Dynamically Analyzing WebAssembly with Wasabi

Daniel Lehmann and Michael Pradel
(TU Darmstadt)
We will offer a tutorial on how to use Wasabi for dynamically analyzing WebAssembly at PLDI 2019!

**Introduction to Wasabi**

Wasabi is a dynamic analysis framework for WebAssembly. What does this mean and why is it useful?

- Dynamic analysis means observing some properties of a program while it is running. Dynamic analysis is routinely used to find and fix bugs, identify performance bottlenecks, or to search for security problems.
- Analysis frameworks, such as Pin, Valgrind, or Jaeger, provide generic APIs on top of which many different analyses can be implemented. They solve low-level problems, e.g., how the analysis is interleaved with the program, once and for all, and let analysis authors focus on the analysis itself.
- WebAssembly is a new programming language (or more precisely, byte code) for browsers. It is easier, more low-level, and a better compilation target than JavaScript, the only option on the client side until now. We expect it to be hugely successful in the near future.

(Click on the bullets for more information. Simplified for the sake of giving a quick intuition.)

**How Does It Work?**

**Conceptually**

The name Wasabi stands for WebAssembly analysis using binary instrumentation, which hints at the two phases Wasabi operates in:

1. It statically instruments a WebAssembly binary (e.g., `program.wasm`). That is, it inserts additional instructions, such as function calls, in between the original instructions of the program. This happens before the execution. Since WebAssembly is a binary format, and to be independent of the source code (which is often not available when analyzing third-party code in websites), we directly modify the byte code.
2. To perform the dynamic analysis, the program is then executed (by opening the website with the now instrumented `program.wasm`). To make the user-written...
Tutorial Goals

- Tell you (a bit) about WebAssembly and Wasabi
- Set-up Wasabi and show that it’s easy “to get going”
- Show examples of what you can do
- Get feedback from (potential) users

→ Active participation and interaction 😊
Part 1:
Introduction to WebAssembly
What is WebAssembly?

Bringing the Web up to Speed with WebAssembly

Andreas Haas  Andreas Rossberg  Derek L. Schuff  Ben L. Titzer
     Google GmbH, Germany / “Google Inc, USA
     {ahaas,rossberg,dschuff,ttitzer}@google.com

Dan Gohman  Luke Wagner  Alon Zakai
     Mozilla Inc, USA
     {sunfishcode,luke,azakai}@mozilla.com

Michael Holman
     Microsoft Inc, USA
     michael.holman@microsoft.com

JF Bastien
     Apple Inc, USA
     jfbastien@apple.com

Abstract

The maturation of the Web platform has given rise to sophisticated and demanding Web applications such as interactive 3D visualization, audio and video software, and games. With that, efficiency and security of code on the Web has become more important than ever. Yet JavaScript as the only built-in language of the Web is not well-equipped to meet these device types. By historical accident, JavaScript is the only natively supported programming language on the Web, its widespread usage unmatched by other technologies available only via plugins like ActiveX, Java or Flash. Because of JavaScript’s ubiquity, rapid performance improvements in modern VMs, and perhaps through sheer necessity, it has become a compilation target for other languages. Through Emscripten 1.431, even C and C++ programs can be compiled to WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

Haas et al., PLDI 2017  https://webassembly.org/
What is WebAssembly?

● “Byte code for the web”
  ○ Very rough idea: “JVM, without 3rd party plugins, directly in the browser”
  ○ Lower-level than JavaScript

● Fast
  ○ Compact binary format → quicker to parse
  ○ Instructions map closely to common hardware
  ○ Linear memory, no garbage collector

● Safe
  ○ Typed, separated code and data, modules, ...

● Portable
  ○ Support by all 4 major browsers, x86/ARM
Use Cases

● Designed as compilation target
  ○ For low-level languages
  ○ From C, C++ with Emscripten
  ○ Rust, Go, …

● Application domains
  ○ Audio/video processing, codecs, e.g., AV1
  ○ Compression, e.g., Brotli
  ○ Games, simulations, e.g., Unreal Engine 4
  ○ Language runtimes, e.g., Blazor (C#)
  ○ Complex web applications, e.g., AutoCAD

Binary and Text Format

https://webassembly.studio/
Text Format: Overview

- Syntax: S-expressions
- 1 file = 1 module
- 1 module : n sections
  - Types
  - Imports/Exports
  - Functions
  - Globals
  - Data

```s
(module
  (type $t0 (func (param i32 i32) (result i32)))
  ...
  (import "env" "b" (func $env.b (type $t4)))
  ...
  (func $f7 (param i32 i32) (result i32)
    (local $l0 i32) ...
    get_global $g2
    i32.const 48
    i32.add
    set_global $g2
    ...
    i32.load
    ...
    (global $g2 (mut i32))
    (export "h" (func $h))
    (data (i32.const 1152) "Hello, world!")
  )
)```
- Code is organized in functions
  - No OOP: no objects or methods
- Globals, per-function locals
- Stack machine
- 4 types: i32, i64, f32, f64
- Linear, unmanaged memory
  - 32-bit addresses

```rml
(module
  (type $t0 (func (param i32 i32) (result i32)))
  ...
  (import "env" "b" (func $env.b (type $t4)))
  ...
  (func $f7 (param i32 i32) (result i32)
    (local $l0 i32) ...
    get_global $g2
    i32.const 48
    i32.add
    set_global $g2
    ...
    i32.load
    ...
    (global $g2 (mut i32))
    (export "h" (func $h))
    (data (i32.const 1152) "Hello, world!")
  )
)
```
Structured Control-Flow

- Well-nested blocks
  - Branches \( (br) \) reference blocks by numerical label

- Block types: block, if, loop

```mermaid
diagram workflow
    block
    | ...  | 0
    | br_if | ...  | 0
    | end   | ...  | ...  | 0

    ;; infinite loop
    loop
    | ...  | ...  | 0
    | ...  | ...  | end |

    if
    | ...  | ...  | 0
    | else |
    | ...  | ...  | end |
    | ...  | ...
```

\[ \text{i32.const \ 0} \]
Host Embedding

- WebAssembly itself is agnostic of its *host*
  - No functions defined by default
  - No system calls/IO routines by default

- Most common embedding: web browsers
WebAssembly API in JavaScript

```javascript
let importObject = {
  env: {
    print: function(s) {
      console.log(s);
    },
  }
};

fetch("hello.wasm")
  .then(response => response.arrayBuffer())
  .then(buffer => WebAssembly.instantiate(buffer, importObject))
  .then(object => {
    let inputs = ...;
    let result = object.instance.exports.somefun(inputs);
    ...
  });
```
WebAssembly API in JavaScript

● WebAssembly global object

● Usual lifecycle:
  ○ Load .wasm file with fetch()
  ○ Instantiate the file = compile and provide imports
  ○ Call exported functions on the instance

● Instantiation methods
  ○ Blocking: new WebAssembly.Instance(...)
  ○ Async: await WebAssembly.instantiate(buffer, imports)
  ○ Streaming: await WebAssembly.instantiateStreaming(fetch(...), imports)
Tooling and Documentation

- **Emscripten: C → WebAssembly compiler**
  - [https://emscripten.org/](https://emscripten.org/)

- **WebAssembly Binary Toolkit**
  - [https://github.com/WebAssembly/wabt](https://github.com/WebAssembly/wabt)
  - wat2wasm
  - wasm2wat
  - wasm-objdump

- **WebAssembly Specification**
  - [https://webassembly.github.io/spec/](https://webassembly.github.io/spec/)
  - E.g., instructions: [https://webassembly.github.io/spec/core/bikeshed/index.html#instructions](https://webassembly.github.io/spec/core/bikeshed/index.html#instructions)
Setup for Hands-On Tasks

- Download VirtualBox [https://www.virtualbox.org/](https://www.virtualbox.org/)
- Copy or download VM image
  - USB stick
- “Import Appliance”
- $ cd ~/tutorial/wasabi/
  $ git pull
  $ cd ./tutorial/
Task 0.1: A Minimal WebAssembly Program

● Goals
  ○ Understand the text format
  ○ Convert between text and binary format
  ○ Embed a WebAssembly binary in a website
  ○ Run it in the browser

● Prerequisites
  ○ WebAssembly Binary Toolkit: https://github.com/WebAssembly/wabt

● Instructions
  ○ See README in https://github.com/danleh/wasabi/tree/master/tutorial/task0/1-hello
Task 0.2: Addition in WebAssembly

● **Goals**
  ○ Write a more useful function in WebAssembly
  ○ How to call WebAssembly functions from JavaScript
  ○ Bonus: WebAssembly i32 != JavaScript number

● **Prerequisites**
  ○ WebAssembly Binary Toolkit: [https://github.com/WebAssembly/wabt](https://github.com/WebAssembly/wabt)

● **Instructions**
  ○ See README in [https://github.com/danleh/wasabi/tree/master/tutorial/task0/2-add](https://github.com/danleh/wasabi/tree/master/tutorial/task0/2-add)
Task 0.3: Compiling C to WebAssembly

● Goals
  ○ Learn about Emscripten
  ○ Compile C code to WebAssembly + HTML harness with Emscripten
  ○ See that you can “run C code in the browser”

● Prerequisites
  ○ Emscripten: https://emscripten.org/docs/getting_started/downloads.html

● Instructions
  ○ See README in https://github.com/danleh/wasabi/tree/master/tutorial/task0/3-hello-c
Part 2: Introduction to Wasabi
Dynamic Analysis Frameworks

- New platform → Need for dynamic analysis tools
  
  ![Diagram]

  Correctness  Performance  Security

- Framework as a basis

<table>
<thead>
<tr>
<th></th>
<th>Pin</th>
<th>Valgrind</th>
<th>RoadRunner</th>
<th>Jalangi</th>
<th>Wasabi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>x86-64</td>
<td>JVM</td>
<td></td>
<td>JavaScript</td>
<td>WebAssembly</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>native binaries</td>
<td>byte code</td>
<td>source code</td>
<td>binary code</td>
<td></td>
</tr>
<tr>
<td>Analysis Language</td>
<td>C/C++</td>
<td>Java</td>
<td>JavaScript</td>
<td>JavaScript</td>
<td>JavaScript</td>
</tr>
</tbody>
</table>
Wasabi: Dynamic Analysis Framework for WebAssembly

- Observe any operation
- Analysis API in JavaScript
- Static, binary instrumentation
  - Why binary? Why static?
    - Different producers of WebAssembly
    - Source code not always available
    - Static instrumentation is reliable
- Open source: [http://wasabi.software-lab.org](http://wasabi.software-lab.org)
Instrument and Analyze

Before

program .wasm

Steps

write analysis

instrumented

run Wasabi

generated

After

analysis.js

program .wasm

wasabi.js

website .html

add analysis.js and wasabi.js

website .html
Analysis API: Overview

- Every instruction can be observed
  - Location, inputs, outputs
- Grouping of similar instructions
  - Similar instructions have a single API hook
  - 23 hooks instead of >100
- Statically pre-computed information
  - E.g., resolve relative branch targets
- Type mapping:
  - WebAssembly → JavaScript

![Type Mapping Diagram]

```
f32
f64
i32
i64
```

```
number
boolean
long.js object
```
Example Analysis: Detecting Cryptominers

- Dynamic analysis to identify cryptomining code
- 11-line re-implementation of Wang et al. [ESORICS’18]
- Gather instruction profile
- No manual instrumentation

```javascript
let inst = {};
Wasabi.binary = function(loc, op, args) {
    switch (op) {
        case "i32.add":
        case "i32.and":
        case "i32.shl":
        case "i32.shr_u":
        case "i32.xor":
            inst[op] = (inst[op] || 0)+1;
    }
};
```
Analysis API: Memory Operations

- **Stack**: Manipulation of the value stack
  - const: Constant is pushed onto the stack
  - drop: Top-of-stack value is discarded
  - select: Conditional push of one of two given values

- **Heap**: Manipulation of global heap values
  - load: Load value from memory
  - store: Store value into memory

- **Locals/globals**: Manipulation of variables
  - local: Reads and writes of local variables
  - global: Reads and writes of global variables

Hooks receive: Involved values, memory addresses, and code location
Analysis API: Control Flow

- **Branches**
  - `br, br_if`: Unconditional and conditional branch
  - `br_table`: Branch via lookup table
  - `if`: Enters block if condition is true

  *Hooks receive: Branch target(s), condition, and code location*

- **Function calls**
  - `call_pre`: Just before a function call (at the call site)
  - `call_post`: Just after a function call (at the call site)
  - `return`: Just before a function return (in the callee)

  *Hooks receive: Arguments, return value, function (for direct calls) or function table (for indirect calls), and code location*
Analysis API: Operations

- **unary**: Operations with a single operand
- **binary**: Operations with two operands

*Hooks receive: Input value(s), output value, and code location*
Analysis API: Other Hooks

- **Blocks:**
  - `begin`
  - `end`

  *Hooks receive: type of block (function, loops, block, if, else) and code location*

- **Memory management:**
  - `memory_grow`
  - `memory_size`

  *Hooks receive: Current memory size, delta in size, code location*

- **Others:**
  - `start`, `nop`, `unreachable`
WASM Instructions versus API Hooks

- 0..1 to N mapping from WASM instructions to API hooks
  - Most instructions trigger exactly one hook
  - Some instructions trigger multiple hooks (e.g., if → if + begin of block)
  - Some hooks are triggered without any explicit instruction (e.g., begin of function)
Code Locations

- All hooks receive the corresponding code locations
- Code locations are w.r.t. original binary, not instrumented binary
- Locations consist of two parts:
  - Function index: Unique identifier of a function within a module
  - Instruction index: Sequential order in function body (start at zero)
    - -1 for hooks that do not correspond to any specific instruction in the body

Example:

```javascript
Object { func: 1, instr: -1 } start
Object { func: 1, instr: -1 } begin function
Object { func: 1, instr: 0 } const, value = 42
Object { func: 1, instr: 1 } direct call to func # 0 args = Array [ 42 ]
```
Static Information

In addition to runtime hooks, Wasabi provides some static information about the analyzed WebAssembly module:

- Functions
- Branch tables
- Global variables

Access this information via `Wasabi.module`

Example:
Part 3: Writing Your Own Wasabi Analyses
Task 1: Your First Wasabi Analysis

● Goal
  ○ Build and run a very simple analysis on a minimalistic WebAssembly module

● Prerequisite
  ○ Wasabi installed
    ■ You have it after the previous tasks

● Once Wasabi is installed:
  See https://github.com/danleh/wasabi/tree/master/tutorial/task1
Task 2: Dynamic Call Graph of a 3D Game

● Goals
  ○ Apply Wasabi to a larger program, here: WebAssembly port of C game engine
  ○ Write dynamic call graph analysis, which is often a building block for other analyses

● Prerequisites
  ○ Wasabi installed
    ■ You have it after the previous tasks
  ○ Graphviz
    ■ For Ubuntu: sudo apt install graphviz

● Instructions
  ○ See README in https://github.com/danleh/wasabi/tree/master/tutorial/task2
Task 3: Reverse Engineering

● **Goals**
  ○ Very simple WebAssembly “reverse engineering”
  ○ Why dynamic analysis is sometimes useful/easier

● **Prerequisites**
  ○ wasm2wat (from WebAssembly Binary Toolkit) or just browser developer tools
  ○ Wasabi

● **Instructions**
  ○ See README in [https://github.com/danleh/wasabi/tree/master/tutorial/task3](https://github.com/danleh/wasabi/tree/master/tutorial/task3)