Wasabi: A Framework for Dynamically Analyzing WebAssembly

Daniel Lehmann
TU Darmstadt, Germany
mail@dlehmann.eu

Michael Pradel
TU Darmstadt, Germany
michael@binaervarianz.de

Abstract

WebAssembly is the new low-level language for the web and has now been implemented in all major browsers since over a year. To ensure the security, performance, and correctness of future web applications, there is a strong need for dynamic analysis tools for WebAssembly. Unfortunately, building such tools from scratch requires knowledge of low-level details of the language, and perhaps even its runtime environment. This paper presents Wasabi, the first general-purpose framework for dynamically analyzing WebAssembly. Wasabi provides an easy-to-use, high-level API that allows implementing heavyweight dynamic analyses that can monitor all low-level behavior. The approach is based on binary instrumentation, which inserts calls to analysis functions written in JavaScript into a WebAssembly binary. Wasabi addresses several unique challenges not present for other binary instrumentation tools, such as the problem of tracing type-polymorphic instructions with analysis functions that have a fixed type, which we address through an on-demand monomorphization of analysis calls. To control the overhead imposed by an analysis, Wasabi selectively instruments only those instructions relevant for the analysis. Our evaluation on compute-intensive benchmarks and real-world applications shows that Wasabi (i) faithfully preserves the original program behavior, (ii) imposes an overhead that is reasonable for heavyweight dynamic analysis (depending on the program and the analyzed instructions, between 1.02x and 163x), and (iii) makes it straightforward to implement various dynamic analyses, including instruction counting, call graph extraction, memory access tracing, and taint analysis.

1 Introduction

WebAssembly [2, 25] is a new, low-level binary instruction format for the web. Its core use case is as a compilation target for systems programming languages like C, C++, or Rust. By providing low-level control over the memory layout and by closely mapping to hardware instructions, WebAssembly provides near-native and predictable performance [25, 27], unlike managed languages, such as JavaScript and Java. As of last year, WebAssembly support is enabled in all major browsers [46], making it portable across different vendors, architectures, and devices, after prior attempts to establish low-level code on the web have failed, e.g., ActiveX [14], (Portable) Native Client [5, 50], or asm.js [26]. To provide some safety guarantees when running untrusted web code, WebAssembly type checks all instructions, strictly separates code from data, verifies that accesses to the linear memory are in-bounds, and offers well-defined interfaces for interaction between modules. Despite its young age, WebAssembly has already been adopted for various applications, including games[1], cryptography [6], machine learning [23], and medical applications [18]. WebAssembly, it seems, will be a ubiquitous and important instruction set for years to come.

Dynamic analysis tools have a long history of success for languages other than WebAssembly, e.g., to check and understand correctness, security, and performance properties [1, 10, 16, 41, 43, 44]. The need for dynamic analysis is particularly strong for highly dynamic languages, such as JavaScript [4] and for languages with a lot of low-level control, such as C and C++. As a compilation target of systems languages and with JavaScript as the host environment, WebAssembly sits exactly at the intersection of these two kinds of languages, making it a prime target for dynamic analysis.

Creators of a dynamic analysis usually can choose between two options. One option is to implement the analysis from scratch. A common strategy is to add instrumentation code to the program, but this requires an in-depth understanding of the instruction set and tools to manipulate it. Another common strategy is to modify the runtime environment of the program, e.g., a virtual machine. Unfortunately, such modifications require detailed knowledge of the virtual machine implementation, and they tie the analysis to a particular version of the runtime environment. Since WebAssembly serves as a compilation target of other languages, source-level instrumentation of these languages might appear to be another possible strategy. However, typical web applications heavily rely on third-party code, for which source code is unavailable at the client-side.

Instead of implementing a dynamic analysis from scratch, the second option is to build upon general-purpose dynamic analysis frameworks. Table 1 lists some popular frameworks: Pin [31] and Valgrind [34] for native programs, DiSL [32] and RoadRunner [19] for JVM byte code, and Jalangi [40] for analyzing JavaScript programs. Building on an existing framework reduces the overall effort required to build an analysis and enables the analysis author to focus on the design of the analysis itself. Unfortunately, there currently is

[1] https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html

	Pin [31]	Valgrind [34]	DiSL [32]	RoadRunner [19]	Jalangi [40]	*Wasabi*
(Primary) platform	x86-64	x86-64	JVM	JVM	JavaScript	WebAssembly
Instrumentation of ...	native binaries	native binaries	byte code	byte code	source code	binary code
Analysis language	C/C++	C	Java	Java	JavaScript	JavaScript
API style	instrumentation + callbacks/hooks	low-level instrumentation	aspect-oriented	event stream	callbacks/hooks	callbacks/hooks

Table 1. Overview of existing dynamic analysis frameworks and Wasabi.

```
const signature = {};
Wasabi.analysis.binary = function(loc, op) {
  switch (op) {
    case "i32.add":
    case "i32.and":
    case "i32.shl":
    case "i32.shr_u":
    case "i32.xor":
      signature[op] = (signature[op] || 0) + 1;
}};
```

Figure 1. Cryptominer detection through instruction profiling, as described in a recent WebAssembly analysis [47].

no general-purpose dynamic analysis framework for WebAssembly.

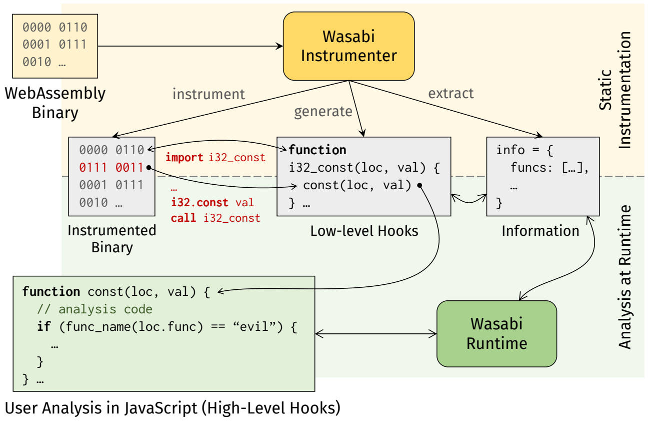
This paper presents *Wasabi*, the first general-purpose framework for dynamic analysis of WebAssembly.[2] Wasabi provides an easy-to-use, high-level API to implement heavyweight analyses that can monitor all low-level behavior. The framework is based on binary instrumentation, which inserts WebAssembly code that calls into analysis functions in between the program's original instructions. The analyses themselves are written in JavaScript and implement analysis functions, called *hooks*, to perform arbitrary operations whenever a particular instruction is executed. To limit the overhead that a dynamic analysis imposes, Wasabi supports *selective instrumentation*, i.e., it instruments only those instructions that are relevant for a particular analysis.

As a simple example of a Wasabi-based analysis, Figure 1 shows our re-implementation of the profiling part of a cryptomining detector [47]. Unauthorized use of computing resources is detected by monitoring the WebAssembly program and gathering an instruction signature that is unique for typical mining algorithms. Implementing this analysis in Wasabi takes ten lines of JavaScript, which use the framework's `binary` hook to keep track of all executed binary operations. In contrast, the original implementation is based on a special-purpose instrumentation of WebAssembly that the authors of [47] implemented from scratch. This and more sophisticated analyses (Section 4.2) show that Wasabi allows implementing analyses with little effort.

Apart from being the first dynamic analysis framework for WebAssembly, Wasabi addresses several unique technical challenges. First, to provide a high-level API for tracking low-level behavior, the approach abstracts away various details of the WebAssembly instruction set. For example, Wasabi bundles groups of related instructions into a single analysis hook, resolves relative target labels of branch instructions into absolute instruction locations, and resolves indirect call targets to actual functions. Second, Wasabi transparently handles the interaction of the WebAssembly code to analyze and the JavaScript code that implements the analysis. A particular challenge is that WebAssembly functions must statically declare fixed parameter types, while some WebAssembly instructions are polymorphic, i.e., they can be executed with different numbers and types of arguments. To insert hook calls for polymorphic instructions, a different monomorphic variant of the hook must be generated for every concrete combination of argument types. Wasabi uses *on-demand monomorphization* to automatically create such monomorphic hooks, but only for those type variants that are actually present in the given WebAssembly code. Third, Wasabi faithfully executes the original program and even preserves its memory behavior, which is useful to implement memory profilers. To this end, none of the inserted instructions requires access or modification of the program's original memory. Instead, analyses can track memory operations in JavaScript, i.e., in a separate heap that does not interfere with the WebAssembly heap.

For our evaluation, we implement eight analyses on top of Wasabi, including basic block profiling, memory access tracing, call graph analysis, and taint analysis. We find that writing a new analysis is straightforward and typically takes at most a few dozens of lines of code. As expected by design, Wasabi faithfully preserves the original program behavior. The framework instruments large binaries quickly (e.g., 40 MB in about 15 seconds), and it increases the code size between less than 1% and 742%, depending on the program and which instructions shall be analyzed. The runtime overhead imposed by Wasabi varies between 1.02x and 163x, again depending on the program and instructions-to-analyze, which is in line with existing frameworks for heavyweight dynamic analysis.

[2]"Wasabi" stands for WebAssembly dynamic analysis using binary instrumentation.



```
0000 0110
0001 0111
0010 …
```

```
0000 0110
0111 0011
0001 0111
0010 …
```

```
import i32_const
…
i32.const val
call i32_const
```

```
function
i32_const(loc, val) {
  const(loc, val)
} …
```

```
info = {
  funcs: […],
  …
}
```

```
function const(loc, val) {
  // analysis code
  if (func_name(loc.func) == “evil”) {
    …
  }
} …
```

Figure 2. Wasabi's instrumentation and analysis phases.

In summary, this paper contributes the following:

- We present the first general-purpose framework for dynamically analyzing WebAssembly code, an instruction format that is becoming a cornerstone of future web applications.
- We present techniques to address unique technical challenges not present in existing dynamic analysis frameworks, including on-demand monomorphization of analysis hooks and static resolution of relative branch targets.
- We show that Wasabi is useful as the basis for a diverse set of analyses, that implementing an analysis takes very little effort, and that the framework imposes an overhead that is reasonable for heavyweight dynamic analysis.
- We make Wasabi available as open-source, enabling others to build on it: http://wasabi.software-lab.org

2 Approach

This section describes our framework for dynamically analyzing WebAssembly programs. We give an overview of Wasabi and the design decisions that have led to it (Section 2.1), introduce Mini-Wasm, a WebAssembly core language, (Section 2.2), describe the analysis API (Section 2.3), and finally present some details of the instrumentation (Section 2.4).

2.1 Overview

Figure 2 gives on overview of Wasabi. The inputs are a WebAssembly binary to analyze (top-left) and a dynamic analysis written in JavaScript (bottom-left). The rationale for choosing JavaScript as the analysis language is twofold. First, it is widely used in the web and hence well-known to people interested in dynamic analysis for the web. Second, JavaScript is the only high-level language that is directly supported by all platforms that currently execute WebAssembly.

The framework has two phases: *static instrumentation* and *analysis at runtime*. The first phase augments the given WebAssembly binary with instructions that call into the analysis implementation. To this end, the instructions of the original program are interleaved with calls to *low-level analysis hooks*. The low-level hooks are implemented in JavaScript code and generated by Wasabi. At runtime, the low-level hooks then call *high-level analysis hooks*, which the analysis author implements. The Wasabi runtime provides further information, e.g., the types of functions in the program, to the analysis.

Ahead-of-time binary instrumentation offers three important advantages over alternative designs. First, it is independent of the execution platform that WebAssembly runs on and robust to changes in current platforms. Suppose we would instead modify a specific implementation of WebAssembly, e.g., in Firefox, then Wasabi could not analyze programs executed elsewhere. Moreover, Wasabi would risk to become outdated when the execution platform evolves. Second, binary instrumentation enables Wasabi to support all languages that compile to WebAssembly, which currently include C/C++ [52], Rust [15], Go [33], and TypeScript [39]. Alternatively, we could rely on source-level instrumentation for these languages, but since the source code of WebAssembly running on websites is often unavailable, this is not an option if we want to support, e.g., security applications like reverse engineering. Third, ahead-of-time instrumentation avoids runtime overhead compared to instrumenting code during the execution [9, 31, 34]. Since WebAssembly, in contrast to other binary formats, does not suffer from language features that make ahead-of-time instrumentation inherently difficult, such as self-modifying code or mixed code and data, Wasabi can reliably instrument code ahead-of-time.

2.2 Mini-Wasm

For a self-contained and concise presentation, we now introduce *Mini-Wasm*, a simplified core of WebAssembly. The Wasabi implementation supports the entire WebAssembly language. Figure 3 shows the grammar of Mini-Wasm modules. A *module* corresponds to a single binary file and contains *functions*, *global* variables, an optional *start* function, and at most one table and memory. Each of these have a name only when they are imported or exported, and otherwise are referenced by integer indices. The *table* maps indices to functions and is used for indirect calls, e.g., to implement function pointers or virtual calls. Similar to native programs, and unlike in managed languages with garbage collection, WebAssembly *memory* is a linear sequence of bytes, which can be increased at runtime with `memory.grow`.

WebAssembly execution is a stack machine with per-function locals, similar to the JVM [29]. One distinctive feature of WebAssembly, which is relevant for Wasabi, is how control-flow is encoded. Unlike in the JVM or native code, instructions are structured into well-nested, implicitly labeled blocks. Instead of unrestricted gotos that directly jump to an instruction offset, branch instructions can only target blocks in which they are enclosed. The destination block is referenced by a non-negative integer label, where zero indicates the immediately

Module and sections

$$\begin{aligned}
\mathit{module} &::= \mathit{function}^* \ \mathit{global}^* \ \mathit{start}^? \ \mathit{table}^? \ \mathit{memory}^? \\
\mathit{function} &::= \mathit{type}_{\mathit{func}} \ (\mathit{import} \mid \mathit{code}) \ \mathit{export}^* \\
\mathit{global} &::= \mathit{type}_{\mathit{val}} \ (\mathit{import} \mid \mathit{init}) \ \mathit{export}^* \\
\mathit{start} &::= \mathit{idx}_{\mathit{func}} \\
\mathit{table} &::= \mathit{import}^? \ \mathit{idx}_{\mathit{func}}{}^* \ \mathit{export}^* \\
\mathit{memory} &::= \mathit{import}^? \ \texttt{byte}^* \ \mathit{export}^* \\
\mathit{import} &::= \texttt{"name"}, \quad \mathit{export} ::= \texttt{"name"} \\
\mathit{code} &::= (\texttt{local} \ \mathit{type}_{\mathit{val}})^* \ \mathit{instr}^* \\
\mathit{init} &::= \mathit{instr}^*
\end{aligned}$$

Instructions

$$\begin{aligned}
\mathit{instr} ::= {} & \texttt{nop} \mid \texttt{drop} \mid \texttt{select} \\
& \mid \mathit{type}_{\mathit{val}}\texttt{.const} \ \mathit{value} \mid \mathit{unary} \mid \mathit{binary} \\
& \mid \mathit{type}_{\mathit{val}}\texttt{.load} \mid \mathit{type}_{\mathit{val}}\texttt{.store} \mid \texttt{memory.grow} \\
& \mid (\texttt{set} \mid \texttt{get} \mid \texttt{tee})\texttt{\_local} \ \mathit{idx}_{\mathit{local}} \mid (\texttt{set} \mid \texttt{get})\texttt{\_global} \ \mathit{idx}_{\mathit{global}} \\
& \mid \texttt{call} \ \mathit{idx}_{\mathit{func}} \mid \texttt{call\_indirect} \ \mathit{type}_{\mathit{func}} \mid \texttt{return} \\
& \mid \texttt{block} \mid \texttt{loop} \mid \texttt{if} \mid \texttt{else} \mid \texttt{end} \\
& \mid \texttt{br} \ \mathit{label} \mid \texttt{br\_if} \ \mathit{label} \mid \texttt{br\_table} \ \mathit{label}^+ \\
\mathit{unary} ::= {} & \texttt{i32.eqz} \mid \ldots \mid \texttt{f32.neg} \mid \ldots \mid \texttt{f32.convert\_s/i32} \mid \ldots \\
\mathit{binary} ::= {} & \texttt{i32.add} \mid \ldots \mid \texttt{i32.eq} \mid \ldots
\end{aligned}$$

Types, labels, indices

$$\begin{aligned}
\mathit{type}_{\mathit{val}} &::= \texttt{i32} \mid \texttt{i64} \mid \texttt{f32} \mid \texttt{f64} \\
\mathit{type}_{\mathit{func}} &::= \mathit{type}_{\mathit{val}}{}^* \rightarrow \mathit{type}_{\mathit{val}}{}^* \\
\mathit{label} &\in \mathbb{N}, \quad \mathit{idx}_{\mathit{func} \mid \mathit{global} \mid \mathit{local}} \in \mathbb{N}
\end{aligned}$$

Figure 3. Abstract syntax of Mini-Wasm.

enclosing block. Depending on the block type, the next executed instruction is either the first one inside the block (for `loop` blocks, rendering the branch a backward jump) or the next instruction after the block's matching `end` instruction (for `block`, `if`, and `else` blocks, thus a forward jump). For example, in Figure 4, the label `1` at line 4 references the block at line 1, and hence is a jump forward to line 8. Section 2.4 describes how Wasabi handles this control-flow encoding.

WebAssembly is statically type checked and knows four primitive types for globals, locals, and stack values: 32 and 64 bit integers (`i32`, `i64`), and single and double precision floats (`f32`, `f64`). Many WebAssembly instructions are polymorphic in the sense that the input and result types vary depending on the context in which the instruction is executed. For example, `call` and `return` pop and push different types depending on the function type of the called and current function, respectively. Similarly, the instruction types for accesses to locals and globals depend on the referenced local and global variable.

For `drop`, which removes the current stack top, and `select`, which pushes one of two values depending on a condition, the types cannot be simply looked up in the module, but depend on the previously executed code. For example, a `drop` following an `i32.const` has `i32` as input type, whereas a `drop` following a call that returns an `f64` value has `f64` as input type. The many possible typed instructions pose a challenge for generating Wasabi's hooks, which we explain along with our solution in Section 2.4.

```
1  block       <---------,
2      block             |
3          get_local 0   |
4          br_if 1    ---' ;; block reference by label
5          ;; next instruction if local #0 == false
6      end
7  end ;; matching end for first block
8  ;; next instruction if local #0 == true
```

Figure 4. Structured control-flow in WebAssembly.

2.3 Analysis API

Wasabi offers analysis authors an API with hooks to be implemented by the analysis. The API is both powerful enough to enable arbitrary dynamic analyses and high-level enough to spare the analysis author dealing with irrelevant details. Table 2 shows the hooks,[3] along with their arguments and types. The hooks can be roughly clustered into six groups: Instructions related to stack manipulation (`const`, `drop`, `select`), operations (`unary`, `binary`), accesses and management of register and memory (`local`, `global`, `load`, `store`, `memory_grow`), function calls (`call_pre`, `call_post`, `return`), control flow (`br*`), and blocks (`begin`, `end`). Each hook implementation receives details about the respective instruction, e.g., its inputs and outputs, as well as the code location of the instruction.

The API is designed to ensure four important properties.

- *Full instruction coverage.* It covers all WebAssembly instructions and provides all their inputs and results to the analysis. This property is crucial to implement arbitrary dynamic analyses that can observe all runtime behavior. We describe in Section 2.4 how selective instrumentation limits the costs to be paid for this flexibility.
- *Grouping of instructions.* The API groups related WebAssembly instructions into a single hook, which significantly reduces the number of hooks analyses must implement. Providing one hook per instruction to the analysis would require a huge number of hooks (e.g., there are 123 numeric instructions alone), whereas Wasabi's API provides 23 hooks only. To distinguish between instructions, if necessary, the hooks receive detailed information as arguments. For example, the `binary` hook receives an op argument that specifies which binary operation is executed. To hide the various variants of polymorphic instructions from analyses authors, Wasabi also maps all variants of the same kind of instruction into a single hook. For example, the `call` instruction can take different numbers and types of arguments, depending on the called function, which are represented in the hook as an array of varying length.
- *Pre-computed information.* Wasabi provides pre-computed information along with some hooks because the runtime

[3]For brevity, the table leaves out five additional hooks that Wasabi supports, `start`, `nop`, `unreachable`, `if`, and `memory_size`, for a total of 23 hooks.

Hook Name	Arguments and Types
`const` / **`drop`**	`value`: $type_{val}$
`select`	`condition`: `boolean`, `first`: $type_{val}$, `second`: $type_{val}$
`unary`	`op`, `input`: $type_{val}$, `result`: $type_{val}$
`binary`	`op`, `first`: $type_{val}$, `second`: $type_{val}$, `result`: $type_{val}$
`local` / **`global`**	`op`, `index`: `number`, `value`: $type_{val}$
where	`op`: instruction `string`, e.g., `i32.add` or `get_local`
`memory_grow`	`delta`: `number`, `previousSize`: `number`
`load` / **`store`**	`op`, `memarg`, `value`: $type_{val}$
where	`memarg`: { `addr`: `number`, `offset`: `number` }
`call_pre`	`func`: `number`, `args`, `tableIndex`: (`number` | `null`)
where	`args`: [$type_{val}$], `tableIndex == null` iff direct call
`call_post` / **`return`**	`results`: [$type_{val}$]
`br`	`target`
`br_if`	`target`, `condition`: `boolean`
`br_table`	`table`: [`target`], `tableIndex`: `number`
where	`target`: { `label`: `number`, `location`: `location` }
`begin`	`type`
`end`	`type`, `begin`: `location`
where	`type`: `string` ∈ {`function`, `block`, `loop`, `if`, `else`}
every hook	`location`: { `func`: `number`, `instr`: `number` }, ...

Table 2. API of the high-level analysis hooks.

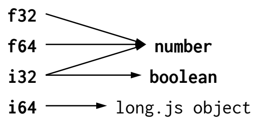


Figure 5. Mapping of WebAssembly types to JavaScript.

values on their own are not informative enough for an analysis. For example, the three branch-related hooks receive `target` objects that contain the statically resolved, absolute location of the next instruction that will be executed, if the branch is taken, alongside the low-level relative branch label. Similarly for indirect calls, Wasabi resolves the runtime table index to the actually called function.

- *Faithful type mappings.* Finally, the API faithfully maps typed values from WebAssembly to JavaScript. Figure 5 shows the four primitive WebAssembly types and how they are represented without loss of precision in a Wasabi analysis. `i32`, `f32`, and `f64` are represented as a JavaScript `number`. Since JavaScript has no native support for 64-bit integers, Wasabi transparently maps them to `long.js`[4] objects. Conditions, which are `i32`s with value 0 or 1 in WebAssembly, are mapped to JavaScript's `boolean`s.

The API gives analysis authors the power to implement sophisticated dynamic analyses with little effort. In particular, it is straightforward to implement memory shadowing [34, 53], a feature useful, e.g,. for tracking the origin of undesired values or for taint analysis. To associate some meta-information with a memory value, all an analysis must do is to maintain a map of memory locations to meta-information, and to update the meta-information on memory-related instructions. One of our example analyses (Section 4.2) is a taint analysis that implements memory shadowing in this way.

2.4 Binary Instrumentation

The following presents the core of Wasabi: its binary instrumentation component, which inserts code that eventually calls the high-level analysis hooks described in the previous subsection. We first describe the instrumentation of individual WebAssembly instructions (Section 2.4.1) and how Wasabi reduces overhead via selective instrumentation (Section 2.4.2). Then, we highlight four instrumentation challenges that are unique to WebAssembly and describe how Wasabi addresses them (Sections 2.4.3 to 2.4.6).

2.4.1 Instrumentation of Instructions

To keep track of all instructions that occur during the execution, Wasabi inserts for each instruction a call to an analysis hook. Table 3 illustrates the instrumentation for a subset of all instructions. Row 1 shows the simplest case: a `const` instruction that pushes an immediate value on the stack. The instrumentation adds a call to the corresponding hook. Since the hook receives the value produced by the `const` instruction as an argument, the value is pushed once more on the stack prior to the call. After the call to the hook, the stack will be the same as in the original, uninstrumented program.

Row 2 of Table 3 shows an instruction that takes inputs and produces results. To pass both to the inserted hook call, we need to duplicate values on the stack. For this purpose, Wasabi generates a fresh local of the appropriate type and writes the current stack top to this local with `tee_local`. Before the hook call, the inserted code retrieves the stored input and its result with `get_local`. Row 3 illustrates how Wasabi instruments `call` instructions. In contrast to other instructions, we surround the original instruction with two calls into the analysis, so that an analysis author can execute analysis behavior before and after the call.

All inserted calls go to JavaScript functions that are imported into the WebAssembly binary. These imported functions are not yet the high-level hooks from Section 2.3, but low-level hooks that are automatically generated by Wasabi. There are several reasons for this indirection. First, it allows Wasabi to map typed WebAssembly instructions to untyped JavaScript hooks in a seamless way (Section 2.4.3). Second, it helps providing information that is useful in high-level hooks but not available at the current instruction (Section 2.4.4). Third, Section 2.4.5 shows that Wasabi sometimes also calls other hooks at runtime, because the necessary information

[4] https://github.com/dcodeIO/long.js

	Instructions Original	⇒ Instrumented	Explanations and other changes made to the module
	Constants ($type_{val}$`.const`), similarly simple instructions that only produce a value:		
1	**`i32.const`** *value*	**`i32.const`** *value*	original instruction
		`i32.const` *value*	duplicate constant value for the hook
		`call` $idx_{\texttt{hooks.i32.const}}$	instrumentation hook, also needs to be added as function import to module
	General instructions (unary, binary, load, store):		
2		**`tee_local`** $idx_{temp\ input\ local}$	store instruction input(s) into freshly generated local(s)
	`f32.abs`	**`f32.abs`**	
		`tee_local` $idx_{temp\ result\ local}$	store instruction result into freshly generated local
		`get_local` $idx_{temp\ input\ local}$	push inputs and results on stack as hook arguments
		`get_local` $idx_{temp\ result\ local}$	
		`call` $idx_{\texttt{hooks.f32.abs}}$	one low-level hook per instruction
	Calls (`call` and `call_indirect`), similarly also `returns`:		
3		**`tee_local`** $idx_{temp\ input\ local}$	store call argument(s) in freshly generated local(s)
		`i32.const` idx_{func}	pass index of called function to hook
		`get_local` $idx_{temp\ input\ local}$	pass stored inputs to monomorphized `call_pre` hook
		`call` $idx_{\texttt{hooks.call\_pre\_}(type_{local})^*}$	
	`call` idx_{func}	**`call`** idx_{func}	
		`tee_local` $idx_{temp\ result\ local}$	store call result(s) in freshly generated local(s)
		`get_local` $idx_{temp\ input\ local}$	pass stored results to monomorphized `call_post` hook
		`call` $idx_{\texttt{hooks.call\_post\_}(type_{local})^*}$	
	Polymorphic `drop` and `select` instructions:		
4	. . .	. . .	type check all instructions to keep track of abstract stack
	(preceding code)	(preceding code)	here: assuming preceding instructions produce a stack top value of type $type_{val}$,
	. . .	. . .	then the following `drop` has type `[`$type_{val}$`]` → `[]`
	`drop`	**`call`** $idx_{\texttt{hooks.drop\_}type_{val}}$	matching monomorphic hook call is inserted (consumes stack top in place of `drop`)
	Blocks/structured control-flow (`block`, `loop`, `if`, `else`, `end`) and branches (`br`, `br_if`, `br_table`):		
5	*label*: **`block`**	*label*: **`block`**	*label* is implicit and not encoded in the Wasm binary
		`call` $idx_{\texttt{hooks.begin\_block}}$	every block type (`block`/`loop`/`if`/`else`) has own low-level `begin` hook
	otherlabel: **`loop`**	*otherlabel*: **`loop`**	
		`call` $idx_{\texttt{hooks.begin\_loop}}$	loop `begin` hook is called once per iteration
	. . .	. . .	
		`i32.const` 1 (*label*)	"raw" (i.e., unresolved relative) target label is passed to hook as integer
		`i32.const` resolve(*label*)	resolved (at instrumentation time) label to next executed instruction is also passed
		`call` $idx_{\texttt{hooks.br}}$	branch hooks must come before the instruction
		`call` $idx_{\texttt{hooks.end\_loop}}$	explicitly call `end` hooks of all "traversed" blocks, for dynamic block nesting
		`call` $idx_{\texttt{hooks.end\_block}}$	
	`br` 1 (*label*)	**`br`** 1 (*label*)	
	. . .	. . .	(not shown:) end hooks receive location of the end and of the matching block begin
		`call` $idx_{\texttt{hooks.end\_loop}}$	every block type (`block`/`loop`/`if`/`else`) has own `end` hook (cf. `begin_*` hook)
	`end`	**`end`**	
		`call` $idx_{\texttt{hooks.end\_block}}$	
	`end`	**`end`**	
	Instructions with `i64` inputs or results, value is split for passing to hook:		
6	**`i64.const`** *value*	**`i64.const`** *value*	if instruction has side-effects, its result is duplicated via a local instead (but const here)
		`i64.const` *value*	push lower 32-bit half of `i64` value as `i32` on stack
		`i32.wrap/i64`	
		`i64.const` *value*	shift upper 32-bit half of `i64` value to right, then push as `i32` on stack
		`i64.const` 32	
		`i64.shr_s`	
		`i32.wrap/i64`	
		`call` $idx_{\texttt{hooks.i64.const}}$	cannot pass `i64` values to hooks, so they take a tuple of (`i32`, `i32`) instead

Table 3. Instrumentation of (a subset of all) WebAssembly instructions. Every hook also takes two `i32`s that represent the original instruction's location. For brevity, the corresponding `i32.const` instructions are omitted in the table.

which hooks to call is available only then. Finally, the low-level hooks can convert values before passing them to the high-level hooks (Section 2.4.6). All of these issues can be solved by automatically generated low-level hooks that are hidden from analysis authors.

2.4.2 Selective Instrumentation

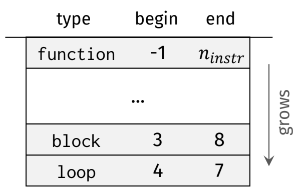
Not every analysis uses all of the hooks provided by the API from Section 2.3. To reduce both the code size and the runtime overhead of the instrumented binary, Wasabi supports *selective instrumentation*. That is, only those kinds of instructions are instrumented that have a matching high-level hook in the given analysis. Wasabi ensures that the instrumentation for different kinds of instructions are independent of each other, so that instrumenting only some instructions still correctly reflects their behavior. Sections 4.5 and 4.6 show that selective instrumentation significantly reduces code size and runtime overhead.

2.4.3 On-demand Monomorphization

An interesting challenge for the instrumentation comes from static typing in WebAssembly. While there are polymorphic instructions, WebAssembly functions, including our hooks, must always be declared with a fixed, monomorphic type. For polymorphic instructions, Wasabi cannot simply generate one hook per kind of instruction: Consider `drop` with the polymorphic instruction type $[\alpha] \rightarrow [\,]$. Inserting a call to the same hook after each `drop` is not possible, because the hook's function type would then be polymorphic. Instead, Wasabi generates multiple monomorphic variants of a polymorphic hook and inserts a call to the appropriate monomorphic low-level hook.[5]

For many polymorphic instructions, determining which monomorphic hook variant to call is straightforward. For example, the instruction type of `set_global` depends only on the type of the referenced variable. The types of `drop` and `select`, however, cannot be simply looked up. Instead, as shown in row 4 of Table 3, their type depends on all preceding instructions. Wasabi thus performs full type checking during instrumentation, that is, it keeps track of the types of all values on the stack [25, 48]. When the `drop` in the last line of the example is encountered, its input type is equal to the top of the abstract stack and Wasabi can insert the call to the matching monomorphic low-level hook.

While creating monomorphic variants of hooks yields type-correct WebAssembly code, doing so eagerly leads to an explosion of the required number of monomorphic hooks. Since functions can have an arbitrary number of arguments and results[6], the number of monomorphic hooks for `call`s and `return`s is even unbounded. One way to address this



type	begin	end
function	-1	n_{instr}
...		
block	3	8
loop	4	7

Figure 6. Abstract control stack at the `br` branch instruction in row 5 of Table 3 (assuming the example is preceded by four other instructions).

problem would be to set a heuristic limit, e.g., by generating hooks for calls with up to ten arguments. However, the resulting $4^{10} = 1,048,576$ call-related hooks would cause unnecessary binary bloat and may still fail to support all calls.

Instead, Wasabi generates monomorphic hooks on-demand only for instructions and type combinations that are actually present in the given binary. We call this approach *on-demand monomorphization* of hooks. During instrumentation, Wasabi maintains a map of already generated low-level hooks. If a required hook, e.g., for a `call` instruction with type `[i32]` $\rightarrow$ `[f32]`, is present in the map, the function index of the hook is returned. Otherwise, Wasabi generates the hook and updates the map. Our evaluation shows that on-demand monomorphization significantly reduces the number of low-level hooks, and hence the code size, compared to the eager approach described above.

2.4.4 Resolving Branch Labels

As described in Section 2.2, WebAssembly relies on structured control-flow, a feature not present in other low-level instruction sets. An interesting challenge that arises from structured control-flow is how and when to resolve the destination of branches. Row 5 of Table 3 illustrates the problem with a few control-flow-related instructions. The `br` instruction jumps to a destination referenced by a relative integer label 1. However, passing this label to the high-level dynamic analysis API would be of limited use, because without additional static information (namely the surrounding blocks), the dynamic analysis cannot resolve the label to a code location.

To enable analysis authors to reason about branch destinations without implementing their own static analysis, Wasabi resolves branch labels during the instrumentation and passes the resulting absolute instruction locations to the high-level API. To resolve branch labels, Wasabi keeps track of an *abstract control stack* while instrumenting WebAssembly code. Whenever the instrumentation enters a new block, an element is pushed to the control stack, consisting of the block type (`function`, `block`, `loop`, `if`, or `else`), the location of the block begin, and the location of the matching `end` instruction. Whenever the instrumentation encounters the end of a block, the topmost entry is popped of the control

[5]This strategy is similar to the compilation of generic functions in Rust or instantiation of function templates in C++ [28, 45].

[6]Strictly speaking, functions in the binary format 1.0 have at most one result, but the formal semantics already support multiple return values [25].

stack. As an example, Figure 6 shows the control stack for the code in row 5 of Table 3.

Given the abstract control stack, Wasabi can determine during instrumentation what code location a branch, if taken, will lead to. At every branch to a label n, Wasabi queries the control stack for its $n + 1$-th entry from the top, to determine the targeted block, and then computes the location of the next instruction from the block type and the locations of the `begin` and `end` instructions. This absolute instruction location is then given as an argument to the branch hook, as shown in the example in Table 3.

2.4.5 Dynamic Block Nesting

Another control-flow-related challenge is about observing the end of the execution of a block. Some analyses may want to observe the block nesting at runtime, i.e., to perform some action when a block is entered and left. For this purpose, Wasabi offers the high-level `begin` and `end` hooks (Section 2.3). The example in row 5 of Table 3 shows that our instrumentation adds the respective hook calls (e.g., `call` $idx_{\texttt{hooks.begin\_block}}$ and `call` $idx_{\texttt{hooks.end\_block}}$) at the beginning of a block and before the matching `end`.

Unfortunately, `branch`ing or `return`ing will jump out of a block and over the inserted `end` hook calls. Consider the last two calls to `hooks.end_loop` and `hooks.end_block` in Table 3. They are not executed because the earlier `br 1` directly transfers control to after the enclosing block. To account for that, Wasabi adds additional calls before each branch and `return` that invoke every end hook of the blocks that will be "traversed" during the jump. That is, as the example shows, Wasabi inserts calls to the `end` hooks for the two enclosing blocks prior to the `br 1` instruction. Again, the *control stack* can tell us which end hooks need to be called, namely all between the current block (stack top, inclusive) and the branch target block (also inclusive). For example, in Figure 6, the instrumented code calls the `loop` and `block` end hooks. For a `return` it would be all blocks on the block stack up to and including the `function` block.

For conditional branches (`br_if`), we call the `end` hooks for traversed blocks only if the branch is actually taken. Similarly, for multi-way branches (`br_table`), which branch is taken (and thus which blocks are left) is known only at runtime. Thus, the instrumentation statically extracts the list of ended blocks for every branch table entry and stores this information. Inside the low-level hook for `br_table`, one of the stored branch table entries will then be selected, before calling the corresponding `end` hooks at runtime.

2.4.6 Handling `i64` Values

As mentioned in Section 2.2, `i64` values cannot be passed to JavaScript functions (and thus our hooks), since JavaScript has only double precision float numbers. To nevertheless enable dynamic analyses to faithfully observe all runtime values, including `i64` values, Wasabi splits a 64-bit integer into two 32-bit integers to pass them to JavaScript. For every `i64` stack value (either produced by a `const` or by any other instruction), we thus insert instrumentation as shown in row 6 of Table 3. The inserted code duplicates the `i64` value twice, from the first of which only the lower 32 bits are extracted and the second of which is shifted to result in the upper 32 bits. Both `i32` values can then be passed to the hook in question. On the JavaScript side, the low-level hook joins the two 32-bit values into a `long.js` object[7], enabling an analysis to faithfully reason about 64-bit integers.

3 Implementation

We have implemented the Wasabi instrumenter, including the static analyses it performs, in about 5000 lines of Rust code. Rust programs themselves can be compiled to WebAssembly, which gives us the option to run Wasabi in the browser and instrument WebAssembly programs at load time in the future. To reduce the time required for instrumenting large binaries, Wasabi can instrument multiple WebAssembly functions in parallel. The only synchronization point is the map of low-level hooks created during on-demand monomorphization, which is guarded by an upgradeable multiple readers/single writer lock.

Our implementation is available to the public under the permissive MIT license at http://wasabi.software-lab.org.

4 Evaluation

To evaluate Wasabi, we focus on five research questions:

RQ 1 How easy is it to write dynamic analyses with Wasabi?

RQ 2 Do the instrumented WebAssembly programs remain faithful to the original execution?

RQ 3 How long does it take to instrument programs?

RQ 4 How much does the code size increase?

RQ 5 What is the runtime overhead due to instrumentation?

4.1 Experimental Setup

We apply Wasabi to 32 programs. 30 of them are from the PolyBench/C benchmark suite[8], which has been used in the paper introducing WebAssembly [25]. In total, the PolyBench benchmark suite comprises 5,163 non-empty, non-comment lines of C code. We compile the PolyBench programs to WebAssembly with emscripten 1.38.8, resulting in 790 KB of WebAssembly binaries. Moreover, we use two complex, real-world programs: the Unreal Engine 4 Zen Garden demo[9], as an example of a major game engine running in the browser,

[7] An alternative would be to use the recently proposed `BigInt` support for JavaScript (https://github.com/tc39/proposal-bigint), but this feature is currently only available in Chrome.

[8] http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[9] https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html

Analysis	Hooks	LOC
Instruction mix analysis	all	42
Basic block profiling	`begin`	9
Instruction coverage	all	11
Branch coverage	`if`, `br_if`, `br_table`, `select`	14
Call graph analysis	`call_pre`	18
Dynamic taint analysis	all	208
Cryptominer detection	`binary`	10
Memory access tracing	`load`, `store`	11

Table 4. Analyses built on top of Wasabi.

and the PSPDFKit benchmark[10], which exercises a commercial library for in-browser rendering and annotation of PDFs. Their WebAssembly binaries are 39.5 MB and 9.5 MB, respectively.

All experiments are performed on a laptop with an Intel Core i7-7500U CPU (2 cores, hyper-threading, 2.7 to 3.5 GHz, 4 MB L3 cache) and 16 GB of RAM. The operating system is Ubuntu 17.10 64-bit. To execute WebAssembly programs, we use a nightly version of Firefox 63.0a1 (2018-08-02).

4.2 Ease of Implementing Analyses (RQ 1)

We have implemented eight dynamic analyses on top of Wasabi. Table 4 lists them, along with the hooks they implement and their total lines of JavaScript code.

Instruction Mix Analysis This analysis counts how often each kind of instruction is executed, which can serve as a basis for performance and security analyses.

Basic Block Profiling A classic dynamic analysis [12] that counts how often each function, block, and loop is executed, which is useful, e.g., for finding "hot" code.

Instruction and Branch Coverage These analyses record for each instruction and branch, respectively, whether it is executed, which is useful to assess the quality of tests.

Call Graph Analysis This analysis creates a dynamic call graph, including indirect calls and calls between functions that are neither imported nor exported. Call graphs are the basis of various other analyses, e.g., to find dynamically dead code or to reverse-engineer malware.

Taint Analysis The analysis associates a taint with every value and tracks how taints propagate through instructions, function calls, and memory accesses, to detect illegal flows from sources to sinks.

Memory Access Tracing The analysis tracks all memory accesses and stores them for a later off-line analysis, e.g., to detect cache-unfriendly access patterns.

[10] https://pspdfkit.com/webassembly-benchmark/

```
const coverage = [];
function addBranch({func, instr}, branch) {
    coverage[func] = coverage[func] || [];
    coverage[func][instr] = coverage[func][instr] || [];
    if (!coverage[func][instr].includes(branch)) {
        coverage[func][instr].push(branch);
    }
}
Wasabi.analysis = {
    if_(loc, cond) { addBranch(loc, cond) },
    br_if(loc, target, cond) { addBranch(loc, cond) },
    br_table(loc, tbl, df, idx) { addBranch(loc, idx) },
    select(loc, cond) { addBranch(loc, cond) },
};
```

Figure 7. Branch coverage analysis with Wasabi.

Cryptominer Detection As discussed in the introduction, this analysis gathers a signature based on the frequency of binary instructions to identify mining of cryptocurrencies [47].

As illustrated by the low numbers of lines of code in Table 4, each of these analyses can be implemented with little effort. For further illustration, Figure 7 shows the implementation of the branch coverage analysis. It implements four hooks, `if`, `br_if`, `br_table`, and `select` to keep track of all branches.

4.3 Faithfulness of Execution (RQ 2)

To validate that Wasabi's instrumentation does not modify the semantics of the original program, we compare the behavior of each unmodified binary with the behavior of the fully instrumented binary. For the PolyBench programs, we compile each program with an option to output intermediate results of every calculation on the console. Similarly, the Unreal Engine demo has a mode to check that the pixel values of rendered frames are the same as pre-defined reference frames. For all these programs, the behavior remains unchanged after instrumentation. The PSPDFKit benchmark does not provide any built-in correctness check; based on our manual observations the behavior of the original and instrumented code appear to be the same.

As another way to validate the instrumented WebAssembly code, we use the static WebAssembly validator, which offers some well-formedness guarantees and checks that the code is type correct [24]. Running `wasm-validate` from the WebAssembly Binary Toolkit[11] on all 32 fully instrumented programs shows that all the instrumented code passes the validator. We also instrument and successfully validate Wasabi's output on all programs of the official WebAssembly specification test suite[12], which consists of 63 additional programs.

4.4 Time to Instrument (RQ 3)

Table 5 shows how long Wasabi takes to instrument the programs. The $x \pm y$ notation means a mean value of x and

[11] https://github.com/WebAssembly/wabt

[12] https://github.com/WebAssembly/spec/tree/master/test

Program	Binary Size (B)	Runtime (ms)	$\frac{\text{MB}}{\text{s}}$
PolyBench (avg.)	26 332 ± 299	23 ± 1.4	1.15
PSPDFKit	9 615 389	5 129 ± 65	1.87
Unreal Engine 4	39 510 398	15 481 ± 293	2.55

Table 5. Time taken to instrument programs, averaged across 20 runs (and 30 programs for PolyBench).

a standard deviation of y after 20 repetitions. For readability, we have summarized the results for all 30 PolyBench programs in one row. While the PolyBench programs are of similar, small size (26.3 KB ± 299 B), the PSPDFKit and Unreal Engine binaries are considerably larger (9.6 MB and 39.5 MB, respectively). Instrumentation takes 23ms, on average, for the PolyBench programs, i.e., it is almost instantaneous, and still quick for the larger PSPDFKit (5s) and Unreal Engine binaries (15.5s). Wasabi's instrumentation is parallelized (Section 3), and these numbers are obtained with four threads running on two physical cores. The single-threaded instrumentation time on the large Unreal Engine binary is on average 26.5s, showing that the parallelization reduces the execution time to $15.5/26.5 \approx 0.58$ of the single-threaded time. The last column of Table 5 reports the throughput, i.e., binary code processed per second, showing that the throughput increases with larger binary sizes.

4.5 Increase of Code Size (RQ 4)

Figure 8 presents the increase in binary code size after instrumenting a program. Since many analyses need only a small subset of all hooks (e.g., block profiling needs only `begin`), we evaluate code size increase per required hook, as provided by selective instrumentation (Section 2.4.2). For each hook on the x-axis, the figure shows on the y-axis the increase in binary size as a percentage of the original program size. That is, 0% means the instrumented binary has the same size as the original one and 100% means the program doubled in size due to instrumentation.

With selective instrumentation, more than half of the hooks increase the binary size only by a negligible amount or not at all (less than 1% increase for `nop`, `unreachable`, `memory_size`, `memory_grow`, `select`, and `br_table`; less than 10% for `drop`, `return`, `unary`, `global`, `if`, `br`, and `br_if`, on average). In fact, in several cases the Unreal Engine binary decreased by 1% because Wasabi encodes indices more compactly than the original binary.[13]

Naturally, hooks for instructions that appear very often in the program have the largest influence on the code size, e.g., memory `load` and `store` (between 39% and 58% increase), `begin` and `end` of blocks (11% – 84%), pushing to the stack with `const` (59 – 71%), operations on `locals` (128 – 180%), and finally `binary` instructions (83 – 190%). The difference for the `binary` hook between PolyBench and the other programs can be explained by the former being mostly numerical computation (thus having more binary instructions such as `i32.mul`), whereas PSPDFKit and the Unreal Engine have more diverse code. When instrumenting for all hooks together, which is not required for many analyses, the size increases between 495 and 743%. This result shows that selective instrumentation is very effective in reducing the binary size, compared to blindly instrumenting all instructions.

To evaluate Wasabi's on-demand monomorphization of hooks, we count how many low-level hooks are inserted during full instrumentation. For PolyBench, between 110 (`floyd-warshall` program) and 122 (`deriche`) hooks are inserted, 302 hooks for PSPDFKit, and 783 hooks for the Unreal Engine. In the original Unreal Engine binary, i.e., a real-world WebAssembly program, the call with the largest number of arguments passes 22 `i32` values, which clearly shows that eagerly generating all possible monomorphic combinations of call hooks ($4^{22} \approx 1.7 \times 10^{13}$) is simply not possible. Even in the small PolyBench programs, calls to functions with 6 arguments are common. For these programs, generating no more than 122 hooks on-demand is much better than generating all $4^6 = 4,096$ hooks for `call` instructions plus some more for other instructions.

4.6 Runtime Overhead (RQ 5)

Figure 9 shows how much runtime overhead the instrumentation imposes. On the y-axis, we show the runtime of the instrumented program relative to the original program, that is, a value of 1.0x means the runtime does not increase due to instrumentation. As for code size, most of the hooks contribute only a small runtime overhead: `nop`, `unreachable`, `memory_size`, `memory_grow`, `select`, `drop`, and `unary` each impose less than 1.02x overhead, on average. Instrumenting for `return` or `call` hooks, which are sufficient for many interesting analyses at the function level, incurs a reasonable overhead of up to 1.3x or 2.8x, respectively. More expensive hooks are `begin` and `end` for observing blocks, which incur between 1.5x and 9.9x runtime overhead, 1.8x – 20x for `load`, up to 6.5x for `store`, 2x – 32x for `const`, 4x – 48.5x for operations on `locals`, and 2.6x – 77.5x for `binary` operations. When instrumenting for all hooks, the runtime overhead is between 49x and 163x. Note that the overheads for the PolyBench programs, which perform only numerical computations, are much higher than for the real-world workloads in PSPDFKit and the Unreal Engine. Typical WebAssembly programs call out to the host environment, e.g., to perform shading in WebGL, modify the DOM, or interact with some other Web API, so any overhead imposed by Wasabi contributes only to parts of the total execution time.

Comparing the overhead results to existing heavy-weight dynamic analysis frameworks for other languages shows that

[13] WebAssembly uses the variable length LEB128 encoding for integers, also known from the DWARF debug information format [49]. This allows for multiple possible encodings with different lengths of the same number.

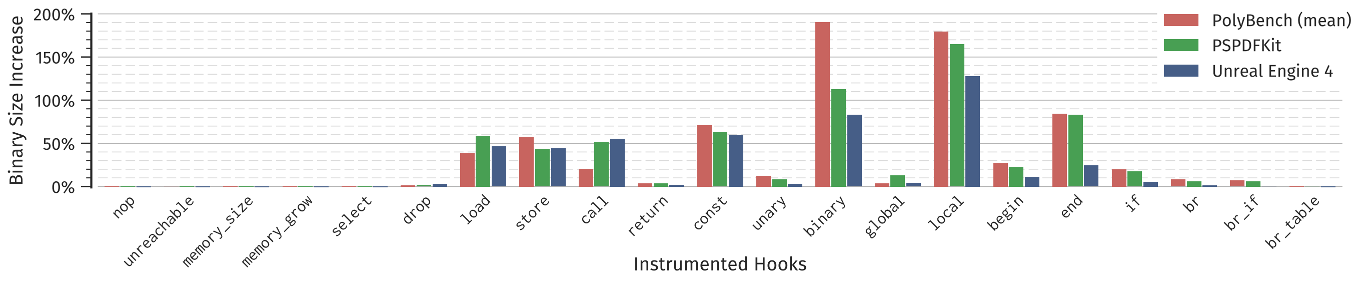


Figure 8. Binary size increase in percent of the original size, when instrumenting the test programs for different analysis hooks. For readability, binary sizes for the 30 PolyBench programs are shown averaged.

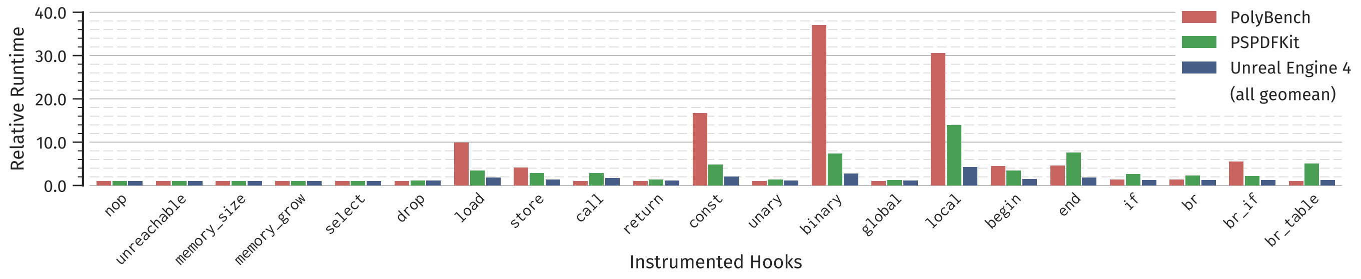


Figure 9. Runtime of the instrumented programs relative to the uninstrumented runtime, per analysis hook. Results are averaged over 20 runs (and again for readability, over the 30 PolyBench programs).

Wasabi's overhead is reasonable. The widely-used JavaScript analysis framework Jalangi reports overheads with the *empty* analysis in the same order of magnitude, namely 26x during record plus 30x during replay, on average [40]. Similarly, the RoadRunner analysis framework for JVM byte code reports an average slowdown of 52x without any analysis [19].

5 Related Work

WebAssembly WebAssembly has been first publicly announced in 2015 and since 2017 is implemented by four major browser engines (Chrome, Edge, Firefox, and Safari). A paper by Haas et al. [25] formalizes the language and its type system, and explains the design rationales. Watt describes the mechanized, formal verification of the WebAssembly specification [48]. Herrera et al. study the performance of WebAssembly, compared to JavaScript, for numerical benchmarks [27].

Dynamic Analysis of WebAssembly Despite its young age, several dynamic analyses for WebAssembly have already been proposed, including two taint analyses [20, 42] and a cryptomining detector [47]. These analyses have been implemented by modifying the V8 engine [20], by implementing a new WebAssembly virtual machine in JavaScript [42], and through custom binary instrumentation [47], respectively. Our evaluation shows that these analyses and others can be implemented in top of Wasabi with significantly less effort.

Binary Instrumentation Tools Binary instrumentation has been a popular strategy to implement dynamic analyses. Often used tools for x86 binaries include DynamoRIO [9], Pin [31], and Valgrind [34], which have provided inspiration for Wasabi. These tools instrument binaries at runtime by translating basic blocks just before their execution, and by storing translations in a code cache. In contrast, Wasabi instruments binaries statically, i.e., ahead-of-time, which avoids any instrumentation overhead during execution. Wasabi also differs w.r.t. the API it provides to analysis authors: While DynamoRIO provides an API to manipulate instructions, Wasabi provides an API to observe the execution of instructions. Analyses written for Pin can specify "instrumentation routines", which determine where to place calls to analysis routines. Instead, Wasabi automatically selects which kinds of instructions to instrument based on the hooks implemented by the analysis. Umbra [53] is a dynamic binary instrumentation tool that focuses on efficient memory shadowing. In contrast, Wasabi provides a general-purpose framework for arbitrary dynamic analysis, including memory shadowing. A difference compared to all the above tools is that in Wasabi, the dynamic analysis is written and executed in a high-level language, JavaScript, instead of being compiled to binary code. The rationale is that JavaScript is already very popular in the web, making it easier for analysis authors to adopt Wasabi.

Dynamic Analysis in General Dynamic analysis [7] has since long been recognized as an effective way to complement static analysis [17]. Various analyses have been proposed, including dynamic slicing [3], taint analyses for x86 binaries [35] and Android [16], tools to find concurrency bugs [11, 30, 36] and heap-related bugs [13], an analysis to track the origin of null and undefined values [8], and analyses to understand performance problems [51]. Given the increasing interest in WebAssembly, we expect an increased demand for dynamic analyses for WebAssembly, for which Wasabi provides a reusable platform.

Dynamic Analysis for the Web Motivated by the dynamic features of JavaScript, such as runtime loading of code, various dynamic analyses for JavaScript-web applications have been presented. For example, recent approaches include analyses to find type inconsistencies [38], JIT-unfriendly code [21], bad coding practices [22], and data races [37]. Many of these analysis are built on top of Jalangi [40], a general-purpose dynamic analysis framework for JavaScript. To the best of our knowledge, there is no comparable tool for WebAssembly yet, a gap this paper aims to fill.

6 Conclusion

This paper presents Wasabi, a general-purpose dynamic analysis framework for WebAssembly, the new low-level instruction set for the web. The framework instruments binaries ahead-of-time and inserts code into the binary that calls into an analysis implemented in JavaScript. Besides being the first dynamic analysis framework for WebAssembly, Wasabi addresses several unique challenges that did not occur in dynamic analysis tools for other platforms. In particular, we handle the problem of tracing polymorphic instructions with analysis functions that have a fixed type via an on-demand monomorphization of analysis hooks, and we statically resolve relative branch labels in control-flow constructs during the instrumentation. The high-level API provided to analyses authors allows for implementing otherwise complex analyses with a few dozens of lines of code, while still providing a complete view of the execution. Our evaluation with both compute-intensive benchmark programs and real-world web applications shows 1.02x to 163x runtime overhead, depending on the program and which instructions are analyzed, which is reasonable for heavyweight dynamic analyses.

We believe that Wasabi provides a solid basis for various analyses to be implemented in the future. As an interesting challenge for future work, we envision cross-language dynamic analysis, in particular, to analyze web applications that run both JavaScript and WebAssembly code.

References

[1] 2018. Using the GNU Compiler Collection (GCC): gcov – a Test Coverage Program. Retrieved August 6, 2018 from https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[2] 2018. WebAssembly website. Retrieved August 6, 2018 from https://webassembly.org/

[3] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/93542.93576

[4] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5, Article 66 (Sept. 2017), 36 pages. https://doi.org/10.1145/3106739

[5] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. 2011. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 355–366. https://doi.org/10.1145/1993498.1993540

[6] Nuttapong Attrapadung, Goichiro Hanaoka, Shigeo Mitsunari, Yusuke Sakai, Kana Shimizu, and Tadanori Teruya. 2018. Efficient Two-level Homomorphic Encryption in Prime-order Bilinear Groups and A Fast Implementation in WebAssembly. In *Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security (ASIACCS '18)*. ACM, New York, NY, USA, 685–697. https://doi.org/10.1145/3196494.3196552

[7] Thoms Ball. 1999. The Concept of Dynamic Analysis. In *ACM SIGSOFT Software Engineering Notes*, Vol. 24. Springer-Verlag, 216–234. https://doi.org/10.1145/318774.318944

[8] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. 2007. Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 405–422. https://doi.org/10.1145/1297027.1297057

[9] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. IEEE Computer Society, Washington, DC, USA, 265–275. http://dl.acm.org/citation.cfm?id=776261.776290

[10] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223. http://dl.acm.org/citation.cfm?id=2190025.2190067

[11] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

[12] P. P. Chang and W. W. Hwu. 1988. Trace Selection for Compiling Large C Application Programs to Microcode. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture (MICRO 21)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 21–29. http://dl.acm.org/citation.cfm?id=62504.62511

[13] Trishul M. Chilimbi and Vinod Ganapathy. 2006. HeapMD: Identifying Heap-based Bugs Using Anomaly Detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 219–228. https://doi.org/10.1145/1168857.1168885

[14] Microsoft Corp. 1996. Microsoft Announces ActiveX Technologies. Retrieved August 6, 2018 from https://news.microsoft.com/1996/03/12/microsoft-announces-activex-technologies/

[15] Alex Crichton. 2017. Enable WebAssembly backend by default. Github Rust repository. Retrieved August 6, 2018 from https://github.com/rust-lang/rust/pull/46115

[16] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking

System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (June 2014), 29 pages. https://doi.org/10.1145/2619091

[17] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*. New Mexico State University Portland, OR, 24–27.

[18] Richard Finney and Daoud Meerzaman. 2018. Chromatic: WebAssembly-Based Cancer Genome Viewer. *Cancer Informatics* 17 (2018). https://doi.org/10.1177/1176935118771972

[19] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '10)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

[20] William Fu, Raymond Lin, and Daniel Inge. 2018. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *ArXiv e-prints* (Feb. 2018). arXiv:cs.CR/1802.01050

[21] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2786805.2786831

[22] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2771783.2771809

[23] Fabian Göttl, Philipp Gagel, and Jens Grubert. 2018. Efficient Pose Tracking from Natural Features in Standard Web Browsers. In *Proceedings of the 23rd International ACM Conference on 3D Web Technology (Web3D '18)*. ACM, New York, NY, USA, Article 17, 4 pages. https://doi.org/10.1145/3208806.3208815

[24] WebAssembly Community Group. 2018. WebAssembly Specification. Retrieved August 6, 2018 from https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[25] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[26] David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js: Working Draft – 18 August 2014. Retrieved August 6, 2018 from http://asmjs.org/spec/latest/

[27] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. 2018. *WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices*. Technical Report. Technical report SABLE-TR-2018-2. Montréal, Québec, Canada: Sable Research Group, School of Computer Science, McGill University.

[28] S. Klabnik and C. Nichols. 2018. *The Rust Programming Language*. No Starch Press. https://books.google.de/books?id=lrgrDwAAQBAJ

[29] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2013. The Java Virtual Machine Specification – Java SE 7 Edition. Retrieved August 6, 2018 from https://docs.oracle.com/javase/specs/jvms/se7/html/

[30] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1168857.1168864

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[32] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2162049.2162077

[33] Richard Musiol. 2018. WebAssembly architecture for Go. Google Docs. Retrieved August 6, 2018 from https://docs.google.com/document/d/131vjr4DH6JFnb-blm_uRdaC0_Nv3OUwjEY5qVCxCup4

[34] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[35] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*.

[36] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[37] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 251–262. https://doi.org/10.1145/2254064.2254095

[38] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 314–324. http://dl.acm.org/citation.cfm?id=2818754.2818795

[39] Micha Reiser and Luc Bläser. 2017. Accelerate JavaScript Applications by Cross-compiling to WebAssembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2017)*. ACM, New York, NY, USA, 10–17. https://doi.org/10.1145/3141871.3141873

[40] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[41] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1247360.1247362

[42] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. Taint Tracking for WebAssembly. *arXiv preprint arXiv:1807.08349* (2018).

[43] EclEmma team. 2018. JaCoCo Java Code Coverage Library. Retrieved August 6, 2018 from https://www.jacoco.org/jacoco/

[44] The Clang Team. 2018. UndefinedBehaviorSanitizer – Clang 8 documentation. Retrieved August 6, 2018 from https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[45] David Vandevoorde and Nicolai M. Josuttis. 2002. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[46] Luke Wagner. 2017. WebAssembly consensus and end of Browser Preview. Retrieved August 6, 2018 from https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html

[47] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. 2018. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*.

[48] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 53–65. https://doi.org/10.1145/3167082

[49] DWARF Debugging Information Format Workgroup. 2015. DWARF Debugging Information Format – Version 3. Retrieved August 6, 2018 from http://dwarfstd.org/doc/Dwarf3.pdf

[50] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, Washington, DC, USA, 79–93. https://doi.org/10.1109/SP.2009.25

[51] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. 2014. Comprehending Performance from Real-world Execution Traces: A Device-driver Case. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/2541940.2541968

[52] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/2048147.2048224

[53] Qin Zhao, Derek Bruening, and Saman Amarasinghe. 2010. Umbra: Efficient and Scalable Memory Shadowing. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, New York, NY, USA, 22–31. https://doi.org/10.1145/1772954.1772960