Wasabi:
A Framework for Dynamically Analyzing WebAssembly

http://wasabi.software-lab.org

Daniel Lehmann and Michael Pradel
TU Darmstadt, Germany

TECHNISCHE UNIVERSITÄT DARMSTADT

SOLA SoftwareLab

TL;DR

- **WebAssembly**: bytecode for the web
  - New and important platform
    ➡ Need for tooling

- **Wasabi**: dynamic analysis framework for WebAssembly
  - Observe any operation
  - Analysis API in JavaScript
  - Binary instrumentation
  - Open source: `http://wasabi.software-lab.org`

WebAssembly in a Nutshell 1/2

- [Haas et al., PLDI 2017]

- **Fast**: within 1.5x – 1.9x of native
  - Binary format: compact, quick to parse
  - Instructions map closely to hardware
  - No GC, predictable performance

```
23 09              get_global 9
41 10              i32.const 16
6a                 i32.add
04 40              if
41 8c a7 ed 03       i32.const 8082316
...                  ...
```

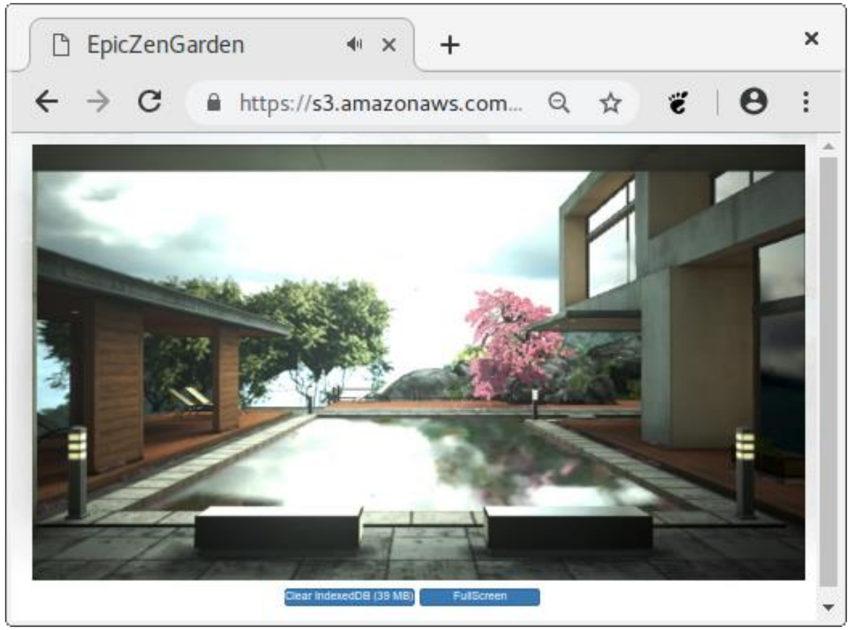
- **Safe**: static types, separated code and data, ...

- **Portable**: all major browsers, ARM/x86

- Designed as a **compilation target**
  - C/C++ via Emscripten
  - Rust, Go, …

```
6100 6d73 0001
0000 7f02 007f
1102 0701 7270
…
```
program.wasm

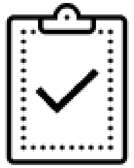
- Many **use cases**:
  - Alternative to JavaScript on the client
  - Audio/video processing, compression, machine learning
  - Games
  - …

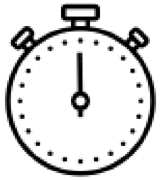


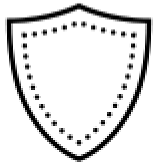
Unreal Engine 4: Zen Garden demo

Dynamic Analysis Frameworks

- New platform ➡ Need for **dynamic analysis** tools

 Correctness

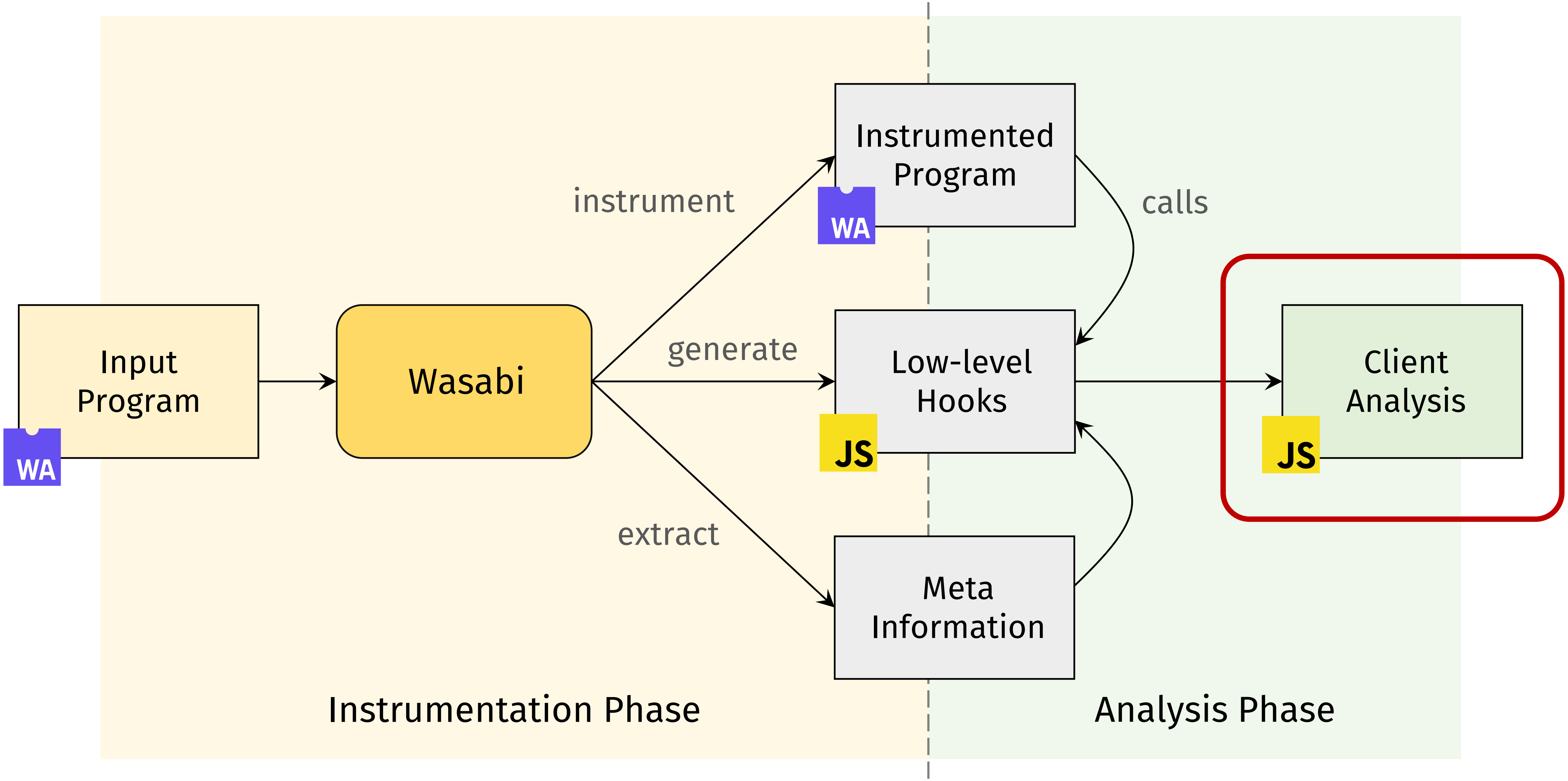
 Performance

 Security

- **Frameworks** as a basis

	Pin	Valgrind	RoadRunner	Jalangi	Wasabi
Platform	x86-64		JVM	JavaScript	WebAssembly
Instrumentation	native binaries		byte code	source code	binary code
Analysis Language	C/C++		Java	JavaScript	JavaScript

Wasabi Overview



Client Analysis Example

- Analysis in **JavaScript**
  - High-level language
  - Familiar to web developers

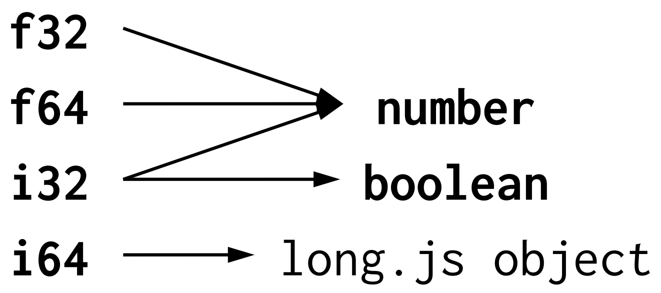
- E.g., **crypto miner detection**
  - [Wang et al., ESORICS ’18]
  - Gather instruction profile
  - 11 LOC
  - No manual instrumentation

```
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;
  }
};
```

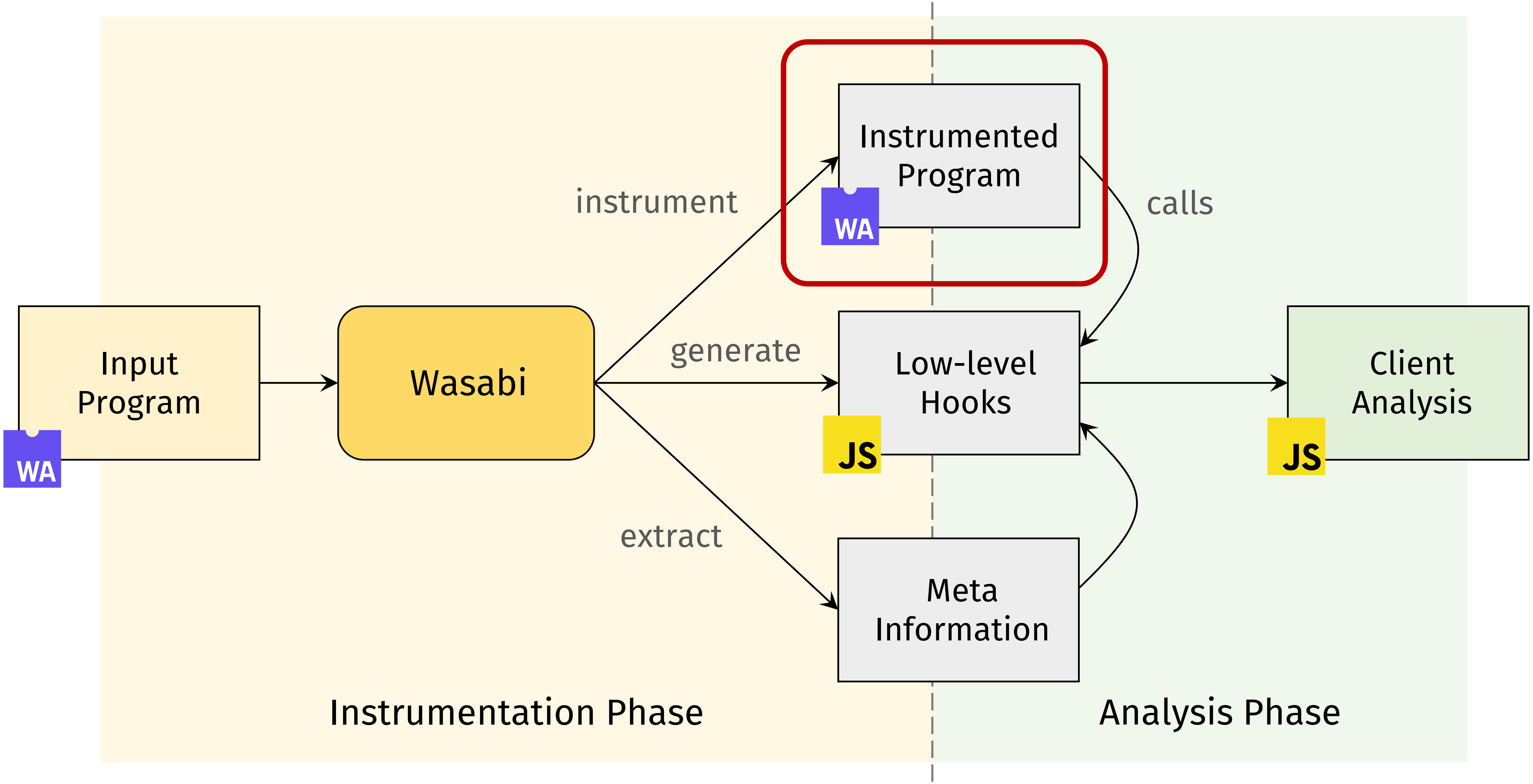
Client Analysis API

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

Hook	Arguments
`call`	loc, func, args...
`binary`	loc, op, arg1, arg2, result
`return`	loc, results...
`br_if`	loc, target, condition
	...



Wasabi Overview



Static Binary Instrumentation

- Why **binary**, why **static**?
  - **Different producers** of WebAssembly
  - **Source code** not always available
  - Static instrumentation is **reliable**

Original Code	Instrumented Code
f32.abs	copy argument to $\text{local}_{\text{arg}}$ f32.abs duplicate stack top copy $\text{local}_{\text{arg}}$ to stack call $\text{hook}_{\text{f32.abs}}$

Add local: $\text{local}_{\text{arg}}$: **f32**

Add WebAssembly function:

```
fun hook_f32.abs(
        arg: f32,
        res: f32) {
    ...
}
```

Instrumentation Challenges

- **Polymorphic** instructions: `drop`, `select`, ...
  - Instrumentation must do type checking

```
...
drop ;; type: [α] -> []
     ;; what is α?
```

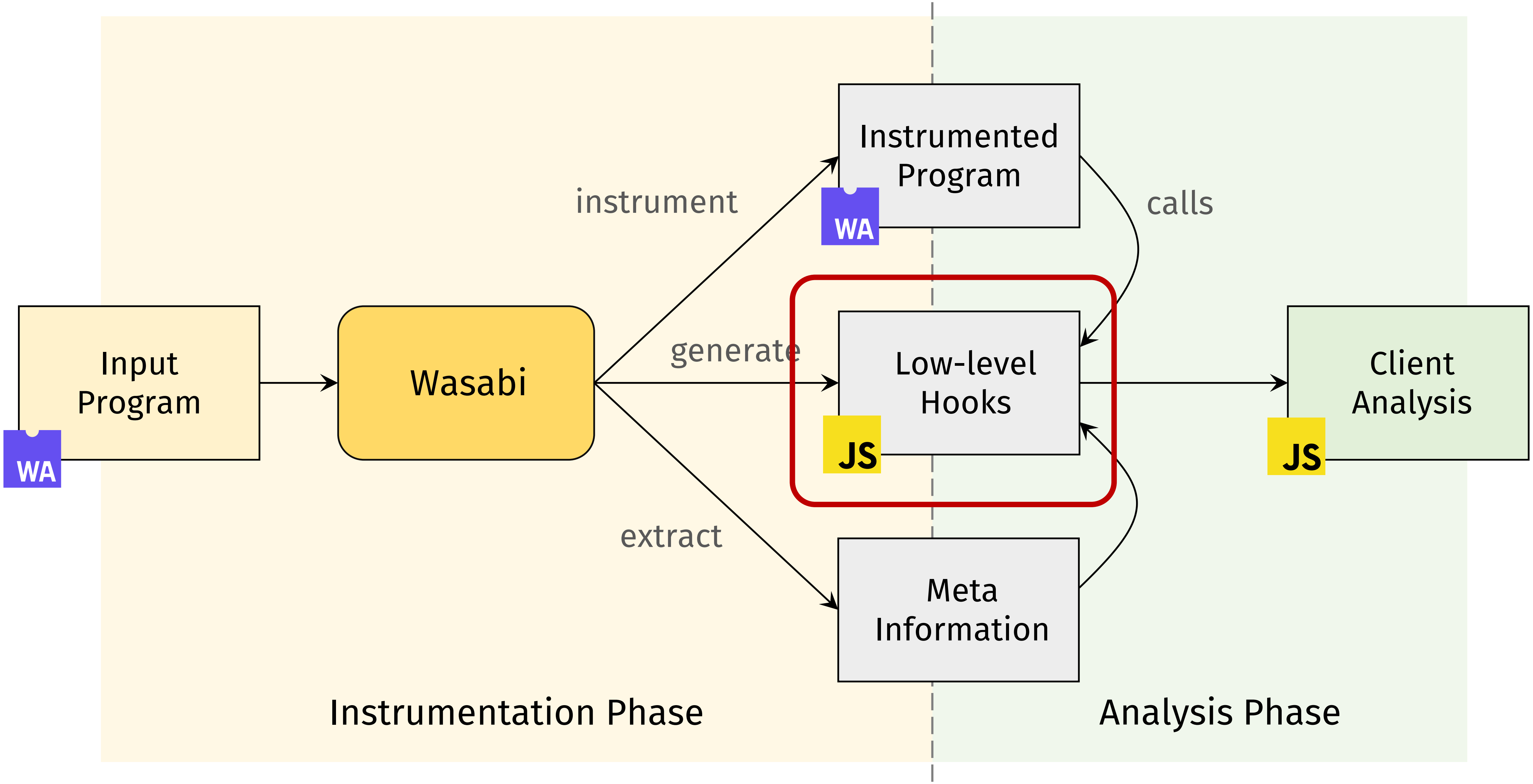
- But functions are **monomorphic** (= fixed type)
  - Monomorphization: 1 hook per concrete type
  - Infinitely many type **combinations** for `call`, `return`
  - On-demand: only for types that appear in the binary

```
fun hook_drop_α
```

```
fun hook_drop_i32
fun hook_drop_f32
...
```

- Other challenges
  - Dynamic block nesting, resolving branch labels, handling i64s

Wasabi Overview



Low-Level Hooks

- Bridge between WebAssembly and JavaScript

Instrumented Program:

```
drop
call hook_drop_f32
```

...

```
drop
call hook_drop_i64
```

Low-Level Hooks:

```
fun hook_drop_f32(arg) {
  call Wasabi.drop(...)
}
```

```
fun hook_drop_i64(arg) {
  convert i64 arg
  call Wasabi.drop(...)
}
```

(High-Level) Client Analysis:

```
Wasabi.drop = (...) => {
  ...
}
```

Evaluation

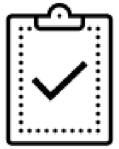
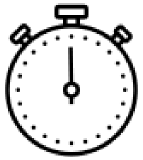
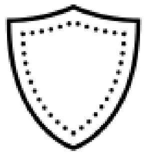
- Program Test Set
- Example Analyses
- Instrumentation Overhead
  - Code size
  - Runtime

Program Test Set

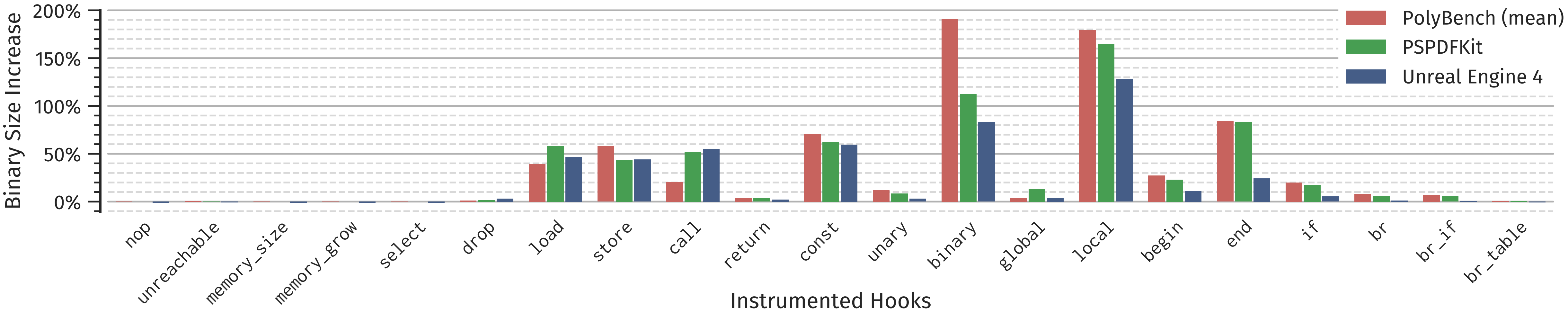
Program		# Instructions	Time to Instr.
PolyBench/C	Set of 30 numerical programs	(mean =) 23 772	23 ms
PSPDFKit	In-browser PDF rendering and editing	7 178 854	5.1 s
Unreal Engine 4	3D game engine demo	20 603 058	15.5 s

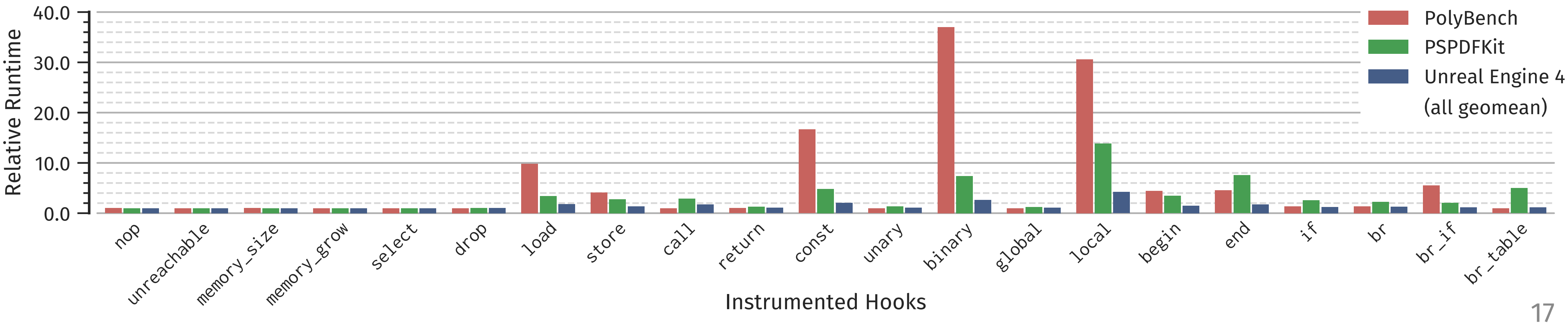
- Also tested on WebAssembly **spec test suite**

Example Analyses

	Analysis	Hooks
	Instruction coverage	*all*
	Branch coverage	`if`, `br_if`, `br_table`, `select`
	Call graph extraction	`call_pre`
	Instruction mix	*all*
	Basic block profiling	`begin`
	Memory access tracing	`load`, `store`
	Dynamic taint analysis	*all*
	Crypto miner instruction profile	`binary`

Instrumentation Overhead





Conclusion

- **WebAssembly**: bytecode for the web
  - New and important platform
    ➡ Need for tooling

- **Wasabi**: dynamic analysis framework for WebAssembly
  - Observe any operation
  - Analysis API in JavaScript
  - Binary instrumentation
  - Open source: http://wasabi.software-lab.org

Links

- https://webassembly.org/
- http://wasabi.software-lab.org/
- https://emscripten.org/
- https://github.com/rustwasm
- https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html
- Icons by https://icons8.com/