Wasm Call Graphs

Jakob Getz
University of Stuttgart / KAIST

Paper

That’s a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly

- Daniel Lehmann University of Stuttgart Stuttgart, Germany
- Frank Tip Northeastern University Boston, MA, USA
- Michelle Thalakottur Northeastern University Boston, MA, USA
- Michael Pradel University of Stuttgart Stuttgart, Germany

What is the paper about?

Constructing Call Graphs from WebAssembly Binaries

Two Sections

1. What **challenges** are there specific to WebAssembly
2. Evaluate existing call graph construction approaches

Outline

1. High level overview on the study
2. Examples on how to construct call graphs for WebAssembly
3. Challenges on call graph construction for WebAssembly
4. Study performed on Microbenchmarks
5. Study performed on Real World Binaries

Outline

1. **High level overview on the study**
2. Examples on how to construct call graphs for WebAssembly
3. Challenges on call graph construction for WebAssembly
4. Study performed on Microbenchmarks
5. Study performed on Real World Binaries

Study Design

```
type Tool = "Wassail" | "WAVM" | "MetaDCE" | "Twiggy"
type Input = WebAssemblyBinary
type Output = { reachableFunc: SetReachableFunc, callGraph: CallGraph? }
type GroundTruth = func(input: Input): Output // Computed manually
type Generate = func(input: Input, tool: Tool): Output
type Results = { sound: boolean, precise: boolean }
type Compare = func(toolOutput: Output, groundTruth: Output): Results
```

Example Workflow

```
const input: Input = "main.wasm"
const groudTruth: Output = getGroundTruth("main.wasm") // computed manually
const toolOutput: Output = generate("main.wasm", "Wassail")
const result: Results = compare(toolOutput, groudTruth)
```

Input

`main.wat` or binary representation `main.wasm`

```
(module
    (func $main (export "main")
        call $reachable
    )
    (func $reachable)
    (func $not-reachable)
)
```

Compute Ground Truth

```
(module
    (func $main (export "main")
        call $reachable
    )
    (func $reachable)
    (func $not-reachable)
)
```

=>

```
{
    reachableFunc: [ "main", "reachable" ],
    callGraph = "main → reachable"
}
```

Ground Truth

```
{
    reachableFunc: [ "main", "reachable" ],
    callGraph: "main → reachable"
}
```

This output is:

- **sound** - contains *all* reachable functions
- **precise** - contains *only* reachable functions

Unsound

```
(module
    (func $main (export "main")
        call $reachable
    )
    (func $reachable)
    (func $not-reachable)
)
```

=>

```
{
    reachableFunc: [ "main" ],
    callGraph: "main"
}
```

Imprecise

```
(module
    (func $main (export "main")
        call $reachable
    )
    (func $reachable)
    (func $not-reachable)
)
```

=>

```
{
    reachableFunc: [ "main", "reachable", "not-reachable" ],
    callGraph: "main → reachable, not-reachable"
}
```

Tools

- **Wassail** Takes input `.wat` file and computes output call graph
- **WAVM+LLVM** Lift the WASM binary to the intermediate representation LLVM IR and give that to the call graph generator WAVM
- **MetaDCE** Takes input `.wasm` file and a list of entry points.
- **Twiggy** Takes input `.wasm` file and computes dominator tree, from which one can extract the Reachable Function Set

Example Workflow

```
const input: Input = "main.wasm"
const groudTruth: Output = getGroundTruth("main.wasm") // computed manually
const toolOutput: Output = generate("main.wasm", "Wassail")
const result: Results = compare(toolOutput, groudTruth)
```

- If our `toolOutput` does not contain all reachable functions as our `groundTruth` the output is unsound
- If our `toolOutput` does contain more reachable functions then our `groundTruth` then the output is imprecise

Outline

1. High level overview on the study
2. **Examples on how to construct call graphs for WebAssembly**
3. Challenges on call graph construction for WebAssembly
4. Study performed on Microbenchmarks
5. Study performed on Real World Binaries

A Second Example

```
(module
    (func $main (export "main")
        i32.const 0
        call_indirect (param) (result)
    )
    (func $a)
    (func $not-reachable)
    (table $table 2 funcref)
    (elem $table (i32.const 0) $a $b)
)
```

Precise call graph:

main -> a

Set reachable functions:

{ main, a }

A Third Example

```
(module
    (func $main (export "main")
        call $index
        i32.load
        call_indirect)
    (func $index (result i32)
        i32.const 1337)
    (func $a)
    (func $not-reachable)
    (memory $memory 1)
    (data $memory (i32.const 1337)
        "\01\00\00\00")
    (table $table 2 funcref)
    (elem $table (i32.const 0)
        $not-reachable $a)
)
```

- One must model the memory
- One must track dataflow through functions

Precise call graph:

main -> index, main -> a

Set reachable functions:

{ main, index, a }

It is getting difficult

```
(module
  (func $main (export "main")
    i32.const 0
    call_indirect)
  (func $export1 (export "export1"))
  (func $export2 (export "export2"))
  (func $not-reachable)
  (table $table (export "table")
    1 funcref)
  (elem $table (i32.const 0) $export1)
)
```

```
const binary = fs
  .readFileSync('./main.wasm');

const result = await WebAssembly
  .instantiate(binary, {});

const exp = result.instance.exports

exp.table.set(0, exp.export2);

exp.main();
```

Outline

1. High level overview on the study
2. Examples on how to construct call graphs for WebAssembly
3. **Challenges on call graph construction for WebAssembly**
4. Study performed on Microbenchmarks
5. Study performed on Real World Binaries

Challenges

There are a lot of challenges like the previous ones.

Our paper identifies `12` challenges ordered into `6` classes.

- Program representation
- Indirect calls and table section
- Types
- Host environment
- Memory
- Source languages

Prevalence in Real World Binaries

Measured on **8.392** binaries from the WasmBench dataset.

Of those binaries...

- 95% have no `.debug` section (*FunctionIndices*)
- 83% have at least one indirect call (*TableIncirection*)
- 92% have at least one imported function (*HostCallbacks*)
- 95% have at least one store instruction (*MemoryMutable*)

Outline

1. High level overview on the study
2. Examples on how to construct call graphs for WebAssembly
3. Challenges on call graph construction for WebAssembly
4. **Study performed on Microbenchmarks**
5. Study performed on Real World Binaries

Microbenchmarks

Our paper introduces a set of **24** microbenchmarks, crafted to reflect the WASM specific challenges.

Example 1

Microbenchmark 1: **Simple direct call**

```
(module
    (func $main (export "main")
        call $reachable
    )
    (func $reachable)
    (func $not-reachable)
)
```

- F_{all}: Set of all functions in the binary
- F_r: Set of reachable functions
- F_e: Set of call edges

Ground Truth

$|F_{all}| = 3, |F_r| = 2, |F_e| = 1$

Tool: Wassail

$|F_r| = 2, |F_e| = 1$, Sound, Precise

Tool: Twiggy

$|F_r| = 2$, Sound, Precise

Example 2

Microbenchmark 14:

Constant table index value

Related to challenges *TableIndexValue* and *TableIndirection*

```
(module
    (func $main (export "main")
        i32.const 0
        call_indirect (param) (result)
    )
    (func $a)
    (func $b)
    (table $table 2 funcref)
    (elem $table (i32.const 0) $a $b)
)
```

Ground Truth

$|F_{all}| = 3, |F_r| = 2, |F_e| = 1$

Tool: Wassail

$|F_r| = 3, |F_e| = 2$, Sound, *Not* Precise

Tool: WAVM+LLVM

$|F_r| = 1, |F_e| = 0$ *Not* Sound, *Not* Precise

Result

- None of the four evaluated tools: **Wassail**, **WAVM+LLVM**, **MetaDCE** and **Twiggy** is sound for all benchmarks.
- None of the tools is precise for all benchmarks.
- The approach **WAVM+LLVM** is unsound for most benchmarks, because lifting WASM to LLVM IR loses crutial information
- Some benchmarks cause the tools to crash
- Common case for imprecision is the challenge class **Indirect calls and table section**

Implications

- Static analysis should likely be performed directly on WASM bytecode
- Future analysis should test against the microbenchmarks
- Future analyses should…
  - use types to constrain indirect call targets
  - track dataflow and memory
  - analyse host code to improve precision

Outline

1. High level overview on the study
2. Examples on how to construct call graphs for WebAssembly
3. Challenges on call graph construction for WebAssembly
4. Study performed on Microbenchmarks
5. **Study performed on Real World Binaries**

Evaluation on Real World Binaries

10 real world binaries collected from NPM

- Ground Truth generation:
  - Exercise each binary with test cases
  - Collect set F_{dyn} (all executed functions by the tests) through dynamic analysis
- Perfom static analysis on the binaries with our tools and report F_r
- Quantify unsoundness: $F_{unsound} = F_{dyn} - F_r$

Result

- All tools but **MetaDCE** are unsound for at least one binary
- **MetaDCE** crashes on two binaries
- **Twiggy** and **MetaDCE** are more conservative in declaring functions as unreachable
- **WAVM+LLVM** is unsound for most binaries

Summary

- First systematic study on challenges in call graph construction for WebAssembly
- How prevalent are those challenges in real world binaries
- How are they handled by existing static analysis
- All exisiting analysis approaches are unsound and imprecise
- Suggestion for future analysis:
  - Rely on WebAssembly bytecode instead of intermediate representation like LLVM IR
  - Take into account dataflow
  - Perform pointer analysis

Thank You