Part 2: Introduction to Wasabi
Dynamic Analysis Frameworks

- New platform → Need for dynamic analysis tools
  - 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
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 to 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
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)

```wasm
(module
  (func $f
    i32.const 3
    i32.const 2
    i32.add
    drop
  )
)

(start $f)
```

- `begin hook`
- `const hook`
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:

```
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](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