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

- program.c
- program.rs
- program.go

WASM compiler

program.wasm

Browser support:
- Chrome
- Firefox
- Safari
- Edge

Node.js
WebAssembly Compilation

Example at: https://mbebenita.github.io/WasmExplorer/
Today’s Use of WebAssembly: Web Applications
Today’s Use of WebAssembly: IoT

Wasmachine: Bring IoT up to Speed with A WebAssembly OS

Elliott Wen
The University of Auckland
gwen929@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 OSes 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 1 (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-
Today’s Use of WebAssembly: Embedded Systems

Gurdeep Singh and Scholliers, MPLR’19
Today’s Use of WebAssembly: Smart Contract Platforms

Ewasm - Ethereum Webassembly

coin/>cap
Today’s Use of WebAssembly: Browser Add-Ons

- gorhill / uBlock
  - Issues: 35
  - Pull requests: 1
  - Actions

- master
  - uBlock / src / js / wasm /

- gorhill Refactor hntrie to avoid the need f… on Aug 10, 2021

- README.md 4 years ago
- biditrie.wasm 2 years ago
- biditrie.wat 2 years ago
- hntrie.wasm 8 months ago
- hntrie.wat 8 months ago
**WebAssembly Support**

https://caniuse.com/wasm

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

[Chart showing WebAssembly support across different browsers and devices, with usage statistics and percentage of users.]
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
- Java
- 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
- Web [Unmaintained]
- Wafl [Unmaintained]
- Wasm [Unmaintained]
- Wase
- WebAssembly
- Wreacket [Unmaintained]
- Zig
Performance

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

As input size increases, JS becomes faster (JIT)

<table>
<thead>
<tr>
<th>Input Size</th>
<th>SD (n)</th>
<th>SD gmean(^a)</th>
<th>SU (n)</th>
<th>SU gmean(^a)</th>
<th>All gmean(^a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra-small</td>
<td>0</td>
<td>0x ↓</td>
<td>30</td>
<td>35.30x ↑</td>
<td>35.30x ↑</td>
</tr>
<tr>
<td>Small</td>
<td>1</td>
<td>1.53x ↓</td>
<td>29</td>
<td>8.33x ↑</td>
<td>7.67x ↑</td>
</tr>
<tr>
<td>Medium</td>
<td>17</td>
<td>1.53x ↓</td>
<td>13</td>
<td>3.68x ↑</td>
<td>1.38x ↑</td>
</tr>
<tr>
<td>Large</td>
<td>15</td>
<td>1.67x ↓</td>
<td>15</td>
<td>1.16x ↑</td>
<td>0.83x ↑</td>
</tr>
<tr>
<td>Extra-large</td>
<td>17</td>
<td>1.22x ↓</td>
<td>13</td>
<td>1.08x ↑</td>
<td>0.92x ↑</td>
</tr>
</tbody>
</table>


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

On a raytracer
Secure Design of WebAssembly: Sandboxing

Applications are sandboxed

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

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

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

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

```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
```


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
...

CROW: Code Diversification for WebAssembly

Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries

Quentin Stívelmann
Vrije Universiteit Brussel
Brussel, Belgium
quentin.stivelmann@vub.ac.be

David W. Bailey
Loyola University Maryland
Baltimore, MD, USA
baillyd@loyola.edu
Cees De Roover
Vrije Universiteit Brussel
Brussel, Belgium
cees.de.roover@vub.ac.be

Orestis Floros
KTH Royal Institute of Technology
foretis@kth.se
Oscar Vera Perez
Univ Rennes, Inria, CNRS, IRISA
oscars-vera-perez@inria.fr

Compositional Information Flow Analysis for WebAssembly Programs

Quentin Stívelmann, Cees De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
quentin.stivelmann, cees.de.roover@vub.be

Fuzzing: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly

Daniel Lehmann
University of Stuttgart, Germany
daniel.lehmann@kit.edu

Martin Tuhola Top
Aalto University, Helsinki, Finland
top@aalto.fi

Michael Pradel
University of Stuttgart, Germany
michael.pradel@uni-stuttgart.de

WebAssembly binaries are often compiled from memory-safe languages, such as C and C++. Because of WebAssembly’s binary memory and instruction protection features, they can be considered as compiled WebAssembly binaries, sometimes even more securely, than the same source code is compiled to native architectures. One reason is the lack of certain features, such as stack canaries, page protection flags, or hardened memory allocations [10]. To that end, this paper proposes an automated state program analysis for finding software defects. This paper describes an effective technique [9, 12, 22, 47, 59]. For example, Google’s Näsijärvi et al compression of vulnerabilities in

Recent work [10] has shown that, surprisingly, memory vulnerabilities in WebAssembly binaries can sometimes be even more easily exploited than when the same source code is compiled to native C code, and more easily exploited than when the same source code is compiled to native architectures. One reason is the lack of certain features, such as stack canaries, page protection flags, or hardened memory allocations [10]. To that end, this paper proposes an automated state program analysis for finding software defects. This paper describes an effective technique [9, 12, 22, 47, 59]. For example, Google’s Näsijärvi et al compression of vulnerabilities in

1- Million Alexa websites rely on WebAssembly. How
same study revealed an alarming finding: In 2019, over two-thirds of WebAssembly applications are exposed to external attacks, including buffer overflows, use of null pointers, and injection attacks.

In this paper, we propose an automated state program analysis for finding software defects. This paper describes a technique [9, 12, 22, 47, 59]. For example, Google’s Näsijärvi et al. compression of vulnerabilities in

In this paper, we propose an automated state program analysis for finding software defects. This paper describes an effective technique [9, 12, 22, 47, 59]. For example, Google’s Näsijärvi et al.

WAFI: Binary-Only WebAssembly Fuzzing with Fast Snapshots

Keno Haßler
tna hassler@campus.tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Dominik Maier
dmaier@tue-berlin.de
Technische Universität Berlin
Berlin, Germany

Wasmati: An efficient static vulnerability scanner for WebAssembly
Tiago Brito, Pedro Lopes, Nuno Santos, José Fragoso Santos
INESC-ID / IST, Universidade de Lisboa, Portugal

Article Info
Article history:
Received 5 January 2022
Revised 26 March 2022
Accepted 24 April 2022
Available online 30 April 2022

ABSTRACT
WebAssembly is a new binary instruction format that allows targeting compiled code with languages to be executed with native speed by the browser’s JavaScript engine. How
WebAssembly binaries can be compiled from unsafe languages like C and, classical code
such as buffer overflows or format strings can be transferred over from the original program

WAFI: Binary-Only WebAssembly Fuzzing with Fast Snapshots
Keno Haßler
keno.haslier@campus.tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Dominik Maier
dmaier@tue-berlin.de
Technische Universität Berlin
Berlin, Germany

Wasmati: An efficient static vulnerability scanner for WebAssembly
Tiago Brito, Pedro Lopes, Nuno Santos, José Fragoso Santos
INESC-ID / IST, Universidade de Lisboa, Portugal

Article Info
Article history:
Received 5 January 2022
Revised 26 March 2022
Accepted 24 April 2022
Available online 30 April 2022

ABSTRACT
WebAssembly is a new binary instruction format that allows targeting compiled code with languages to be executed with native speed by the browser’s JavaScript engine. How
WebAssembly binaries can be compiled from unsafe languages like C and, classical code
such as buffer overflows or format strings can be transferred over from the original program

WAFI: Binary-Only WebAssembly Fuzzing with Fast Snapshots
Keno Haßler
keno.haslier@campus.tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Dominik Maier
dmaier@tue-berlin.de
Technische Universität Berlin
Berlin, Germany

Wasmati: An efficient static vulnerability scanner for WebAssembly
Tiago Brito, Pedro Lopes, Nuno Santos, José Fragoso Santos
INESC-ID / IST, Universidade de Lisboa, Portugal

Article Info
Article history:
Received 5 January 2022
Revised 26 March 2022
Accepted 24 April 2022
Available online 30 April 2022

ABSTRACT
WebAssembly is a new binary instruction format that allows targeting compiled code with languages to be executed with native speed by the browser’s JavaScript engine. How
Simplicity of WebAssembly: Size of the Specification

WebAssembly core is a small, well-defined standard

Semantics defined formally, along with a reference implementation

Feasible to support the entirety of the standard
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
WebAssembly has no instruction for arbitrary jumps

Local control-flow instructions:
- Scopes: block, loop, if
- Jumps: br, br_if, br_table
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.
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

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:

- ✔ Local jumps (if, br, ...)
- ✔ Direct function calls
- ✔ Function returns
- 🚫 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
)

👎 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");
}
```
WebAssembly in Practice: Interfacing with JavaScript

WebAssembly object provides way of interacting with WebAssembly

```javascript
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; } }
};
```

👎 Need to support multi-lingual applications
Wassail: WebAssembly Static Analysis and Inspection Library

Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries

Quentin Stévenart
Veaje Universiteit Brussel
Brussels, Belgium
quentin.stevenart@vub.be

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

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

WebAssembly [20], “is a binary instruction format for a stack-based virtual machine” [21], designed as a compilation target for high-level languages. The specification of its core has been a W3C standard since December 2019 [22]. WebAssembly was designed for the purpose of embedding browsers in web applications as a portable manner, thereby enabling intensive computation on the Web. A 2021 empirical study by Billing et al. [23] found our tools on the web as diverse as game engines, natural language processing, and media players. There is also a notable growth in the ecosystem of tools and libraries available to developers. WebAssembly has also seen varied research applications, including the value of analysis for WebAssembly. Examples include static analysis [24], smart contracts [25], WebAssembly stacks [26], and embedded software [27].

Program slicing [28, 34] is a program comprehension technique that, based on a specific program point, selects the source code traces to a specific variable. Program slicing has numerous applications, including automated testing, static analysis, and debugging. While program slicing is an important study area, other research issues exist. For example, Wasm instrumentation [25, 36], WebAssembly debugging [37], and data flow analysis [38] are important research issues in the area of WebAssembly compilation.

The design and implementation of this toolsingle tool produced in our approach, we have

Abhijit, WebAssembly is a new W3C standard, providing a portable target for compilation for various languages. All major browsers can run WebAssembly programs, and its implementation beyond the web: there is interest in embedding cross-platform development applications, server applications, and embedded applications. WebAssembly, because of its performance and portability, is more appealing compared to other solutions. In recent years, WebAssembly has been carefully designed with security in mind. In particular, WebAssembly applications are sandboxed from their host environment. However, recent work has brought in light several limitations that impose WebAssembly to traditional attack vectors. Volumes of malware 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 based on the concept of information flow analysis, which tracks the flow of information through the program. In this section, we present our approach to information flow analysis. Our approach is based on the concept of program slicing. In this approach, we dynamically instrument WebAssembly code to enable information flow analysis. We then execute the program and analyze the slices produced. Our tool, called WasmSlice, provides a way to analyze WebAssembly programs and detect potential security vulnerabilities. WasmSlice is implemented as a tool that instrumented WebAssembly code with additional slices. WasmSlice can be used to analyze WebAssembly programs and detect potential security vulnerabilities. WasmSlice is implemented as a tool that instrumented WebAssembly code with additional slices. WasmSlice can be used to analyze WebAssembly programs and detect potential security vulnerabilities.