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. However, building such tools from scratch requires knowledge of low-level details of the language and 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 supports heavyweight dynamic analyses. It is based on binary instrumentation, which inserts calls to analysis functions written in JavaScript into a WebAssembly binary. Dynamically analyzing WebAssembly comes with several unique challenges, such as the problem of tracing type-polymorphic instructions with analysis functions that have a fixed type, which we address through on-demand monomorphization.

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, and (iii) makes it straightforward to implement various dynamic analyses, including instruction counting, call graph extraction, memory access tracing, and taint analysis.

CCS Concepts • Software and its engineering → Dynamic analysis; Software maintenance tools; Frameworks.

ACM Reference Format:

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 games1, 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

Wasabi takes ten lines of JavaScript, which use the frame-binary work's weight analyses that can monitor all low-level behavior. The Assembly serves as a compilation target of other languages, 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.

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

<table>
<thead>
<tr>
<th>Analysis framework</th>
<th>(Primary) platform</th>
<th>Instrumentation of ...</th>
<th>Analysis language</th>
<th>API style</th>
<th>(Primary) platform</th>
<th>Instrumentation of ...</th>
<th>Analysis language</th>
<th>API style</th>
<th>(Primary) platform</th>
<th>Instrumentation of ...</th>
<th>Analysis language</th>
<th>API style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin [31]</td>
<td>x86-64</td>
<td>native binaries</td>
<td>C/C++</td>
<td>instrumentation + callbacks/hooks</td>
<td>native binaries</td>
<td>x86-64</td>
<td>C</td>
<td>low-level instrumentation</td>
<td>JVM</td>
<td>byte code</td>
<td>JavaScript</td>
<td></td>
</tr>
<tr>
<td>Valgrind [34]</td>
<td>x86-64</td>
<td>native binaries</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JVM</td>
<td>byte code</td>
<td>binary code</td>
</tr>
<tr>
<td>DiSL [32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JavaScript</td>
<td>callbacks/hooks</td>
<td></td>
</tr>
<tr>
<td>RoadRunner [19]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JavaScript</td>
<td>callbacks/hooks</td>
<td></td>
</tr>
<tr>
<td>Jalangi [40]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JavaScript</td>
<td>callbacks/hooks</td>
<td></td>
</tr>
<tr>
<td>Wasabi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JavaScript</td>
<td>callbacks/hooks</td>
<td></td>
</tr>
</tbody>
</table>

1. const signature = {};
2. Wasabi.analysis.binary = \texttt{function}(loc, op) {
3. \quad \texttt{switch} (op) {
4. \quad \quad \texttt{case} "i32.add":
5. \quad \quad \texttt{case} "i32.and":
6. \quad \quad \texttt{case} "i32.shl":
7. \quad \quad \texttt{case} "i32.shr_u":
8. \quad \quad \texttt{case} "i32.xor":
9. \quad \quad \texttt{signature}[op] = (\texttt{signature}[op] \| \ 0) + 1;
10. \}\};

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

\footnotetext{\(\text{Wasabi}\) stands for WebAssembly dynamic analysis using binary instrumentation.
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). The increase in binary size and the runtime overhead imposed by Wasabi depend greatly on the program and which instructions in it shall be analyzed. For most instructions, thanks to selective instrumentation, code size increases by less than 1%, but in the worst case, when every single instruction in PolyBench/C is instrumented, code size increases by 742%. The runtime overhead can be below 1.02x for analyzing instructions such as `drop` and `select`, 2.8x for analyzing calls, and up to 163x when analyzing every single instruction of `heat-3d` from the PolyBench/C benchmark suite. These results are in line with existing frameworks for heavyweight dynamic analysis.

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 an 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
Module and sections

```module ::= function* global* start? table? memory?
function ::= typefunc (import | code) export*
global ::= typeval (import | init) export*
start ::= idxfunc
table ::= import? idxfunc* export*
memory ::= import? byte* export*
import ::= "name", export ::= "name"
code ::= (local type_val)* instr*
init ::= instr*
```

In instructions:
```
instr ::= nop | drop | select
| typeval:const value | unary | binary
| typeval:load | typeval:store | memory.grow
| (set | get | tee)_local idx_local | | (set | get)_global idx_global
| call idxfunc | call INDIRECT typefunc | return
| block | loop | if | else | end
| br label | br_if label | br_table label+
```

Types, labels, indices
```
typeval ::= i32 | i64 | f32 | f64
typefunc ::= typeval → typeval*
label ∈ ℕ, idxfunc | global | local ∈ ℕ
```

**Figure 3.** Abstract syntax of Mini-Wasm.

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, call1 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.

### 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, 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 variables and memory (local, global, load, store, memory_grow), function calls (call_pre, call_post, return), control flow (br, br_if, br_table), 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.

```
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.
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 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. 32, 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.js⁴ objects. Conditions, which are 32s with value 0 or 1 in WebAssembly, are mapped to JavaScript’s booleans.

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.

---

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

<table>
<thead>
<tr>
<th>Hook Name</th>
<th>Arguments and Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>const/drop</td>
<td>value: typeval</td>
</tr>
<tr>
<td>select</td>
<td>condition: boolean, first: typeval, second: typeval</td>
</tr>
<tr>
<td>unary</td>
<td>op, input: typeval, result: typeval</td>
</tr>
<tr>
<td>binary</td>
<td>op, first: typeval, second: typeval, result: typeval</td>
</tr>
<tr>
<td>local/global</td>
<td>op, index: number, value: typeval</td>
</tr>
<tr>
<td>memory_grow</td>
<td>delta: number, previousSize: number</td>
</tr>
<tr>
<td>load/store</td>
<td>op, memarg, value: typeval, memarg: { addr: number, offset: number }</td>
</tr>
<tr>
<td>call-pre</td>
<td>func: number, args, tableIndex:(number</td>
</tr>
<tr>
<td>call_post/return</td>
<td>results:[typeval]</td>
</tr>
<tr>
<td>br</td>
<td>target</td>
</tr>
<tr>
<td>br_if</td>
<td>target, condition: boolean</td>
</tr>
<tr>
<td>br_table</td>
<td>table:[target], tableIndex: number where target:(label: number, location: location)</td>
</tr>
<tr>
<td>begin</td>
<td>type, begin: location</td>
</tr>
<tr>
<td>end</td>
<td>type: string ∈ {function, block, loop, if, else}</td>
</tr>
<tr>
<td>every hook</td>
<td>location:{func: number, instr: number},...</td>
</tr>
</tbody>
</table>

---

⁴https://github.com/dcodeIO/long.js
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Original</th>
<th>Instrumented</th>
<th>Explanations and other changes made to the module</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constants</strong> (<strong>type_val</strong>. <strong>const</strong>), similarly simple instructions that only produce a value:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i32. const value</td>
<td>i32. const value</td>
<td>duplicate constant value for the hook</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>call idx_hooks.i32.const</td>
<td>instrumentation hook, also needs to be added as function import to module</td>
<td></td>
</tr>
<tr>
<td><strong>General instructions</strong> (unary, binary, load, store):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f32. abs</td>
<td>tee_local idx_temp input local</td>
<td>store instruction input(s) into freshly generated local(s)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>f32. abs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_local idx_temp input local</td>
<td>get_local idx_temp input local</td>
<td>store instruction result into freshly generated local</td>
<td></td>
</tr>
<tr>
<td></td>
<td>call idx_hooks.f32.abs</td>
<td>one low-level hook per instruction</td>
<td></td>
</tr>
<tr>
<td><strong>Calls</strong> (call and call_indirect), similarly also returns:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tee_local idx_temp input local</td>
<td>tee_local idx_temp input local</td>
<td>store call argument(s) in freshly generated local(s)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>get_local idx_temp input local</td>
<td>pass index of called function to hook</td>
<td></td>
</tr>
<tr>
<td>get_local idx_temp input local</td>
<td>call idx_hooks.call_pre(type_val)*</td>
<td>pass stored inputs to monomorphized call_pre hook</td>
<td></td>
</tr>
<tr>
<td>call idx_hooks.call_post(type_val)*</td>
<td>call idx_hooks.call_post(type_val)*</td>
<td>pass stored results to monomorphized call_post hook</td>
<td></td>
</tr>
<tr>
<td><strong>Polymorphic drop and select instructions:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 (preceding code)</td>
<td>...</td>
<td>type check all instructions to keep track of abstract stack</td>
<td></td>
</tr>
<tr>
<td>(preceding code)</td>
<td>...</td>
<td>here: assuming preceding instructions produce a stack top value of type type_val, then the following drop has type [type_val] → []</td>
<td></td>
</tr>
<tr>
<td>drop</td>
<td>call idx_hooks.drop_type_val</td>
<td>matching monomorphic hook call is inserted (consumes stack top in place of drop)</td>
<td></td>
</tr>
<tr>
<td><strong>Blocks/structured control-flow</strong> (block, loop, if, else, end) and branches (br, br_if, br_table):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>label: block</td>
<td>label: block</td>
<td>label is implicit and not encoded in the Wasm binary</td>
<td></td>
</tr>
<tr>
<td>otherlabel: loop</td>
<td>otherlabel: loop</td>
<td>every block type (block/loop/if/else) has own low-level begin hook</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>call idx_hooks.begin_block</td>
<td>loop begin hook is called once per iteration</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>call idx_hooks.begin_loop</td>
<td>“raw” (i.e., unresolved relative) target label is passed to hook as integer</td>
<td></td>
</tr>
<tr>
<td>i32. const 1</td>
<td>call idx_hooks.resolve(label)</td>
<td>resolved (at instrumentation time) label to next executed instruction is also passed</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>call idx_hooks.br</td>
<td>branch hooks must come before the instruction</td>
<td></td>
</tr>
<tr>
<td>call idx_hooks.end_loop</td>
<td>call idx_hooks.end_block</td>
<td>explicitly call end hooks of all “traversed” blocks, for dynamic block nesting</td>
<td></td>
</tr>
<tr>
<td>br 1 (label)</td>
<td>(not shown) end hooks receive location of the end and of the matching block begin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>end</td>
<td>every block type (block/loop/if/else) has own end hook (cf. begin_* hook)</td>
<td></td>
</tr>
<tr>
<td>call idx_hooks.end_loop</td>
<td>call idx_hooks.end_block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions with i64 inputs or results, value is split for passing to hook:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i64. const value</td>
<td>i64. const value</td>
<td>if instruction has side-effects, its result is duplicated via a local instead (but const here)</td>
<td></td>
</tr>
<tr>
<td>i64. const value</td>
<td>push lower 32-bit half of i64 value as i32 on stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i32. wrap/i64</td>
<td>i64. const value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i64. const 32</td>
<td>shift upper 32-bit half of i64 value to right, then push as i32 on stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i64. shr_s</td>
<td>cannot pass 16 values to hooks, so they take a tuple of (i32, i32) instead</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Instrumentation of (a subset of) WebAssembly instructions. Every hook also takes two i32s that represent the original instruction’s location. For brevity, we have omitted the corresponding i32. const instructions here.
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 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 $[\alpha] \rightarrow [\cdot]$. (That is, it pops an argument of any type $\alpha$ from the stack and pushes no value.) 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 `returns` is even unbounded. One way to address this 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 $[132] \rightarrow [f32]$, 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),

\(^5\)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].
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 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 `idxhooks.begin_block` and call `idxhooks.end_block`) at the beginning of a block and before the matching end.

Unfortunately, branching or returning 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 1oop 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, 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.

---

5 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.
### Analysis Hooks LOC

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Hooks</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction mix analysis</td>
<td>all</td>
<td>42</td>
</tr>
<tr>
<td>Basic block profiling</td>
<td>begin</td>
<td>9</td>
</tr>
<tr>
<td>Instruction coverage</td>
<td>all</td>
<td>11</td>
</tr>
<tr>
<td>Branch coverage</td>
<td>if, br_if, br_table, select</td>
<td>14</td>
</tr>
<tr>
<td>Call graph analysis</td>
<td>call_pre</td>
<td>18</td>
</tr>
<tr>
<td>Dynamic taint analysis</td>
<td>all</td>
<td>208</td>
</tr>
<tr>
<td>Memory access tracing</td>
<td>load, store</td>
<td>11</td>
</tr>
<tr>
<td>Cryptominer detection</td>
<td>binary</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4. Analyses built on top of Wasabi.

### 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, 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 in size, 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.

```
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.

**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.

**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

[^8]: [http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/](http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/)
Table 5 shows how long Wasabi takes to instrument the code processed per second, showing that the throughput increases with larger binary sizes.

<table>
<thead>
<tr>
<th>Program</th>
<th>Binary Size (B)</th>
<th>Runtime (ms)</th>
<th>MB x</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyBench (avg)</td>
<td>26 332 ± 299</td>
<td>23 ± 1.4</td>
<td>1.15</td>
</tr>
<tr>
<td>PSPDFKit</td>
<td>9 615 389</td>
<td>5 129 ± 65</td>
<td>1.87</td>
</tr>
<tr>
<td>Unreal Engine 4</td>
<td>39 510 398</td>
<td>15 481 ± 293</td>
<td>2.55</td>
</tr>
</tbody>
</table>

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

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 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 = 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, 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.\footnote{https://github.com/WebAssembly/wabt}

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 49.5% (Unreal Engine 4) and 74.3% (mean across the 30 PolyBench/C programs). 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

\footnote{https://github.com/WebAssembly/spec/tree/master/test}

\footnote{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.
When instrumenting for all hooks, the runtime overhead attributes only to parts of the total execution time. Other Web API, so any overhead imposed by Wasabi contributes only a small runtime overhead: 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 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].

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.

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 analyses 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.

Acknowledgments

This work was supported by the German Research Foundation within the Perf4JS and ConcSys projects, by the Hessian LOEWE initiative within the Software-Factory 4.0 project, and by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP.

References


