# Dead Weight: Analyzing Code Bloat and Its Security Implications in WebAssembly Binaries

Pratik M. Kamble[1(✉)][0009−0004−5375−1673] and
Aravind Prakash[1][0000−0002−2994−0480]

Binghamton University, Binghamton, NY, USA
{pkamble1, aprakash}@binghamton.edu

**Abstract.** Code bloat in WebAssembly (Wasm) binaries is not merely a size concern—it is a measurable security liability. Unused code expands the attack surface by inflating control-flow targets, preserving security-relevant operations, and retaining unnecessary host privileges. Despite these risks, the security implications of Wasm bloat have not been systematically quantified.

This paper presents an analysis of code bloat across a diverse corpus of over 8,000 real-world binaries, including benchmark suites, widely used libraries, and a Linux kernel compiled to Wasm. We combine staged static reachability analysis with dynamic coverage measurement using two independent pipelines. Our results reveal a fundamental gap: while conservative static analysis finds near-zero provably dead code under realistic assumptions, dynamic analysis shows that 70–85% of functions remain unexecuted under typical workloads.

To assess the security impact, we perform an analytical debloating simulation on optimized library binaries. Removing dead code significantly hardens the binary: it reduces average Control-Flow Integrity (CFI) equivalence class sizes by 4.9–11.6×, limiting the targets available for indirect-call hijacking; eliminates up to 40.7% of sensitive WASI host imports reachable only via dead code paths; and removes up to 81% of memory-store and 84% of `call_indirect` operations that constitute exploitation primitives. These findings demonstrate that effective debloating requires profile-guided techniques, and that such debloating yields substantial security hardening.

**Keywords:** WebAssembly · Code Bloat · Security · Attack Surface · Control-Flow Integrity · Static/Dynamic Analysis

## 1 Introduction

WebAssembly (Wasm) is a low-level instruction set architecture (ISA) designed to be a browser-portable compilation target for source languages like C, C++, and Rust [6]. Over the years, it has achieved widespread adoption and is a core technology in modern web development. It is supported by all major web browsers, including Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge, and Opera, and is increasingly deployed in server-side and edge environments [14,15].

Although Wasm executes inside a sandbox, binaries compiled from memory-unsafe languages can still preserve source-level vulnerabilities such as buffer overflows and use-after-free errors [11]. Prior work has proposed stronger memory-safety mechanisms for Wasm [4], and implementation-level bugs in browser Wasm engines have enabled remote code execution, as illustrated by CVE-2024-2887 [17]. Empirical studies further show that real-world Wasm modules often originate from memory-unsafe languages and import security-sensitive host APIs [8].

In this setting, "code bloat"—excessive unused code [19,10]—is a serious concern. Bloat can introduce exploitable vulnerabilities that harm dependability.

In this paper, we perform a multi-level analysis of code bloat in Wasm binaries to provide a holistic view of bloat in real-world Wasm code and to understand the limits of static and dynamic approaches in this setting.

We perform principled static analysis using a staged reachability model (four stages, from strict entry-only roots to conservative "exports plus all table entries"). We also perform dynamic analysis at instruction, basic-block, and function levels using two independent pipelines: the Wizard research engine [26] (non-intrusive monitors) and Walrus-based [23] static binary instrumentation. Two pipelines allow cross-checking and reduce the risk of engine-specific artifacts.

Our test set comprises: more than 8,000 real-world Wasm binaries drawn from the WasmBench dataset [8]; the 27-program Wasm-R3 benchmark suite [1]; three real applications—SQLite, zlib, and Duktape—compiled to wasm32-wasi [22] at -O0, -O2, and -Oz; microbenchmarks and a libc feature ladder; and a Linux kernel compiled to WebAssembly [20].

**Contributions.**

- A staged static reachability analysis (S1–S4) that exposes a fundamental limitation: once Wasm tables are modeled conservatively, static bloat collapses to near zero despite 70–85% dynamic bloat, demonstrating that static DCE is structurally inadequate for Wasm.
- Dual-pipeline dynamic analysis (Wizard + Walrus) measuring bloat at function, basic-block, and instruction levels, cross-validated to within 0.14% mean delta across 27 benchmarks.
- Evaluation across a diverse corpus: 8,461 WasmBench binaries, 27 Wasm-R3 benchmarks, three libraries at three optimization levels, microbenchmarks, and a Linux kernel compiled to Wasm.
- A security assessment showing that dead code inflates CFI equivalence classes by up to 37×, exposes up to 40.7% of WASI imports exclusively through dead paths, and retains over 80% of memory-write and indirect-call operations as exploitation primitives.
- An analytical debloating simulation demonstrating that profile-guided removal of dead code substantially reduces all three attack-surface dimensions without binary modification.

*Why WebAssembly Bloat Differs from Native Bloat.* While bloat is well-documented in native binaries [19], Wasm's architecture makes the consequences fundamentally different: (1) Wasm's table-driven `call_indirect` defeats conservative

static DCE—once exports and tables are modeled conservatively, static bloat collapses to near zero even when 70–85% of code is dynamically dead; native LTO removes 14–68% of functions in our experiments (Section 8.1), while Wasm S4 removes 0%. (2) Wasm modules explicitly declare host imports; dead code retains transitive paths to these imports, creating an attack vector with no direct native analog. (3) Wasm provides only coarse-grained forward-edge CFI via type-signature checks; native platforms deploy shadow stacks, pointer authentication, and per-call-site whitelists.

**Summary of Results.** Dynamic measurements show only 20–25% of functions execute (median bloat ≈77%). For Linux-on-Wasm, ≈84% of instructions remain unexecuted. Even at `-Oz`, the majority of code is unused. Static bloat collapses to near zero under conservative table models. Dead code inflates CFI classes by up to 37×, retains up to 41% of WASI imports, and harbors over 80% of security-relevant operations.

## 2 Background

### 2.1 Wasm Architecture

A Wasm module is a binary-encoded program designed for safe, fast, and portable execution [6,28]. A module consists of types (function signatures), imports, functions, function-pointer tables, memory, globals, exports, an optional start function, element segments, code, and data segments.

The *code section* contains function bodies as sequences of Wasm instructions, each with a type signature. Functions can be imported from the host or defined locally; the index space interleaves imported (lower indices) and defined (higher indices) functions. WebAssembly uses *structured control flow*—`block`, `loop`, `if/else`, and branch instructions [6]—and does not permit arbitrary jumps.

The `call_indirect` instruction enables *dynamic dispatch* by indexing into a function table populated via element segments. Since the table index is computed at runtime, static resolution of call targets is non-trivial. This mechanism is widely used for function pointers, virtual methods, and callbacks. The *export section* declares functions, tables, memories, and globals accessible to the host. For WASI modules, `_start` typically serves as the primary entry point; browser-embedded modules may export multiple functions for JavaScript to call.

### 2.2 Code Bloat and Security

Software bloat—code that consumes resources without providing value [19]—expands the attack surface, since vulnerabilities in unused code remain exploitable if an attacker can reach them [11]. Prior work shows only 20–30% of native code executes under typical workloads [19]. For Wasm, many binaries originate from memory-unsafe languages and import powerful host APIs [8], making precise static debloating difficult.

We use three notions of bloat throughout this paper. *Static bloat*: code unreachable from a chosen root set, provably removable. *Dynamic bloat*: statically reachable but unexecuted under a given workload—rarely used features, error paths, and dormant subsystems. *Byte bloat*: binary bytes (including debug/custom sections) not contributing to executed behavior.

### 2.3 Challenges of Bloat Analysis in Wasm

Analysing bloat in Wasm presents several challenges.

*Indirect calls and tables:* Wasm uses function tables and the `call_indirect` instruction for dynamic dispatch. Conservative assumptions about tables (treating all table entries as potential targets) can make almost all functions appear reachable, trivializing static bloat [21,8]. Constructing sound call graphs in the presence of host-mutated tables and complex embeddings remains an open problem.

*Entry-point variability:* WASI-style modules typically have a single canonical entry point (`_start`), whereas library-style modules may export many functions and be invoked from arbitrary host code. Static reachability is highly sensitive to this choice.

*Workload dependence:* Dynamic bloat depends on the workloads used. A function that appears dead under one workload may execute under another. Representative workload selection is therefore critical for meaningful dynamic measurements.

*Granularity and stack validation:* Bloat can be measured at function, basic-block, and instruction levels. In WebAssembly, measuring at basic-block and instruction level is harder than in native code because of stack validation. Wasm is a typed stack machine: the engine verifies that every instruction sees the right number and types of values [28,27]. Any instrumentation must preserve stack effects and structured control flow, whereas native instrumentation can usually add probes at basic-block boundaries without such structural checks.

## 3 Methodology

### 3.1 Working Example

To make these ideas concrete before turning to larger benchmarks, we use a simple "addition" program as a running example. In all variants, the program defines a small `add(a, b)` function that returns $a + b$, and a `main` that calls it once and prints the result. The variants differ only in language and toolchain: C (Emscripten), C (wasi-sdk), C++ (Emscripten), and Rust.

Table 1 shows that each binary contains significant code well beyond what is necessary to add two numbers. The excess functions come from runtime support and standard library code. The Rust binary contains 200 functions (1.8 MB) versus 42 for C/WASI (93 KB). The C++ variant, compiled with Emscripten, includes 1,710 functions due to iostream and exception handling support.

**Table 1.** Function Reachability and Execution Across Analysis Stages

| Binary | Total | S1 | S2 | S3 | S4 | Dyn. |
|---|---|---|---|---|---|---|
| **c** | 62 | 40 | 47 | 59 | 62 | 28 |
| **c_wasi** | 42 | 29 | 29 | 38 | 42 | 20 |
| **cpp** | 1710 | 461 | 468 | 1492 | 1710 | 407 |
| **rust** | 200 | 84 | 84 | 200 | 200 | 39 |

These results highlight three points that recur throughout the paper. First, even when conservative static analysis (S4) marks almost all functions as reachable (static bloat near zero), dynamic coverage shows that most compiled code never executes. Second, dynamic bloat is already large in trivial programs: only 20–45% of functions run, and the rest is runtime initialization and unused library machinery. Third, the choice of source language and toolchain has a dramatic effect on bloat magnitude. The Rust binary is substantially larger (200 functions, 1.8 MB vs. 42 functions, 93 KB for C/WASI), primarily due to Rust's runtime infrastructure—panic handler, allocator, and string formatting machinery—that is statically linked into every binary. While Rust's safety features (bounds checking, `Option`/`Result` types) contribute some additional code, the dominant source of bloat is runtime and standard library support.

### 3.2 Staged Static Analysis (S1–S4)

The static analysis uses four stages with progressively relaxed assumptions: **S1**—only designated entry points as roots; **S2**—all exported functions as roots; **S3**—type-compatible table entries added as roots (type-filtered indirect calls); **S4**—all table entries as roots (conservative exports+tables model). For each stage, the analysis computes the set of reachable functions and aggregates static instruction counts. Under S4, any unreachable function is a sound lower bound on dead code. The difference between S4 and dynamic coverage reflects reachable-but-unexecuted code.

### 3.3 Dynamic Measurement with Dual Pipelines

Dynamic bloat is measured using two independent pipelines: (a) **Wizard engine monitors**, which collect instruction counts and basic-block coverage non-intrusively inside a production-grade Wasm engine, and (b) **Walrus-based static instrumentation**, which inserts `mark_bb` calls at basic-block entries, allowing engines and embeddings that do not support Wizard (including Linux-on-Wasm running in a browser) to be instrumented. Using two pipelines allows cross-checking coverage measurements for the same binaries and workloads and increases confidence in the results.

### 3.4 Research Questions

Our analysis is guided by five research questions:

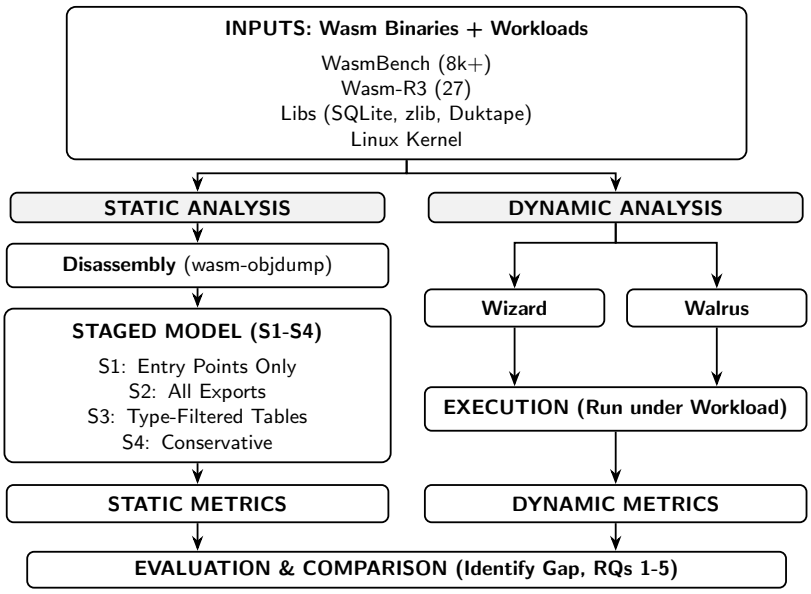


**Fig. 1.** Methodology: staged static reachability analysis combined with dual-pipeline dynamic coverage.

- **RQ1** (§6.3): How much bloat exists in real-world Wasm binaries at function, basic-block, and instruction levels?
- **RQ2** (§6.4): How do static estimates change across S1–S4, and how do they compare to dynamic bloat?
- **RQ3** (§6.5): How do optimization levels and toolchains affect bloat in real applications?
- **RQ4** (§6.6): What are the implications for large systems such as Linux-on-Wasm, and how does byte-level bloat relate to code-execution bloat?
- **RQ5** (§7): What is the security impact of code bloat, and how effectively does profile-guided debloating reduce the attack surface?

Figure 1 summarizes this methodology.

## 4 Static Bloat Detection

Static bloat detection employs reachability analysis to identify functions that cannot be executed from designated entry points under a given model. The analysis builds a call graph from roots and marks all transitively reachable functions. Functions not in the reachable set constitute static bloat.

### 4.1 Four-Stage Reachability Model

The static analysis implements four stages with progressively relaxed assumptions.

*Stage 1 (S1): Entry Points Only.* Roots consist of designated entry points: the `_start` function for WASI modules, or a benchmark-specific `main` function. Direct calls are followed transitively; `call_indirect` is ignored. This stage reflects a strict single-entry model.

*Stage 2 (S2): All Exports.* Roots expand to include all exported functions. This models library-style usage where any export may be called by the host. Direct calls are followed; indirect calls are still ignored.

*Stage 3 (S3): Type-Filtered Indirect Calls.* When a `call_indirect` is encountered, potential targets are constrained to functions in the table with compatible type signatures. This exploits Wasm's type system to approximate dynamic dispatch while remaining conservative.

*Stage 4 (S4): Conservative Exports+Tables Model.* All functions that appear in any element segment (i.e., any table entry) are added to the root set. This stage models a closed-world scenario where any table entry may be called. Under this model, any function unreachable at S4 is provably dead with respect to the exports+tables model and constitutes a static lower bound on bloat.

### 4.2 Implementation

The static analyzer is a Python tool that processes disassembly and metadata produced by `wasm-objdump` [24]. It extracts the function index space, export table, type section, and element segments, distinguishing imported from local functions. For each local function, it scans instructions to identify direct call targets and `call_indirect` sites, recording type indices used. For each stage S1–S4, a worklist algorithm starts from the corresponding root set and marks reachable functions by following direct calls and stage-specific indirect targets. Static function bloat is computed as the fraction of local functions not reachable; instruction counts per function are aggregated to compute static instruction bloat.

### 4.3 Interpreting Staged Results

S1 and S2 highlight potential bloat under restricted entry assumptions (e.g., a specific embedding that uses only a few exports). S3 illustrates how type-aware modeling of tables can drastically shrink apparent static bloat, especially in programs that heavily use function pointers and virtual dispatch. S4 provides a conservative lower bound on dead code under the exports+tables model: S4-unreachable code can be removed without affecting any execution that respects this model.

However, in many Wasm modules, S3 and S4 mark almost all functions reachable, especially when large libraries are exposed via tables. In such cases, static S4 bloat is negligible, and static analysis alone cannot identify substantial dead code.

## 5 Dynamic Bloat Detection

Dynamic bloat detection measures which functions, basic blocks, and instructions execute under concrete workloads.

### 5.1 Metrics

Dynamic bloat is computed at three granularities: (1) **Function-level bloat** = (total functions − executed functions) / total functions; (2) **Basic-block-level bloat** = (total basic blocks − executed basic blocks) / total basic blocks; (3) **Instruction-level bloat** = (total static instructions − executed static instructions) / total static instructions. Basic-block and instruction-level metrics reveal dead regions inside otherwise "live" functions.

### 5.2 Wizard Engine Monitors

The Wizard research engine provides built-in monitoring for Wasm execution [26]. Coverage monitors used in this work include: a *coverage monitor* reporting the number of executed and total instructions and basic blocks, along with percentages; and an *instruction-count monitor* reporting, for each function, static instruction count and dynamic instruction count. A function is considered executed if its dynamic instruction count is greater than zero. Wizard instrumentation is non-intrusive and integrated into the execution engine, minimizing overhead.

### 5.3 Walrus-Based Instrumentation

Walrus [23] is a Rust library for manipulating Wasm modules. For engine-agnostic dynamic analysis and for complex embeddings, a second pipeline uses Walrus to instrument binaries.

**Basic-block identification:** Within each local function, basic block entry points are the function's entry point, successors of unconditional and conditional branches (`br`, `br_if`, `br_table`, `return`, `unreachable`), and structured constructs (`block`, `loop`, `if`/`else`) at entry and fall-through positions.

**Instrumentation:** For each basic block, a unique `bb_id` is created. Two instructions (`i32.const bb_id` and `call mark_bb`) are inserted before the block's first original instruction.

**Runtime collection:** When a basic block executes, a bit corresponding to the block in a bitset is marked. After workload execution, the bitset is used to compute executed vs total basic blocks, instruction counts (summing per-block counts), and function coverage (functions with at least one executed block).

### 5.4 Cross-Engine Consistency

Both Wizard and Walrus pipelines are run on identical workloads and binaries. Across all 27 Wasm-R3 benchmarks, the mean function-level delta between

Wizard and Walrus is 0.13% and the mean instruction-level delta is 0.14%, with maximums of 1.55% and 0.44% respectively (full per-benchmark data in the Appendix). This agreement confirms that the observed bloat patterns are not artifacts of a specific engine or instrumentation methodology.

### 5.5 Linux-on-Wasm Instrumentation

Linux compiled to WebAssembly [20] is deployed as part of a browser-based environment with JavaScript glue, a terminal UI, and at least one Web Worker for the main kernel execution. Walrus-based instrumentation is applied to the Linux Wasm binary, inserting `mark_bb` probes at basic-block entries. The JavaScript harness exports `mark_bb` to the module as an imported function. Coverage bitsets are collected per-worker using atomic operations and merged after execution for offline analysis. This setup enables measurement of kernel coverage under realistic browser-based execution without re-hosting the module in a standalone engine.

## 6 Evaluation

### 6.1 Datasets

We perform our evaluation on five datasets comprising programs with varying degrees of complexities. **Wasm-R3 benchmarks:** 27 binaries with embedded replay workloads [1], including real-world applications compiled to Wasm with captured traces. **WasmBench-derived corpus:** Over 8,000 unique binaries collected from public repositories, NPM packages, and websites [8]. Used for large-scale static staged analysis. **Libraries (SQLite, zlib, Duktape):** Compiled to wasm32-wasi at -O0, -O2, and -Oz. SQLite is exercised with an SQL script; zlib compresses and decompresses a large text file; Duktape runs a JavaScript workload. **Linux kernel:** A Linux kernel (v6.1) compiled to Wasm [20] and executed in a browser environment, instrumented via Walrus. **Microbenchmarks:** Simple addition programs from Section 3.1 and a feature ladder that incrementally enables libc features (malloc, printf, file I/O, environment access).

Table 2 summarizes which datasets are used for each analysis pipeline. All 8,461 WasmBench binaries are successfully parsed and used for static staged analysis (S1–S4). Dynamic analysis requires reproducible workloads to produce meaningful coverage measurements; the WasmBench binaries lack standardized inputs and execution harnesses. Dynamic measurement is therefore performed on datasets where we control workloads: Wasm-R3 benchmarks (embedded replay traces), libraries (custom test scripts), and Linux-on-Wasm (interactive shell session).

Workloads were designed to exercise core functionality of each application. The SQLite workload covers DDL, DML, joins, subqueries, CTEs, window functions, aggregations, and transactions. The Duktape workload exercises strings, arrays, objects, JSON, regular expressions, exceptions, closures, prototypes, Date/Math,

**Table 2.** Dataset Usage Across Analysis Pipelines

| Dataset | Samples | Static | Dynamic | Purpose |
|---|---|---|---|---|
| WasmBench | 8,461 | S1–S4 | — | Large-scale static reachability |
| Wasm-R3 | 27 | S1–S4 | Wizard + Walrus | Cross-validated bloat measurement |
| Libraries | 9 | S1–S4 | Wizard | Optimization-level analysis |
| Linux kernel | 1 | S1–S4 | Walrus | Large-system case study |
| Microbenchmarks | 8 | S1–S4 | Wizard | Feature-ladder analysis |

**Table 3.** Summary of Dynamic Bloat in Wasm-R3 Benchmarks

| Metric | All ($n$=27) | | Reasonable ($n$=21) | |
|---|---|---|---|---|
| | Mean | Median | Mean | Median |
| Function Bloat | 77.0% | 77.2% | 70.6% | 69.8% |
| Instruction Bloat | 76.9% | 86.0% | 70.4% | 77.3% |
| Block Bloat | 81.3% | 86.5% | 76.0% | 79.0% |

"Reasonable" excludes 6 trivial or initialization-only benchmarks (<10 functions executed).

`eval`, and property descriptors. zlib compresses and decompresses a 1 MB text file, exercising both compression paths. Dead code corresponds primarily to optional subsystems (e.g., SQLite's FTS and JSON extensions, Duktape's ES5.1 edge-case handlers) that are statically linked but unused by any single workload.

### 6.2 Experimental Setup

Emscripten 4.0.15 [29], wasi-sdk 20.1.8 [25], rustc 1.90.0 [13] (wasm32-wasi). Static analysis: Python 3.12.3 + WABT 1.0.37 [24]; DCE baseline: Binaryen 124 [30]. Dynamic: Wizard engine [26] + Walrus 0.19 [23]. Hardware: Intel i7-4790 (4C/8T, 3.6 GHz), 32 GiB RAM, SSD, Linux 6.14.

### 6.3 RQ1: Extent of Dynamic Bloat

Across the Wasm-R3 benchmarks, dynamic bloat is prevalent. The majority of code remains unexecuted under realistic workloads, leaving substantial room for debloating and attack-surface reduction. Table 3 shows aggregated results. In a typical Wasm-R3 benchmark, only about 20–25% of functions and instructions execute under its embedded workload (i.e., the workload it comes packaged with), and only ≈15–20% of basic blocks execute.

### 6.4 RQ2: Static vs Dynamic Bloat

This section examines the relationship between staged static bloat and dynamic bloat.

**Table 4.** Static vs Dynamic (Wasm-R3)

| Stage | Mean | Median |
|---|---|---|
| S1: Entry only | 55.6% | 49.1% |
| S2: All exports | 42.9% | 43.8% |
| S3: Type-filtered | 3.4% | 0.1% |
| S4: Conservative | 0.0% | 0.0% |
| **Dynamic** | **77.0%** | **77.2%** |

**Table 5.** Static Bloat (WasmBench, $n$=8,461)

| | Func % | | Instr % | |
|---|---|---|---|---|
| **Stage** | Mean | Med | Mean | Med |
| S1 | 42.1 | 44.3 | 40.7 | 40.4 |
| S2 | 35.3 | 32.1 | 34.6 | 23.7 |
| S3 | 3.0 | 0.0 | 1.4 | 0.0 |
| S4 | 1.8 | 0.0 | 1.1 | 0.0 |

**Wasm-R3 and WasmBench.** Tables 4 and 5 present staged static bloat for the Wasm-R3 benchmarks and the WasmBench corpus, respectively.

Staged static analysis on the Wasm-R3 benchmarks reveals strong sensitivity to entry-point selection. At S2 (exports-only), median static function bloat is ≈40–45%. Moving to S3 (typed tables) causes a dramatic reduction: median function bloat falls below 0.2% and median instruction bloat below 0.01%. S4 (all tables) further collapses residual static bloat to effectively 0% for most benchmarks.

The WasmBench corpus (Table 5) confirms this pattern at scale: S1 and S2 report significant static bloat, but S3 and S4 aggressively shrink it, particularly in modules with heavy table usage. Under conservative assumptions (S3/S4), static analysis declares nearly all code reachable, even though dynamic analysis shows ≈70–80% bloat. Static lower bounds on bloat are therefore small for table-heavy modules.

**Feature Ladder.** The feature ladder microbenchmarks stress how quickly bloat appears even in small programs. Table 6 shows a single C program compiled in four variants that progressively enable more of libc: a baseline with no heap or I/O, then configurations adding printf, file I/O, and finally environment/strcmp.

At the function level, S1 reachability is sound for all variants: every function that executes is included in the S1 reachable set. For the lightest configuration (baseline), there is only one defined function, it is reachable, and it executes, so both static and dynamic function bloat are 0%. As more features are enabled, dynamic bloat rises to ≈40% (functions) and 80–90% (instructions)—`printf` contains lengthy switch cases for format specifiers, most of which are unused. Under S3/S4, all library functions become reachable and static bloat collapses to 0%, while dynamic bloat remains high. This illustrates the central gap: the staged static model has perfect recall for executed functions, but instruction-level static bloat severely underestimates dynamic bloat once libc features are enabled.

### 6.5 RQ3: Toolchains and Optimization Levels

This section examines how compilers and optimization levels influence bloat. Table 7 summarizes static and dynamic bloat for SQLite, zlib, and Duktape

**Table 6.** Feature Ladder Microbenchmarks (WASI C)

| | | Functions | | Dyn. Bloat (%) | | Static S4 (%) | |
|---|---|---|---|---|---|---|---|
| **Variant** | **Features** | S1 | Exec | Func | Instr | Func | Instr |
| `feat_base` | None (No Heap/IO) | 1 | 1 | 0.0 | 0.0 | 0 | 0 |
| `feat_printf` | + printf | 13 | 8 | 38.5 | 82.5 | 0 | 0 |
| `feat_file` | + file I/O | 22 | 13 | 40.9 | 87.2 | 0 | 0 |
| `feat_all` | + env, strings | 24 | 14 | 41.7 | 88.0 | 0 | 0 |

**Table 7.** Static and Dynamic Bloat Across Optimization Levels

| | | Binary | Static Bloat (%) | | | | Dynamic Bloat (%) | | |
|---|---|---|---|---|---|---|---|---|---|
| **App** | **Opt.** | **Size** | **S1** | **S2** | **S3** | **S4** | **Func** | **Instr** | **Block** |
| SQLite | -O0 | 3.09 MB | 30.1 | 30.1 | 0.1 | 0.0 | 91.2 | 96.8 | 96.2 |
| | -O2 | 1.45 MB | 43.8 | 43.8 | 0.2 | 0.0 | 62.9 | 78.2 | 77.2 |
| | -Oz | 1.01 MB | 36.5 | 36.5 | 0.2 | 0.0 | 57.8 | 72.0 | 71.2 |
| zlib | -O0 | 0.40 MB | 7.0 | 7.0 | 0.0 | 0.0 | 40.9 | 67.1 | 71.2 |
| | -O2 | 0.26 MB | 14.0 | 14.0 | 0.0 | 0.0 | 39.5 | 72.2 | 74.4 |
| | -Oz | 0.25 MB | 12.4 | 12.4 | 0.0 | 0.0 | 40.2 | 70.2 | 72.0 |
| Duktape | -O0 | 1.21 MB | 54.0 | 54.0 | 0.0 | 0.0 | 73.8 | 83.6 | 84.8 |
| | -O2 | 0.74 MB | 63.6 | 63.6 | 0.0 | 0.0 | 74.6 | 85.6 | 84.2 |
| | -Oz | 0.49 MB | 55.7 | 55.7 | 0.0 | 0.0 | 65.6 | 78.9 | 78.1 |

compiled with wasi-sdk at -O0, -O2, and -Oz and executed under their respective workloads.

*SQLite.* Binary size decreases from 3.09 MB at -O0 to 1.01 MB at -Oz (3× reduction). Static S3/S4 bloat collapses to near 0%, while dynamic function bloat ranges from 91.2% (-O0) to 57.8% (-Oz). Even at -Oz, over half of functions and 70% of instructions remain unexecuted, corresponding to optional subsystems (FTS, JSON extensions, ALTER TABLE) statically linked but unused by the workload.

*zlib.* Dynamic function bloat is ≈40% across all optimization levels, with instruction bloat between 67–72%. Static S1 bloat is modest (7–14%), indicating most code is reachable from the entry point but never exercised.

*Duktape.* Dynamic function bloat is 74–75% at -O0/-O2 and 66% at -Oz. Instruction bloat remains 80–86% even at -Oz. Re-measurement with an expanded workload exercising 20 language subsystems produced identical results (74.6% function bloat at -O2), confirming that dead code corresponds to structural engine internals rather than workload-dependent paths.

**Table 8.** Linux-on-Wasm Bloat Statistics (Interactive Workload)

| Metric | Total | Executed | Unused | Bloat (%) |
|---|---|---|---|---|
| Functions | 9,842 | 2,182 | 7,660 | 77.8 |
| Basic Blocks | 153,437 | 23,924 | 129,513 | 84.4 |
| Instructions | 1,080,432 | 176,105 | 904,327 | 83.7 |
| Code Bytes | 3,221,612 | 518,481 | 2,703,131 | 83.9 |

*Toolchain and optimization level affect bloat but do not change the fundamental pattern.* Across C libraries and a JS engine, the majority of code remains unused under realistic workloads, even in size-optimized configurations.

### 6.6 RQ4: Large System — Linux-on-Wasm

The Linux OS compiled to Wasm [20] serves as an extreme case study. The instrumented binary contains 9,842 functions, 153,437 basic blocks, 1,080,432 static instructions, and ≈3.22 MB of code bytes. Under an interactive shell workload consisting of typical commands (ls, cat, echo, basic file operations, and /proc queries), Table 8 summarizes the results. Only 22.2% of functions and 16.3% of instructions execute, leaving over 80% of code unused.

*Even in a large, complex system like Linux-on-Wasm, more than 80% of instructions and bytes remain unexecuted under a typical interactive workload.* Function-level coverage (22%) overestimates how much of the code is "hot"; most executed functions contain internal dead blocks. For Wasm deployments, this unused code inflates network transfer and compilation time and represents latent attack surface if additional features or workloads become reachable.

*Byte Bloat vs. Code-Execution Bloat.* To separate byte-level from execution-level bloat, we stripped debug and custom sections from SQLite at `-O2` using `wasm-tools strip`, reducing file size by ≈200 KB. Dynamic bloat metrics changed by less than 0.1 percentage points. This is expected: debug and custom sections contain metadata (symbol names, source mappings, producer information) but no executable code. Stripping them reduces file size but does not change which functions, basic blocks, or instructions execute. Separately, `wasm-opt -dce` removed negligible code because, under S4, nearly all functions are considered reachable. Existing DCE tools operating at S4 cannot eliminate workload-specific dead code; addressing dynamic bloat requires profile-guided techniques.

## 7 Security Assessment

*Threat Model.* We assume the presence of a memory safety vulnerability (e.g., buffer overflow or type confusion) within a Wasm module. The attacker may influence inputs and corrupt memory, including table indices used in `call_indirect`. We do not assume binary modification or runtime compromise.

Under this model, dead code increases exploitability in three ways: (1) it enlarges the set of type-compatible indirect-call targets, weakening the forward-edge control-flow integrity (CFI) that Wasm's type system provides; (2) it retains transitive paths to privileged host imports that would otherwise be unreachable; and (3) it preserves security-relevant operations (memory writes, calls) that can be reached via redirected indirect calls. Although these functions are never executed under intended workloads, they remain fully validated, type-compatible, and executable under attacker-controlled control flow.

### 7.1 Debloating Simulation Methodology

Using Wizard `icount` coverage profiles from Section 6, we partition functions into *live* (dynamic count $> 0$) and *dead* (count $= 0$), then analytically compute three post-debloating metrics: (1) **CFI equivalence classes**—for each `call_indirect` type, count type-compatible table entries before (all) and after (live only); (2) **WASI import reachability**—compute transitive reachability from live/dead functions; imports reachable *only* from dead code are eliminable (classified as high/medium/low sensitivity); (3) **Security-relevant opcodes**—partition instructions into dead/live functions, counting memory stores, `call_indirect`, and direct calls. This requires no binary rewriting; all parsing uses `wasm-objdump` [24].

### 7.2 Strengthening Control-Flow Integrity

WebAssembly enforces coarse-grained forward-edge CFI through its type system: a `call_indirect` instruction traps unless the target function's signature matches the call site's declared type [6]. The set of type-compatible functions for a given call site constitutes a *CFI equivalence class*. Larger classes give an attacker who corrupts a table index more valid redirection targets.

Table 9 quantifies the inflation. In SQLite, the average class drops from 36.7 to 7.5 (**4.9**× inflation). Duktape shows **11.6**× average and **37.2**× maximum inflation (186→5). zlib shows minimal inflation (1.1×), consistent with its low bloat. Documented implementation vulnerabilities such as CVE-2024-2887 [17] show that WebAssembly engines can contain exploitable type-confusion bugs; our analysis quantifies how much the set of valid type-compatible redirection targets shrinks after debloating. Debloating reduces average class sizes by 79.6% (SQLite) to 91.4% (Duktape). Notably, S4 static analysis classifies nearly all table functions as reachable, failing to expose this inflation.

### 7.3 Attack Surface Reduction: WASI Imports

The principle of least privilege requires that a module retain access only to host capabilities it actually uses. Dead code violates this principle by preserving transitive call paths to WASI imports that no live function ever invokes.

**Table 9.** Security Impact of Code Bloat: CFI Equivalence Class Inflation and WASI Import Exposure (`-O2` Binaries)

| Binary | Avg CFI Class Size | | | Max CFI Class Size | | | Reachable WASI Imports | | |
|---|---|---|---|---|---|---|---|---|---|
| | Before | After | Inflation | Before | After | Inflation | Before | After | Reduction |
| SQLite | 36.7 | 7.5 | 4.9× | 139 | 26 | 5.3× | 27 | 16 | 40.7% |
| Duktape | 29.1 | 2.5 | 11.6× | 186 | 5 | 37.2× | 13 | 11 | 15.4% |
| zlib | 2.2 | 2.0 | 1.1× | 5 | 4 | 1.25× | 14 | 14 | 0.0% |

*Before* = original binary; *After* = debloated. *Inflation* = Before / After. CFI classes over all `call_indirect` sites; WASI imports counted as transitively reachable via direct calls.

We classify WASI imports by sensitivity: *high* includes file-system operations (`path_open`, `fd_write`, `path_symlink`, `path_create_directory`), network operations (`sock_send`, `sock_recv`), and process control (`proc_exec`); *medium* includes environment access, clock queries, and random number generation. For each import, we compute whether it is transitively reachable from any live function, any dead function, or both. An import reachable *only* from dead code is eliminable by debloating.

In SQLite, debloating eliminates 11 of 27 reachable WASI imports (40.7%), including high-severity file-system APIs (`path_symlink`, `path_create_directory`, `fd_seek`) unused by the SQL workload but retained as pre-linked entry points to host capabilities. In Duktape, 2 of 13 imports are eliminable (15.4%). zlib shows no reduction, consistent with its tightly-used codebase. These dead-only imports represent latent capability paths: redirected control flow can transitively invoke host APIs the module never requires. Removing dead code eliminates both the intermediate functions and the transitive import reachability.

### 7.4 Security-Relevant Operations in Dead Code

Beyond control-flow targets and import paths, dead code contains individual instructions that constitute exploitation primitives. Unlike native binaries where ROP/JOP gadgets are arbitrary instruction subsequences, Wasm's structured control flow constrains code reuse to function-granularity redirection via `call_indirect`. Once an attacker redirects an indirect call to a dead function, the *entire body* executes, including all memory operations, calls, and side effects. The security concern is not individual gadgets but dead functions containing security-relevant operations that are reachable via the function table.

Table 10 quantifies these operations. In SQLite (`-O2`), dead code contains 10,955 memory store instructions (81.2% of all stores) and 920 `call_indirect` instructions (83.7%). Duktape harbors 4,631 dead memory writes. These operations—writes to linear memory, indirect calls that could chain further redirections, and direct calls to import wrappers—are unnecessary for the module's intended functionality but remain available if control flow is diverted.

**Table 10.** Security-Relevant Operations in Dead Code (`-O2` Binaries)

| Binary | Memory Writes | | `call_indirect` | | Direct Calls | |
|---|---|---|---|---|---|---|
| | Dead | % of Total | Dead | % of Total | Dead | % of Total |
| SQLite | 10,955 | 81.2% | 920 | 83.7% | 8,412 | 79.5% |
| Duktape | 4,631 | 76.8% | 512 | 89.1% | 5,203 | 78.2% |
| zlib | 198 | 42.1% | 0 | — | 312 | 41.8% |

Counts derived from `wasm-objdump` disassembly, partitioned into dead vs. live functions using Wizard `icount` coverage profiles. "Memory Writes" includes all `*.store*` variants; "Direct Calls" includes all `call` instructions (excluding `call_indirect`).

Debloating replaces dead function bodies with `unreachable` traps, eliminating these operations entirely. After debloating, any attempt to redirect an indirect call to a formerly dead function results in an immediate trap.

### 7.5 Discussion

These findings demonstrate that WebAssembly bloat is not merely a performance issue but a measurable security liability. The three dimensions interact: inflated CFI classes (Section 7.2) provide the *mechanism* for redirecting control flow; dead-only imports (Section 7.3) provide *capability paths* to host-level APIs; and dead operations (Section 7.4) provide *payloads*.

Across `-O2` binaries, debloating reduces indirect-call targets by up to 97.3% (Duktape: 186→5) and eliminates up to 40.7% of WASI imports (SQLite: 27→16). The contrast with zlib—minimal bloat, minimal security impact—suggests severity scales with application complexity and breadth of statically linked code. These results reinforce profile-guided debloating as a security hardening measure for WebAssembly deployments.

## 8 Related Work

*Software Bloat Studies.* Quach et al. [19] present a multi-OS study showing that only 20–30% of code executes under typical workloads, and that static and dynamic approaches offer complementary debloating opportunities. Other work examines bloat in Java and managed languages, focusing on object allocation and feature-based customization [10,18]. These studies establish the prevalence of bloat in native and managed settings but do not consider WebAssembly, where structured control flow and explicit tables change the shape of static analysis.

*WebAssembly Analyses.* Hilbig et al. [8] and Musch et al. [16] conduct empirical studies of real-world Wasm binaries, reporting that many originate from memory-unsafe languages and import powerful host APIs. Lehmann et al. [11] show that classic vulnerabilities such as buffer overflows remain exploitable in Wasm.

**Table 11.** Native vs. Wasm Bloat Comparison (-O2 Binaries)

| | Static Removal | | Dynamic Function Bloat | |
|---|---|---|---|---|
| **App** | **Native (LTO)** | **Wasm (S4)** | **Native** | **Wasm** |
| SQLite | 14.4% | 0% | 59.6% | 62.9% |
| zlib | 68.1% | 0% | 57.5% | 39.5% |
| Duktape | 30.5% | 0% | 55.8% | 74.6% |

Prior work on Wasm call-graph construction, slicing, and practical size-analysis tools highlights the difficulty of reasoning about indirect calls, table entries, and stack-preserving transformations [5,30,21]. Breitfelder et al. [2] introduce WasmA, a static analysis framework for WebAssembly binaries. Jangda et al. [9] study Wasm performance and usage patterns.

*Tools and Debloating Techniques.* Twiggy [5] is a code-size profiler that treats table entries as roots, limiting dead-code detection with indirect calls. Binaryen's wasm-opt and wasm-metadce implement DCE at S4 and cannot address workload-specific bloat in table-heavy modules [30]. Wasabi [12] and Wizard [26] provide dynamic analysis capabilities; this paper builds on Wizard and Walrus. Debloating techniques for native code include static feature-based removal, dynamic tracing, and profile-guided optimization [19,7]. Our work provides the measurement layer for applying similar ideas to WebAssembly.

### 8.1 Native Bloat vs. WebAssembly Bloat

To isolate Wasm-specific factors from general software bloat, we compile SQLite, zlib, and Duktape as native x86-64 binaries using the same source versions and optimization levels (-O0, -O2, -Oz) used for the Wasm evaluation. Native static dead code removal is measured using LTO with `gc-sections`; native dynamic bloat is measured using `llvm-cov` under the same workloads. Table 11 presents the side-by-side results at -O2.

*Dynamic Bloat is Comparable.* Dynamic function bloat ranges from 55–60% in native binaries and 39–75% in Wasm, confirming that code bloat is a general software property rather than a Wasm-specific artifact. The majority of compiled code remains unexecuted under realistic workloads regardless of the compilation target.

*Static Debloating is Structurally Harder for Wasm.* Native LTO removes 14.4% of functions in SQLite, 68.1% in zlib, and 30.5% in Duktape. In contrast, Wasm S4 removes 0% across all three applications. The fundamental reason is that Wasm's `call_indirect` mechanism and element tables create a dense set of potential call targets that conservative analyses cannot prune, whereas native LTO has whole-program visibility to determine which functions' addresses are never taken.

*Security Consequences are Amplified in Wasm.* Although dynamic bloat is comparable, the security impact of leaving dead code in place differs substantially. Native platforms deploy fine-grained CFI mechanisms including shadow stacks, pointer authentication (ARM PA), and per-call-site target whitelists [3]. These secondary defenses constrain exploitation even when dead code inflates the target set. Wasm provides type-signature checking on `call_indirect`; in the deployment model studied here, we do not assume additional runtime-enforced per-call-site target whitelists or hardware-assisted pointer protections. Additionally, native binaries link against shared libraries available process-wide; dead code does not meaningfully retain access to `libc`. In Wasm, host imports are explicitly declared per-module and reachable only through transitive call paths. Removing dead code can sever these paths entirely (Section 7.3), eliminating access to host capabilities the module never requires—a debloating benefit specific to the Wasm execution model.

*Measurement Methodology.* Native coverage is measured at source level using `llvm-cov` (source functions and lines), while Wasm coverage is measured at binary level (compiled functions and instructions). These are not directly comparable point-by-point—the compiler may inline, split, or merge functions differently for each target. However, both measure the same underlying phenomenon: what fraction of compiled code executes under a given workload. The comparison establishes that (1) dynamic bloat is prevalent in both settings, and (2) native LTO achieves significant static removal while Wasm S4 achieves none, demonstrating that static debloating is structurally harder for Wasm.

### 8.2 Wasm Dispatch Tables vs. Native Vtables and Function Pointers

Wasm's element tables, C++ vtables, and C function-pointer arrays are all indexed tables of function pointers resolved at runtime. Conservative static analysis faces challenges in all three cases. However, several structural differences make Wasm's case harder to analyze and more consequential for security.

*Class Hierarchy Constraints.* C++ vtables are organized by class hierarchy. Well-known devirtualization techniques—Class Hierarchy Analysis (CHA) and Rapid Type Analysis (RTA)—exploit inheritance structure to narrow indirect call targets to type-compatible subclasses. Wasm element tables have no such hierarchy: they are flat integer-indexed arrays where the only filtering available is coarse type-signature matching. A single signature such as `(i32, i32)` $\rightarrow$ `i32` may match hundreds of unrelated functions from different subsystems, all sharing the same table.

*Whole-Program Visibility and Host Mutability.* In native toolchains with LTO, the linker can determine whether a function's address is ever taken and remove it if not. Our experiments confirm this: native LTO removes 14–68% of functions across SQLite, zlib, and Duktape (Table 11). In Wasm, element tables can be mutated by the host environment at runtime via `table.set` and `table.grow`.

Any static analysis limited to the Wasm module alone cannot soundly account for these external mutations. More precise techniques such as field-sensitive pointer analysis (tracking individual table indices) and context-sensitive analysis (distinguishing call paths) face the same fundamental limitation—they cannot account for host-side table modifications without analyzing the host environment, which is generally unavailable for deployed modules.

*Security Impact.* Even when native vtable resolution is imprecise, secondary CFI mechanisms—shadow stacks, pointer authentication (ARM PA), per-call-site whitelists—limit exploitability. Wasm provides type-signature checking on `call_indirect`; in the deployment model studied here, we do not assume additional runtime-enforced per-call-site target whitelists or hardware-assisted pointer protections. Consequently, imprecision in table resolution has a more direct security impact in Wasm: each additional type-compatible function in the table is a viable redirection target with no further constraint. As shown in Section 7.2, dead code inflates CFI equivalence classes by up to 37×, increasing the number of valid redirection targets available under table-index corruption.

## 9 Conclusion

This paper presented a multi-level study of code bloat in WebAssembly binaries, combining staged static reachability analysis with dual-pipeline dynamic coverage measurement across Wasm-R3, real libraries, microbenchmarks, Linux-on-Wasm, and a large public corpus. The staged S1–S4 model shows that, under realistic exports+tables assumptions, conservative static analysis finds little provably dead code in table-heavy modules. In contrast, dynamic analysis using Wizard and Walrus shows that 70–85% of functions, basic blocks, and instructions remain unexecuted under realistic workloads.

Our security assessment shows that this bloat is a measurable attack-surface liability. Dead code inflates CFI equivalence classes by up to 37×, retains up to 40.7% of WASI imports exclusively through dead paths, and contains over 80% of memory-store and `call_indirect` operations. A comparison with native binaries confirms that dynamic bloat is comparable across targets, but native LTO removes 14–68% of functions while Wasm's conservative table model removes none.

These results highlight a fundamental gap between conservative static debloating and workload-dependent dynamic bloat in WebAssembly. Effective Wasm debloating will require profile-guided or semantics-aware techniques that safely remove dynamically unused code while preserving intended host interactions.

**Acknowledgment.** We thank the shepherd and anonymous reviewers for their feedback. This work is supported in part by the National Science Foundation award #2047205. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

# Appendix

### Cross-Engine Validation Data

Table 12 reports the per-benchmark deltas between the Wizard and Walrus dynamic coverage pipelines. The mean deltas of 0.13% (function) and 0.14% (instruction) confirm that the observed bloat measurements are not artifacts of a specific engine or instrumentation methodology.

**Table 12.** Delta of Dynamic Bloat Measurements (Wizard vs. Walrus)

| **Benchmark** | **Func $\Delta$ (%)** | **Instr $\Delta$ (%)** |
|---|---|---|
| boa | 0.01 | 0.00 |
| bullet | 0.02 | 0.01 |
| commanderkeen | 0.00 | 0.39 |
| factorial | 1.28 | 0.20 |
| ffmpeg | 0.00 | 0.00 |
| fib | 0.01 | 0.01 |
| figma-startpage | 0.08 | 0.04 |
| funky-kart | 0.07 | 0.44 |
| game-of-life | 0.00 | 0.38 |
| guiicons | 0.00 | 0.21 |
| hydro | 0.01 | 0.01 |
| jqkungfu | 0.00 | 0.00 |
| jsc | 0.01 | 0.17 |
| mandelbrot | 0.28 | 0.27 |
| multiplyDouble | 0.01 | 0.00 |
| multiplyInt | 0.01 | 0.01 |
| pacalc | 0.07 | 0.12 |
| parquet | 0.00 | 0.09 |
| pathfinding | 1.55 | 0.33 |
| rfxgen | 0.00 | 0.29 |
| rguilayout | 0.00 | 0.36 |
| rguistyler | 0.00 | 0.15 |
| riconpacker | 0.00 | 0.13 |
| rtexpacker | 0.00 | 0.00 |
| rtexviewer | 0.00 | 0.00 |
| sandspiel | 0.10 | 0.26 |
| sqlgui | 0.06 | 0.03 |
| **Mean $\Delta$** | **0.13** | **0.14** |
| **Max $\Delta$** | **1.55** | **0.44** |

# References

1. Baek, D., Getz, J., Sim, Y., Lehmann, D., Titzer, B.L., Ryu, S., Pradel, M.: Wasm-R3: Record-reduce-replay for realistic and standalone WebAssembly benchmarks.

Proceedings of the ACM on Programming Languages **8**(OOPSLA2), 2156–2182 (2024). https://doi.org/10.1145/3689787
2. Breitfelder, F., Roth, T., Baumgärtner, L., Mezini, M.: WasmA: A static WebAssembly analysis framework for everyone. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 753–757 (2023). https://doi.org/10.1109/SANER56733.2023.00085
3. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Computing Surveys **50**(1), 1–33 (2017). https://doi.org/10.1145/3054924
4. Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A., Stefan, D.: Position paper: Progressive memory safety for WebAssembly. In: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP). pp. 1–8 (2019). https://doi.org/10.1145/3337167.3337171
5. Fitzgerald, N., The Rustwasm Team: Twiggy: A code size profiler for WebAssembly. https://github.com/rustwasm/twiggy (2018), software tool
6. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 185–200 (2017). https://doi.org/10.1145/3062341.3062363
7. Heo, K., Lee, W., Pashakhanloo, P., Naik, M.: Effective program debloating via reinforcement learning. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 380–394 (2018). https://doi.org/10.1145/3243734.3243838
8. Hilbig, A., Lehmann, D., Pradel, M.: An empirical study of real-world WebAssembly binaries: Security, languages, use cases. In: Proceedings of the Web Conference 2021. pp. 2696–2708 (2021). https://doi.org/10.1145/3442381.3450138
9. Jangda, A., Powers, B., Berger, E.D., Guha, A.: Not so fast: Analyzing the performance of WebAssembly vs. native code. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 107–120 (2019), https://www.usenix.org/conference/atc19/presentation/jangda
10. Jiang, Y., Wu, D., Liu, P.: JRed: Program customization and bloatware mitigation based on static analysis. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC). pp. 12–21 (2016). https://doi.org/10.1109/COMPSAC.2016.146
11. Lehmann, D., Kinder, J., Pradel, M.: Everything old is new again: Binary security of WebAssembly. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 217–234 (2020), https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann
12. Lehmann, D., Pradel, M.: Wasabi: A framework for dynamically analyzing WebAssembly. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 1045–1058 (2019). https://doi.org/10.1145/3297858.3304068
13. Matsakis, N.D., Klock, F.S.: The rust language. ACM SIGAda Ada Letters **34**(3), 103–104 (2014). https://doi.org/10.1145/2692956.2663188
14. Mendki, P.: Evaluating WebAssembly enabled serverless approach for edge computing. In: 2020 IEEE Cloud Summit. pp. 161–166 (2020). https://doi.org/10.1109/IEEECloudSummit48914.2020.00031
15. Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: Twine: An embedded trusted runtime for WebAssembly. In: 2021 IEEE 37th International Conference on Data

Engineering (ICDE). pp. 205–216 (2021). https://doi.org/10.1109/ICDE51399.2021.00025
16. Musch, M., Wressnegger, C., Johns, M., Rieck, K.: New kid on the web: A study on the prevalence of WebAssembly in the wild. In: Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). pp. 23–42 (2019). https://doi.org/10.1007/978-3-030-22038-9_2
17. NIST National Vulnerability Database: CVE-2024-2887: Type confusion in Google Chrome WebAssembly. https://nvd.nist.gov/vuln/detail/CVE-2024-2887 (2024), accessed: 2026-05-27
18. Qian, C., Hu, H., Alharthi, M., Chung, P.H., Kim, T., Lee, W.: Razor: A framework for post-deployment software debloating. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1733–1750 (2019), https://www.usenix.org/conference/usenixsecurity19/presentation/qian
19. Quach, A., Prakash, A., Yan, L.: Debloating software through piece-wise compilation and loading. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 869–886 (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/quach
20. Severin, J.: Linux on WebAssembly: Running the linux kernel in the browser. https://github.com/joelseverin/linux-wasm (2024), software repository
21. Stiévenart, Q., Binkley, D.W., De Roover, C.: Static stack-preserving intra-procedural slicing of WebAssembly binaries. In: Proceedings of the 44th International Conference on Software Engineering (ICSE). pp. 2031–2042 (2022). https://doi.org/10.1145/3510003.3510070
22. The WASI Subgroup: WASI: The WebAssembly system interface. https://wasi.dev (2019)
23. The wasm-bindgen Project Developers: Walrus: A WebAssembly transformation library. https://github.com/wasm-bindgen/walrus (2019), software tool
24. The WebAssembly Community: WABT: The WebAssembly binary toolkit. https://github.com/WebAssembly/wabt (2016), software tool
25. The WebAssembly Community: WASI SDK: WASI-enabled WebAssembly c/c++ toolchain. https://github.com/WebAssembly/wasi-sdk (2019), software tool
26. Titzer, B.L.: The wizard research engine. https://github.com/titzer/wizard-engine (2022), software repository
27. Watt, C.: Mechanising and verifying the WebAssembly specification. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP). pp. 53–65 (2018). https://doi.org/10.1145/3167082
28. World Wide Web Consortium: WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/ (2019), w3C Recommendation
29. Zakai, A., Emscripten Contributors: Emscripten: A compiler toolchain for the web platform. https://emscripten.org (2011), software tool
30. Zakai, A., The Binaryen Contributors: Binaryen: Compiler infrastructure and toolchain library for WebAssembly. https://github.com/WebAssembly/binaryen (2016), software tool