Program Analysis of WebAssembly Applications

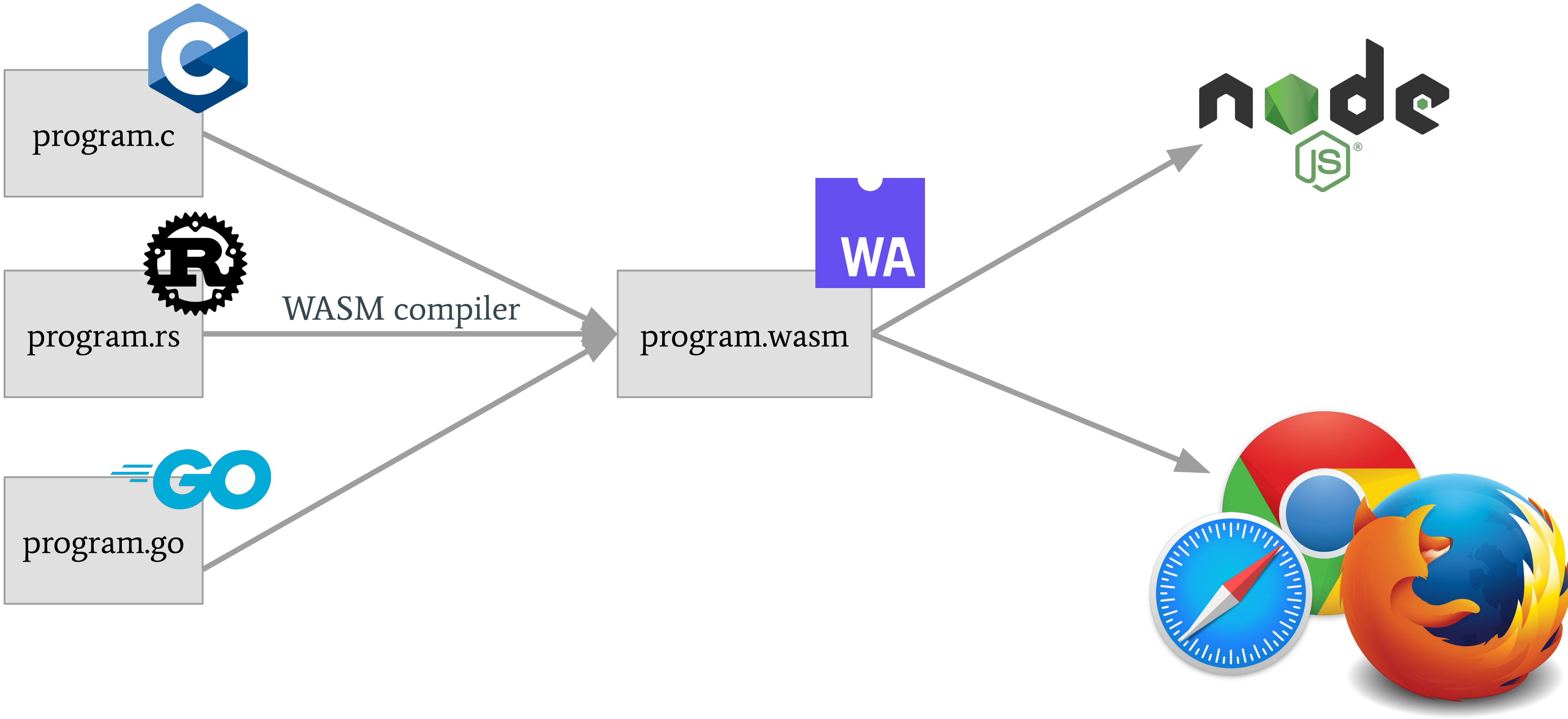
Quentin Stiévenart

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/

WebAssembly Usage in a Nutshell



WebAssembly Compilation

Example at: https://mbebenita.github.io/WasmExplorer/

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 byte-code 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.

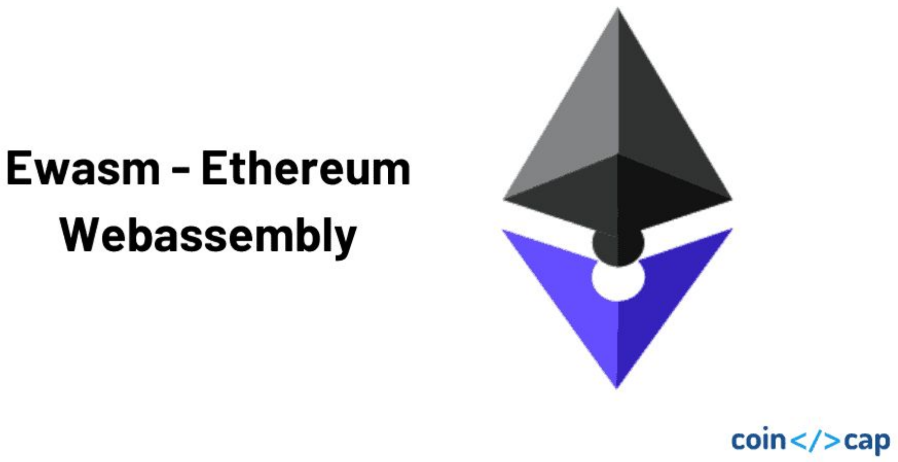
A runtime also assists a WebAssembly program with system call operations (e.g., networking or file access). Specifi-

Wen and Weber, PerCom 2020

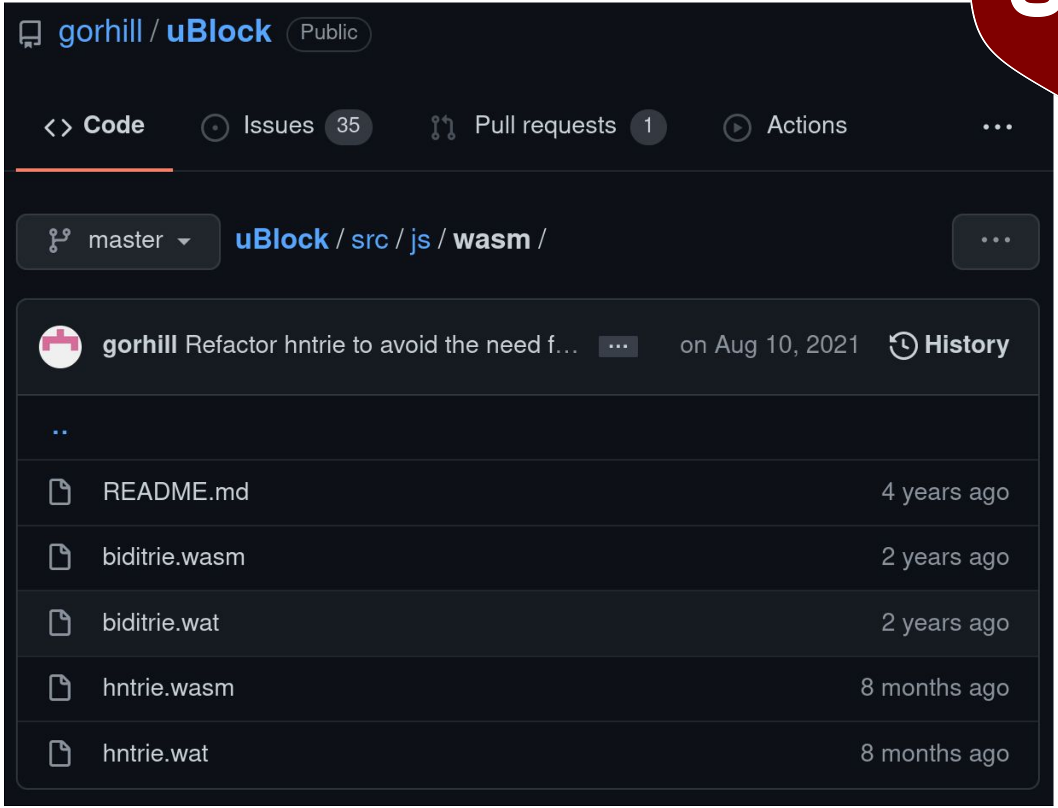
Today’s Use of WebAssembly: Embedded Systems

Gurdeep Singh and Scholliers, MPLR’19

Today’s Use of WebAssembly: Smart Contract Platforms



Today's Use of WebAssembly: Browser Add-Ons



WebAssembly Support

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										

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

Performance

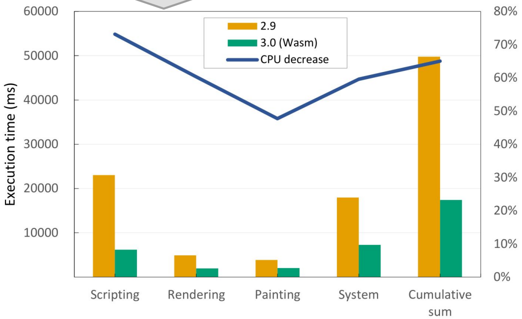
Plenty of room for improvements, while JS engines have been heavily optimized

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 ↑

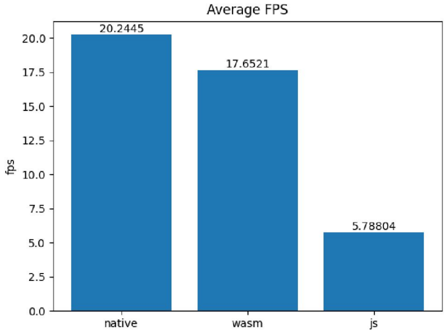
Wang, Weihang. "Empowering Web Applications with WebAssembly: Are We There Yet?." 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021.

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



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

On a raytracer

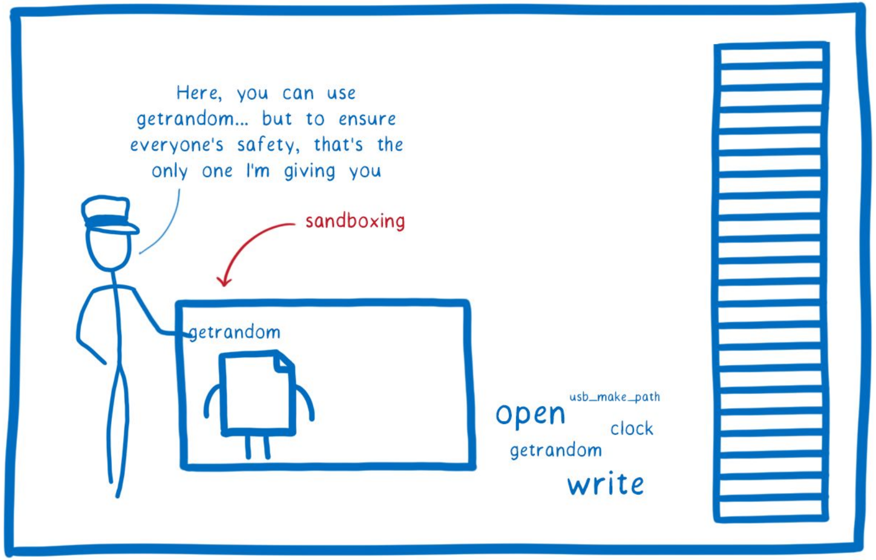


Johansson, Ludwig. "Ray tracing in WebAssembly, a comparative benchmark." (2022).

Secure Design of WebAssembly: Sandboxing

Applications are sandboxed

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

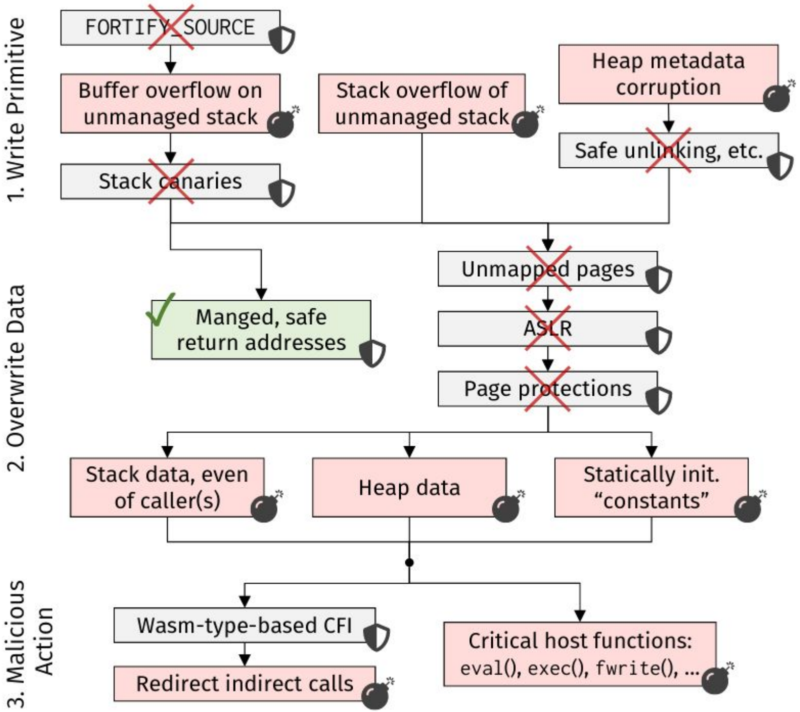


Clark, Lin. "Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly" (2019)

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?



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: 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).

```
// 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
```

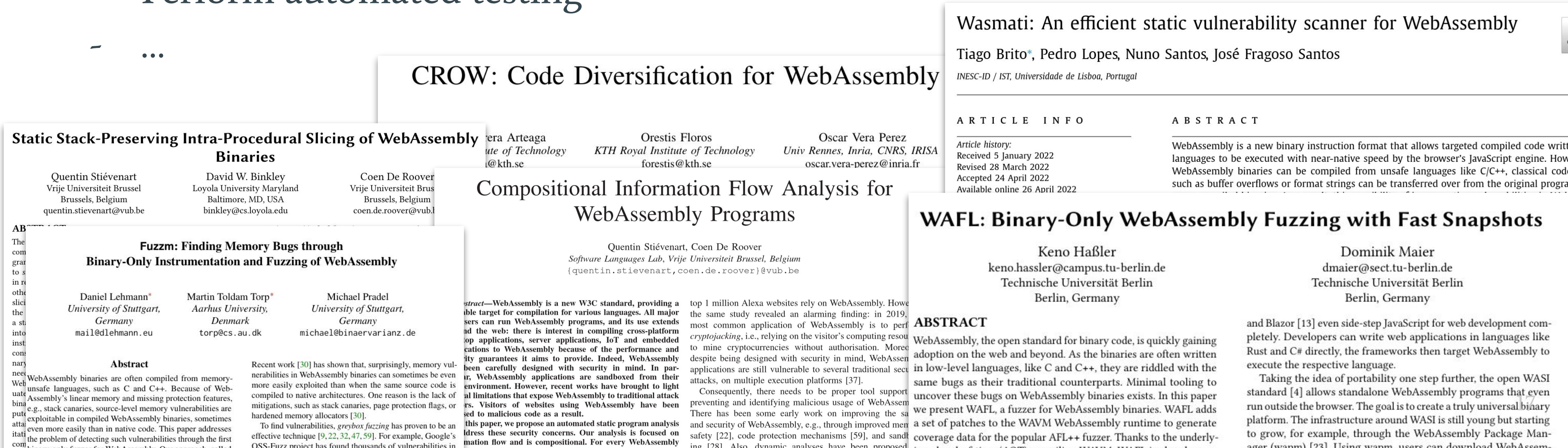
```
(data (i32.const 65536) "%[^\0a]\00
                        file.txt\00a\00
                        Append constant text.\00...")
```

Read-only in native code
Can be overwritten in WASM

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
- ...



Simplicity of WebAssembly: Size of the Specification

WebAssembly core is a small, well-defined standard

Semantics defined formally, along with a reference implementation

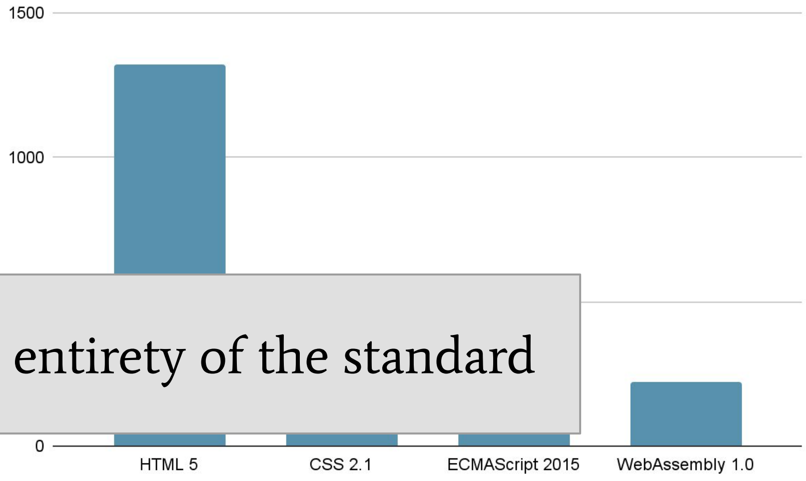
local.get x

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.

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

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

👍 Feasible to support the entirety of the standard



Specification size (number of pages)

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

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
```

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

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))
```

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

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

Design of WebAssembly: Indirect Function Calls

```
(func (;0;) (type 0) (param i32) (result i32)
  local.get 0
  i32.load
  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

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

👍 Less branching points in static analysis

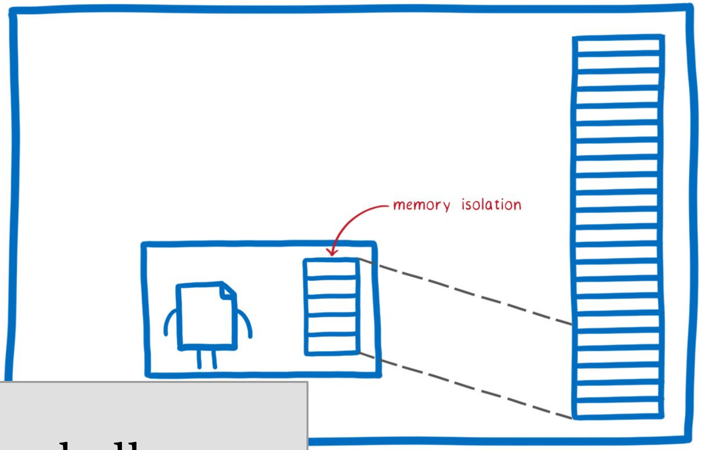
Design of WebAssembly: Memory Model

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
)
```



ecode Alliance: Building a secure by
VebAssembly" (2019)

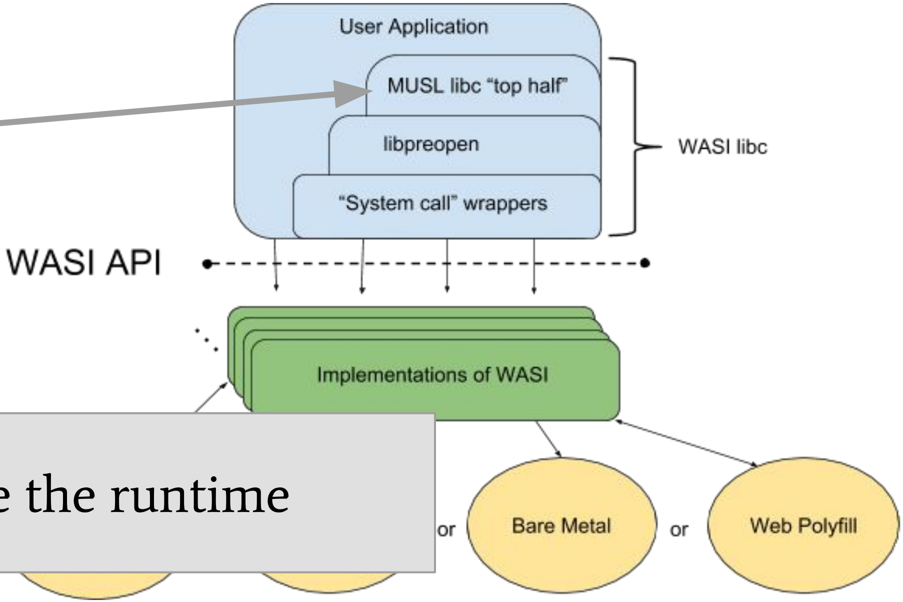
👎 Pointer analysis remains a challenge

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");
}
```



👎 Need to analyze the runtime

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)
  …)
```

👎 Need to support multi-lingual applications

```
turn x + y; } }
};
```

Wassail: WebAssembly Static Analysis and Inspection Library

https://github.com/acieroid/wassail

Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries

Quentin Stiévenart
Vrije Universiteit Brussel
Brussels, Belgium
quentin.stievenart@vub.be

David W. Binkley
Loyola University Maryland
Baltimore, MD, USA
binkley@cs.loyola.edu

Coen De Roover
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

ABSTRACT

The recently introduced WebAssembly standard aims to be a portable compilation target, enabling the cross-platform distribution of programs written in a variety of languages. We propose an approach to *slice* WebAssembly programs in order to enable applications in reverse engineering, code comprehension, and security among others. Given a program and a location in that program, program slicing produces a minimal version of the program that preserves the behavior at the given location. Specifically, our approach is a static, intra-procedural, backward slicing approach that takes into account WebAssembly-specific dependences to identify the instructions of the slice. To do so it must correctly overcome the considerable challenges of performing dependence analysis at the binary level. Furthermore, for the slice to be executable, the approach needs to ensure that the stack behavior of its output complies with WebAssembly's validation requirements. We implemented and evaluated our approach on a suite of 8 386 real-world WebAssembly binaries, finding that the average size of the 495 204 868 slices computed is 53% of the original code, an improvement over the 60% attained by related work slicing ARM binaries. To gain a more qualitative understanding of the slices produced by our approach, we

WebAssembly [25] "*is a binary instruction format for a stack-based virtual machine*" [65] designed as a compilation target for high-level languages. The specification of its core has been a W3C standard since December 2019 [49]. WebAssembly was designed for the purpose of embedding binaries in web applications in a portable manner, thereby enabling intensive computations on the web. A 2021 empirical study by Hilbig et al. [30] found use cases on the web as diverse as game engines, natural language processing, and media players. Thanks to its ability to incorporate runtime functions exported by the host environment, WebAssembly has also found usage beyond web applications, broadening the value of analyses for WebAssembly. Examples include desktop applications [63], smart contracts [19], IoT back ends [27], and embedded software [52].

Program slicing [12, 66] is a program decomposition technique that, based on a specific program point called the *slicing criterion*, identifies a subprogram of the code relevant to the slicing criterion. Program slicing has numerous applications, in debugging [32, 37, 67], program comprehension [11, 16, 31, 36, 59], software maintenance [23, 26], re-engineering [14], refactoring [20], testing [4, 28, 29], reverse engineering [2, 3], tierless or multi-tier programming [45, 46], and vulnerability detection [50].

Compositional Information Flow Analysis for WebAssembly Programs

Quentin Stiévenart, Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{quentin.stievenart,coen.de.roover}@vub.be

Abstract—**WebAssembly is a new W3C standard, providing a portable target for compilation for various languages. All major browsers can run WebAssembly programs, and its use extends beyond the web: there is interest in compiling cross-platform desktop applications, server applications, IoT and embedded applications to WebAssembly because of the performance and security guarantees it aims to provide. Indeed, WebAssembly has been carefully designed with security in mind. In particular, WebAssembly applications are sandboxed from their host environment. However, recent works have brought to light several limitations that expose WebAssembly to traditional attack vectors. Visitors of websites using WebAssembly have been exposed to malicious code as a result.**

In this paper, we propose an automated static program analysis to address these security concerns. Our analysis is focused on information flow and is compositional. For every WebAssembly function, it first computes a summary that describes in a sound manner where the information from its parameters and the global program state can flow to. These summaries can then be applied during the subsequent analysis of function calls.

top 1 million Alexa websites rely on WebAssembly. However, the same study revealed an alarming finding: in 2019, the most common application of WebAssembly is to perform *cryptojacking*, i.e., relying on the visitor's computing resources to mine cryptocurrencies without authorisation. Moreover, despite being designed with security in mind, WebAssembly applications are still vulnerable to several traditional security attacks, on multiple execution platforms [37].

Consequently, there needs to be proper tool support for preventing and identifying malicious usage of WebAssembly. There has been some early work on improving the safety and security of WebAssembly, e.g., through improved memory safety [22], code protection mechanisms [59], and sandboxing [28]. Also, dynamic analyses have been proposed for detecting cryptojacking [16], [67] or for performing taint tracking [25], [60]. However, not a single *static* analysis for WebAssembly has been proposed so far.