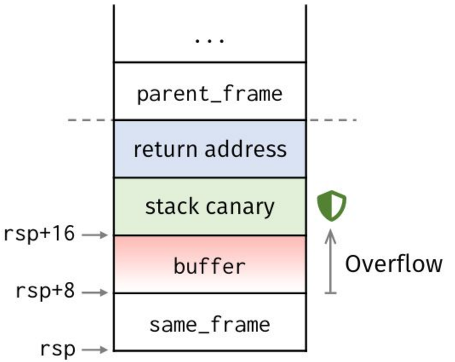
```c
void parent() {
  char parent_frame[8] = "BBBBBBBB"; // Also overwritten
  vulnerable(readline());
  // Dangerous if parent_frame is passed, e.g., to exec
}
void vulnerable(char* input) {
  char same_frame[8] = "AAAAAAAA"; // Can be overwritten
  char buffer[8];
  strcpy(buffer, input); // Buffer overflow on the stack
}
```

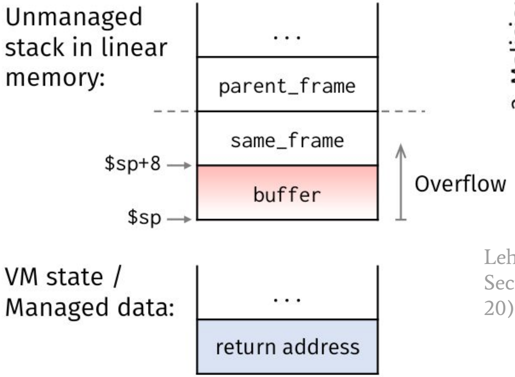


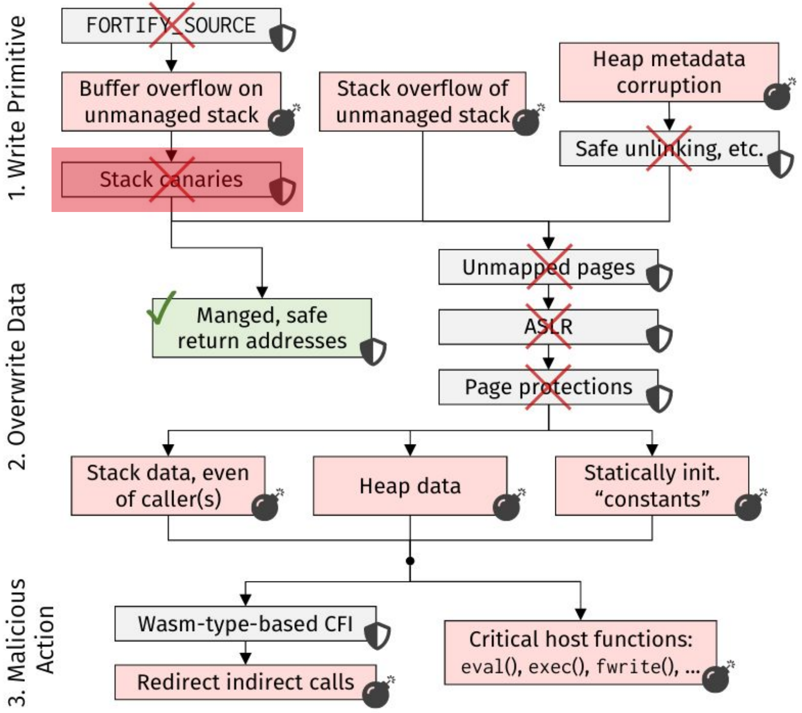
Missing Protection: Stack Canaries

In native code: stack smashing is prevented through stack canaries

In WebAssembly: return address cannot be rewritten, but data can still be overwritten







Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

Memory Allocators in WebAssembly

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

Memory is manually managed → need for an allocator

Binary size is an important factor → use of minimal allocators

Default (native) allocator dlmalloc are hardened against many attacks

Minimal allocators (wee_alloc, emmalloc) are not...

Attack: Heap Metadata Corruption

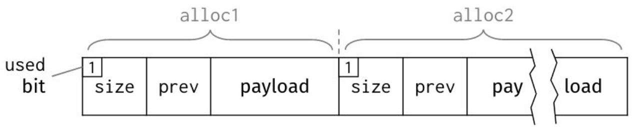
Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

“Unlink exploit” possible against emmalloc

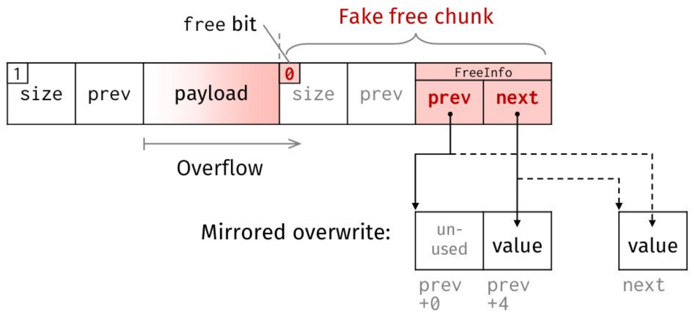
After an overflow, allocator merges free block with another non-free one

Value of prev and next can be used for writing to arbitrary locations

```
// Called on alloc2, before merging it into alloc1.
void removeFromFreeList(Chunk* chunk) {
  FreeInfo* freeInfo = chunk->freeInfo;
  freeInfo->prev->next = freeInfo->next; // mirrored
  freeInfo->next->prev = freeInfo->prev; // write
}
```



(a) Heap layout before the overflow: two adjacent chunks.



(b) Heap layout after an overflow of `alloc1`: manipulated metadata causes mirrored write to a chosen location on `free`.

What Can be Overwritten?

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

- Any stack data
  → But no return addresses!
- Any heap data
- Constant data! Read-only linear memory does not exist in WebAssembly

Arbitrary Code Execution?

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

Three main approaches to achieve it for WebAssembly:

- Redirect indirect calls
- Inject code in the host environment
- Application-specific

Arbitrary Code Execution through Indirect Calls

Not really “arbitrary”

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

```
(func (;0;) (type 0) (param i32) (result i32)
  local.get 0
  call_indirect (type 0))
(func (;1;) (type 0) (param i32) (result i32) ...)
(func (;2;) (type 0) (param i32) (result i32) ...)
(func (;3;) (type 1) (param i32 i32) (result i32) ...)
(table (;0;) 4 4 funcref)
(elem (;0;) (i32.const 1) 1 2 3)
```

Call target must have the right type

Arbitrary Code Execution through Host Environment

McFadden, B., Lukasiewicz, T., Dileo, J., & Engler, J. (2018). Security chasms of wasm. NCC Group Whitepaper.

```
char *payload = “alert(‘XSS’);// ”
                “                ”
                “                ”
                “                ”
                “   \x40\x00\x05\x00\x00\x00”;
⚡memcpy(comms.msg, payload, 72); // comms.msg is 64 bytes long!
emscripten_run_script(“console.log(‘Porting my program to WASM!’);”);
…
```

Arbitrary Code Execution through Application-Specific Means

Example: WebAssembly issues a web request through an imported function

Different host could be contacted through overwrites

Example: WebAssembly modules contain interpreter/runtime for CLI/.NET

Manually crafted code could be interpreted

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

End-to-End Case Study: XSS in the Browser

Including vulnerable code may lead to XSS

Example: image manipulation website that depends on vulnerable version of libpng

- Specific version of libpng suffers from a buffer overflow

```
void main() {
  std::string img_tag = "<img src=’data:image/png;base64,";
  pnm2png("input.pnm", "output.png");  // CVE-2018-14550
  img_tag += file_to_base64("output.png") + "’>";
  emcc::global("document").call("write", img_tag);
}
```

Overwrites the img_tag buffer

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

End-to-End Case Study: Remote Code Execution in Node.js

Including vulnerable code in server-side WebAssembly can enable RCE

Example: server accepts log requests from clients

```c
// Functions supposed to be triggered by requests
void log_happy(int customer_id) { /* ... */ }
void log_unhappy(int customer_id) { /* ... */ }

void handle_request(char *input1, int input2, char *input3) {
  void (*func)(int) = NULL;
  char *happiness = malloc(16);
  char *other_allocation = malloc(16);
  memcpy(happiness, input1, input2); // Heap overflow
  if      (happiness[0] == ’h’) func = &log_happy;
  else if (happiness[0] == ’u’) func = &log_unhappy;
  free(happiness); // Unlink exploit overwrites func
  func(atoi(input3)); // 3rd input is passed as argument
}

// Somewhere else in the binary:
void exec(const char *cmd) { /* ... */ }
```

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

End-to-End Case Study: Arbitrary File Write in VM

Some attacks impossible on native code become possible in WebAssembly

Example: writing to a file

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 217-234).

```c
// Write "constant" string into "constant" file
FILE *f = fopen("file.txt", "a");
fprintf(f, "Append constant text.");
fclose(f);

// Somewhere else in the binary:
char buf[32];
scanf("%[^\n]", buf); // Stack-based buffer overflow
```

```
(data (i32.const 65536) "%[^\0a]\00
                         file.txt\00a\00
                         Append constant text.\00...")
```

Read-only in native code
Can be overwritten in WASM

Execution Differences

Stiévenart, Q., De Roover, C., & Ghafari, M. (2022, April). Security risks of porting C programs to webassembly. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (pp. 1713-1722).

Dataset of 17 802 programs
(terminating and deterministic)

Compile & run .wasm

Compile & run native

Compare return code and stdout

Programs exhibiting differences
4 911 programs

Execution Differences: Wide Characters

Most differences are caused by files using wide characters

```
wprintf(L"hello\n");
```

WebAssembly

Native

Displays nothing

Displays `hello`

Requires calling `fwide(stdout, 1)`

Stiévenart, Q., De Roover, C., & Ghafari, M. (2022, April). Security risks of porting C programs to webassembly. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (pp. 1713-1722).

Execution Differences: puts

Documentation states that `puts` returns a "non-negative value upon success"

```
puts("hello\n");
```

WebAssembly

Native

Returns 0

Returns 6

Stiévenart, Q., De Roover, C., & Ghafari, M. (2022, April). Security risks of porting C programs to webassembly. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (pp. 1713-1722).

Execution Difference: Pointer Size

WebAssembly is a 32-bit architecture

```
sizeof(void *);
```

WebAssembly

Returns 4

Native

Returns 8

Stiévenart, Q., De Roover, C., & Ghafari, M. (2022, April). Security risks of porting C programs to webassembly. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (pp. 1713-1722).

Execution Difference: Uninitialised Data

WebAssembly's memory is zero initialized, reducing the chance of seeing "garbage"

```
printf("%s", malloc(5*sizeof(char)));
```

WebAssembly

Prints the empty string

Native

Prints garbage

Stiévenart, Q., De Roover, C., & Ghafari, M. (2022, April). Security risks of porting C programs to webassembly. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (pp. 1713-1722).

Execution Difference: Malloc/Free Implementation

```c
char *data = malloc(100 * sizeof(char));
data += 10;
free(data);
```

WebAssembly

Runs successfully

Native

`free(): invalid pointer`

Execution Difference: Memory Protections

```c
char *data = malloc(100 * sizeof(char));
data -= 10;
strcpy(data, other);
```

WebAssembly → Runs successfully

Native → `SIGSEGV`

Summary of Differences Encountered

Stiévenart, Q., De Roover, C., & Ghafari, M. (2022, April). Security risks of porting C programs to webassembly. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (pp. 1713-1722).

Root cause	Due to	Programs affected
Different standard library		3574
	Wide characters	3253
	`malloc/free`	259
	`puts`	36
	`printf`	26
Security protections		769
	Stack smashing	626
	Memory protections	143
Execution environment		444
	Uninitialised data	382
	Size of pointers	26
	Size of numbers	18
	OS' environment	18
	Memory layout	18

Compiler Bugs

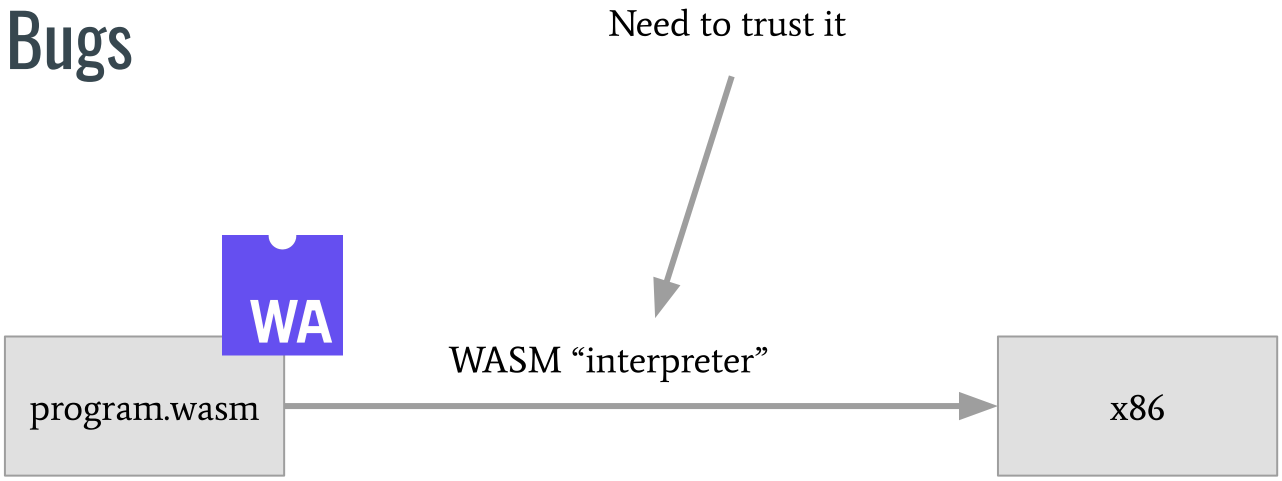
Romano, Alan, et al. "An Empirical Study of Bugs in WebAssembly Compilers." 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021.

Study of 146 bugs in WebAssembly compilers

TABLE II
FINDINGS AND IMPLICATIONS OF OUR STUDY.

	Findings	Implications
1	Data type incompatibility bugs account for 15.75% of the 146 bugs (Section IV-B2).	Interfaces (e.g., APIs) passing values between WebAssembly and JavaScript caused type incompatibility bugs when their data types are mishandled in one of the languages. Such interfaces (e.g., `ftell`, `fseek`, `atoll`, `llabs`, and `printf`) require more attention.
2	Porting synchronous C/C++ paradigm to event-loop paradigm causes a unique challenge (Section IV-B1).	While automated tools support the synchronous to event-loop conversion (e.g., Asyncify), bugs in them may cause concurrency issues (e.g., race condition, out-of-order events). Programs going through this conversation require extensive testing.
3	Supporting (or emulating) linear memory management models is challenging (Section IV-B3).	WebAssembly emulates the linear memory model (of the native execution environment). Many bugs reported in this regard require a particular condition (e.g., allocation of a large memory to trigger heap memory size growth), calling for more comprehensive testing.
4	Changes of external infrastructures used in WebAssembly compilers lead to unexpected bugs (Section IV-B4).	Compiler developers should stay on top of developments that occur in the existing infrastructure used within the compiler. In particular, valid changes (in one context) of existing infrastructure can introduce unexpected bugs in WebAssembly. Rigorous testing is needed.
5	Despite WebAssembly being platform independent, platform differences cause bugs (Section IV-B8).	The default Emscripten Test Suite focuses on testing V8 browser and Node.js, while there are bugs reported due to the platform differences (e.g., caused by other browsers and OSes). The test suite should pay attention to cover broader aspects of the platform differences.
6	Unsupported primitives not properly documented lead to bugs being reported in the compiler (Section IV-D9).	WebAssembly compiler developers should pay attention to keeping the document consistent with the implementation (e.g., mentioning sigsetjmp and function type bitcasting are not supported).
7	Some bug reports failed to include critical information, leading to a prolonged time of debugging (Section IV-C).	We observe that the current bug reporting practice can be improved. In particular, an automated tool that collects critical information (e.g., inputs, compilation options, and runtime environments) would significantly help in the bug reproduction process.
8	Bugs that manifest during runtime made up a significant portion (43%) of the bugs inspected (Section V-B).	Many bugs in the compilers cause runtime bugs in the compiled programs, which are more difficult to detect and fix. To mitigate these bugs, compiler developers should be sure to test the emitted modules in the test suites more exhaustively.
9	77.1% of bug-inducing inputs were less than 20 line and developers manually reduce the size of inputs (Section V-D).	In many cases, bugs can be successfully reproduced by relatively small inputs that are less than 20 lines. Currently, developers often manually reduce large inputs. Automated bug-inducing input reduction (e.g., delta debugging) would be beneficial.

Runtime Bugs



Any break in the isolation guarantee can result in malicious modules corrupting or stealing other modules’ data

Runtime Bugs

Flaws in x86 code generation can happen!

Memory access due to code generation flaw in Cranelift module

Critical **cfallin** published **GHSA-hpqh-2wqx-7qp5** on May 21, 2021

Package	Affected versions	Patched versions
cranelift-codegen (crates.io)	**<= 0.73.0**	**0.73.1, 0.74.0**

Description

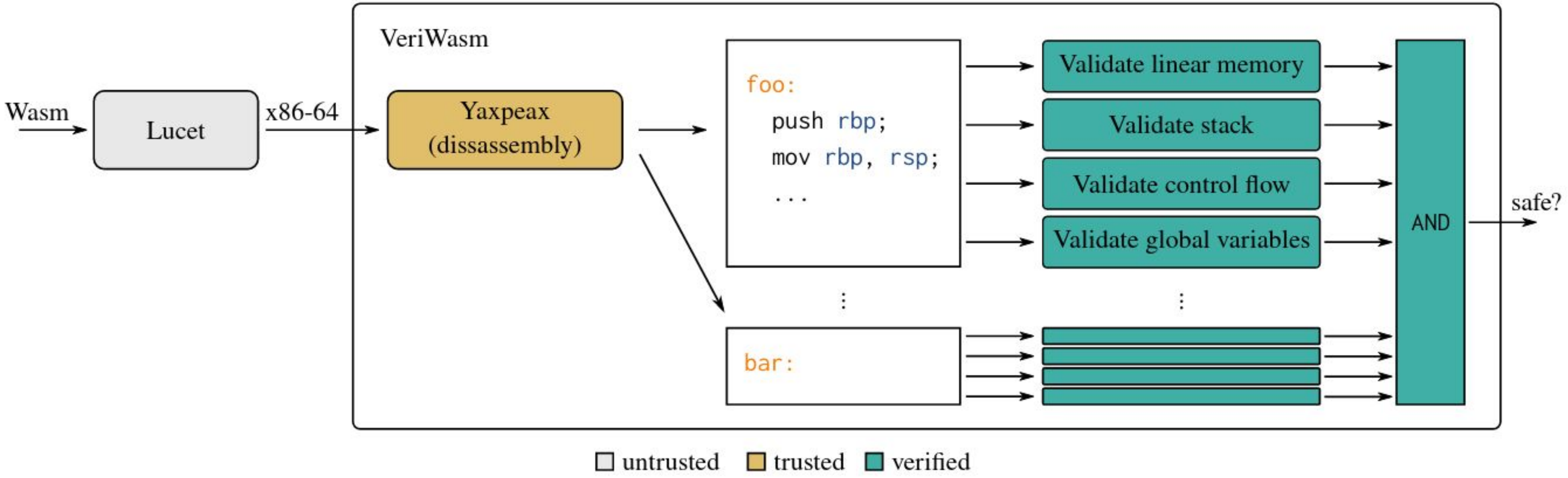
There is a bug in 0.73.0 of the Cranelift x64 backend that can create a scenario that could result in a potential sandbox escape in a WebAssembly module. Users of versions 0.73.0 of Cranelift should upgrade to either 0.73.1 or 0.74 to remediate this vulnerability. Users of Cranelift prior to 0.73.0 should update to 0.73.1 or 0.74 if they were not using the old default backend.

Detected because programs were crashing in the wild

Runtime Bugs

Johnson, E., Thien, D., Alhessi, Y., Narayan, S., Brown, F., Lerner, S., ... & Stefan, D. (2021, February). Доверяй, но проверяй: SFI safety for native-compiled Wasm. In Network and Distributed System Security Symposium (NDSS). Internet Society.

VeriWasm moves the trust further along the chain



Is WebAssembly That Insecure?

Most security aspects also affect native binaries!

The picture looks much better for WebAssembly than native

WebAssembly is still young: lots of room for improvements & better stability

The Future of WebAssembly

- Feature proposals and WebAssembly 2.0
- Tools for WebAssembly

Extensible, Open Standard

Anyone can submit proposals to extend WebAssembly!

Proposal Process

- Phase 0 (“Pre-Proposal”): someone has an idea, discussion is initiated
  Move to next phase if CG deem that it is in-scope and feasible
- Phase 1 (“Feature-Proposal”): iteration over the features
  Move to next phase once spec *text* has been extended
- Phase 2 (“Proposed Spec Text Available”): implementation + tests
  Move to next phase once test suite has been updated and runs against 1 impl.
- Phase 3 (“Implementation”): implementation in other VMs + toolchains
  Move to next phase once 2 VMs + 1 toolchain support it
- Phase 4 (“Standardization”): more discussion, edge cases, minor changes
  Moves to next phase when consensus is reached
- Phase 5 (“Feature Standardized”): update to W3C recommendation

Proposals Implementations

	Your browser	Chrome 96	Firefox 90	Safari 15.2	Wasmtime 0.33	Wasmer 2.0
Standardized features						
JS BigInt to Wasm i64 integration	✓	✓	✓	✓	n/a	n/a
Bulk memory operations	✓	✓	✓	✓	✓	✓
Multi-value	✓	✓	✓	✓	✓	✓
Import & export of mutable globals	✓	✓	✓	✓	✓	✓
Reference types	✓	✓	✓	✓	✓	✓
Non-trapping float-to-int conversions	✓	✓	✓	✓	✓	✓
Sign-extension operations	✓	✓	✓	✓	✓	✓
Fixed-width SIMD	✓	✓	✓	✗	✓	✓
In-progress proposals						
Exception handling	✓	✓	✓	✓	✗	✗
Extended constant expressions	✗	✗	⏳	✗	✗	✗
Memory64	✗	✗	⏳	✗	⏳	✗
Multiple memories	✗	✗	✗	✗	⏳	✗
Module Linking	✗	✗	✗	✗	⏳	✗
Relaxed SIMD	✗	✗	⏳	✗	✗	✗
Tail calls	✗	⏳	✗	✗	✗	✗
Threads and atomics	✓	✓	✓	✓	✗	⏳
Type reflection	✗	✗	⏳	✗	✗	✗

Feature Proposals: Accepted Proposals

Accepted proposals will make it to the next standard specification

Mostly focus on important features missing in 1.0

Proposal	Champion	Meeting notes
Import/Export of Mutable Globals	Ben Smith	WG 2018-06-06
Non-trapping float-to-int conversions	Dan Gohman	WG 2020-03-11
Sign-extension operators	Ben Smith	WG 2020-03-11
Multi-value	Andreas Rossberg	WG 2020-03-11
JavaScript BigInt to WebAssembly i64 integration	Dan Ehrenberg & Sven Sauleau	WG 2020-06-09
Reference Types	Andreas Rossberg	WG 2021-02-10
Bulk memory operations	Ben Smith	WG 2021-02-10
Fixed-width SIMD	Deepti Gandluri and Arun Purushan	WG 2021-07-14

Feature Proposals: Proposals In-Progress

Other in-progress proposals are in an earlier phase

Extend CSP with policies specific for WebAssembly

Phase 3 - Implementation Phase (CG + WG)

Proposal	Champion
Tail call	Andreas Rossberg
Multiple memories	Andreas Rossberg
Custom Annotation Syntax in the Text Format	Andreas Rossberg
Memory64	Sam Clegg
Exception handling	Heejin Ahn
Web Content Security Policy	Francis McCabe
Branch Hinting	Yuri Iozzelli
Extended Constant Expressions	Sam Clegg
Relaxed SIMD	Marat Dukhan & Zhi An Ng

Feature Proposals: In-Progress

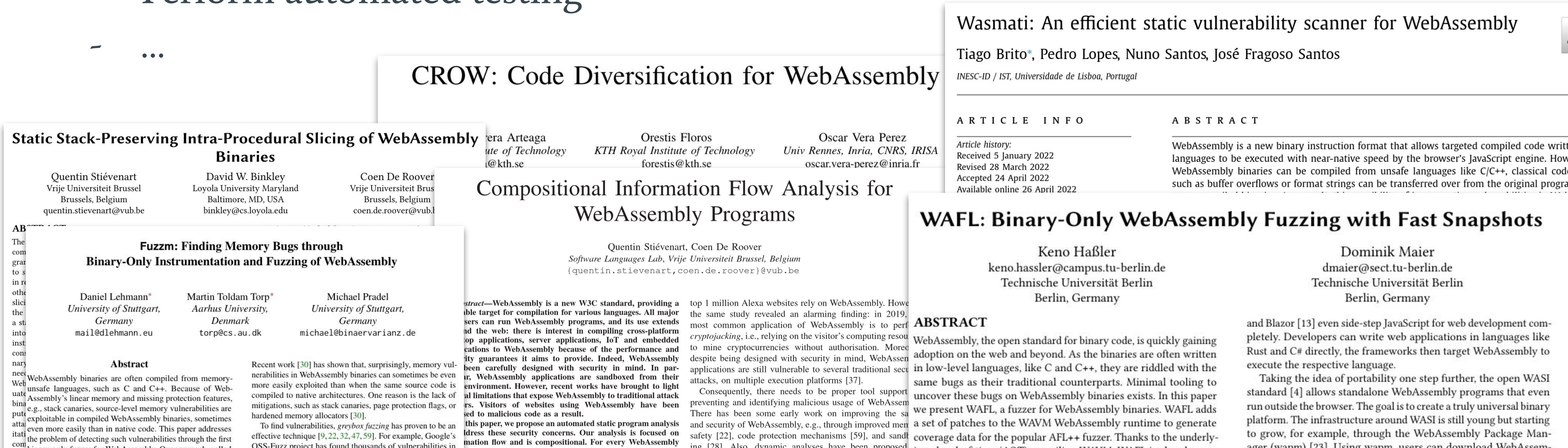
Other noteworthy proposals:

- Threads
- Garbage collection
- Feature detection
- Constant time

Tools for WebAssembly

There is a lot of ongoing research towards tool support for WebAssembly in order to

- Analyze binaries
- Increase their security
- Perform automated testing
- ...



Program Analyses for WebAssembly

Mon 6 Jun

Displayed time zone: Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna change

09:00 - 10:30 PAW Welcome and Keynote at Pine — PAW
Add Session Information

09:00 90m *Keynote* ☆ Andreas Rossberg: WebAssembly 2.0 and Beyond
K: Andreas Rossberg Dfinity Stiftung

11:00 - 12:30 Session 1 at Pine — PAW
Add Session Information

11:00 30m *Talk* ☆ MEWE: Multi-variant Execution for WebAssembly
Javier Cabrera Arteaga KTH Royal Institute of Technology, Martin Monperrus KTH Royal Institute of Technology, Benoit Baudry KTH

11:30 30m *Talk* ☆ Dynamic Analysis for WebAssembly with Wasabi
Daniel Lehmann University of Stuttgart, Michael Pradel University of Stuttgart

12:00 30m *Talk* ☆ A Type System with Subtyping for WebAssembly's Stack Polymorphism
Yasuaki Morita Reykjavik University, Dylan McDermott Reykjavik University, Tarmo Uustalu Reykjavik University

13:30 - 15:00 Session 2 at Pine — PAW
Add Session Information

13:30 30m *Talk* ☆ Wimpl: A Simple IR for Static Analysis of WebAssembly Binaries
Michelle Thalakottur Northeastern University, Daniel Lehmann University of Stuttgart, Frank Tip Northeastern University, Michael Pradel University of Stuttgart

14:00 30m *Talk* ☆ A Modular Static Analysis Platform for WebAssembly
Sebastian Erdweg JGU Mainz, Katharina Brandl JGU Mainz, Sven Keidel TU Darmstadt, Germany

14:30 30m *Talk* ☆ Building Static Analyses for WebAssembly Binaries with Wassail
Quentin Stiévenart Vrije Universiteit Brussel, Coen De Roover Vrije Universiteit Brussel

15:30 - 17:00 Session 3 at Pine — PAW
Add Session Information

15:30 30m *Talk* ☆ SecWasm: Information Flow Control for WebAssembly
Iulia Bastys Chalmers University of Technology, Maximilian Algehed Chalmers University of Technology, Sweden, Alexander Sjösten TU Wien, Andrei Sabelfeld Chalmers University of Technology

16:00 30m *Talk* ☆ Static Execution Costs of WebAssembly Functions
John Shortt Carleton University, Anil Somayaji Carleton University, Amy Felty University of Ottawa

16:30 30m *Day closing* ☆ Open Discussion on Program Analyses for WebAssembly

Security of WebAssembly Applications

Quentin Stiévenart, Vrije Universiteit Brussel
SecAppDev 2022

@acieroid

quentin.stievenart@vub.be