Wastrumentation: Portable WebAssembly Dynamic Analysis with Support for Intercession

Aäron Munsters
Vrije Universiteit Brussel, Brussels, Belgium

Angel Luis Scull Pupo
Vrije Universiteit Brussel, Brussels, Belgium

Elisa Gonzalez Boix
Vrije Universiteit Brussel, Brussels, Belgium

Abstract

Dynamic program analyses help in understanding a program's runtime behavior and detect issues related to security, program comprehension, or profiling. Instrumentation platforms aid analysis developers by offering a high-level API to write the analysis, and inserting the analysis into the target program. However, current instrumentation platforms for WebAssembly (Wasm) restrict analysis portability because they require concrete runtime environments. Moreover, their analysis API only allows the development of analyses that observe the target program but cannot modify it. As a result, many popular dynamic analyses present for other languages, such as runtime hardening, virtual patching or runtime optimization, cannot currently be implemented for Wasm atop a dynamic analysis platform. Instead, they need to be built manually, which requires knowledge of low-level details of the Wasm's semantics and instruction set, and how to safely manipulate it.

This paper introduces Wastrumentation, the first dynamic analysis platform for WebAssembly that supports intercession. Our solution, based on source code instrumentation, weaves the analysis code directly into the target program code. Inlining the analysis into the target's source code avoids dependencies on the runtime environment, making analyses portable across Wasm VMs. Moreover, it enables the implementation of analyses in *any* Wasm-compatible language. We evaluate our solution in two ways. First, we compare it against a state-of-the-art source code instrumentation platform using the WasmR3 benchmarks. The results show improved memory consumption and competitive performance overhead. Second, we develop an extensive portfolio of dynamic analyses, including novel analyses previously unattainable with source code instrumentation platforms, such as memoization, safe heap access, and the removal of NaN non-determinism.

2012 ACM Subject Classification Software and its engineering → Object oriented frameworks; Software and its engineering → Dynamic analysis; Information systems → Web applications; Security and privacy → Information flow control

Keywords and phrases WebAssembly, dynamic analysis, instrumentation platform, intercession

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.23

Funding *Aäron Munsters*: Funded by the Research Foundation Flanders, project number 1S53725N. *Angel Luis Scull Pupo*: Funded by the Cybersecurity Research Program Flanders (CRPF) from the Flemish Government.

1 Introduction

WebAssembly (Wasm) [13] has become a compilation target for many high-level languages such as C, C++, and Rust, thanks to its portability and predictable performance [31]. It was originally designed as a binary instruction set for the web and is currently supported by all major browsers.

Wasm offers predictable performance through its compact program representation, enabling efficient loading, validation and execution as parts of the program are being loaded. To safely execute untrusted code, it features control-flow integrity (ensuring programs cannot jump to unintended code locations), a modular system with strict encapsulation and type safety, and a sandboxed linear memory layout that isolates data from code and prevents unauthorized memory access [13].

Although Wasm was designed with built-in security and efficient mechanisms, it does not inherently address questions of software quality. For example, recent research reports performance issues in server-side Wasm applications [14] and memory safety issues [18, 23]. Dynamic analysis tools are increasingly being leveraged for Wasm to enhance security [9, 2], improve program comprehension [20, 30, 26, 39] and enable profiling [21].

To implement dynamic analyses, developers often rely on code transformation tools or general-purpose analysis platforms. In the context of Wasm, transformation tools like Brewasm [5] and WasmManipulator [29] allow developers to instrument the target program code by injecting additional Wasm instructions implementing the analysis. However, this approach requires a deep understanding of the instruction set and how it can be manipulated, such that the injected instructions uphold type safety and do not unintentionally interfere with the target program.

Alternatively, developers can use a general-purpose dynamic analysis framework for Wasm, either Wizard [38], or Wasabi [19]. These frameworks provide developers with a high-level API to implement analyses while avoiding unintentional analysis interference and ensuring type-safe instrumentation. Wizard's instrumentation framework is implemented within the Wizard execution engine. The integration at the level of the execution engine allows the instrumentation code to be turned on and off at runtime. Wasabi on the other hand instruments the target program with hooks that call into a JavaScript analysis which co-exists with the instrumented input program. Unfortunately, analyses implemented with these frameworks are restricted to concrete runtime environments. Wizard's instrumentation executes only on top of the Wizard engine, while Wasabi requires a JavaScript Virtual Machine (VM) along the Wasm VM. Moreover, analyses implemented with Wizard and Wasabi can only observe a program execution but not modify it, i.e. they do not support intercession. As a result, a wide range of dynamic analyses cannot be implemented with these frameworks, including runtime hardening [27, 18], virtual patching [34, 42], or runtime optimization [34, 43].

This paper presents Wastrumentation, the first general-purpose framework for dynamic analysis of WebAssembly that supports intercession and is portable among the many Wasm environments through its design as a source code instrumentation platform. To avoid the requirement of a JavaScript VM to be present for the instrumentation, Wastrumentation exposes its instrumentation API as a Wasm application binary interface (ABI). Furthermore, targeting a Wasm ABI has the benefit that any language that compiles to Wasm can be used to implement the analysis. Wastrumentation then merges both the target program and the analysis implementation as one Wasm program, making the instrumentation portable to any Wasm VM. To enable the construction of dynamic analyses requiring intercession, the instrumentation API of Wastrumentation enables the analysis to alter or skip operations happening at the target program side.

In summary, the main contributions of the paper are:

- We develop a novel instrumentation approach for Wasm programs to deploy dynamic analyses through source code rewriting featuring an instrumentation ABI akin to state-of-the-art high-level APIs but language-agnostic and extended with support for intercession.
- We show the usefulness of Wastrumentation by building an extensive portfolio of dynamic analyses, including a variety of existing analyses from Wasabi, as well as the implementation of three novel dynamic analyses that require intercession support, previously

unattainable with existing source code instrumentation platforms for Wasm. These analyses cover the domains of runtime optimization (memoization analysis), security (safe-heap analysis) and program repair ("DeNaN" analysis).
- We conducted a performance evaluation of Wastrumentation for the developed dynamic analyses on real-world programs from the Wasm-R3 benchmark suite [1], which shows improved memory consumption and competitive performance overhead compared to the state-of-the-art source code instrumentation platform, Wasabi [19].

2 Approach

This section describes our source code instrumentation platform for dynamic analysing Wasm applications. We first introduce Wastrumentation from the point of view of the analysis developer. Section 2.1 shows how to build a dynamic analysis atop of Wastrumentation, and Section 2.2 details the analysis API. Then, we turn our attention to the platform itself. Section 2.3 shows its architecture and gives an overview of the design choices. Section 2.4 details the transformations at the Wasm level where the instrumentation code is injected.

2.1 Developing a Dynamic Analysis with Wastrumentation

We now explain Wastrumentation from the perspective of an analysis developer implementing a concrete dynamic analysis. To this end, we present the "DeNaN" dynamic analysis, a novel dynamic analysis inspired by the "`--denan`" pass from the Wasm compiler framework Binaryen [10]. The "DeNaN" analysis ensures determinism among different Wasm VMs regarding floating point operations, as the Wasm language specification for "NaN Propagation"[1] does not enforce a canonical floating-point NaN representation. To address this, the "DeNaN" analysis replaces all runtime floating-point NaN values with a deterministic value 0.0.

To implement the "DeNaN" analysis, all Wasm instructions that may operate on runtime floating-point values must be intercepted. For each intercepted instruction, the analysis distinguishes between integer types (`i32`, `i64`), which are unaffected by NaN propagation, and floating-point type (`f32`, `f64`). Integer values remain unchanged, while floating-point NaN values must be replaced with 0.0. The relevant Wasm instructions include:
- Stack operations: `const`, pushes a constant on the value stack.
- Local and global variable operations: `local.get`, `local.set`, `global.get`, `global.set`, a "get" reads from a variable or global and pushed onto the value stack, a "set" does the opposite.
- Memory operations: `load`, `store`, a "load" reads from linear memory to the value stack, a "store" does the opposite.
- Arithmetic operations: `unary`, `binary`, operates on value(s) atop of the value stack.
- Function calls: passing arguments and receiving results for local, imported or exported functions.

Listing 1 shows a Rust implementation of the "DeNaN" analysis using the Wastrumentation API. Each Wastrumentation analysis implements *traps* that allow developers to register for specific Wasm events. The "DeNaN" analysis registers traps corresponding to the aforementioned instructions (Sections 2.1–2.1): `const_`, `local`, `global`, `load`, `store`, `unary`, `binary` and `apply`. The traps are defined using the `advice!` macro and are invoked by the instrumentation platform when corresponding events occur.

[1] WebAssembly 2.0 Specification – NaN Propagation (accessed on 28 Nov 2024).

The analysis instructs each trap function to capture each involved runtime value `v` and replace `v` with the result of `v.denan()`. It implements the method `denan` for Wasm values as an extension trait for the type `Value` (Sections 2.1–2.1). In Rust, extension traits allow to extend types with additional methods. This analysis declares the extension trait `Denan` (Section 2.1) and implements it for the type `Value` (Section 2.1) to allow invoking `denan` on Wasm values.

The implementation of `denan` (Section 2.1) matches on the type of the value. Integer values (matching `WasmType::I32` or `WasmType::I64`) are left untouched (Section 2.1). Floating point values (matching `WasmType::F32` or `WasmType::F64`) are cast to Rust's respective floating point type. On this floating type the method `is_nan` is called to intercept NaN instances. If `is_nan()` identifies a NaN instance, the value is replaced with 0.0, otherwise, it is returned unchanged (Section 2.1 and Section 2.1).

Listing 1 The Rust implementation of the "DeNaN" analysis. This analysis replaces all NaN floating-point values with 0.0 to ensure deterministic NaN-propagation across Wasm VMs.

```rust
#![no_std]
use wastrumentation_rs_stdlib::*;
advice! {
    const_ (v: Value, _l: Loc) {
        v.denan() }
    local (v: Value, _i: LocalIndex, _l: LocalOp, _l: Loc) {
        v.denan() }
    global (v: Value, _i: GlobalIndex, _g: GlobalOp, _l: Loc) {
        v.denan() }
    load (i: LoadIndex, o: LoadOffset, op: LoadOperation, _l: Loc) {
        op.perform(&i, &o).denan() }
    store (i: StoreIndex, v: Value, o: StoreOffset, op: StoreOperation, _l: Loc) {
        op.perform(&i, &v.denan(), &o); }
    unary (opt: UnaryOperator, opnd: Value, _l: Loc) {
        opt.apply(opnd.denan()).denan() }
    binary (opt: BinaryOperator, l_opnd: Value, r_opnd: Value, _l: Loc) {
        opt.apply(l_opnd.denan(), r_opnd.denan()).denan() }
    apply (func: WasmFunction, args: MutDynArgs, ress: MutDynResults) {
         args.update_each_arg(|_index, v| v.denan());
         func.apply();
         ress.update_each_res(|_index, v| v.denan()); } }
trait Denan { fn denan(self) -> Self; }
impl Denan for Value {
    fn denan(self) -> Self {
        use WasmType::{I32, I64, F32, F64};
        match self.type_() {
            I32 | I64 => self,
            F32 => self.as_f32().is_nan().then(|| 0_f32.into()).unwrap_or(self),
            F64 => self.as_f64().is_nan().then(|| 0_f64.into()).unwrap_or(self),
        } }}
```

2.2 Dynamic Analysis API

Table 1 shows the trap functions supported by Wastrumentation to build dynamic analyses. The table shows the high-level conceptual API, referring to Wasm constructs such as "location", "operator" and "operand". The trap functions are named after their corresponding program event. Each trap function gets as arguments details on the corresponding Wasm instruction and its code location. Such a conceptual API can then be implemented for a concrete high-level language that compiles to Wasm. Currently, our platform offers a Rust and AssemblyScript implementation of that API to implement dynamic analysis. The concrete language API is mapped onto the low-level Wasm instrumentation ABI by Wastrumentation at instrumentation time. The actual ABI for the conceptual API in Table 1 is specificied in Appendix A in [25].

Table 1 The dynamic analysis API supported by Wastrumentation. The columns show the trap name, the arguments provided to the trap, the type of value the trap must return and the implementation to ensure a transparent execution.

Trap name	Typed arguments	Return Type	Transparent Base
`if-then`$_{\texttt{pre}}$, `if-then-else`$_{\texttt{pre}}$	c:$type_{bool}$, input:$i32$, arity:$type_{i32}$, l:Loc	$type_{bool}$	c
`br`	l:$label$, l:Loc	$\emptyset$	
`br-if`	c:$type_{bool}$, l:$label$, l:Loc	$type_{bool}$	c
`br-table`	t:$i32$, effective-target:$label$, d:$label$, l:Loc	$type_{i32}$	t
`select`	c:$type_{bool}$, l:Loc	$type_{bool}$	c
`call`$_{\texttt{pre}}$, `call`$_{\texttt{post}}$	fn:idx_{func}, l:Loc	$\emptyset$	
`call-indirect`$_{\texttt{pre}}$	fn-idx:$i32$, tbl:idx_{tbl}, l:Loc	$type_{i32}$	fn-idx
`call-indirect`$_{\texttt{post}}$	tbl:idx_{tbl}, l:Loc	$\emptyset$	
`unary`	op:$unary$, o:$type_{val}$, l:Loc	$type_{val}$	op.apply(o)
`binary`	op:$binary$, l:$type_{val}$, r:$type_{val}$, l:Loc	$type_{val}$	op.apply(l, r)
`const`	v:$type_{val}$, l:Loc	$type_{val}$	v
`local`	v:$type_{val}$, i:idx_{local}, l:op_{local}, l:Loc	$type_{val}$	v
`global`	v:$type_{val}$, i:idx_{global}, g:op_{global}, l:Loc	$type_{val}$	v
`load`	i:idx_{load}, o:$offset_{store}$, op:op_{load}, l:Loc	$type_{val}$	op.perform(i, o)
`store`	i:idx_{store}, v:$type_{val}$, o:$offset_{store}$, op:op_{store}, l:Loc	$\emptyset$	op.perform(i, v, o)
`memory-size`	s:$type_{val}$, i:idx_{memory}, l:Loc	$type_{val}$	s
`memory-grow`	a:$type_{val}$, i:idx_{memory}, l:Loc	$type_{val}$	i.grow(a)
`block`$_{\texttt{pre}}$, `loop`$_{\texttt{pre}}$	input:$i32$, arity:$type_{i32}$, l:Loc	$\emptyset$	
`if-then`$_{\texttt{post}}$, `if-then-else`$_{\texttt{post}}$, `block`$_{\texttt{post}}$, `loop`$_{\texttt{post}}$, `drop`, `return`	l:Loc	$\emptyset$	
`apply`	f:$Function$, args:$Arguments$, ress:$Results$,	$\emptyset$	f.apply()

Our high-level API is inspired by Wasabi's instrumentation API, but it supports *intercession*, i.e. the ability of the analysis to modify the target program. This has two major implications for the design of the instrumentation API. We need to conceive (1) how the analysis controls the target program given the built-in safety and security characteristics of Wasm (detailed in Section 2.2.1), and (2) how the analysis can forward the intercepted operation to the target Wasm program (detailed in Section 2.2.2). The column "Transparent Base" in Table 1 denotes the operation that the analysis can call to forward the intercepted operation (captured at the trap function) to the target program.

2.2.1 Intercession: Modifying the Target Program

Dynamic analyses written on Wastrumentation can not only observe all the runtime behaviour of a target application but also control it. More concretely, analyses can change *part of* the target program behavior during the execution of the trap function. To illustrate this, consider the trap function `const_` that intercepts pushing a constant on the Wasm value stack. This trap function receives two arguments: the constant value `v` that would be pushed and the source code location `l` of the instruction. Wastrumentation allows the analysis to decide what value is pushed on the stack when the control flow returns to the target program by using the return value of the trap function. For example, the "DeNaN" analysis (cf. Section 2.1) uses this to alter the constant value `v` to a deterministic NaN value.

Generally, the API of Wastrumentation uses the return value of trap functions to alter the behavior of the target instruction. Whereas this is not always applicable (due to reasons further explained in Section 2.4), Table 1 shows when a return value dictates (part of) the target program behavior in the column "Return Type". Below, we further outline how the analysis is in control once a trap function is invoked.

- Intercession of control flow: For the trap functions `if`$_{\texttt{pre}}$, `if-then-else`$_{\texttt{pre}}$ `select`, and `br-if`, the return value dictates control flow of the branch expression as a boolean. For the trap function `br-table` the return value dictates the target label that control-flow will jump to. The return value of the trap function `call-indirect`$_{\text{pre}}$ determines the index in the function table of the instruction that will be invoked.
- Intercession of value stack manipulation: The following trap functions intercept instructions that push on the value stack: `unary`, `binary`, `const`, `local`, `global`, `load`, `store`, `memory-size` and `memory-grow`. The return value of the corresponding trap function is used as the outcome value that will be pushed on the stack.
- Introspection only: Aside for general intercession, the following trap functions cannot influence their respective instruction: `br`, `call`$_{\text{pre}}$, `call`$_{\text{post}}$, `call-indirect`$_{\text{post}}$, `block`$_{\text{pre}}$, `loop`$_{\text{pre}}$, `if`$_{\text{post}}$, `if-then-else`$_{\text{post}}$, `block`$_{\text{post}}$, `loop`$_{\text{post}}$, `drop` and `return`. This is due to how the respective program events do not depend on runtime information (eg. `br`) or report the termination of a program event (eg. `if-then-else`$_{\text{post}}$).
- Intercession of function application: The trap function `apply` can control a Wasm function application. When a function application takes place, this trap receives two pointers. The former points to an array containing the function arguments, and the latter points to the array containing the function results. This trap can alter the passed arguments before the function application takes place, and it can alter the return values before the control flow returns to the caller. Furthermore, this trap function is in control of whether the function application takes place at all.

2.2.2 Forwarding Intercepted Operations Back to the Target Program

Even if an analysis does not need to modify the target program, it remains responsible for the execution of intercepted program events. For example, if the trap function `const_` must remain transparent then the unaltered argument `v` must be the return value of the trap function. For the traps involved, the analysis is thus responsible for ensuring that the target operation takes place within the execution of the trap function. As mentioned before, Table 1 shows within the column "Transparent Base" the functions to be called within the trap function to forward the intercepted operation back to the target program.

Actually, the functions in the column "Transparent Base" represent the *continuation* of the intercepted operation that yields a return value. This design enables the analysis developer to add analysis code around the continuation of the intercepted operation and decide either to call the continuation or skip it altogether. This is most notable for the program events "apply" and "memory-grow". For "apply" the analysis controls function applications. This may further imply calling recursive functions. For "memory-grow", the return value either points to the next allocated page in linear memory or -1 if no allocation could be performed. This enables the analysis to artificially limit the available memory by not calling the memory to growth continuation and early returning with -1. For all other operations, the transparent base continuation is side-effect free and has no significant impact on analysis code that may go before or after the continuation.

2.3 Wastrumentation Design

Figure 1 depicts at a high-level the instrumentation process of Wastrumentation. We first describe the input provided to Wastrumentation, followed by the instrumentation process and the additional code that is generated. Then we elaborate on how the different modules are merged into a single output program. The numerical labels in the figure allow us to reference to visual elements in the following text.

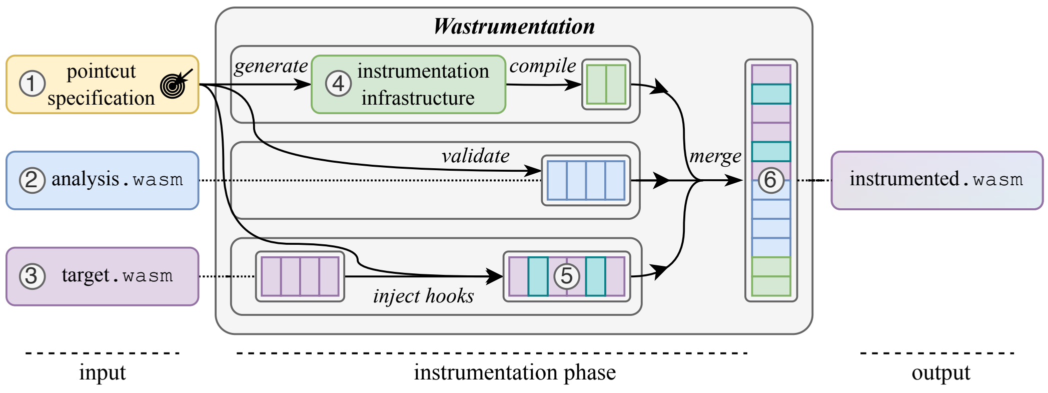


Figure 1 High-level overview of the architecture of Wastrumentation.

Input and Pointcut specification. In what follows, we adopt the pointcut specification terminology from aspect-oriented programming (AOP) to describe Wastrumentation inputs and their relationship. Note, however, that our implementation is limited compared to AOP, as our pointcut specification only supports static instruction targeting. As Figure 1 depicts, the input for Wastrumentation consists of the pointcut specification ①, the analysis program ②, and the target program to instrument ③. The pointcut specification enables selective instrumentation, by declaring the set of program instructions I and the set of functions of interest F within the target program. For example, the Rust implementation of the "denan" analysis in Listing 1 can be compiled to a Wasm module ②, along with a pointcut specification to target all implemented traps $I = \{const, local, global, load, store, unary, binary, apply\}$, and restrict instrumentation to the Wasm functions 4, 5, and 6 of the input program, $F = \{4, 5, 6\}$.

Instrumentation Process. For a given pointcut specification, Wastrumentation validates that the analysis module implements the trap functions corresponding to the specified target program instructions I. This validation takes place by iterating over the set I and validating that each corresponding trap function is exported by the analysis with the proper signature. If validation fails, Wastrumentation aborts the current instrumentation. If validation succeeds, Wastrumentation rewrites the identified instructions I within target functions F to inject hooks to placeholder analysis functions ⑤, according to the strategy explained in Section 2.4.

On-demand Instrumentation Code Generation. Some hooks pass compound data structures to traps functions that go beyond what the primitive number types of Wasm (i32, i64, i64, f64) can represent. For example, the *apply* trap function receives an array of Wasm arguments passed to the intercepted function call (cf. Section 2.4). Wastrumentation will generate additional instrumentation infrastructure as a new module ④. This infrastructure is generated on-demand, based on the input modules at hand. The module is responsible for managing such complex data structures, leaving the other modules untouched.

Bundling and Output. The final phase of Wastrumentation *bundles* the intermediate Wasm modules as a single instrumented module ⑥. To *bundle* these modules, Wastrumentation leverages the tool `wasm-merge` that is provided as part of the Binaryen tools [10]. This tool acts like a linker where multiple modules are merged into one. In doing so, `wasm-merge` replaces module imports that cross-reference other modules' exports with inline references to local functions in the resulting Wasm binary.

Preserving the Program State Space. Injecting analysis code into a target program comes with the challenge of preserving its original behavior. Any changes made during rewriting must not be observable for the input program. This requirement is critical to prevent deviations that could, for example, hide the program from a security analysis. The applied changes during instrumentation cannot interfere with the program state, including its memory, value stack, or call stack. This is unique to Wastrumentation. Other tools, e.g. Wasabi and Wizard, avoid this complexity because their analysis operates in a separate address space within the JavaScript interpreter or at the level of the Wasm interpreter, respectively. Our solution to preserve the input program state departs from the following properties of Wasm.

No Reflection. Wasm disallows runtime reflection regarding program code, the value stack, or the call stack [13]. This ensures that program code cannot inadvertently observe nor modify these components. Consequently, at execution time, the original input program code within the instrumented module remains unaware of the additional instrumentation code structure, values on the value stack and activation records on the call stack.

Separation of Linear Memory. Each memory load and write instruction in the instrumented module is statically tied to a unique memory view. When bundling the different code units (the target program, the analysis module, and additional instrumentation code), Wastrumentation leverages `wasm-merge` to assign distinct memory regions to each code unit. This separation leverages the Wasm standardized Multi-Memory proposal [12]. Multi-Memory enables to statically associate a unique memory index for every memory-related Wasm-instruction. This allows each module to operate exclusively on its unique linear memory index, ensuring complete isolation at runtime.

Figure 2 shows the statically enforced memory separation of merged program modules before and after program bundling. In the figure, the memory-relevant instructions such as `i32.load` and `i32.store` have a static memory index and a dynamic value that dictates the offset in the memory. Whereas each module has instructions addressing on its own set of linear memories, the instrumented module will contain the input program addressing the first indices of linear memory, followed by instrumentation and analysis modules operating at an on-demand decided memory index.

We ensure that the input program remains at its own offset, as external interoperation with the linear memory might make assumptions in the memory index, such as is the case for the widely adopted Wasm System Interface[2].

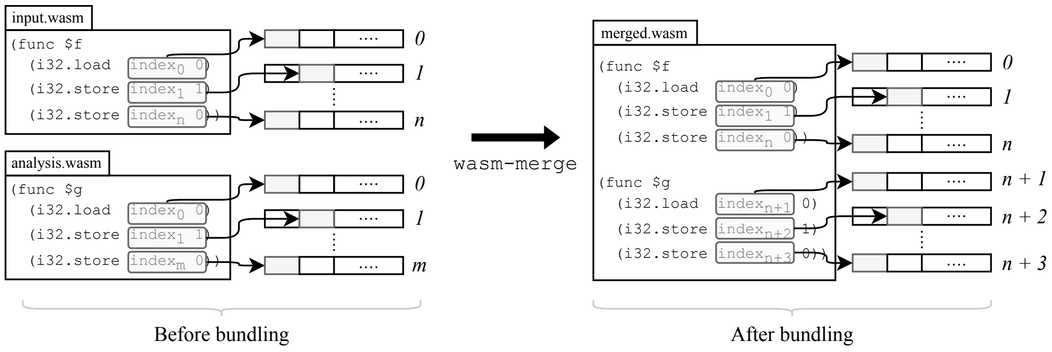


Figure 2 Visualization of the multi-memory access model before and after transformation.

[2] For our work we target the WASI P1, `https://wasi.dev` (accessed on 17 Dec 2024).

Listing 2 A analysis program that intercepts function applications and forwards the application.

```
(module ;; input analysis
  (func $apply (type $f_base:i32 $buf_sig:i32 $buf_sigtype:i32 $c_arg:i32 $c_res ⇒ void:i32)
    (; pre apply analysis code ;)
    (call $apply_base $f_base $buf_sig)
    (; post apply analysis code ;))
  (import "instrumented_input" "apply_base") (func $apply_base (param i32 i32)))
```

2.4 Instrumentation through Transformation

In what follows we detail the transformation strategy taken by Wastrumentation of the input program. The transformation follows two distinct patterns, one for function applications and another more general pattern for all other operations.

2.4.1 Function Application Instrumentation

The API of the “apply” trap receives a pointer to the *continuation function* and a pointer to a buffer containing the runtime arguments and results. The continuation function represents the next step in the execution flow. For the apply hook, this is the original function being intercepted. As Section 2.2.2 explains, the “apply” trap is responsible for calling the *continuation function* to allow a transparent execution of the target program. Inspired by the CLOS auxiliary “`around:`” method definition [3], the analysis code can (a) manipulate the arguments *before* the *continuation*, (b) determine *when* or *if at all* the function application takes place and (c) manipulate the results *after* the function application.

Listing 2 shows the minimal transparent analysis as a Wasm program that instruments function applications. The only implementation is the trap definition $\$apply$ (Section 2.4.1) that receives five arguments. The first argument (f_{base}) is a pointer to the *continuation function*. The second and third arguments (buf_{sig}, $buf_{sigtype}$) are pointers to buffers that contain the arguments and results for the current intercepted function application. The last two arguments (c_{arg}, c_{res}) dictate the number of arguments and results for that call respectively (to bound the value access of buf_{sig}, $buf_{sigtype}$). Section 2.4.1 calls the continuation, which may be preceded or succeeded with additional analysis code that is left out of the listing for brevity. Since the implementation of $\$apply_{base}$ resides in the transformed target program, the analysis must declare it as a function import (Section 2.4.1).

Listing 3 shows the input program for the transformation, containing a single function definition $\$f$. Whereas the transformation generalizes to any input set of function definitions, we restrict the example transformation to that of $\$f$ for brevity. Some intentional design considerations that influenced the transformation approach are the following:

- The analysis must be notified of all function applications of a target function, including (a) internal function applications through `call` or `call_indirect` statements local to the target module, (b) external function applications that may stem from the host and (c) for target functions that have no implementation details specification but are defined through an import statement.
- The API of the $\$apply$ trap is polymorphic for the target function application signature. This design choice enables low coupling between the analysis and the target input program, as the analysis program needs not to monomorphize the $\$apply$ trap implementation per signature as shown in [19]. Consequently, the analysis implementation must dispatch on function-specific behavior at runtime which may incur a performance penalty.

Listing 4 shows the input program result after transformation. The transformation of target function $\$f$ from Listing 3 results in two definitions $\$f_{original}$ (Section 2.4.1) and $\$f$ (Section 2.4.1). It is transparent to client code of $\$f$ (both internal and external to the Wasm

Listing 3 An input program candidate for dynamic analysis on function applications.

```
(module ;; input program
  (func $f (export "f") (type f_i^m ⇒ f_o^n) body_f))
```

module) since after transformation $\$f$ preserves its identity[3], its signature and its export declaration. However, $\$f$ will now call the trap $\$apply$ defined in Listing 2 with the required information.

In preparation for the call to $\$apply$, $\$f$ first stores arguments and runtime types to buffers (Sections 2.4.1–2.4.1) that $\$apply$ may manipulate at will. Next, $\$f$ yields control flow to $\$apply$ (Section 2.4.1), after which the results are loaded from the buffers (Sections 2.4.1–2.4.1) and eventually $\$f$ returns. The arguments to $\$apply$ (Section 2.4.1) are 0, the function pointer to the continuation function index in the table $\$table_{base-functions}$, the pointers $\$buf_{sig}$ and $\$buf_{sigvalues}$ pointing to the value- and types-buffers respectively and the number of arguments m and results n.

If the trap $\$apply$ in Listing 2 calls the continuation $\$apply_{base}$, in Listing 4 the applicable continuation is resolved at runtime with the function pointer argument $\$f_{base}$ that is looked up in the function table $\$table_{base-functions}$ (Section 2.4.1). The call to the resolved applicable continuation $\$f_{base}$ loads the arguments from the buffer $\$buf_{sig}$ (Section 2.4.1) and calls $\$f_{original}$ (Section 2.4.1), after which the results are stored to $\$buf_{sig}$ (Section 2.4.1) and control is yielded to the post-analysis code in the trap $\$apply$.

Listing 4 The input program from Listing 3 after the instrumentation phase.

```
(module ;; input program, instrumented
  (func $f_{original} (type f_i^m ⇒ f_o^n) body_f)
  (func $f (export "f") (type f_i^m ⇒ f_o^n)
    (local $buf_{sig} i32) (local $buf_{sigtype} i32)
    ;; push values on stack
    (local.get 0) (local.get 1) … (local.get m)
    ;; store args & types to instr-module-heap from value stack
    (local.set $buf_{sig} (call $alloc-vals-f_i^m-f_o^n))
    (local.set $buf_{sigtype} (call $alloc-typs-f_i^m-f_o^n))
    ;; call analysis apply
    (call $apply 0 $buf_{sig} $buf_{sigtype} m n)
    ;; retrieve results from instr-module-heap onto value stack
    (call $free-vals-f_i^m-f_o^n)
    (call $free-typs-f_i^m-f_o^n))
  (func $f_{base} (type $buf_{sig}:i32 ⇒ void)
    ;; retrieve args from instr-module-heap onto value stack
    (call $load-args-f_i^m-f_o^n (local.get $buf_{sig}))
    ;; call original body
    (call $f_{original})
    ;; store results to instr-module-heap from value stack
    (call $store-results-f_i^m (local.get $buf_{sig})))
  (import "analysis" "apply" (func $apply (type i32 i32 i32 i32 i32 ⇒ void)))
  (import "instr" "alloc-vals-f" (func $alloc-vals-f_i^m-f_o^n (type f_i^m ⇒ i32)))
  (import "instr" "alloc-typs-f" (func $alloc-typs-f_i^m-f_o^n (type void ⇒ i32)))
  (import "instr" "load-args-f" (func $load-args-f_i^m-f_o^n (type i32 ⇒ f_i^m)))
  (import "instr" "store-results-f" (func $store-results-f_i^m (type i32 ⇒ f_o^n)))
  (import "instr" "free-vals-f" (func $free-vals-f_i^m-f_o^n (type i32 ⇒ void)))
  (import "instr" "free-typs-f" (func $free-typs-f_i^m-f_o^n (type i32 ⇒ void)))
  (table $table_{base-functions} [$f_{base}] funcref)
  (func (export "apply_base") (param $f_{base} i32) (param $buf_{sig} i32) ;; => void
    (call_indirect $table_{base-functions} (type i32 ⇒ void) $f_{base}))
```

[3] Identifiers are defined as alphanumeric tokens prefixed with a \$-sign. In practice, Wasm functions are defined by an index as an i32-value, so a function name is actually a number that dictates its index.

The buffers used to communicate the runtime arguments and results between the function application, the analysis trap and the continuation function are handled by an external "instrumentation" module. This "instrumentation" module can orchestrate its linear memory while leaving the linear memory of the input program and the analysis code untouched. Functions to allocate and free memory, to load and store values in the buffers are imported in the instrumented module (Sections 2.4.1–2.4.1) and can be imported by the input analysis, though the pre- and post-analysis code is left out.

2.4.2 General Program Operation Instrumentation

All other instructions follow more concise rewrite strategies by Wastrumentation. These rewrite strategies rewrite each target operation into instructions that prepare context information on the value stack, followed by a call to an associated analysis trap function.

Table 2 summarizes the rewrite strategy, grouping instructions with similar rewrite patterns. The "Original instruction" column lists the unaltered instructions, while the "Rewritten instruction" column shows the instrumented result. The trap function is denoted as "`$trap`" for brevity but is specific to each instruction, such as "`$return`" for "return" or "`$br`" for "br".

As shown in Table 1, each trap function also receives the instruction's location. To provide this information, two constants (function index and instruction offset) are pushed on the stack before each trap call. These details are omitted from Table 2 for simplicity.

Maintaining the transformed program's well-typed property is essential. Thus the instrumented hooks must preserve the type correctness, including their interaction with the VM stack. Therefore, the analysis may modify the values handled by the target program, but it cannot alter their types. For instance, the rewrite of t.`const` c involves pushing the constant onto the value stack, followed by a call to the trap function "`$const`". This trap function receives the constant, returning a value of the same type while controlling its final value. Similarly, instructions like $t.unop$ and $t.binop$ are replaced with trap function calls that take the instruction operands and compute the resulting value on behalf of the program. The operands are available on the stack as they must have been for the original instruction.

For control-flow-related instructions such as "br-if", "br-table", and "select", the rewrite introduces a preceding trap function that controls the dynamic control flow. For "br-if" and "select," this is an `i32` value representing the condition. For "br-table," it is an `i32` value selecting a branch label from the statically predefined table associated with the instruction.

Instructions like "if-then-else", "if-then", "call-indirect", "call", "block", and "loop", include both pre- and post-trap functions. This design allows the analysis to trace the flow of execution per instruction, as additional instrumented code can be executed in between. For "if-then-else" and "if-then," the pre-trap function may modify the boolean condition controlling the flow. In contrast, trap functions for "return" and "drop" are observational, i.e. unable to alter control flow. For instructions like "local-get," "global-get," "local-set," "local-tee," "global-set," and "global-tee," the trap functions can alter the values being read or written to the respective variables.

3 Implementation

We implemented Wastrumentation in Rust and modularized the platform across different crates for improved maintainability and reusability. The core crate, as described in Figure 1, is responsible for transforming an input program based on a given analysis specification and pointcut specification. This crate consists of 5,118 lines of Rust code.

Table 2 The transformations applied by Wastrumentation.

Original Instruction	Rewritten instruction	Transformation description
	const	
t.const c	t.const c	Push constant on top of stack (TOS).
	call \$trap	Call trap with constant, return value is result.
	unary, binary	
		Operands are on TOS, will be args to trap.
t.*unop*	*call* \$trap	Call trap, return value is instruction result.
	br, br-if, br-table, select	
	(i32.const l)	Serialize target label (“br”).
	call \$trap	Call trap, passing serialized target label.
br l	br l	Return value of trap can change control flow
		for “br-if” and “br-table” and “select”.
	if-then-else, if-then	
	(*call* $\$trap_{pre}$	Call pre-trap with blocktype arity: n input
	(i32.const n) (i32.const m))	values on stack, m result values on stack.
if *blocktype*	if *blocktype*	Return value of pre-trap can alter control flow.
instr*	*instr**	
	(*call* $\$trap_{post}$)	Call post-trap.
else	else	
instr*	*instr**	
	(*call* $\$trap_{post}$)	Call post-trap.
end	end	
	call-indirect, call	
	i32.const x	Push f-idx (“call”) or tbl-idx (“call-indirect”).
	(*call* $\$trap_{pre}$)	Trap does not take call args, only context.
call_indirect x y	call_indirect x y	To view or alter args / results, use trap *apply*.
	(*call* $\$trap_{post}$ (i32.const x))	Call post-trap.
	block, loop	
	(*call* $\$trap_{pre}$	Call pre-trap with blocktype arity.
	(i32.const n) (i32.const m))	
block *blocktype*	block *blocktype*	
instr*	*instr**	
end	end	
	(*call* $\$trap_{post}$	Call post-trap with blocktype arity.
	(i32.const n) (i32.const m))	
	return, drop	
	call \$trap	Call trap before instruction.
return	return	Trap can only observe, not alter semantics.
	local-get, global-get	
local.get x	local.get x	Push value on the value stack.
	(*call* \$trap (i32.const x))	Call trap, return value is instruction result.
	local-set, local-tee, global-set, global-tee	
	(*call* \$trap)	Instruction argument is TOS, call trap.
local.set x	local.set x	Trap may alter argument for instruction.

The core crate accepts only Wasm modules as analysis inputs, as explained in Section 2.3. However, these analyses must adhere to the analysis ABI (see Appendix A in [25]), which is typically not written by hand. To facilitate interaction with this ABI, we developed a domain-specific language (DSL) that exposes the high-level API described in Section 2.2. The primary and most mature implementation of this DSL is written in Rust as a set of Rust macros, and it is used in our evaluation (in Section 4). We also have a less mature implementation written in AssemblyScript as an AssemblyScript library, demonstrating the language agnosticism of the analysis ABI. The Rust DSL comprises 1,943 lines of code, and the AssemblyScript DSL, which currently only partially abstracts the low-level ABI, consists of 450 lines of code. Figure 3 illustrates how the analysis, along with the DSL implementation, is compiled into a Wasm module that serves as input for Wastrumentation. Note that once an analysis is compiled to a Wasm module, the trap functions must adhere to the statically typed signature in the ABI (Appendix A in [25]), or otherwise, Wastrumentation will reject the analysis.

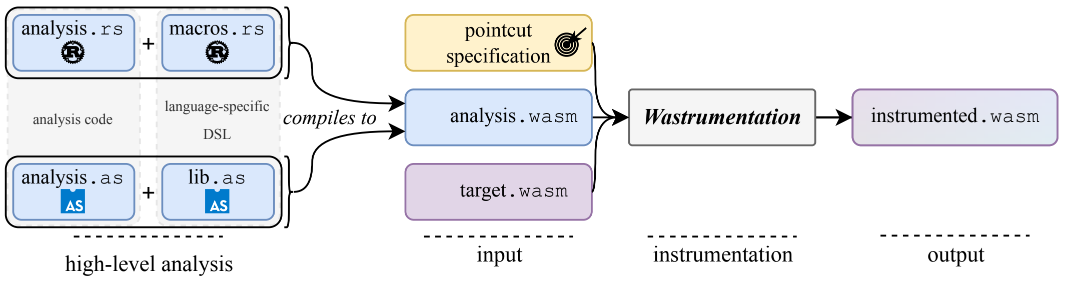


Figure 3 Overview of implementing analyses for Wastrumentation using a high-level language.

Rust and AssemblyScript are also supported for generating the instrumentation infrastructure. The code to generate the instrumentation infrastructure in either language amounts to a total of 4,451 lines of Rust code. Note that other languages can be supported, as long as a Rust-wrapper library can built around them, that implements the compiler interface used by Wastrumentation to compile the high-level program to a Wasm module.

4 Evaluation

We evaluate our work to check whether Wastrumentation is suitable for building dynamic analyses for Wasm applications. To this end, we assess the practicality and performance of our approach. We conduct several experiments to answer the following research questions (inspired by the ones found to evaluate other instrumentation platforms [19, 33, 35, 8, 38]):

RQ1. Does Wastrumentation aid the development of dynamic analyses compared to state-of-the-art approaches?
RQ2. Do the instrumented Wasm programs remain faithful to their original execution?
RQ3. How much does the code size increase after transformation?
RQ4. What is the runtime overhead due to instrumentation?
RQ5. What is the memory overhead due to instrumentation?

4.1 Methodology

We performed our experiments on two machines. The first is a server running Ubuntu 24.04.1 LTS (GNU/Linux 6.8.0-48-generic x86_64). The server has an AMD Ryzen 9 7950X processor (16 cores, 32 threads, 4.5–5.7GHz) and 128GB of main memory. The second machine is an Apple Mac Mini, with an M4 processor (4 performance and 6 efficiency cores) and 24GB of main memory.

We employed WasmR3 [1] commit 299be52, as target for a set of dynamic analyses. This benchmark suite includes 27 programs and serves as a reference for "realistic and standalone Wasm benchmarks" based on a 2024 record-reduce-replay pass of real-world programs.

To compare our approach to three state-of-the-art tools operating at three different levels:

- We use Wasabi to compare our approach to another source code instrumentation platform. We patched it to be compatible with WebAssembly 2.0 programs, commit ee2fb70
- We use Binaryen v122 to compare to a bytecode rewriting framework.
- We use Wizard [38] commit 1810b61 (compiled with Virgil [37] commit e64483e) to compare to a VM-level Wasm instrumentation framework.

In terms of runtimes, we execute Wasabi on Node V22.5.1, and we execute standalone Wasm programs on Wasmtime V30.0.2 and the aforementioned version of Wizard.

To avoid skewed results due to the VM warmup time or noise from other processes on the system, we compute the median of 30 executions of the target Wasm program for each individual runtime instance. When deploying an analysis with an instrumentation platform, we target every function of the input program and enable instrumentation for every program operation that the analysis depends on.

4.2 RQ1: Evaluating the Development of Dynamic Analysis Using Wastrumentation

To evaluate how Wastrumentation aids the development of dynamic analyses compared to state-of-the-art approaches, we implemented twelve dynamic analyses summarized in Table 3. The analysis code is written in Rust.

Table 3 The dynamic analyses implemented in Wastrumentation. We include the type of hooks and lines of code for the analyses in Wastrumentation and Wasabi. Those cells marked with ✗ correspond to analyses not possible to implement in Wasabi. The hooks used in [a] ("Basic Block Profiling") are `if_then_else, if_then, call pre, call_indirect pre, block pre, loop_ pre`, in [b] ("Branch Coverage") `if_then_else, if_then, br_if, br_table, select` and in [c] ("DeNaN") `apply, unary, binary, const_, local, global, load, store`.

Analysis Name	Hooks		Lines of code	
	Wasabi	Wastrumentation	Wasabi (JavaScript)	Wastrumentation (Rust)
Instruction Mix	all	all (introspection)	62	81
Basic Block Profiling	`begin`	6[a]	11	33
Instruction Coverage	all	all (introspection)	16	52
Branch Coverage	`if, br_if, br_table, select`	5[b]	21	30
Call Graph	`call_pre`	`call pre`	29	52
Dynamic Taint	all	all (introspection)	272	500
Memory Access Tracing	`load, store`	`load, store`	99	107
Cryptominer Detection	`binary`	`binary`	10	32
Forward	all	all	26	33
Denan	✗	8[c]	✗	35
Safe Heap	✗	`load, store`	✗	69
Memoization	✗	`apply`	✗	82

The first eight analyses in Table 3 ("Instruction Mix", "Basic Block Profiling", "Instruction Coverage", "Branch Coverage", "Call Graph", "Dynamic Taint", "Memory Access Tracing", "Cryptominer Detection") are analyses we ported from Wasabi [19][4]. We use the Wasabi analysis implementations available on the latest commit 21a322b in their public repository. The lines of code for the ported implementation in Rust remain in the same order of magnitude than those implemented in JavaScript for Wasabi. The main reason for the code size increase is to conform with the Rust type system and borrow checker (e.g., strict typing, synchronize on mutable global state). The ported analyses show that Wastrumentation does not sacrifice in terms of its ability to implement existing analyses.

The latter four analyses in Table 3 ("Forward", "DeNaN", "Safe Heap", "Memoization") are new analyses. More concretely, we implemented "Forward" in both platforms, and the last three are novel analyses that require intercession (which is not supported in Wasabi). We describe those analyses in what follows.

[4] Due to space constraints, the code for those analyses is included in the artifact. The semantics of the analyses are the same as they were implemented for Wasabi for the evaluation [19].

- **Forward**. This analysis implements all trap functions without altering the original program (cf. Table 1, column "Transparent Base"). It provides insight into the instrumentation platform's minimum performance overhead as no additional analysis code is included. Section 4.3 also uses it as a reference analysis to validate that input programs remain faithful to their execution semantics after instrumentation.
- **Denan**. This analysis replaces all `NaN`-floating point values in the program with the floating point value zero. The analysis is inspired by the "`--denan`" instrumentation pass from Binaryen. Section 2.1 showed our implementation and Section 4.2.2 compares it to the ad-hoc implementation in Binaryen.
- **Safe Heap**. This analysis checks for incorrect heap access. It is inspired by the "`--safe-heap`" instrumentation pass from Binaryen. Listing 5 shows its implementation and Section 4.2.2 compares it to the ad-hoc implementation in Binaryen. For every load or store operation, the analysis checks null dereferencing, reading past available memory and usage of addresses with incorrect memory alignment (Sections 4.2–4.2).
- **Memoization**. This analysis implements a memoization strategy of pure functions [24]. Its goal is to cache the outcome of pure function applications to save on repeated computation efforts. Section 4.2.1 details our implementation.

For didactical purposes, we omit (1) synchronization primitives on global state, (2) type casting and (3) method invocations for type conversion in Listing 5 and Listing 6.

Listing 5 Implementation of Binaryen's "`--safe-heap`" pass in Wastrumentation.

```rust
use wastrumentation_rs_stdlib::*;

const WASM_PAGE_SIZE:i64 = 65_536;
fn bounds(idx:i64,byts:i64,ofst:i64) {
 // strict_add: assert no overflow
 let last_idx_byte =
  idx.strict_add(byts).strict_add(ofst);
 assert!((last_idx_byte != 0) &&
  (last_idx_byte <=
   (base_memory_size(0)) *
    WASM_PAGE_SIZE)); }

// turn size into bitmask (‘-1‘), then
// assert mask-region is zero-only (‘&‘)
fn alignment(idx:i64, size:i64) {
 // size == 4 (_32) | 8 (_64)
 assert!(idx & (size - 1) == 0); }

advice! {
 load (i:LoadIndex,o:LoadOffset,
       op:LoadOperation,l:Loc) {
  bounds(i,op.target_size(),o.offset());
  alignment(i,op.target_size());
  op.perform(&i,&o)
 }
 store (i:StoreIndex,v:WasmValue,
        o:StoreOffset,op:StoreOperation,
        l:Loc) {
  bounds(i,op.target_size(),o.offset());
  alignment(i,op.target_size());
  op.perform(&i,&v,&o); } }
```

Listing 6 A dynamic analysis implementation that caches the result of stateless function.

```rust
use std::collections::HashMap;
use wastrumentation_rs_stdlib::*;

struct Key{fn_idx:i32,ags:Vec<Byte>}
static mut CACHE:HashMap<Key,Vec<Value>>
 = HashMap::new();

fn cache_hit(key:&Key) -> bool {
 CACHE.contains_key(&key) }

fn cache_retrieve(
 key:&Key, results:&mut Results) {
 CACHE.get(&key).unwrap().into_iter()
  .enumerate().for_each(|(i, v)|
   results.set_res(i, v.to_owned()));
}

fn cache_insert(key:Key, rss:&Results) {
 let rss = rss.rss_iter().collect();
 CACHE.insert(key, rss);
}

advice! { apply (
 fnc:WasmFn, ags:Args, rss:Results) {
  let ags = ags.map(to_bytes).collect();
  let key = Key {fn_idx: fnc.idx, ags};
  if cache_hit(&key) {
   cache_retrieve(&key, &mut rss);
  } else {
   fnc.apply();
   cache_insert(key, &rss); } } }
```

4.2.1 The Memoization Analysis

We now discuss the details of the memoization analysis. Listing 6 shows its implementation in Wastrumentation. The analysis maintains a global hashmap as *cache* where it associates a function index and its arguments with the results for that function call (Sections 4.2–4.2). The analysis targets the *apply* trap since only function applications will be memoized (Section 4.2). The trap implementation computes the cache key (Sections 4.2–4.2) and returns the associated result values if the key is stored in the cache. If the cache had no associated entry for the computed key, the function application continues and the computed result is stored in the cache for successive function applications (Sections 4.2–4.2).

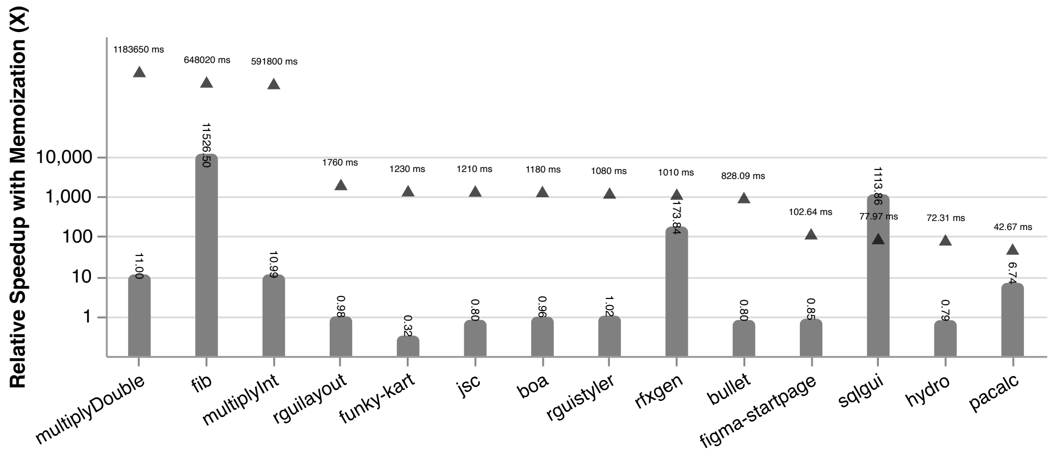


Figure 4 Execution times after instrumentation with the memoization analysis. The programs are sorted by their absolute runtime in miliseconds (▲) from left to right. The bar charts indicate the relative speedup on a logarithmic scale.

To assess the effectiveness of the memoization analysis, we conduct an experiment to check whether it would benefit the programs in the WasmR3 benchmark suite executed on Wasmtime. We first perform a static analysis phase to identify the programs with pure functions which could be good candidates for memoization. This phase identified 14 candidate programs (out of the 27) featuring pure functions. Figure 4 plots the measured runtimes of those programs and their instrumented variants. Our results show that the analysis can incur a performance penalty for some programs, and it can speed up others. The performance penalty ranges from 1.02x slowdown (rguilayout) to 3.10x slowdown (funky-kart), while the speed-up ranges from 1.02x speedup (rguistyler) to 1,1526.50x speedup (fib). This experiment shows that such analyses can be developed at low cost for exploration, and yield the potential to then be ported within concrete VM implementations if they offer promising results.

4.2.2 Comparison to Bytecode Rewriting Frameworks

As mentioned before, state-of-the-art dynamic analysis platforms for Wasm do not support intercession, but bytecode rewriting frameworks such as Binaryen [10] and Walrus [32] do. We observe that Binaryen v122 offers six instrumentation passes: "`--safe-heap`", "`--denan`", "`--instrument-locals`", "`--instrument-memory`", "`--log-execution`", "`--trace-calls`". Out of the six, all offer support for intercession except for "`--log-execution`" . Unfortunately, the "`--denan`" analysis is the only instrumentation pass that rewrites the input program into a variant that can execute standalone. All other instrumentation passes rewrite the target program such the host must implement a part of the analysis, eg. what to perform when observing reads or writes into locals for the "`--instrument-locals`" pass. This couples the analysis with host-provided functionality. Titzer et al. developed in [38] two standalone ad-hoc analyses, "branches" and "opcodes", in the Walrus bytecode rewriting framework. To further evaluate RQ1, we compare the implementation of these three ad-hoc analysis implementations ("DeNaN", "branches", and "opcodes") that can execute standalone with an equivalent analysis on Wastrumentation.

When comparing the implementation of a given analysis on Wastrumentation to an ad-hoc solution, we observe three main limitations of bytecode rewriting frameworks.

Higher implementation effort. We observe repeated efforts of constructing the low-level Wasm constructs and injecting these within the target program when using bytecode rewriting frameworks. This most notably affects the code size of the analyses: "DeNaN" (195 LoC on Binaryen, 35 on Wastrumentation), "branches" (291 LoC on Walrus, 24 on Wastrumentation), and "opcodes" (242 LoC on Walrus, 81 on Wastrumentation). The lines of code for analyses in Wastrumentation are smaller since the transformation efforts are reused among analyses as they happen within Wastrumentation and Wasm-Merge.

Semantic abstraction loss. We observe a notable loss of high-level semantic expressiveness in bytecode rewriting systems because analysis implementations are restricted to low-level, elementary operations due to the lack of access to rich abstractions, such as complex data structures and a strong type system. For example, "DeNaN" analysis is limited to changing a value to 0 while the "branches" and "opcodes" increment a fixed set of global counters by 1. The other analyses implemented by Binaryen ("`--safe-heap`", "`--instrument-locals`", "`--instrument-memory`", "`--log-execution`", "`--trace-calls`") avoid additional complexity by forcing the analysis developer to implement more complex data structures within the host, such as logging infrastructure, hashmap dictionaries, or debugging infrastructure. In contrast, implementing the analyses on a general-purpose analysis platform is easier since developers do not need to manipulate Wasm instructions nor ensure the type safety of the analysis code. In Wastrumentation, developers can implement the analysis using a high-level API and in a high-level language like Rust. This reduces the complexity of the analysis by raising the abstraction level, allowing for a compiler to type-check, validate, and optimize the analysis code.

Lack of Separation of Concerns. We observe the analysis logic is tightly interwoven with the transformation pass, resulting in reduced modularity and maintainability. This means that instructions such as 'increment a counter' (analysis code) are scattered among instructions such as 'extend target body with increment' (transformation code). Wastrumentation decouples these concerns by treating the analysis implementation as a unit separate from the instrumentation phases. As a result, it becomes easier to maintain, comprehend, and evolve the semantics of the analysis independently from instrumentation.

Finally, we would like to note that the Binaryen instrumentation pass "DeNaN" is incomplete. The implementation of the "`--denan`" pass in Binaryen documents does not support the instrumentation of `local.get`, as doing so "would cause problems if we ran this pass more than once (the added functions use gets, and we don't want to instrument them)" [11]. Since Wastrumentation decouples instrumentation from the analysis implementation, the instrumentation of `local.get` operations do not run the risk that analysis code and its instructions of `local.get` are instrumented too.

Table 4 Violated assertions when deploying the Forward analysis using Wastrumentation on the Wasm official test suite.

	Tool	Execution Stage	Warning	# Modules	# Assertions
(1)	Wastrumentation	Transformation	Expected type f64, got f32	1	0
(2)	Wasm-Merge	Transformation	Unknown misc operation	8	36
(3)	Wasmtime	Runtime	Uninitialized element	4	4
(4)	Wasmtime	Runtime	Equality assertion failure	2	2
(5)	Wasmtime	Validation	Undeclared func. reference	2	8

4.3 RQ2: Evaluating the Faithful Execution

We take an experimental approach to validate that input programs remain faithful to their execution semantics after instrumentation. We validate that instrumented programs maintain structural integrity and correct execution semantics across (a) the 4,082 programs in the official Wasm test suite and (b) the 27 programs in the WasmR3 benchmark suite.

To validate the faithful execution of the 4,082 programs from the official Wasm test suite, we instrumented them with the "Forward" analysis (presented in Section 4.2). This analysis targets every supported trap without altering the semantics of the original program. The official Wasm test suite includes 26,632 assertions. We disabled those that assert failures (i.e., assertions that expect an invocation to trap)[5], as the goal of the experiment is to show that the "Forward" analysis does not alter successful tests. Disabling assertions on failures ensures proper execution of subsequent assertions. Otherwise, an exception in the input program halts the module's execution, preventing the instrumentation code from being notified and unable to clean up its internal state[6]. While the forward analysis is stateless, the instrumentation infrastructure remains stateful with respect to the apply trap (cf. Listing 4). For this reason, we disable failure assertions to retain all success assertions.

For the remaining 21,392 assertions, Wastrumentation has a success rate of 99.77%, with only 50 assertions failing across 17 modules. Table 4 categorizes the failures by the tool reporting the failure (Wastrumentation, Wasm-Merge, or the Wasmtime runtime), the stage at which the failure occurred (Transformation, Validation, or Runtime), the reported warning, the number of affected modules, and the number of affected assertions.

Failure (1) stems from `wasabi-wasm` [17], the underlying transformation library used by Wastrumentation which fails to infer the types of the input program. Failure (2) originates from Wasm-Merge, which cannot identify an input operation. We suspect this can be solved by enabling more feature flags provided by Wasm-Merge as they enable more Wasm instructions. Failure (3) results from uninitialized elements (static declared Wasm memory segments), which Wastrumentation does not directly manipulate. This issue could be addressed with further investigation into the serialization of our transformed module or the merge step. Failure (4) is a runtime failure caused by disabling previous assertions after they failed, violating the expected state. Finally, failure (5) requires more investigation.

In a second experiment, we verify Wastrumentation using the 27 programs of the WasmR3 benchmark suite across all 12 analyses listed in Table 3. For each input program in the WasmR3 and each analysis, we used `wasm-validate`, a Wasm binary validation tool, to verify the structural integrity of the code after instrumentation. The results showed no errors reported by `wasm-validate` for any of the analyses, confirming that Wastrumentation produces valid Wasm code that fully adheres to the requirements of the Wasm binary format.

[5] The following assertions (including count) were disabled: "assert_invalid" (1,476), "assert_trap" (2,388), "assert_exhaustion" (15), "assert_malformed" (1,277), "assert_unlinkable" (84).

[6] A proposal for Wasm exception handling is not yet standardized. If it were standardized, a set of exceptions could be accommodated by the instrumentation platform to clean up the analysis state.

4.4 RQ3: Evaluating Code Size Increase

We now compare the code size increase of Wastrumentation with respect to the state-of-the-art source code instrumentation platform Wasabi. To compare binary output size after instrumentation for both platforms, we computed $s = \frac{\text{code-size}_{\text{wasabi-instrumented}}}{\text{code-size}_{\text{wastrumentation-instrumented}}}$. This ratio s indicates where Wastrumentation-instrumented programs require fewer bytes ($s > 1$) or when Wasabi-instrumented programs require fewer bytes ($s < 1$).

We calculated the s ratio for each instrumentation of the WasmR3 benchmark suite using the dynamic analyses supported by both platforms. More concretely, the dynamic analyses used are the first eight analyses in Table 3 along with an adapted version of the "Forward" analysis. To ensure a fair comparison, we modified our "Forward" analysis to only include hooks that are compatible with Wasabi.

Note that, for Wasabi, the analysis logic is implemented on the JavaScript side, whereas, in Wastrumentation, it is embedded as Wasm code within the instrumented binary. Since JavaScript and Wasm serve different execution models and cannot be directly compared, we exclude the JavaScript analysis implementation from the code size comparison for Wasabi's instrumented programs to ensure a fair evaluation.

4.4.1 Results

Figure 5 shows the ratio s for each instrumentation in the cross-product. In cases where Wasabi failed to instrument a specific input program, we leave the cell blank as no ratio can be computed. Additionally, Figure 6 shows the size of the input programs prior to any instrumentation. The code size increases for each platform individually are presented in Appendix B in [25].

Out of a total of 180 successful ratios, 140 indicate the output Wasm module code generated by Wastrumentation is smaller, while the remaining 40 favor Wasabi. The analyses "Basic Block Profiling", "Branch Coverage" and "Call Graph" tend to favor Wasabi, which aligns with the low number of target operations in these analyses (6, 5 and 1, respectively). For the other analyses, which involve a larger number of traps, the size ratio generally favor Wastrumentation, resulting in smaller binary sizes compared to Wasabi.

The noticeable "red stripe" for the input program "game-of-life" indicates a significant advantage for Wasabi's instrumentation. This is largely due to the significant difference in binary sizes among the input programs. Whereas "game-of-life" has an uninstrumented size of just 1.199 bytes, the other input programs range from 38.965–22.264.896 bytes. This suggests that Wastrumentation introduces a relatively larger size overhead for smaller programs compared to Wasabi, which may be expected as this metric counts for Wastrumentation the additional analysis code, whereas it does not count the JavaScript analysis code for Wasabi.

4.5 RQ4: Evaluating the Runtime Overhead of Wastrumentation

To assess the runtime overhead of Wastrumentation, we performed 4 experiments. The first experiment evaluates the runtime overhead for Wastrumentation for its load, validate and execution of the "Forward" analysis. The second experiment computes the execution overhead of Wastrumentation for the 12 analyses discussed in Table 3. The third experiment compares the runtime overhead to the Wasabi source code instrumentation platform. Lastly, we compare the performance overhead to state-of-the-art bytecode rewriting frameworks.

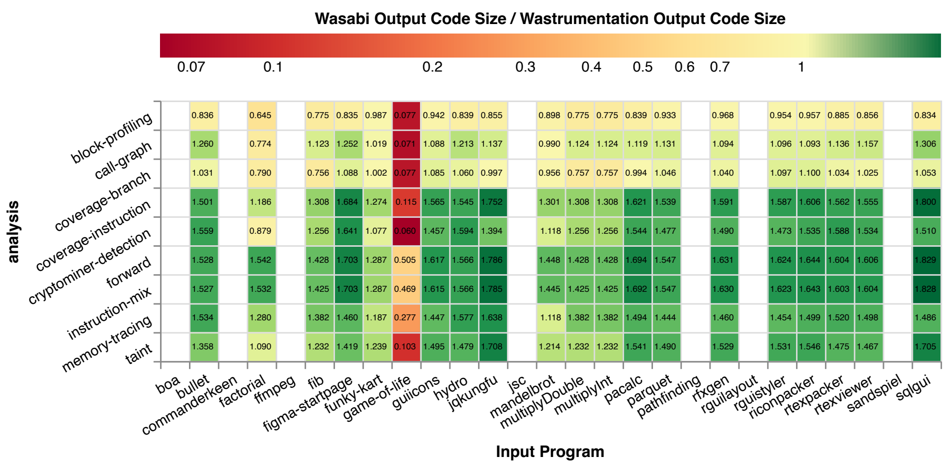


Figure 5 Code size ratio of Wasabi-over Wastrumentation-analyzed programs.

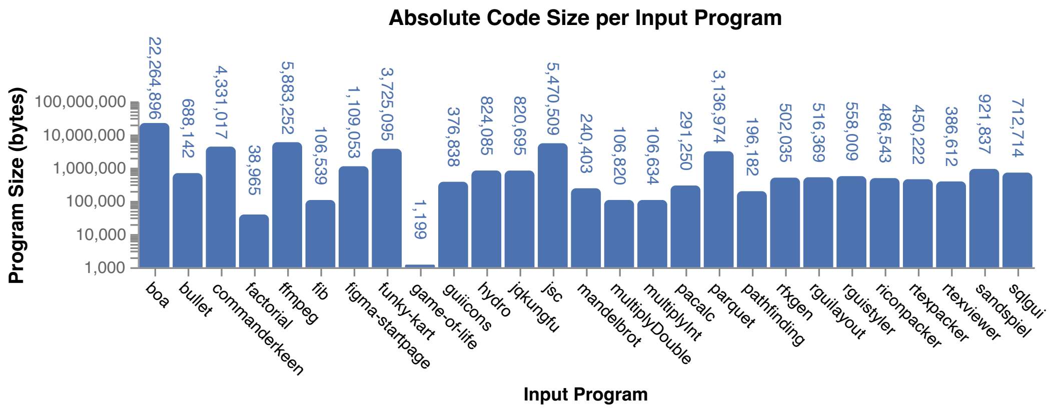


Figure 6 Code size of input programs from the WasmR3 benchmark suite.

4.5.1 Load, Validate and Execute Time

In this section, we evaluate the cost of loading, validating and executing programs for each stage for the input programs of the WasmR3 input suite when instrumenting with the "Forward" analysis. Figure 7 plots the overhead for each stage for 20 instrumented input programs with Wastrumentation. The overhead is computed using the Wizard engine. The median is computed for both the uninstrumented and instrumented variant over 30 runs. For the remaining input programs, 5 timed out, and 2 could not execute.

We observe that the relative increase for loading and validating an instrumented variant correlates with the absolute code size increase from Section 4.4. This is because both phases require reading the input program into the execution engine and performing a static validation on it. For loading, this is in the range 1.13–5.37, for validating in the range 1.84–5.65.

Second, we observe that the increase in execution time spent for the input programs differs significantly depending on the input program. We further ported 10 analyses other than "Forward" to Wastrumentation for further comparison, however other analyses execution failed with the message "`!trap[MEM_OUT_OF_BOUNDS]`". We believe this error does not originate for the "Forward" analysis since it does not reserve additional linear memory in that analysis code. To the best of our knowledge, this may point out a regression for the Wizard baseline engine, since no such exceptions occurred when executing the same experiments on other execution engines such as NodeJS, Wasmtime, and Wasmer.

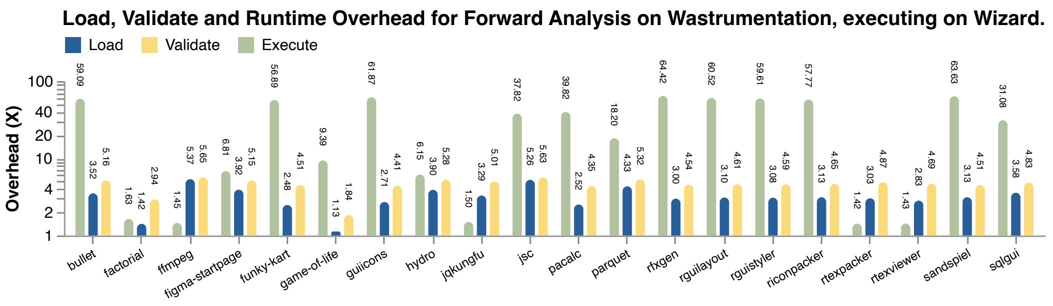


Figure 7 Overhead for loading, validating and executing time after instrumenting WasmR3 programs with the "Forward" analysis on Wastrumentation, executing on Wizard.

4.5.2 Setup for NodeJS experiments

To determine the runtime overhead of Wastrumentation, we compute the ratio o as the performance of instrumented execution relative to uninstrumented execution. An o value of 2 means instrumentation slows the execution time down by a factor of 2.

To compare the performance overhead for Wastrumentation and Wasabi, we computed a ratio p as the execution time of a Wasabi-instrumented module relative to the execution time of a Wastrumentation-instrumented module. A value $p > 1$ indicates Wastrumentation-instrumented execution requires less time to complete, and a value of $p < 1$ indicates Wasabi-instrumented execution requires less time to complete. For example, a p value of 0.5 means that "Wasabi is twice as fast at runtime to analyse the program", while a p value of 2 means that "Wastrumentation is twice as fast at runtime to analyse the program".

Similar to Section 4.4, we compute the ratio's o and p for the programs in WasmR3 using the analyses supported by both platforms. We profile the execution of uninstrumented and instrumented programs using the performance observation API from NodeJS "perf_hooks". The recorded times do not include the program loading from main memory or the verification or compilation of the module. As mentioned in Section 4.1, we calculate the median of 30 executions of the program within a single NodeJS instance. During our experiments, the execution of an instrumented program was limited to a timeout of 300 seconds.

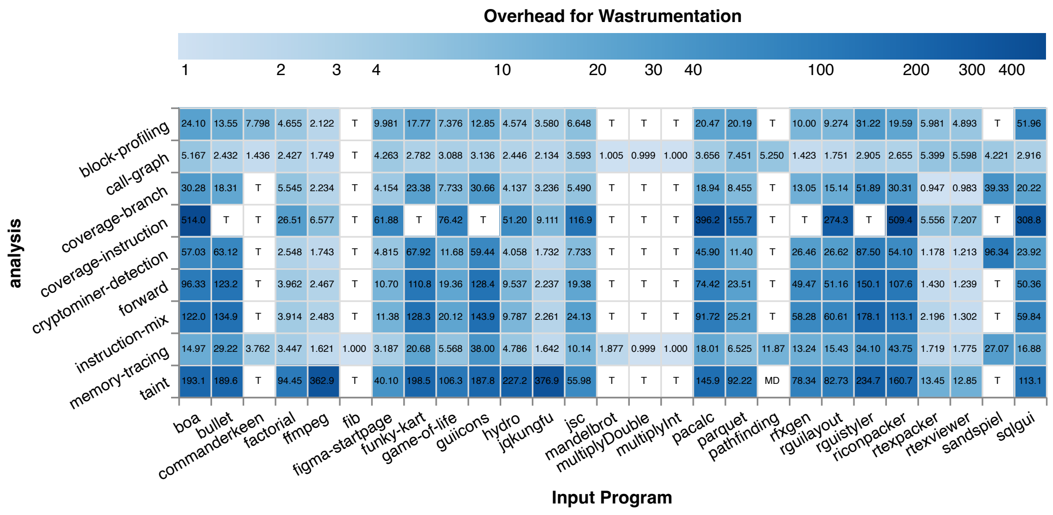
4.5.3 Runtime overhead of Wastrumentation

Figure 8 shows the results for the overhead ratio o. The white cells indicate missing data, either due to the instrumented execution exceeding the timeout (T) or due to raising an uncaught exception (MD). In total, 51 combinations timed out, while 1 raised an uncaught exception ("memory-tracing" for the input program "pathfinding").

The results show that the instrumentation overhead for Wastrumentation ranges from 1x–514.0x. The analyses "coverage-instruction" and "taint" generally incur a higher overhead of 5.5x–514.0x and 12.85x–376.9x, respectively. These analyses are, however, heavy-weight as they implement all traps and maintain heap-allocated data structures that grow during the target program execution (a coverage hashmap and growable vectors, respectively). Nevertheless, Wastrumentation's overhead aligns with expectations for similar source code instrumentation platforms such as Wasabi [19], Jalangi [33] and Aran [6].

4.5.4 Comparison to Wasabi

Figure 9 shows the results for p ratio used to compare the runtime overhead for Wastrumentationand Wasabi. Note that figure only plots 20 input programs, as 7 input programs could not be instrumented by Wasabi so no ratio could be computed. The white cells indicate missing data, labeling T for "timeout", followed by the platform "M" (Wastrumentation), "S" (Wasabi) or "M&S" (both). For a total of 180 combinations where both Wastrumentation and Wasabi yield a valid instrumented module, 140 combinations yield a measurement. For 31 combinations both Wastrumentation and Wasabi timeout. For Wastrumentation an additional 3 combinations timeout and for Wasabi an additional 6. The performance overhead individually per platform for additional reference are further detailed in Appendix C in [25].



analysis	boa	bullet	commanderkeen	factorial	ffmpeg	fib	figma-startpage	funky-kart	game-of-life	guiicons	hydro	jqkungfu	jsc	mandelbrot	multiplyDouble	multiplyInt	pacalc	parquet	pathfinding	rfxgen	rguilayout	rguistyler	riconpacker	rtexpacker	rtexviewer	sandspiel	sqlgui
block-profiling	24.10	13.55	7.798	4.655	2.122	T	9.981	17.77	7.376	12.85	4.574	3.580	6.648	T	T	T	20.47	20.19	T	10.00	9.274	31.22	19.59	5.981	4.893	T	51.96
call-graph	5.167	2.432	1.436	2.427	1.749	T	4.263	2.782	3.088	3.136	2.446	2.134	3.593	1.005	0.999	1.000	3.656	7.451	5.250	1.423	1.751	2.905	2.655	5.399	5.598	4.221	2.916
coverage-branch	30.28	18.31	T	5.545	2.234	T	4.154	23.38	7.733	30.66	4.137	3.236	5.490	T	T	T	18.94	8.455	T	13.05	15.14	51.89	30.31	0.947	0.983	39.33	20.22
coverage-instruction	514.0	T	T	26.51	6.577	T	61.88	T	76.42	T	51.20	9.111	116.9	T	T	T	396.2	155.7	T	T	274.3	T	509.4	5.556	7.207	T	308.8
cryptominer-detection	57.03	63.12	T	2.548	1.743	T	4.815	67.92	11.68	59.44	4.058	1.732	7.733	T	T	T	45.90	11.40	T	26.46	26.62	87.50	54.10	1.178	1.213	96.34	23.92
forward	96.33	123.2	T	3.962	2.467	T	10.70	110.8	19.36	128.4	9.537	2.237	19.38	T	T	T	74.42	23.51	T	49.47	51.16	150.1	107.6	1.430	1.239	T	50.36
instruction-mix	122.0	134.9	T	3.914	2.483	T	11.38	128.3	20.12	143.9	9.787	2.261	24.13	T	T	T	91.72	25.21	T	58.28	60.61	178.1	113.1	2.196	1.302	T	59.84
memory-tracing	14.97	29.22	3.762	3.447	1.621	1.000	3.187	20.68	5.568	38.00	4.786	1.642	10.14	1.877	0.999	1.000	18.01	6.525	11.87	13.24	15.43	34.10	43.75	1.719	1.775	27.07	16.88
taint	193.1	189.6	T	94.45	362.9	T	40.10	198.5	106.3	187.8	227.2	376.9	55.98	T	T	T	145.9	92.22	MD	78.34	82.73	234.7	160.7	13.45	12.85	T	113.1

Figure 8 Execution overhead for instrumenting programs using Wastrumentation. Instrumentation is done for the cross-product of shared dynamic analyses and the WasmR3 benchmark suite.

The results of this experiment show that the distribution does not favor any particular instrumentation platform. Overall, the most suitable platform is determined by the specific input program, rather than the type of analysis applied. For example, for some input programs ("figma-startpage"–"hydro") Wastrumentation incurs a notable lower overhead across all analyses, while for other input programs ("jqkungfu", "rtexpacker", "rtexviewer") the opposite holds. Some notable outliers are ["jqkungfu","taint"] where Wasabi-instrumented execution is 17.5x faster and the "memory-tracing" analysis for "bullet", "funky"-kart, "guiicons", "ffxgen", and "rguistyler" where Wastrumentation is >9.0x faster than Wasabi.

To coroborate that our findings are consistent across virtual machines, we also deployed the "Forward" analysis for Wastrumentation on Wasmtime. Appendix D in [25] plots our findings, and confirms that across NodeJS and Wasmtime the overhead for the "Forward" analysis remains within the same order of magnitude.

4.5.5 Comparison to bytecode rewriting systems

Lastly, we compare the performance overhead of Wastrumentation with that of analyses implemented in bytecode rewriting systems. To this end, we compute the overhead of the bytecode rewriting analyses that we ported to Wastrumentation as described in Section 4.2.2.

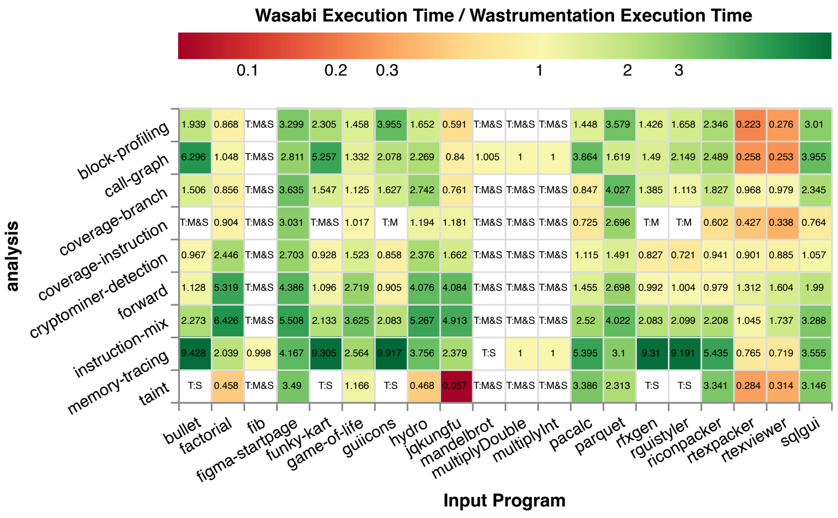


Figure 9 Execution time ratio of Wasabi-analyzed programs over Wastrumentation-analyzed programs.

Figure 10 plots the performance overhead for various input programs of the WasmR3 benchmark suite. For all analyses and input programs, the runtime overhead of bytecode rewriting is one or two orders of magnitude lower. These results are confirmed by earlier experiments, as pointed out by Titzer et al. [38].

The performance advantage of bytecode rewriting systems stems from two aspects. First, bytecode rewriting systems allow developers to have more fine-grained control over which specific locations to inject instrumentation code statically. Second, they can directly inline the analysis within the target operations. In contrast, Wastrumentation, requires runtime dispatching on generic operations. For example, our "binary" trap encapsulates 76 different instructions requiring additional dispatch at runtime.

4.6 RQ5: Evaluating the Memory Overhead of Wastrumentation

We now assess the runtime memory overhead incurred by Wastrumentation and compare it to Wasabi. To evaluate the memory overhead, we used the "Forward" analysis on the WasmR3 benchmark suite for both platforms. We measured the runtime memory using Node.js's built-in `process.memoryUsage()` method after executing each program. The memory overhead for each instrumented program was calculated relative to an uninstrumented baseline, representing the increase in memory usage (expressed as x times more memory usage).

4.6.1 Results

Figure 11 shows the results for each input program. For Wasabi, memory overhead could not be computed for the seven programs that failed to be instrumented, and these programs show no value in the figure. We observe that Wastrumentation's memory overhead ranges from 1.01x to 37.65x, while the memory overhead for Wasabi ranges from 1.03x to 12.34x. We suspect that the higher memory overhead in Wasabi is due to Node.js JIT employing more memory, as the analysis is written in JavaScript, but further experimentation is needed to quantify this hypothesis.

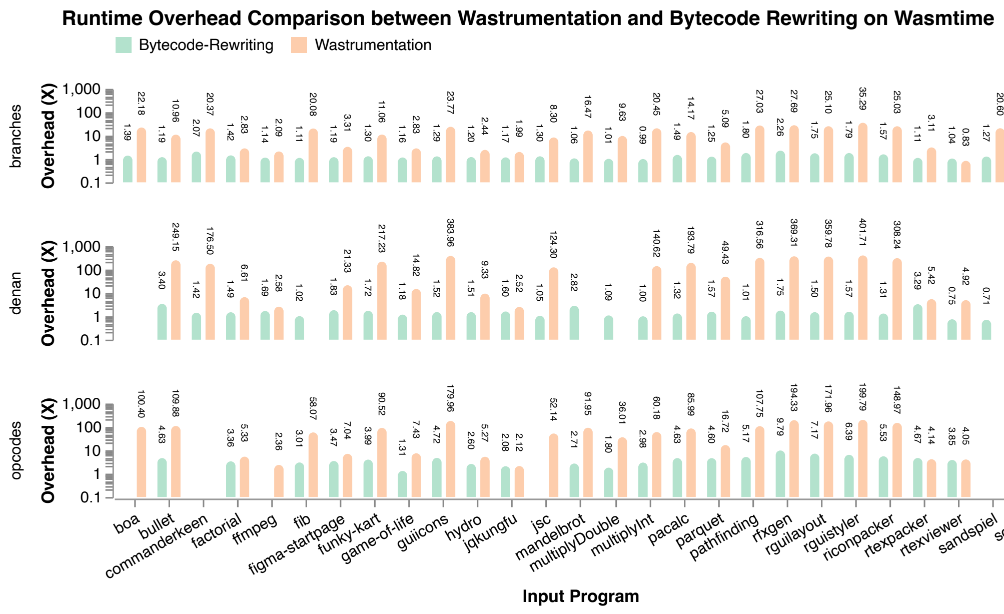


Figure 10 Performance overhead comparison between Wastrumentation and bytecode rewriting systems across various input programs from the WasmR3 benchmark suite.

In the context of Wasm programs, runtime memory is often a limited resource, particularly in environments such as cloud-based platforms or microcontrollers. As a result, minimizing memory overhead is critical for ensuring efficient execution. These results indicate that Wastrumentation may be better suited for deployment in memory-constrained environments.

5 Related Work

In recent years, WebAssembly has become a popular target for dynamic analysis. These analyses span various domains, including security [2], taint tracking [9, 36], program comprehension [20, 30, 26, 39], and profiling [21]. This section compares Wastrumentation to other instrumentation platforms for Wasm and discusses closely related work beyond Wasm.

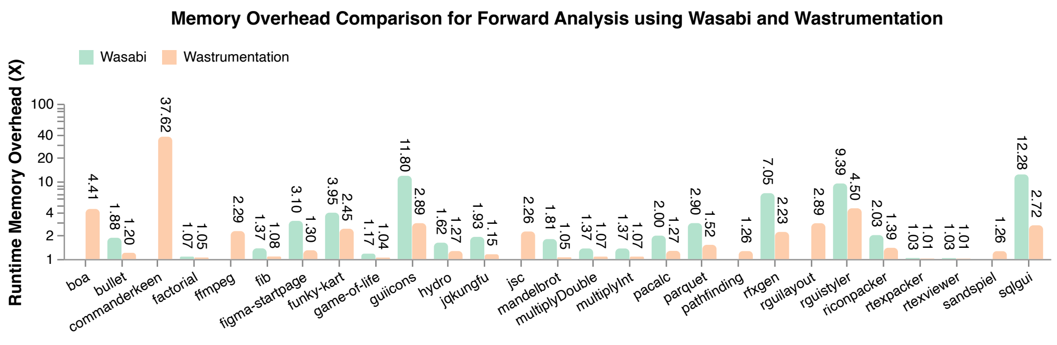


Figure 11 Runtime memory overhead of programs analyzed with Wasabi and Wastrumentation using the "Forward" analysis on the WasmR3 benchmarks. Baseline runs uninstrumented on NodeJS.

5.1 Instrumentation Platforms for WebAssembly

As mentioned before, there exist several tools that enable code transformation and instrumentation for Wasm binaries, Brewasm [5], WasmManipulator [29], and Binaryen [10]. These tools facilitate code injection by providing low-level APIs to modify the input program binary code. However, the analysis developer is responsible for the correctness of the transformation.

Wasabi [19] was the first dynamic analysis platform for Wasm, enabling developers to implement analysis against a high-level instrumentation API and ensuring a correct transformation. The analyses are written in JavaScript by implementing functions that execute code for specific program events of interest. Wasabi achieves instrumentation through source-level code rewriting, generating an instrumented version of the target program that calls the analysis code at runtime. Like Wasabi, Wastrumentation also performs source code instrumentation. In contrast, Wastrumentation does not require a specific language for the analysis, as long as the language can compile to Wasm. Moreover, analyses can operate in environments outside the web, where the Wasm VM may run without a JavaScript VM.

Wizard [38] follows a different instrumentation approach than that of Wasabi and Wastrumentation. Wizard is a Wasm VM with built-in support for instrumentation. In Wizard, analyses are implemented as instrumentation "probes". These probes are developed as extensions to classes within the Wizard VM, which upon execution of a target program will invoke the relevant probe. Wizard focuses on minimizing instrumentation overhead through features that VM-level instrumentation can benefit from, e.g., dynamically adapting analysis code during execution and JIT intrinsification of instrumentation code. However, with this approach, applications executed in the Wizard VM are the only ones that could be analysed. In contrast, dynamic analyses implemented with Wastrumentation do not have additional requirements for the runtime environment, making them portable to any Wasm VM.

5.2 Instrumentation Platforms Supporting Intercession

Dynamic analysis frameworks supporting intercession have been explored for languages other than Wasm. The GraalVM [41] is a polyglot VM that supports instrumentation and dynamic language execution. GraalVM's Truffle framework [40] enables high-performance instrumentation and runtime adaptation [7]. This capability has been used for building a taint analysis platform [16, 15]. In contrast to GraalVM, Wastrumentation performs the instrumentation ahead of time, allowing the instrumented program to run on any Wasm VM, at the cost of not being able to disable instrumentation once the program is executed.

Pin [22], Valgrind [28], and DynamoRIO [4] are dynamic binary instrumentation frameworks for native code analysis. Valgrind provides tools like Memcheck, Callgrind, and Helgrind, which allow intercession and monitoring of x86, ARM, and other instruction set architectures. Its approach of translating code into an intermediate representation enables detailed runtime analysis, including memory checking and profiling. Wastrumentation differs from these approaches for its target domain of applications. The distinct semantics of Wasm, such as its static strong type system and lack of reflection, challenge instrumentation.

6 Conclusion

This paper presented Wastrumentation, a general-purpose source code instrumentation platform for Wasm. Developers can build dynamic analyses using a high-level API that compiles to the Wasm application binary interface supported by Wastrumentation. As a result, analyses can be implemented in *any* language that compiles to Wasm. The platform then merges the target program and the analysis code into a single Wasm program. The resulting instrumented code can be run in *any* Wasm VM. The high-level API allows for implementing analysis requiring intercession, e.g., altering or skipping operations at the target program. Our evaluation with real-world applications from the WasmR3 benchmark suite shows a competitive performance overhead compared to the state-of-the-art source code instrumentation platforms. We also implemented three novel analyses requiring intercession, which were unattainable with existing source code instrumentation platforms for Wasm.

In future work, we aim to improve the platform's modularity so that analyses that do not need intercession do not incurr any performance costs. Currently, it is not possible to eliminate intercession when it is not needed. We also aim to explore a more fine-grained specification language, enabling analysis developers to specify precisely what needs to be instrumented. This could further reduce performance overhead and the binary size of the instrumented application. Finally, we aim to gain more experience and insights in intercession analyses and explore ways to prevent them from compromising overall integrity or efficiency, e.g., detect analyses that pollute the target linear memory or impose a significant overhead.

References

1 Doehyun Baek, Jakob Getz, Yusung Sim, Daniel Lehmann, Ben L. Titzer, Sukyoung Ryu, and Michael Pradel. Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks. *Proc. ACM Program. Lang.*, 8(OOPSLA2):2156–2182, October 2024. doi:10.1145/3689787.

2 Iulia Bastys, Maximilian Algehed, Alexander Sjösten, and Andrei Sabelfeld. SecWasm: Information Flow Control for WebAssembly. In Gagandeep Singh and Caterina Urban, editors, *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings*, volume 13790 of *Lecture Notes in Computer Science*, pages 74–103, Cham, 2022. Springer. doi:10.1007/978-3-031-22308-2_5.

3 Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification X2JI3 document 88-002r. *ACM SIGPLAN Notices*, 23(SI):1.1–2.94, 1988. doi:10.1145/885631.885632.

4 Derek Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, USA, 2004. AAI0807735. URL: https://hdl.handle.net/1721.1/30160.

5 Shangtong Cao, Ningyu He, Yao Guo, and Haoyu Wang. BREWasm: A General Static Binary Rewriting Framework for WebAssembly. In Manuel V. Hermenegildo and José F. Morales, editors, *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*, volume 14284 of *Lecture Notes in Computer Science*, pages 139–163, Cham, 2023. Springer. doi:10.1007/978-3-031-44245-2_8.

6 Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 260–270. IEEE Computer Society, 2016. doi:10.1109/SANER.2016.91.

7 Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *Art Sci. Eng. Program.*, 2(3):14, March 2018. doi:10.22152/programming-journal.org/2018/2/14.

8 Aryaz Eghbali and Michael Pradel. DynaPyt: a dynamic analysis framework for Python. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 760–771. ACM, 2022. doi:10.1145/3540250.3549126.

9 William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *CoRR*, abs/1802.01050, 2018. `doi:10.48550/arXiv.1802.01050`.

10 WebAssembly Group. Binaryen: Optimizer and compiler/toolchain library for WebAssembly, 2025. URL: `https://github.com/WebAssembly/binaryen`.

11 WebAssembly Group. DeNaN.cpp. `https://github.com/WebAssembly/binaryen/blob/8c0429ac09d06d6056687e36fd4fb37f61681233/src/passes/DeNaN.cpp#L44-L46`, 2025.

12 WebAssembly Group. Multi-Memory Proposal: Multiple per-module memories for Wasm. `https://github.com/WebAssembly/multi-memory`, 2025.

13 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM. `doi:10.1145/3062341.3062363`.

14 Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazhen Gu, Yangfan Zhou, and Michael R. Lyu. Revealing Performance Issues in Server-Side WebAssembly Runtimes Via Differential Testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 661–672. IEEE, 2023. `doi:10.1109/ASE56229.2023.00088`.

15 Jacob Kreindl, Daniele Bonetta, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. Polyglot, Label-Defined Dynamic Taint Analysis in TruffleTaint. In Elisa Gonzalez Boix and Tobias Wrigstad, editors, *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, MPLR '22, pages 152–153, New York, NY, USA, 2022. ACM. `doi:10.1145/3546918.3560807`.

16 Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Low-overhead multi-language dynamic taint analysis on managed runtimes through speculative optimization. In Herbert Kuchen and Jeremy Singer, editors, *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2021, pages 70–87, New York, NY, USA, 2021. ACM. `doi:10.1145/3475738.3480939`.

17 Daniel Lehmann. Wasabi_Wasm. `https://github.com/danleh/wasabi/tree/master/crates/wasabi_wasm`, 2025.

18 Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything Old is New Again: Binary Security of WebAssembly. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020. `doi:10.5555/3489212.3489225`.

19 Daniel Lehmann and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1045–1058, New York, NY, USA, 2019. ACM. `doi:10.1145/3297858.3304068`.

20 Daniel Lehmann and Michael Pradel. Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries. In Ranjit Jhala and Isil Dillig, editors, *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 410–425. Association for Computing Machinery, 2022. `doi:10.1145/3519939.3523449`.

21 Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. Exploring Missed Optimizations in WebAssembly Optimizers. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 436–448, New York, NY, USA, 2023. ACM. `doi:10.1145/3597926.3598068`.

22 Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. `doi:10.1145/1065010.1065034`.

23 Quentin Michaud, Yohan Pipereau, Olivier Levillain, and Dhouha Ayed. Securing Stack Smashing Protection in WebAssembly Applications. *CoRR*, abs/2410.17925, 2024. `doi:10.48550/arXiv.2410.17925`.

24 Donald Michie. "Memo" functions and machine learning. *Nature*, 218(5136):19–22, April 1968. `doi:10.1038/218019a0`.

25 Aäron Munsters, Angel Luis Scull Pupo, and Elisa Gonzalez Boix. Wastrumentation: Portable WebAssembly Dynamic Analysis with Support for Intercession (Appendix), 2025. URL: `https://soft.vub.ac.be/Publications/2025/vub-tr-soft-25-03.pdf`.

26 Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. Thieves in the Browser: Web-based Cryptojacking in the Wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, pages 4:1–4:10, New York, USA, 2019. ACM. `doi:10.1145/3339252.3339261`.

27 Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450. USENIX Association, August 2021. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/narayan`.

28 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. `doi:10.1145/1250734.1250746`.

29 João Rodrigues and Jorge Barreiros. Aspect-Oriented Webassembly Transformation. In *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2022. `doi:10.23919/CISTI54924.2022.9820136`.

30 Alan Romano and Weihang Wang. WasmView: Visual Testing for WebAssembly Applications. In Gregg Rothermel and Doo-Hwan Bae, editors, *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 13–16, Seoul, Korea (South), 2020. ACM. `doi:10.1145/3377812.3382155`.

31 Mike Rourke. *Learn WebAssembly: Build web applications with native performance using Wasm and C/C++*. Packt Publishing, 2018.

32 Rust and WebAssembly community. Walrus. `https://github.com/rustwasm/walrus`, 2025.

33 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498. ACM, 2013. `doi:10.1145/2491411.2491447`.

34 Suhyeon Song, Seonghwan Park, and Donghyun Kwon. metaSafer: A Technique to Detect Heap Metadata Corruption in WebAssembly. *IEEE Access*, 11:124887–124898, 2023. `doi:10.1109/ACCESS.2023.3327817`.

35 Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for Node.js. In Christophe Dubach and Jingling Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction*, CC '18, pages 196–206. ACM, 2018. `doi:10.1145/3178372.3179527`.

36 Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint Tracking for WebAssembly. *CoRR*, abs/1807.08349, 2018. `doi:10.48550/arXiv.1807.08349`.

37 Ben L. Titzer. Virgil: objects on the head of a pin. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, volume 41, pages 191–208. ACM, October 2006. `doi:10.1145/1167473.1167489`.

38 Ben L. Titzer, Elizabeth Gilbert, Bradley Wei Jie Teo, Yash Anand, Kazuyuki Takayama, and Heather Miller. Flexible Non-intrusive Dynamic Instrumentation for WebAssembly. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pages 398–415, New York, NY, USA, 2024. ACM. doi:10.1145/3620666.3651338.

39 Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, volume 11099 of *Lecture Notes in Computer Science*, pages 122–142, Cham, 2018. Springer. doi:10.1007/978-3-319-98989-1_7.

40 Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In Gary T. Leavens, editor, *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM. doi:10.1145/2384716.2384723.

41 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509581.

42 Ziyao Zhang, Wenlong Zheng, Baojian Hua, Qiliang Fan, and Zhizhong Pan. VMCanary: Effective Memory Protection for WebAssembly via Virtual Machine-assisted Approach. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 662–671. IEEE, 2023. doi:10.1109/QRS60937.2023.00070.

43 Wenxuan Zhao, Ruiying Zeng, and Yangfan Zhou. Wapplique: Testing WebAssembly Runtime via Execution Context-Aware Bytecode Mutation. In Maria Christakis and Michael Pradel, editors, *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 1035–1047. ACM, 2024. doi:10.1145/3650212.3680340.