DWasm: Portable Debugging for the Web

Aäron Munsters
Vrije Universiteit Brussel
Brussels, Belgium
amunster@vub.be

Nikita Servais
Vrije Universiteit Brussel
Brussels, Belgium
Nikita.Servais@vub.be

Carlos Rojas Castillo
Vrije Universiteit Brussel
Brussels, Belgium
crojcas@vub.be

Angel Luis Scull Pupo
Vrije Universiteit Brussel
Brussels, Belgium
Angel.Luis.Scull.Pupo@vub.be

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.be

Abstract

WebAssembly has established itself as a portable compilation target across a wide variety of environments, from browsers and cloud platforms to hardware-constrained devices such as microcontrollers. Its formal specification enables developers to reliably target a broad range of runtimes while compiler toolchains and runtime developers rely on the same formal model. Yet debugging support for WebAssembly remains at an early stage. Source maps often lack sufficient context for debugging, whereas DWARF offers rich type, scope, and location data that developers need for effective debugging. However, adding DWARF support requires non-trivial, time-consuming runtime-specific efforts that must be repeated independently for each runtime. As a result, the set of debugging features available across runtimes is highly fragmented, while bugs that manifest exclusively in environments without debugging support cannot be analyzed at all.

In this paper, we present DWasm, a runtime-independent debugger for WebAssembly. DWasm is realized as a binary transformation pass that instruments the target application with debugging support, without being implemented as part of a concrete runtime engine. By decoupling the debugger from the runtime, DWasm brings portable, DWARF-based debugging across WebAssembly environments. This approach further reduces tool development costs and enables debugging features to be customized for the application-runtime combination at hand. For our benchmark programs, DWasm exhibits 2x memory overhead, which is low when compared with state-of-the-art debuggers for WebAssembly. In contrast, we observe a performance overhead consistently over 300x, where the shadow-execution instrumentation dominates the added overhead.

CCS Concepts: • **Software and its engineering → Dynamic analysis**; **Software testing and debugging**; • **Information systems → Web applications**.

Keywords: debugging, DWARF, WebAssembly, source code instrumentation, shadow execution

1 Introduction

WebAssembly (Wasm) [21] is a bytecode compilation target, originally designed to offer better support for the implementation of resource-intensive browser-based applications. Wasm is formally specified, in a language-, hardware- and platform-independent way. A Wasm module has no I/O and makes no assumption of the host. These characteristics make Wasm modules deterministic and portable across execution runtimes. This has favored the adoption of Wasm to other environments beyond the browser [32, 34], including resource-constrained microcontrollers [6, 26, 27], the cloud [11], confidential computing [7, 38], and blockchain [7].

Despite WebAssembly's rapid adoption across a diverse set of environments, debugging support for WebAssembly remains underdeveloped. However, it is known that debugging takes a significant portion of the time in software development [9, 23]. Existing debugging protocols, such as Chrome DevTools Protocol (CDP) and Debugger Adapter Protocol (DAP), enable the reuse of a debugger front-end across language runtimes. Yet, the debugger backend requires VM-specific support, leading to duplicate efforts across the entire ecosystem of Wasm execution runtimes to support debugging.

Wasm runtimes like Wasmtime [11] and V8 [5] rely on DWARF [13], a debugging format originally designed for Unix systems, to enable Wasm debugging. DWARF provides extensive debugging information, including information about types, scopes, and variable locations, which is essential for effective source-level debugging. Furthermore, DWARF provides mechanisms such as call stack unwinding, bytecode-to-source-location mapping, and a DWARF context interpreter (DCI) to reconstruct source-level debugging contexts. However, integrating DWARF into WebAssembly runtimes is tedious and time-consuming, as all runtimes must undertake the repetitive task of implementing support for these DWARF features. As a result, the set of debugging features available across Wasm execution runtimes is highly fragmented, with varying levels of debugging support. From a set of Wasm runtimes compiled from a recent survey by Zhang et al. [44] and [8], only 7 out of the 18 runtimes exhibit support for some form of source-level language debugging.

These numbers are reflected in recent surveys [3, 4], showing developers' need for better debugging support within the WebAssembly ecosystem.

Our approach. This paper presents a runtime-independent portable debugging solution for Wasm. We solve the fragmentation of debugging support for WebAssembly by removing the need for a debugging support within the runtime and instead embedding it directly in the target application. Our approach instruments the target application with all necessary debugging infrastructure except for a thin I/O layer that must be provided by the host VM. In practice, such an I/O layer may be implemented through the WASI interface [41] to enable communication with the debugger front-end, such as VSCode, through the DAP. Our approach reduces the software development costs for debugging support, which does not need to be redone for each runtime, and creates opportunities for unified debugging experiences across different WebAssembly environments.

The contributions of this paper are:

- We design a portable and extensible back-end debugger for WebAssembly, DWasm. The extensibility of DWasm's I/O layer enables the reuse of debugger front-ends through existing protocols (e.g., DAP, Chrome DevTools).
- We develop several debugging features atop of DWasm, the combination of which is exclusive to DWasm, and deploy it on three existing WebAssembly engines.
- We evaluate DWasm for the overhead on performance, binary size, and memory footprint. Our evaluation shows that DWasm incurs a high runtime overhead, up to 300x slowdown, while maintaining reasonable memory overhead (around 2x) and binary sizes remain dominated by DWARF debugging information.

2 Background

In what follows, we briefly introduce the necessary background on WebAssembly and DWARF to understand the rest of the paper.

2.1 WebAssembly

Wasm [21] is a stack-based, structured, and typed instruction set that serves as a compilation target for numerous programming languages. A Wasm module, i.e., the Wasm bytecode emitted by a compiler, consists of several sections, the most relevant being:

types Type signatures, including those of Wasm functions.

globals Global variables or constants.

functions Functions, each containing a link to a type signature, local variables, and a sequence of instructions.

memories Linear memory also called *memory pages*.

exports Symbols that are exposed to the host environment. For instance, functions marked with the `export` keyword.

imports External dependencies that are needed by the Wasm module to be correctly instantiated. For instance, if a module needs to use I/O. The module can then import the relevant functions from the host environment. However, this is only possible if the host environment implements these functions, i.e., implements the WASI interface [41].

Wasm instructions and functions are *typed*, each satisfying a type signature. For numeric values, Wasm introduces four primitive types: `i32`, `i64`, `f32`, and `f64`.

At runtime, the Wasm engine maintains an internal stack that cannot be accessed directly. Instead, values are implicitly pushed or popped from the stack as instructions execute.

2.2 DWARF

To debug a Wasm application, the compiler generates the Wasm module alongside *debugging information*, typically DWARF [13] or Source Maps [37]. DWARF is the most popular standard that is supported by several languages that compile to Wasm, including C/C++ [14], Rust [24], Zig [25], and Go [20]. Debuggers use this information to translate between Wasm bytecode and source code, enabling source-level debugging. For instance, a step operation advances program execution to the next source line rather than the next Wasm instruction. At the same time, the debugger can present the developer with source-level lexical scope, including variables, arguments, and the call stack.

DWARF is divided into different sections, each enabling different debugging functionality. Two key sections are:

debug lines is a map that relates the address of one Wasm instruction to its source location. For instance, if address $0x3f$ maps to line 7 (in *hello.c*), the debugger reports that the execution is paused at that line when the PC of a Wasm runtime is at $0x3f$.

debug info describes how to reconstruct source-level state (e.g., local variables, arguments) from the underlying low-level Wasm state (e.g., memory page, globals, locals). This section presents the reconstructed information as a sequence of instructions (e.g., read the top of the stack, read global, read a range of memory pages) that the debugger must execute. The DWARF Context Interpreter (DCI) executes this sequence of instructions. For example, the DCI may reconstruct a C source-level *struct* by reading state ranging from the stack, memory pages, and globals.

DWARF includes further sections such as *frame*, *debug ranges*, and *debug str*. The more sections a debugger supports, the richer the debugging experience it can offer to the developer.

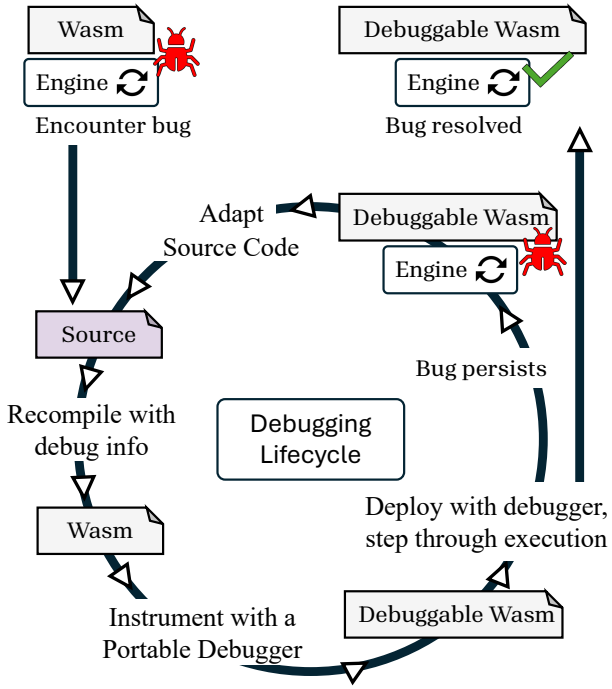


Figure 1. The portable debugging lifecycle.

```
┌─────┬──────┬──────────────────────────────────────┐
│ Brk │ Line │ Source                               │
├─────┼──────┼──────────────────────────────────────┤
│     │    1 │ fn fac(n: i32) -> i32 {              │
│     │    2 │     if n <= 1 {                     │
│   ● │    3 │         return 1;                    │
│     │    4 │     } else {                         │
│     │    5 │         let res = fac(n - 1);        │
│ ▶ ● │    6 │         return n * res;              │
│     │    7 │     }                                │
│     │    8 │ }                                    │
└─────┴──────┴──────────────────────────────────────┘

┌────────┬──────┬─────┬──────┐
│ Scope  │ Name │ Val │ Type │
├────────┼──────┼─────┼──────┤
│ Param  │ n    │   2 │ i32  │
│ Local  │ res  │   1 │ i32  │
└────────┴──────┴─────┴──────┘
[c(ontinue)] [s(tep-over)] [q(uit)]
```

Figure 2. The debugger UI of DWasm.

3 Portable Debugging for the Web

This section provides an overview of our portable debugging solution for Wasm programs, built on DWARF. Instead of integrating DWARF support within the runtime, we propose embedding DWARF-based debugging directly into the target application. Not only does decoupling the debugging support from the runtime reduce software development efforts, but it also creates opportunities to bring source-level debugging features across the entire ecosystem of Wasm engines.

We first describe our approach from the user's perspective, i.e., a developer using a portable debugger for their programs compiled to WebAssembly. We then detail the debugging interface required to offer an interactive online debugging experience.

3.1 The portable debugging lifecycle

Figure 1 illustrates the overall lifecycle for debugging a WebAssembly program using our portable debugger. Initially, the developer encounters a fault in the program behavior while executing it on a WebAssembly engine. Before debugging, the developer recompiles the source code and instructs the compiler to emit DWARF debugging information. For example, Rust's compiler `rustc` and the C/C++ compilers `clang` and `gcc` all accept the `-g` flag to emit source-level debug information. The output binary, which now includes DWARF debugging information, is then passed onto a portable debugger. The debugger performs a binary rewrite pass on the target program to inject the debugger infrastructure, producing a *debuggable Wasm binary*.

Upon launching the debuggable binary on a Wasm engine, the developer is presented with a debugger front-end that allows them to live-debug the application. The debugging front-end is further described in Section 3.2. Figure 2 shows the front-end of our current prototype implementation, DWasm. The developer may adapt the source code to fix the bug, then restart the debugging cycle: recompile, instrument, and launch on an engine, repeating the process until the bug is resolved.

3.2 The Debugger Front–End

DWasm's front-end shown to developers upon launching the debuggable binary provides functionality typically found in online debuggers, such as state inspection and step-wise execution. This front-end issues debugging commands in response to the user's actions. For example, Figure 2 shows that the user has issued two "break" commands for lines 3 and 6, resulting in the installation of two breakpoints. These commands are forwarded to DWasm's backend interface, which will perform an action (e.g., pausing the program upon reaching a breakpoint) and may also notify the front-end. For instance, the front-end in Figure 2 was informed that execution was paused at line 6, as indicated by a triangle. The portable DWasm's front-end interface supports the following operations grouped by functionality:

Breakpoint management. Manipulating the breakpoint table can happen through three operations: `break`, `delete`, and `info breakpoints`. The arguments to these operations may be either source-level locations (e.g., `src/main.rs:42`) or WebAssembly offsets (e.g., `0x1234`).

Stepwise execution control. Performing step-by-step execution of a running program can be done through the operations `step` and `continue`. The `step` command advances to the next WebAssembly instruction, while `continue` resumes execution until the next breakpoint is hit or the program terminates.

State inspection. The operations `print_source_context_with_breakpoints` and `list_locals` respectively shows the current source code, and variables currently accessible in scope with their bound values.

4 Implementation

In this section, we explain the implementation of portable debugging for Wasm through DWasm, our debugger prototype. We first provide a high-level overview of the pipeline for injecting a debugger into a target Wasm program via source-code instrumentation. Next, we elaborate on how the portable debugger is implemented as a dynamic analysis.

4.1 DWasm pipeline

Figure 3 shows DWasm as a five-stage pipeline that translates the target binary with debugging information into a variant with a live debugger embedded into the application. Concretely, Figure 3a shows the dataflow diagram of the pipeline and Figure 3b the corresponding Rust implementation of the pipeline. We describe each stage of the pipeline in more detail below.

① **Validation.** The target program (i.e., a Wasm module) is parsed and verified to include DWARF debugging information, while the WebAssembly sections are validated to ensure they are well-structured and well-typed.

② **Extract Debug Information.** The Wasm module, composed of multiple sections, is split into the module logic and the DWARF data custom section. The DWARF data is then parsed and transformed into a format suitable for processing by the embedded debugger. We retain the DWARF information that our current implementation of the debugger relies on, which includes the `debug_lines` and `debug_info` sections, whereas other debug information entries are discarded.

③ **Generate a Specialized Debugger.** The debugging logic itself is implemented as a dynamic analysis that is woven into the target program. Its implementation is designed as a template that is reusable across different target programs, with the DWARF data left as the template argument. The implementation of the analysis is further explained in Section 4.2. Once the DWARF data is provided to the debugger generation pipeline, the debugger is then compiled to WebAssembly, resulting in the `debugger-analysis.wasm` module in Figure 3a, which is provided as analysis code to a source-code instrumentation framework for Wasm.

④ **Instrument.** The original input program code, along with the specialized debugger analysis, is passed to the instrumentation pass that injects hooks throughout the program. Our debugger prototype, DWasm, uses Wastrumentation [33] for the instrumentation pass. Wastrumentation thus takes as input the original program and the analysis code, together with a set of hooks that specify where the analysis code should be woven into the target program.

⑤ **Emit.** The instrumentation pass emits an instrumented variant of the original target module with the debugger analysis code embedded. This final binary can run on a WebAssembly runtime to provide an interactive online debugging experience during execution.

4.2 Specialized Debugger

The specialized debugger embedded with the target application consists of two main components: the debugger front-end and the debugger backend. As shown in Figure 2, DWasm's debugger front-end is a REPL-like command-line interface that allows a developer to control the target application's execution. This command-line interface accepts debugging commands that map directly onto the debugging interface operations described in Section 3.2.

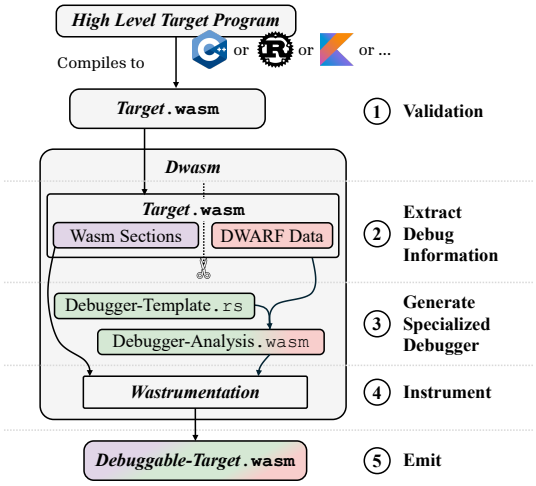
The debugger backend controlling a target program is implemented as an event listener, where each received event corresponds to the execution of a single WebAssembly instruction in the target program. The events themselves are emitted by the shadow execution environment, which is described in more detail in Section 4.4.

Each time the input program executes an instruction, the debugger updates the program counter (PC) and then queries the debugging state. If the set of currently active breakpoints contains the current PC, the developer is presented with the debugger front-end; otherwise, execution continues in the target program.

4.3 Lazy loading of DWARF data

At runtime, the debugger parses the relevant portions of the DWARF data on demand. That is, when a breakpoint is hit or a variable is inspected, the debugger decodes only the entries relevant to the active instruction. Upon the first usage of debugging features that rely on the DWARF information, the DWARF data is parsed and loaded into memory. We employ a lazy-loading strategy because the DWARF data can be large, and parsing it can be expensive. The DWARF data itself is embedded into the debugger analysis source code as a read-only data structure.

Figure 4 shows a snippet of the debugger implementation. This code defines a static variable `DWARF_INFO`, which is a `LazyLock`. The `LazyLock` type is a synchronization primitive



(a) A dataflow diagram of the DWasm pipeline

```rust
fn dwasm_instrument(target: &[u8]) ->  Result<WasmModule> {
    log::info!("(1) Parse module, validate DWARF presence");
    parse_and_validate_module(target)?;

    log::info!("(2) Split DWARF from rest of module");
    let (stripedmodule, dwarf) = split_dwarf(target)?;

    log::info!("(3) Compile debugger based on DWARF");
    let dbg_analysis = generate_debugger_analysis(&dwarf)?;

    log::info!("(4) Perform instrumentation");
    let module = instrument(&stripedmodule, &dbg_analysis)?;

    log::info!("(5) Yield output binary");
    return Ok(module);
}
```

(b) The Rust implementation of the DWasm pipeline

Figure 3. The pipeline of DWasm that transforms a target Wasm program into a debuggable variant. The pipeline extracts the DWARF debug information from the target binary and emits a new binary with the debugger and hooks embedded into the target program.

```rust
use std::sync::LazyLock;
use postcard::from_bytes as deserialize_from_bytes;
use serde_dwarf_info::DwarfInfo;

type GlobalDwarfInfo = LazyLock<DwarfInfo>;

static DWARF_INFO: GlobalDwarfInfo = LazyLock::new( {
  let serialized = include_bytes!("DWARF_TEMPLATE.bin");
  from_bytes(serialized).expect("parsing DWARF bytes")
});
```

Figure 4. The lazily loaded debugging information, which is left as a template for specializing per input program.

that allows lazy initialization, delaying parsing of DWARF data until it is first accessed. The closure passed to the `LazyLock` (lines 7 – 10) keeps a reference to the data (line 8) and deserializes the data to a Rust data structure (line 9). The reference itself is declared using the `include_bytes!` macro, which includes the contents of a file as a byte array at compile time. During the debugger specialization stage of the pipeline (cf. Figure 3), the debugger's source code is cloned, and the template file is replaced with the actual DWARF data for the input program.

4.4 Shadow Execution Environment

As mentioned before, we use the Wastrumentation framework to create an instrumented variant of the original program with the analysis code embedded into it. This instrumentation framework weaves the analysis code into the target program, separating the address spaces of the target program and the analysis code using the multi-memory feature of WebAssembly[1]. The framework also provides a *shadow execution environment* that maintains a parallel representation of the WebAssembly state, including the VM stack (values, labels, frames), linear memory, and the global values. Upon execution of an instruction, the corresponding shadow state is updated to mirror the changes to the real state. The debugger analysis extends Wastrumentation's shadow execution environment to access the program state at runtime.

Figure 5 shows an excerpt of the shadow execution environment implementation for binary operator instructions. The implementation defines a hook (line 1) that is called when a binary operator instruction is executed by Wastrumentation. The implementation of the hook first calls the client of the shadow execution environment (lines 4–6), which in this case is the debugger, to inform it about the execution of the instruction. The remainder of the hook implementation (lines 8–27) further updates the shadow state to mirror the changes to the real state. The implementation follows the WebAssembly specification. Figure 6 displays an excerpt of the WebAssembly specification that specifies the semantics of binary operator instructions, showing the one-to-one correspondence with the shadow execution implementation in Figure 5.

Figure 7 illustrates how the debugger implementation extends the shadow execution environment for binary operator instructions. This hook extends shadow execution to further ensure that the debugger state stays in sync with it, e.g., by keeping the program counter up to date.

[1]The multi-memory feature of WebAssembly allows modules to define multiple linear memories, which can be used to separate the address spaces of the target program and the analysis code.

```
advice! { binary (binop: BinaryOperator, c_1: WasmValue,
      c_2: WasmValue, location: Location
  ) {
    unsafe {
      shadow_traps::binary(&binop, &c_1, &c_2, &location)
    };

    SHADOW_STACK.with_borrow_mut(|shadow_stack| {
      // 1. Assert: due to validation, two values
      //  of value type `t` are on the top of the stack.
      "handled by validation";
      // 2. Pop the value `t.const c_{2}` from the stack.
      let shadow_c_2 = shadow_stack.pop_value_from_stack();
      debug_assert_eq!(shadow_c_2, c_2);
      // 3. Pop the value `t.const c_{1}` from the stack.
      let shadow_c_1 = shadow_stack.pop_value_from_stack();
      debug_assert_eq!(shadow_c_1, c_1);
      // 4. If `binop_{t}(c_{1},c_{2})` is defined, then:
        // a. Let `c` be a possible result of
        //  computing `binop_{t}(c_{1},c_{2})`.
        let c = binop.apply(c_1, c_2);
        // b. Push the value `t.const c` to the stack.
        shadow_stack.push_value_on_stack(c.clone());
      // 5. Else:
        "handled by VM";
        // a. Trap.
      c
    })
  }
}
```

Figure 5. The shadow execution environment implementation of observing binary operator instructions, built on top of the Wastrumentation instrumentation framework [33].

$t.binop$

1. Assert: due to validation, two values of value type t are on the top of the stack.
2. Pop the value $t.\mathsf{const}\ c_2$ from the stack.
3. Pop the value $t.\mathsf{const}\ c_1$ from the stack.
4. If $binop_t(c_1, c_2)$ is defined, then:
   a. Let c be a possible result of computing $binop_t(c_1, c_2)$.
   b. Push the value $t.\mathsf{const}\ c$ to the stack.
5. Else:
   a. Trap.

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \hookrightarrow (t.\mathsf{const}\ c) \quad (\text{if } c \in binop_t(c_1, c_2))$$
$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \hookrightarrow \mathsf{trap} \quad (\text{if } binop_t(c_1, c_2) = \{\})$$

Figure 6. The formal specification of WebAssembly binary operator instructions, as the WebAssembly formal specification defines [21].

Different categories of instructions for shadow execution are handled specially in our extension: arithmetic operations update the shadow value stack, memory operations access the shadow memory, control-flow operations record

```
#[no_mangle]
pub extern "Rust" fn binary(
    binop: &BinaryOperator, c_1: &WasmValue, c_2: &WasmValue,
    location: &Location,
) {
    crate::handle_hook_signal(location.instruction_index());
}
```

Figure 7. The debugger implementation extending the shadow execution environment.

branches to targets, and call or return operations modify the shadow call stack.

We further detail each of these categories and how we maintain the shadow state synchronized below.

Stack and frames. The shadow stack mirrors Wasm's operand stack with three entry kinds: values (`i32`, `i64`, `f32`, `f64`), labels (to mark `block` or `loop` constructs), and frames. Frames encapsulate function index, parameter values, and locals. The instructions `local.get`, `local.set` and `local.tee` translate into reads and writes on the active frame. The instructions `call`, `call_indirect` and `return` push and pop frames while respecting the Wasm calling convention.

Memory and globals. Memory operations (`load`, `store`) are mirrored in the shadow memory, ensuring the debugger can inspect memory contents at any breakpoint. The global state is similarly maintained in parallel, allowing queries to global variables during debugging sessions.

By keeping the shadow execution environment in sync with the execution of the target program, the debugger can access complete program state information without modifying the original program's semantics or its performance characteristics.

4.5 Interfacing with the developer

The debugger front-end mediates the interactions between the debugger and the developer, leveraging the WebAssembly System Interface (WASI) [41]. WASI defines an interface between Wasm modules and their host to share widely supported system features, such as file system access and standard input/output operations. DWasm uses two WASI features. First, operations to access the host's file system are used to display the source code context where one is debugging. Second, the standard input/output is used to accept developer commands to proceed execution (input) and visualize the terminal interface (output).

Note that while the current command-line interface assumes the WebAssembly engine supports WASI, the debugger back-end is fully decoupled from the interface used to communicate with the user. As such, alternative interfaces could be supported to interface with the developer, e.g., by targeting the Debugger Adapter Protocol (DAP) [31]. The

DAP outlines an interface between debugger adapters and debugger front-ends, most commonly used by the VS Code editor to provide a debugger front-end for different runtimes.

5 Evaluation

In this section, we evaluate DWasm and compare it to state-of-the-art DWARF-based debugging techniques for WebAssembly. In our evaluation, we seek to assess the feasibility of DWasm as an alternative debugger for WebAssembly. We focus on answering the following research questions:

RQ1: What is the runtime performance overhead of DWasm compared to state-of-the-art DWARF-based debugging techniques for WebAssembly?

RQ2: What is the memory overhead of DWasm compared to state-of-the-art DWARF-based debugging techniques for WebAssembly?

RQ3: What is the program size increase of the target program after instrumentation by DWasm?

5.1 Experimental Setup

For our evaluation, we select a set of input programs for which the source code is available, can be compiled to WebAssembly, and can be debugged with DWARF-based techniques. We furthermore restrict the set of input programs to those that do not depend on the WASI interface. As explained in Section 4.4, the instrumentation platform used by DWasm relies on the multi-memory feature of WebAssembly to separate the address spaces of the target program and the debugger. Moreover, the debugger relies on the WASI interface to communicate with the developer (cf. Section 4.5). Therefore, we select benchmarks that do not depend on the WASI interface to avoid interference of the WASI interface used by the debugger with the WASI interface used by the target program, as WASI is not designed to support interleaving through multiple memories.

The selected input programs for our evaluation come from the Computer Language Benchmarks Game [15] and are as follows:

- **Binary Trees**: a simple binary tree implementation, implemented in Rust.
- **Fibonacci**: a simple Fibonacci implementation, implemented in Rust.
- **Mandelbrot**: a simple Mandelbrot set implementation, implemented in Rust.
- **N Body**: a simple N Body simulation, implemented in Rust.

The experiments were conducted on a 2023 MacBook Pro with the following specifications:

- **Processor:** Apple M2 Max System on Chip (SoC), featuring a 12-core CPU (8 performance cores and 4 efficiency cores). Externally performed benchmarks report clock speeds up to 3.7 GHz for the performance cores and up to 3.4 GHz for the efficiency cores [22].

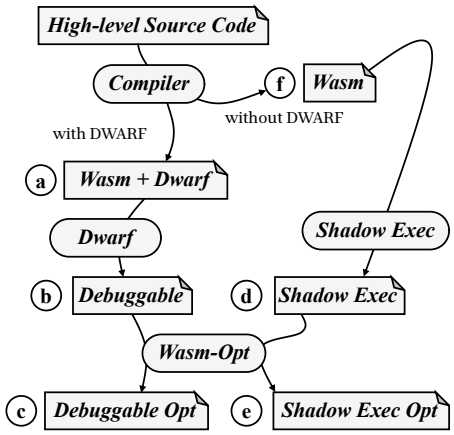


Figure 8. The benchmark program pipeline for discussion in the evaluation section. Each input program is compiled to WebAssembly, either with (ⓐ) or without DWARF debugging information (ⓕ). The compiler output is then further subject to instrumentation, either exclusively with shadow-execution instrumentation (ⓓ) or with an embedded debugger through instrumentation by DWasm (ⓑ). These instrumented binaries are then further optimized by "wasm-opt" (ⓒ , ⓔ).

- **Memory:** 32 GB of unified RAM.
- **Storage:** 1 TB Solid State Drive (SSD).
- **Operating System:** macOS 26.3.

The software environment for the experiments includes:

- **Runtimes:** Wasmtime (v41.0.2), Node.js (v24.13).
- **Debugging Tools:** LLDB (v15.0.7).
- **Instrumentation:** Wastrumentation (commit `56e108e`).

We retain distinct artifacts for the different stages of the transformation performed by DWasm in an effort to tease apart the implications of each stage with respect to the research questions. Figure 8 summarizes the benchmark program pipeline. Each input program is compiled to WebAssembly, either with DWARF debugging information (ⓐ) or without DWARF debugging information (ⓕ). Next, we perform the instrumentation pass with DWasm to generate a debuggable binary artifact (ⓑ). As mentioned in Section 4.4, DWasm relies on a shadow execution environment to maintain the shadow state of the target program. An artifact is generated with only the shadow execution instrumentation (ⓓ), omitting debugging information. The output binary of Wastrumentation can be further optimized with post-processing tools such as "wasm-opt" from the Binaryen toolkit [43], which performs optimizations including dead code elimination and function inlining. These passes generate either a more optimized debuggable binary (ⓒ) or a more optimized binary with shadow-execution instrumentation (ⓔ).

5.2 Runtime Overhead

We begin by evaluating the runtime performance overhead of DWasm compared to state-of-the-art DWARF-based debugging techniques for WebAssembly (cf. RQ1). The runtime performance overhead of a debugger can affect developer responsiveness and, in turn, the usability and effectiveness of the debugging process. We compare DWasm against the debugging capabilities of the Node.js and Wasmtime runtimes, both of which support DWARF-based debugging for WebAssembly.

Node.js setup. To measure the performance overhead of Node.js, we use the "`--inspect`" flag to start the Node.js runtime in debugging mode. This flag tells Node.js to start, as part of its process, a debugging server that allows developers to attach a remote debugger. A debugger front-end can then communicate to the debugging server using the Chrome DevTools Protocol (CDP) [18]. With the flag "`--inspect`" in use, the Node.js runtime *tiers down* its multi-tier JIT execution from TurboFan to Liftoff. This is to avoid the debugger observing that higher JIT tiers re-order code, eliminate variables or perform dead-code elimination [12, 19]. The tierdown prevents introducing ambiguity when breakpoints are set on locations that TurboFan might otherwise re-order or eliminate altogether.

Wasmtime setup. To measure the performance overhead of the Wasmtime runtime, we start the Wasmtime runtime with the flag "`-D debug-info`". This flag tells Wasmtime to enable support for DWARF-based debugging by translating the DWARF debugging information for the JIT-generated native code. That is, Wasmtime JIT-compiles the WebAssembly instructions to native code, while the "`-D debug-info`" flag ensures the provided DWARF debugging information is translated to map the high-level source code to the JIT-generated native code. This translation allows developers to attach a live debugger such as LLDB to the running Wasmtime process and debug the target program [12], which is how we benchmark this runtime.

Both Node.js and Wasmtime runtimes have in common that they must take as input program the module that is not optimized by "wasm-opt". The optimization passes would otherwise break the DWARF mapping, as the tool does not preserve this mapping during transformation [10]. However, for the output from DWasm, further optimizations can be made using "wasm-opt" since the dwarf-mapping is embedded within the analysis code, preserving the DWARF mapping.

Experiments. To measure the performance overheads, we set as a baseline the execution time of the target program without debugging enabled. Relative to this baseline, we measure the execution time of the target program when debugging with DWasm and when debugging with the runtime support for DWARF-based debugging. In addition, we measure the relative slowdown when instrumenting with the shadow execution (ⓓ). Per runtime, we measure a separate baseline execution time, as it can vary across runtimes due to differences in their implementations and optimizations.

Timing the program run happens at the level of the WebAssembly benchmark program, meaning that two WebAssembly instructions mark the start and end of a benchmark run. This means that the measured runtime overhead excludes the startup time of the engine and the surrounding infrastructure, such as starting the debugging server for Node.js or LLDB for Wasmtime. We exclude the startup time as the measured startup time of the engine and related infrastructure can vary significantly between runs and is typically much longer (on the order of seconds) than the actual benchmark execution time (on the order of milliseconds).

Results. Figure 9 plots the collected measurements of 30 runs per runtime-program configuration. As our runtime performance overhead experiments have low variance, we plot the median as a coloured shape and individual outliers as dots.

From the figure, we observe the following trends:

- Debugging with DWasm (DWasm) incurs a significantly higher runtime overhead compared to DWARF-based debugging with LLDB in both Node.js and Wasmtime across all benchmarks. This is expected as DWasm relies on source code instrumentation using Wastrumentation, which reports that instrumentation overhead can range from 1x to 514x [33].
- The overhead of DWasm is most pronounced on the Wasmtime runtime, reaching over 300x slowdown, compared to Node.js, with a slowdown of around 30x. This is due to the baseline execution time on Wasmtime being much faster than on Node.js, which amplifies the relative overhead of DWasm. In other words, the absolute overhead of DWasm is similar across both runtimes, but the relative overhead is much higher on Wasmtime.
- Shadow execution instrumentation (S-E) alone accounts for $>$ 90% of the overhead, indicating that maintaining shadow state is the dominant cost, not the embedded debugger.
- The runtime overhead for DWARF-based debugging (LLDB) remains consistently low and close to baseline, demonstrating that native runtime support for debugging is much more efficient.
- Across all input programs, the relative ordering of overheads is consistent: DWasm > S-E > LLDB.

These results suggest that while DWasm enables portable debugging in contexts where DWARF-based debugging is unavailable, it incurs a substantial performance cost, driven primarily by shadow-execution instrumentation.

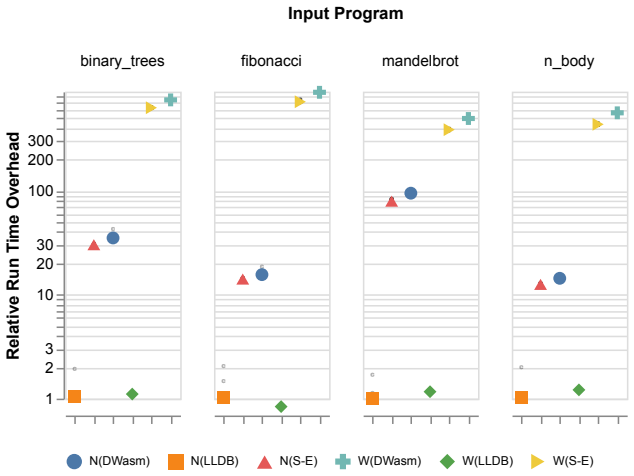


Figure 9. The relative runtime performance overhead of DWasm to the baseline performance per runtime. The format is *Runtime (Instrumentation)*. *Runtime* is either *W(asmtime)* or *N(odejs)*, while *Instrumentation* is *LLDB*, *DWasm*, or *S(hadow)-E(xecution)*.

Figure 10. The relative runtime memory overhead of DWasm to the baseline performance per runtime. The format is *Runtime (Instrumentation)*. *Runtime* is either *W(asmtime)* or *N(odejs)*, while *Instrumentation* is *LLDB*, *DWasm*, or *S(hadow)-E(xecution)*.

5.3 Memory Overhead

In the following, we evaluate the memory overhead of DWasm compared to state-of-the-art DWARF-based debugging techniques for WebAssembly (cf. RQ2).

Experiments. Using the same setup as the previous experiment, we evaluate the memory overhead of DWasm compared to DWARF-based debugging in Node.js and Wasmtime. In a separate experiment, we measure the resident set size (RSS) of the process running the target program, with debugging disabled, as the baseline memory usage of the target program. The RSS was sampled every 10 microseconds[2] during the execution of the target program, and the maximum RSS value observed during the execution was recorded as the memory usage for that run. The highest samples RSS value captures the peak memory usage of the process. We then compute the relative memory overhead of DWasm and the runtime support for DWARF debugging, compared to the target program's baseline peak memory usage without debugging enabled.

Results. Figure 10 plots the collected measurements per runtime-program configuration. From this figure, we observe the following trends:

- The memory overhead of Node.js and Wasmtime debugging with their respective DWARF debugging support consistently incurs a 50x memory overhead.
- The shadow execution instrumentation (N(S-E), W(S-E)) adds minimal memory overhead, close to the baseline, indicating that the shadow execution instrumentation does not significantly increase the memory usage of the target program.
- The memory overhead of debugging with DWasm is consistently around 2x, which is likely due to the fact that the DWARF information is kept in the running process.

[2]On a separate OS thread, the RSS of the child process was sampled using the 'sysinfo' crate [17] in Rust, which provides a cross-platform API for querying system information, including memory usage.

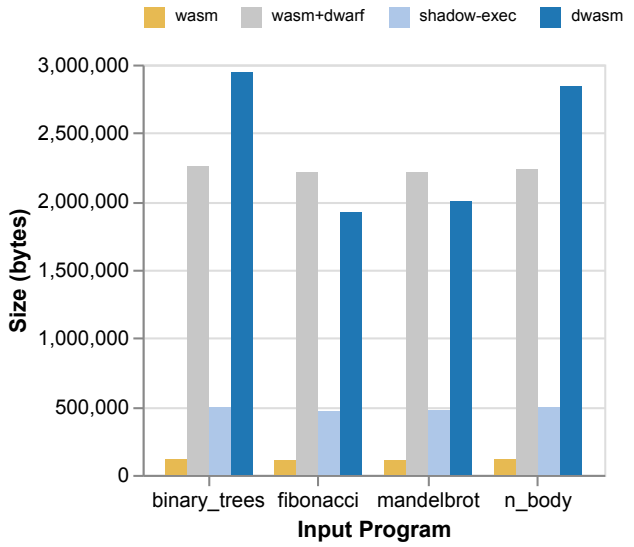


Figure 11. The binary sizes of the target program after the different instrumentation stages.

5.4 Binary Size Overhead

Finally, we evaluate the increase in the target program's binary size after instrumentation with DWasm (cf. RQ3).

Experiments. We measure four different configurations to isolate different sources of size increase. To distinguish between code and the debugging information, we measure the size of the compilation output with and without DWARF debugging information, denoted "wasm+dwarf" and "wasm" respectively. To tease apart the debugger's program size increase from shadow execution within DWasm, the size of weaving shadow execution into the module without DWARF information is plotted as "shadow-exec". The binary program size after instrumentation with the debugger woven in by DWasm, is plotted as "dwasm".

Results. Figure 11 plots the binary size of the various configurations. From this figure, our observations are the following:

- Across all programs, the DWARF sections dominate the binary size: the "wasm+dwarf" configuration (with DWARF) is 8–40× larger than "wasm" binaries. For example, "fibonacci.wasm" grows from 106 kB (no dwarf) to 1.9 MB (wasm + dwarf), and "binary_-trees.wasm" from 115 kB to 2.9 MB.
- The shadow-execution instrumentation ("shadow-execution") adds a constant overhead on top of the stripped module, increasing the size to about 1.9–2.2 MB, which remains below the size of the full DWARF debugging information.
- The size of the output of DWasm ("dwasm") is not always larger than the input program with DWARF ("wasm+dwarf"), as is the case in "fibonacci" and "mandelbrot". This illustrates that DWasm retains most of the debugging information, but the weaving of the debugger and shadow execution can add additional overhead, while non-retained DWARF information can shrink the overall outcome (cf. Section 4.1).
- Overall, the correlation in sizes between the output from DWasm and the "wasm+dwarf" configuration shows that the majority of the debugging information is retained, but the binary size overhead is dominated by retained debugging information, rather than by the instrumentation itself.

Note that the optimized versions of the instrumented binaries (cf. Figure 8, ⓒ and ⓔ) are not plotted in the figure, as the optimization passes performed by "wasm-opt" can significantly reduce the binary size of the instrumented binaries, making it difficult to compare the increase in binary size of DWasm to the baseline binary size of the target program without instrumentation.

In summary, the binary size overhead introduced by DWasm is primarily due to the inclusion of debugging information, with the instrumentation itself contributing only a modest increase. While DWasm retains most of the original DWARF data, its output binaries can become smaller or remain comparable in size to the original DWARF-including binaries.

6 Related Work

This section discusses debugging approaches in the context of WebAssembly. The current landscape of WebAssembly debugging tools spans multiple environments and methodologies, from browser-integrated solutions to specialized frameworks for embedded systems. Debugging support for WebAssembly has taken two approaches, depending on how the WebAssembly code is executed.

A first approach is to build debugging support into the virtual machine (VM) [2, 5]. In V8[5], the debugger is built as part of the interpreter and relies on the engine's internal structures. The benefit is that the debugger implementation has full access to the runtime state. However, depending on internal structures makes the debugger susceptible to changes unrelated to language semantics, which may require maintenance of the debugger, even if the language does not change. Moreover, the debugger may not benefit at all from full access to the internals. For example, running V8 in debugging mode disables more advanced optimizations (e.g., instruction reordering), which can affect the "link" between low-level instructions and high-level constructs [19]. Within the same category of VM-level debugging support, the Wizard [39] engine took a different approach. Instead of hard-coding the debugger directly as a component of the VM, the engine offers a set of instrumentation hooks and an API that enable the implementation of dynamic analyses, which they used to implement a debugger.

A second approach found in systems based on ahead-of-time (AoT) compilation [11, 42] is to delegate to existing debugging infrastructure from well-established DWARF-based debuggers like GDB/LLDB [16, 30]. In practice, however, this is not straightforward, since each runtime may apply optimizations to the Wasm bytecode or internalize the Wasm state in ways that complicate integration with GDB/LLDB. For instance, Wasmtime [11] relies on a code generator, Cranelift [1], to compile a given Wasm program to native code. The given WebAssembly-specific DWARF information is then transformed to match the native code and provide a proper mapping to the source language (e.g., C, Rust). Like in V8, debugging an AoT-compiled Wasm program with Wasmtime requires using the baseline compiler (Winch).

Some work [28, 35, 36] has focused on debugging support for WebAssembly VMs running on microcontrollers, as the available computing resources require targeted debugging solutions. Rojas Castillo et al. [35] introduce out-of-place debugging to IoT applications, in which the state of a running program on a microcontroller is moved to a more powerful

machine for debugging. This approach is built as an extension to the WARDuino VM [27]. The interaction with the microcontroller peripherals was achieved via a pull mechanism that requested information from the microcontroller just before the application interacted with those hardware components. Lauwaerts et al. [28] adopted a push event-based approach to handle peripherals, enabling debugging of hardware interrupts as they occur.

Other approaches have explored debugging support for WebAssembly beyond the traditional features offered by interactive online debuggers. For example, Lauwaerts et al. [29] designed a *multiverse debugger* that can handle input/output operations called MIO. A multiverse debugger facilitates the debugging of non-deterministic programs by enabling the exploration of all execution paths within one debugging session [40]. MIO is also implemented on top of the WARDuino VM.

A shared limitation of all debugging approaches discussed in this section is their portability, since they are tightly implemented within the execution environment. In contrast, DWasm is inlined in the target application; therefore, it can be ported to *any* environment without requiring changes to the target VM.

7 Conclusion and Future Work

WebAssembly achieves portability for program execution but not for debugging, leading to fragmented debugging support across runtimes. This paper presented DWasm, the first portable debugging solution for the WebAssembly ecosystem. DWasm attains portability by inlining debugger services directly into the Wasm bytecode via source-code instrumentation. This approach exhibits two main benefits. First, the debugger achieves the same portability as the target program, giving developers a consistent debugging experience and services across all execution engines. Second, the burden of implementing and maintaining a debugger is lifted from VM engineers, reducing tool development costs.

Our evaluation shows that DWasm enables portable debugging across WebAssembly runtimes at the cost of significant runtime overhead (up to a 300x slowdown, primarily due to shadow execution), while maintaining reasonable memory overhead (around 2x) and binary sizes dominated by retained DWARF information. Compared to native DWARF-based debugging, DWasm trades performance for portability in environments that lack runtime debugging support.

Future Work. We plan to extend the capabilities of DWasm in different directions. First, we plan to incorporate selective debugging, which allows instrumenting only the required parts of the target program, thereby reducing latency. For example, in a large program, a considerable section of the code that must be executed before reaching a breakpoint can be sped up by removing instrumentation from it. Since our debugger already maintains a shadow representation of the program state and Wasm execution remains deterministic between I/O operations, an interesting avenue for future work is to add debugging services that enable time-travel debugging.

Acknowledgement

Aäron Munsters is funded by the Research Foundation Flanders, project number 1S53725N. Carlos Rojas Castillo is funded by the Research Foundation Flanders, project number 1SHEU24N. Angel Luis Scull Pupo is funded by the Cybersecurity Research Program Flanders (CRPF) from the Flemish Government.

References

[1] 2026. Cranelift. https://cranelift.dev/
[2] 2026. Home. https://spidermonkey.dev/
[3] 2026. The State of WebAssembly 2021. https://blog.scottlogic.com/2021/06/21/state-of-wasm.html
[4] 2026. The State of WebAssembly 2023. https://blog.scottlogic.com/2023/10/18/the-state-of-webassembly-2023.html
[5] 2026. V8 JavaScript engine. https://v8.dev/
[6] Bytecode Alliance. 2019. WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime. Accessed: August 24, 2025.
[7] Lennart Almstedt, Kai Bleeke, Mohammad Mahhouk, Leander Jehl, Rüdiger Kapitza, and Lars Wolf. 2023. ContractBox: Realizing accountable data sharing on the edge using a small scale blockchain. *Computer Networks* 229 (June 2023), 109768. doi:10.1016/j.comnet.2023.109768
[8] Matteo Basso. 2026. mbasso/awesome-wasm. https://github.com/mbasso/awesome-wasm original-date: 2017-04-25T11:57:40Z.
[9] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094&rep=rep1&type=pdf
[10] Bytecode Alliance. 2026. Running wasm-opt. https://bytecodealliance.github.io/cargo-wasi/wasm-opt.html. [Accessed 13-03-2026].
[11] bytecodealliance. 2017. Wasmtime. https://docs.wasmtime.dev/. Accessed: August 24, 2025.
[12] Jonas Devlieghere. 2026. WebAssembly Debugging. https://jonasdevlieghere.com/post/wasm-debugging. [Accessed 13-03-2026].
[13] DWARF. 1989. DWARF Debugging Format. https://dwarfstd.org/. Accessed: August 24, 2025.
[14] emscripten. 2015. emscripten. https://emscripten.org/. Accessed: August 24, 2025.
[15] Brent Fulgham and Isaac Gouy. 2026. Measured : Which programming language is fastest? (Benchmarks Game) — benchmarksgame-team.pages.debian.net. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. [Accessed 06-03-2026].
[16] GNU. 1986. The GNU Project Debugger. https://www.gnu.org/software/gdb/. Accessed: August 24, 2025.
[17] Guillaume Gomez. 2026. sysinfo - Cross-platform library to fetch system information. https://github.com/GuillaumeGomez/sysinfo. [Accessed 14-03-2026].
[18] Google. 2024. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/. Accessed: Dec 24, 2024.
[19] Google. 2026. WebAssembly compilation pipeline. https://v8.dev/docs/wasm-compilation-pipeline. [Accessed 13-03-2026].
[20] Robert Griesemer, Rob Pike, and Ken Thompson. 2009. Go. https://go.dev/. Accessed: August 24, 2025.
[21] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF

Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. doi:10.1145/3062341.3062363

[22] Klaus Hinum. [n. d.]. Apple M2 Max Processor - Benchmarks and Specs — notebookcheck.net. https://www.notebookcheck.net/Apple-M2-Max-Processor-Benchmarks-and-Specs.682771.0.html. [Accessed 28-05-2025].

[23] Thomas Hirsch and Birgit Hofer. 2021. What we can learn from how programmers debug their code. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. 37–40. doi:10.1109/SER-IP52554.2021.00014

[24] Graydon Hoare. 2012. Rust. https://www.rust-lang.org/. Accessed: August 24, 2025.

[25] Andrew Kelley. 2016. Zig. https://ziglang.org/. Accessed: August 24, 2025.

[26] Wasm3 Labs. 2019. wasm3. https://github.com/wasm3/wasm3. Accessed: August 24, 2025.

[27] Tom Lauwaerts, Robbert Gurdeep Singh, and Christophe Scholliers. 2024. WARDuino: An Embedded WebAssembly Virtual Machine. *Journal of Computer Languages* (Feb. 2024), 101268. doi:10.1016/j.cola.2024.101268

[28] Tom Lauwaerts, Carlos Rojas Castillo, Robbert Gurdeep Singh, Matteo Marra, Christophe Scholliers, and Elisa Gonzalez Boix. 2022. Event-Based Out-of-Place Debugging. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 85–97. doi:10.1145/3546918.3546920

[29] Tom Lauwaerts, Maarten Steevens, and Christophe Scholliers. 2025. MIO: Multiverse Debugging in the Face of Input/Output. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (Oct. 2025), 2396–2425. doi:10.1145/3763136

[30] LLVM. [n. d.]. The LLDB Debugger. https://lldb.llvm.org/. Accessed: August 24, 2025.

[31] Microsoft. 2026. The Debug Adapter Protocol (DAP) defines the abstract protocol used between a development tool (e.g. IDE or editor) and a debugger. https://microsoft.github.io/debug-adapter-protocol/. Accessed: March 12, 2026.

[32] Mozilla. 2004. Firefox. https://www.mozilla.org/. Accessed: August 24, 2025.

[33] Aäron Munsters, Angel Luis Scull Pupo, and Elisa Gonzalez Boix. 2025. Wastrumentation: Portable WebAssembly Dynamic Analysis with Support for Intercession. In *39th European Conference on Object-Oriented Programming (ECOOP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:29. doi:10.4230/LIPIcs.ECOOP.2025.23

[34] Opera. 1995. Opera. https://www.opera.com/. Accessed: August 24, 2025.

[35] Carlos Rojas Castillo, Matteo Marra, Jim Bauwens, and Elisa Gonzalez Boix. 2022. Out-of-things debugging: A live debugging approach for Internet of Things. *arXiv preprint arXiv:2211.01679* (2022).

[36] Carlos Rojas Castillo, Matteo Marra, and Elisa Gonzalez Boix. 2025. A Control-Flow Graph Approach to Language-Agnostic Debugging for Microcontrollers. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Singapore, Singapore) *(MPLR '25)*. Association for Computing Machinery, New York, NY, USA, 38–56. doi:10.1145/3759426.3760979

[37] SourceMap. 2025. SourceMap. https://tc39.es/source-map/. Accessed: August 24, 2025.

[38] Bala Subramanyan, Arne Hollum, Giovanni Mazzeo, Matt Ficke, and Darshan Vaydia. 2023. Enabling Trusted TEE-as-a-Service Models with Privacy Preserving Automatons. In *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 252–260. doi:10.1109/CloudCom59040.2023.00048 ISSN: 2380-8004.

[39] Ben L. Titzer. 2022. A fast in-place interpreter for WebAssembly. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 148 (Oct. 2022), 27 pages. doi:10.1145/3563311

[40] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. 2019. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:30. doi:10.4230/LIPIcs.ECOOP.2019.27

[41] WASI Subgroup. 2026. WASI.dev is an introduction to the WebAssembly System Interface (WASI). https://wasi.dev/. Accessed: March 12, 2026.

[42] Wasmer. 2024. Run your apps secure. Fast. At scale. https://wasmer.io/. Accessed: Dec 24, 2024.

[43] WebAssembly. 2015. Binaryen. https://github.com/WebAssembly/binaryen. Accessed: August 24, 2025.

[44] Yixuan Zhang, Mugeng Liu, Haoyu Wang, Yun Ma, Gang Huang, and Xuanzhe Liu. 2025. Research on WebAssembly Runtimes: A Survey. *ACM Trans. Softw. Eng. Methodol.* 34, 8 (Oct. 2025), 239:1–239:47. doi:10.1145/3714465