WebAssembly as a Fuzzing Compilation Target (Registered Report)

Florian Bauckholt
CISPA
Saarbrücken, Germany
florian.bauckholt@cispa.de

Thorsten Holz
CISPA
Saarbrücken, Germany
holz@cispa.de

Abstract

By monitoring the execution of the program under test, fuzzers can gather feedback on how different inputs affect the program's behavior and detect crashes and other abnormal behaviors. To achieve these objectives, fuzzers typically rely on a *static* instrumentation phase, which can be cumbersome to extend and experiment with. In this paper, we explore a different strategy: By compiling to a *common high-level compilation target*, we can retain most of the instrumentation opportunities with the potential for *dynamic* instrumentation. Compiling to an intermediate target disentangles instrumentation from the harness build process and produces fuzzer-independent harness artifacts. More specifically, we propose to use WebAssembly (WASM) as a suitable target due to its widespread language support, deterministic and isolated nature, and simple and easy-to-JIT instruction set. To explore this approach, we present and discuss WasmFuzz, a fuzzer for WebAssembly binaries that bridges the gap between native and WASM fuzzing. To enable meaningful WebAssembly fuzzer comparisons, we demonstrate a generic way to retrofit WASM modules into source-based fuzzers through `wasm2c`. This approach already raises the performance baseline of WebAssembly fuzzing significantly. In our preliminary evaluation, WasmFuzz achieves, on average, more basic blocks per target compared to other WebAssembly fuzzers and seems competitive with native setups like `cargo-fuzz` (LibFuzzer).

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

WebAssembly, Fuzzing, Instrumentation, Dataset

ACM Reference Format:
Florian Bauckholt and Thorsten Holz. 2024. WebAssembly as a Fuzzing Compilation Target (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24), September 16, 2024, Vienna, Austria*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3678722.3685531

FUZZING '24, September 16, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1112-1/24/09
https://doi.org/10.1145/3678722.3685531

1 Introduction

Fuzzing is a very active research field in software security that develops methods to automatically discover "interesting" inputs for target programs to find or trigger security vulnerabilities [6, 8, 9, 18, 49]. Feedback-guided fuzzing, in which inputs are mutated from a corpus of entries that are considered "novel" through a feedback mechanism, requires two fundamental components: *input generation* and *instrumentation*. Input generation is difficult as it aims to create meaningful and valid inputs for arbitrary, potentially complex target programs under test. However, input generation is typically easy to experiment with, e.g., the Fuzzing Book [50] sketches various approaches in a few lines of Python code.

In contrast, while instrumentation is conceptually simple, workflows that instrument real-world targets can introduce friction: Instrumentation implementations usually rely on large and complicated libraries and interact with target build systems. Trying to evaluate a new feedback technique can result in per-target work, and most implementations are tied to a single compiler toolchain (e.g., LLVM at a specific version), so they are not easily usable with targets in multiple programming languages.

Generally speaking, existing fuzzing methods are based on three instrumentation paradigms:

Compiler-based instrumentation: During compilation, the compiler can insert code gadgets into the control flow graph to track coverage via fuzzer callbacks or bitmaps shared between the fuzzer and the harness program [12, 18, 49]. Compiler-based instrumentation is usually implemented as an LLVM [25] pass. Source-based approach benefits from mature sanitizers like AddressSanitizer and UBSan that can act as bug oracles [13, 42].

Binary rewriting: After compilation or for executable binaries, this approach recovers the control flow graph by statically analyzing and inserting coverage gadgets into the disassembled binary [15, 34]. This process is difficult to accomplish as recovering and rewriting binaries is, at least in the case of Intel x86 assembly, inherently a heuristics-based workflow. Recent work solves this problem to a satisfactory degree for binary executables and aims to reduce the heuristics required [16, 40].

Dynamic Binary Instrumentation (DBI): Instead of rewriting binaries via static analysis, DBI combines emulation with just-in-time compilation [18, 29, 38]. Tools like QEMU's Tiny Code Generator (TCG) lift assembly code blocks as they are executed and re-emit augmented versions of the code with a special-purpose just-in-time (JIT) instrumentation. DBI is used both for instrumentation as well as for binary translation: Emulators like QEMU [7] can lift code from one platform's assembly to the host's architecture, thus gaining significant performance advantages over naive emulation.

While these instrumentation paradigms work well in practice, each approach comes with several implementation and usability challenges. When experimenting with fuzzer architecture designs, we have to either:

(1) deal with a custom LLVM compiler pass: LLVM [25] is a complex and moving target, and available instrumentation implementations frequently require old versions of LLVM. The compile-time instrumentation approach leads to fuzzers that are tightly intertwined with project build systems. For many projects, this means that swapping out the instrumentation requires target-specific build system changes.

(2) deal with the complexities of real ISAs: Lifting assembly to control flow graphs and re-emitting code is highly complex and inherently lossy. Much of the information of interest for instrumentation (e.g., instruction boundaries, control flow graphs, functions, data types, etc.) has to be inferred, so we cannot reliably experiment with rich instrumentation.

(3) rely on a DBI implementation and make-do with the reduced instrumentation fidelity due to low-level assembly code: In practice, this means depending on Intel's Pin Tool. Pin is a complex C++ codebase that is deeply intermingled with glibc and kernel implementation details, leading to unfortunate compatibility issues [4]. Some projects patch QEMU instead and apply instrumentation in QEMU's TCG IR. The `libTCG` patchset [17] tried to expose TCG as a library and address the need for forking QEMU, but unfortunately, it was never maintained or merged upstream.

These caveats mean that interesting and promising instrumentation ideas (symbolic execution tools, program slicers, new feedback types) [20, 28, 37, 48] are rarely seen in FuzzBench [32]—the leading fuzzing competition benchmark—due to complex and incompatible setups. OSS-Fuzz [1], one of the largest and most effective fuzzing efforts, also suffers from this issue: Every project can specify an optional list of compatible fuzzing engines and can thus opt out of the default supported list. As of the time of writing, OSS-Fuzz supports four fuzzers, but only 157 out of 1259 projects run on all engines (the default). In fact, 751 out of 1259 projects only list one engine. This could be due to compatibility issues, deliberate decisions, or misunderstandings related to project configuration.

1.1 WebAssembly as a Compilation Target

In this paper, we aim to improve this situation by proposing an alternative workflow for harness instrumentation. As a middle-ground between source-based (e.g., through LLVM IR compiler passes) and binary-only (e.g., Intel x86 assembly rewriting) instrumentation, we observe in our experiments that WebAssembly retains most of the semantics present in the intermediate representation (IR) used by compilers while providing a stable and well-designed instruction set. We propose compiling fuzzing harnesses to WebAssembly modules and applying instrumentation passes to the WASM module through dynamic binary instrumentation or static rewriting. For C, C++, and Rust projects, LLVM's `wasm32-wasip1` target provides the necessary support for harness compilation.

We find that WebAssembly's design goals for in-browser usage align with fuzzing harnesses in several ways:

Simplicity: The WebAssembly 1.0 release was designed to be easy to reason about and easy to JIT. The WASM ISA is not a minimalist design, but the instruction set is complete, well-structured, and much simpler to implement than other high-level targets like LLVM IR.

Isolation: WebAssembly does not provide a standard library. Any interaction with the system has to be routed through explicitly defined external API functions. This is beneficial for fuzzing, as harnesses that interact with or rely on system behaviour can be hazardous (e.g., delete files or make network requests) or interfere with the fuzzing process (e.g., external state that results in execution instability or calls out to code that the fuzzer is not aware of and does not gather coverage feedback from). With WebAssembly modules, we can be sure that we have explicitly captured and handled all system interactions.

Determinism: WebAssembly modules are fully deterministic, as there is no support for threading (in the base ISA) or other sources of randomness. We do not need to take extra care in order to achieve full execution stability in a snapshotting fuzzer design.

No arbitrary control flow: In contrast to common ISAs, there is no "jump to address" instruction in WebAssembly. This decision allows browsers to implement single-pass compilers without complex control flow handling. Jumps in WebAssembly either target a static control flow block, a static function, or one of a specified set of functions. These restrictions mean that control flow graphs are accurate and available statically, which is essential for directed fuzzing.

Stability and portability: WebAssembly was designed for in-browser use, where the underlying engine needs to be backward compatible and run on any platform. For fuzzing harnesses, this means that we can distribute harness modules for benchmarking suites without the need to re-compile each project for fuzzing campaigns. This reduces the risk of accidentally invalidating benchmark results or comparisons due to slight but significant differences in code generation decisions, for example.

WebAssembly is gaining popularity as a compilation target and is supported by many systems-type languages (e.g., C, C++, Rust, Zig). With LLVM's `wasm32-wasip1` target, which targets WebAssembly 1.0 with the WebAssembly System Interface environment [46], we can compile many programs without modification, even if they interact with the system via files, stdio, and other UNIX APIs. For fuzzing, it generally suffices to handle `stdout` and `getrandom`, as other interactions indicate improper harness behaviour.

1.2 Evaluating a WASM-based Fuzzer Architecture

To support our design proposal, we implement WasmFuzz, a proof-of-concept fuzzer for WebAssembly harnesses (see Figure 1 for a high-level overview). WasmFuzz is a snapshot fuzzer for `LLVMFuzzerTestOneInput`-style harnesses and implements several instrumentation passes in a custom JIT. We pose two main research questions to assess the feasibility of this approach and discuss our PoC in this regard:

RQ-1 *Is the WebAssembly ISA a good fit for instrumentation?* Does WebAssembly retain enough high-level information to implement standard instrumentation passes? Can WASM-based fuzzing campaigns be competitive in fuzzing efficacy compared to native fuzzers?

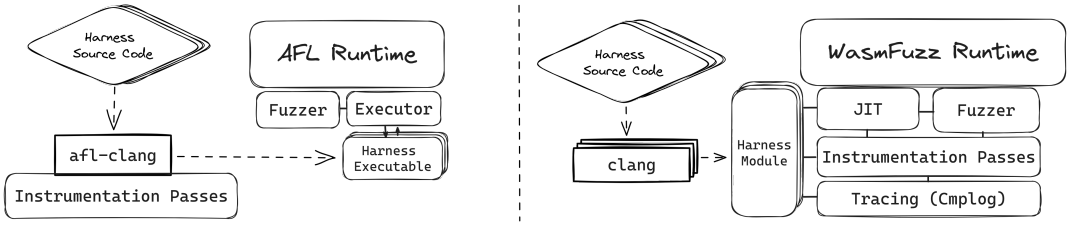


Figure 1: Architecture overview: AFL's fork server (left) compared to WasmFuzz (right)

RQ-2 *Is compatibility an issue?*
Can we fuzz established (i.e., FuzzBench) projects as WASM modules? What kind of modifications are required to compile project harnesses to WebAssembly, and what types of projects are compatible with the approach?

In a preliminary evaluation, we compare the performance of multiple WASM fuzzers with some Rust fuzzing harnesses and also compare the fuzzing performance between existing native fuzzers and WasmFuzz. The initial results are promising, and for the full version of this work, we plan to conduct code coverage experiments on a comprehensive set of harnesses and evaluate the results using statistically sound methods [24, 39].

1.3 Contributions

In summary, we make the following contributions:

- We explore the concept of instrumenting on-demand by compiling the program under test to WebAssembly. To this end, we introduce WasmFuzz, a fuzzer for WebAssembly modules that aims to compete with native fuzzers and provides a framework for further instrumentation experiments.
- In a preliminary evaluation, we present initial results to answer the outlined research questions. For the full version of this work, we plan to thoroughly evaluate the concept.
- We create a suite of WASM fuzzer harnesses in Rust, C, and C++ as a dataset for WASM fuzzer evaluations.

To foster research, we release our prototype, the harness suite, and all other artifact data at https://github.com/CISPA-SysSec/wasmfuzz.

2 Approach

To evaluate the effectiveness of WASM modules as fuzzing harnesses, we design and implement WasmFuzz, a JIT-based WebAssembly fuzzer. We begin with a discussion of the drawbacks of instrumentation outside the compiler toolchain and then highlight relevant design decisions for our implementation.

2.1 WebAssembly as Instrumentation Target

Instrumentation approaches vary in many aspects. We are primarily interested in the quality of the instrumentation, as this influences the effectiveness of the fuzzing process, but we also consider practical aspects such as ease of implementation and portability between different fuzzing harnesses. In the following, we discuss these aspects with respect to compiler-assisted instrumentation, binary rewriters, and dynamic binary instrumentation.

Instrumentation Quality. The compiler-assisted instrumentation in the style of LibFuzzer has several advantages in terms of instrumentation quality and opportunity. Many fuzzer feedbacks require high-level semantics as part of their instrumentation, and some of this information gets lost when looking at compiled artifacts.

Static rewriters, like RetroWrite [15] for Intel x86 binaries, need to first recover control flow and instruction boundary information before they can insert instrumentation snippets. This process relies on heuristics, requires complex modeling, and is undecidable for x86 in general [36]. Recent works aim to reduce the required information for correct patches and instrumentation, but precise instruction location information is inherently necessary [16]. While *undecidability* is not a concern in practice (after all, we are dealing with compiler-generated assembly that does not play any tricks with arbitrary jumps or self-modifying code), relying on heuristics still makes static rewriters brittle and complicated. WebAssembly, by design, retains most of this information. Since all jump targets and instruction boundaries are provided by the module structure, building a static rewriter for WASM modules does not require any heuristics. Dynamic binary instrumentation can also avoid the heuristics required for static rewriters since control flow is always handled in the DBI engine.

Retrofitting a method such as Address Sanitizer (ASAN) is challenging for both native and WASM binaries since much of the datatype-level information is lost when dealing with the assembly level. For example, a useful sanitizer implementation should be able to recognize accesses to uninitialized memory. This does not usually manifest as a crash and cannot be detected by a sanitized heap implementation. However, for commonly used feedback types (such as those supported by AFL++), we do not miss any instrumentation opportunities when we compile to WASM modules. The implementation of edge coverage, for example, is straightforward due to the control flow design of WebAssembly.

Implementation Complexity. Dynamic binary instrumentation is a powerful technique that eliminates the heuristics of static rewriting. However, it is difficult to implement in practice as it requires a complete model of the target machine instructions. This means that most fuzzing tools that employ dynamic binary instrumentation rely on Intel's Pin Tool [29], a DBI implementation for Intel x86.

Compiler-based instrumentation passes are usually implemented as an LLVM pass or compiler plugin. While these implementations are not inherently complicated in the case of LLVM and Intel Pin, relying on a complex framework potentially introduces more friction. LLVM-based fuzzing instrumentation passes often run into problems when trying to maintain compatibility with multiple LLVM

versions, and Intel Pin-based tools suffer from complicated system requirements (e.g., specific Linux kernel releases, glibc versions). Note that FuzzBench [32] does not currently include any concolic execution tool and previously removed QSYM due to its dependency on Pin [4].

Instrumenting WASM vs. LLVM IR.. LLVM IR is characterized by a large number of instrumentation options but also poses a few operational challenges. Replacing the compiler is not trivial (Rust ships with a modified LLVM version, for example), so a certain amount of effort is required when making a fuzzing harness compatible with an LLVM-IR-based fuzzer. This is mainly related to LLVM versions, but LLVM's IR changes frequently. For example, LLVM replaced pointer types with a single *opaque* pointer type in release 17. As a result, fuzzers usually lag behind in support and require different compiler versions for different fuzzers.

The high-level semantics of LLVM's IR are useful for instrumentation but make the instruction set quite complex. In practice, this means that fuzzers typically choose to ignore some of these high-level details, resulting in missed instrumentation. We encountered one such issue related to SanitizerCoverage's handling of the `uint128_t` type while investigating coverage differences between fuzzers. While unimplemented IR instructions slightly degrade fuzzer performance in some cases, they pose correctness issues for taint engines and concolic executors. Modeling the entire LLVM IR instruction set is not feasible for solver-based fuzzers, so they usually ignore operations like SIMD data processing or floating point operations.

Some LLVM IR-based tools use whole-program LLVM bitcode workflows to decouple instrumentation from the build process. These workflows (i.e., `wLLVM` and `gLLVM`) store and collect all frontend-generated LLVM IR and produce a single LLVM IR bundle. This allows fuzzing workflows that require dynamic instrumentation or modification of the target program in some way and are usually found in mutant generators and program slicers [21, 44].

Harness-Tool Compatibility and Tool Stability. WebAssembly is emerging as a compilation target for several system-like programming languages. With the WasmGC extension, this also includes languages that were previously out of reach for LLVM-based instrumentation, such as Go, TypeScript, .NET, and Haskell.

The small instruction set of WebAssembly and the self-contained modules means that there are fewer opportunities for incompatibilities due to uncommon harness features. This type of setup is ideal for fully automated fuzzer solutions: The Cyber Grand Challenge [3], an entirely hands-off fuzzing competition, provided a small custom system interface set with a handful of specified system calls to enable deterministic and holistic tools. With WASM modules, we get deterministic and holistic harnesses at zero cost. The relevant system call interface for fuzzing harnesses is very small (mainly I/O for debug and error prints), and WASM modules are fully self-contained. The WebAssembly 1.0 ISA is designed to be deterministic regardless of the host platform and contains no sources of randomness like data races or built-in timers.

2.2 Build Once, Instrument On-demand

In our WebAssembly-based fuzzing workflow, the harness build process is separate and independent from the fuzzing tool, i.e., the fuzzer operates only on a WASM module and cannot interact with the build system. In contrast, a separate "build" and "run" stage is standard in fuzzing campaigns, and existing tools do not try to interact with build systems at run time. Instead, tools like ensemble fuzzers (EnFuzz [10]) usually use multiple binary executables that have previously been instrumented with several instrumentation and sanitizer configurations. In the following, we elaborate on the design objectives of our approach.

Works With Most Fuzzing Workflows. We want to explore whether WebAssembly modules contain enough information to implement standard instrumentation passes so that we can perform typical fuzzing workflows. As part of the evaluation, we investigate whether our on-demand instrumentation on an architecture like WASM can compete with instrumentation in the build system.

To facilitate this, we extend `cargo-fuzz` to provide drop-in compatibility with WASM harnesses. `cargo-fuzz` is a framework for Rust fuzzing harnesses and is integrated with LibFuzzer. With our changes, switching from LibFuzzer to WasmFuzz works by simply building the harness with `--target=wasm32-wasip1` and does not require any project-specific changes for targets that do not depend on specific platform APIs.

Simplifies Fuzzer Benchmarking. In the current fuzzer benchmarking ecosystem, each fuzzer works with a custom build of the target. While the compiler flags are usually fixed, the resulting binaries can differ both in the structure of the control flow graph and in behavior due to optimization decisions related to inserted instrument snippets and different compiler environments or versions.

Implementations that apply coverage instrumentation at a later stage (like x86 assembly for RetroWrite [15] or Nyx [41]) may have a different "view" of coverage and, therefore, perform worse when evaluated with a different coverage metric of the fuzzer. Although the effect of this discrepancy may be small, it can still be useful to be able to communicate exact *code locations* between two fuzzers. The sandbox of WebAssembly is also helpful here: Since the system interaction is minimal, fuzz executions should be deterministic for fuzzers that implement a reset mechanism. With deterministic execution, no inconsistencies can occur in the reproduction of bugs and coverage metrics.

The specification of WebAssembly 1.0 is a ratified standard, so we can use the artifacts of the fuzzing harness module for future benchmarks as well. Reproducing today's project versions may no longer be trivial in a few years, and current versions may be incompatible with years-old toolchains. As a result, our approach enables better reproducibility of the experimental results.

Enables New Workflows. On-demand instrumentation enables fuzzers to change their feedback configuration at runtime. While we do not investigate instrumentation scheduling in this paper, we envision that applying a multi-armed bandit reward model for fuzzer feedback might yield similar improvements as the scheduling approaches for mutators (e.g., MOpt [30]).

```c
/*
clang -O1 -g example.c -o example.wasm \
      --target=wasm32-wasi -mexec-model=reactor \
      -Wl,--export=malloc,--export=LLVMFuzzerTestOneInput
*/
#include <stdlib.h>
#include <stdint.h>
void LLVMFuzzerTestOneInput(char* data, size_t size) {
    if (size != 4)
        return;
    if (*(uint32_t*)data == 0xdeadbeef)
        __builtin_trap();
}
```

Figure 2: Example harness setup for WasmFuzz

A stable and self-contained harness module format also opens up new practical workflows. For example, we can create a project harness and send the resulting WASM module to a collection of fuzzers or keep multiple versions of a harness and compare execution traces. These ideas are theoretically all possible with existing fuzzer architectures, but WASM could allow us to do this with a single, lightweight module format.

2.3 WasmFuzz Prototype

To validate our proposed approach, we design and evaluate a fuzzer for WebAssembly harnesses. We implement several feedback types (e.g., basic block, edge, and value coverage, etc.) as customizable passes in a custom JIT compiler. Our prototype tool called WasmFuzz is implemented in ~15k lines of Rust. WasmFuzz requires a Linux host and supports the `x86_64` and `aarch64` platforms. The Rust ecosystem has mature libraries (so-called *crates*) for our use case. For example, the `libafl` crate by Fioraldi et al. [19] provides fuzzer components for input mutation and corpus scheduling. Moreover, `cranelift` provides a compiler toolchain similar to LLVM. The intermediate representation of Cranelift fits well with our use case, as the project was originally developed as a code generation backend for a WebAssembly JIT [14].

In the following, we illustrate implementation and design decisions based on an example. WebAssembly modules can define a function interface via *exports*. For simplicity and compatibility, we stick to LibFuzzer's `LLVMFuzzerTestOneInput` API. Figure 2 shows an example C harness, including the command required to compile it down to a compatible WASM module. Notably, we do not require any code changes for existing LibFuzzer harnesses.

To compile a harness for use in WasmFuzz, we instruct LLVM as follows:

- `--target=wasm32-wasi`: Target WebAssembly with the WebAssembly System Interface platform. While our example harness does not use any POSIX APIs, most real-world harnesses depend on some of them.
- `-mexec-model=reactor`: The execution model determines details related to module initialization and expected lifecycle. The `reactor` model allows us to call into exported functions repeatedly.
- `--Wl,--export=LLVMFuzzerTestOneInput` + `malloc`: Export the harness entry point. This allows us to identify the target function in the fuzzer, even when running on modules without debug information. We require an additional export called `malloc` in order to pass data into the harness function. WebAssembly modules operate on a linear address space, so we cannot just pass in a pointer to our data in the host address space. Explicitly allocating space in the target's linear memory is an established solution for these kinds of WebAssembly interfaces.

Table 1: Fuzzer versions used in our experiments. Native fuzzers appear in combination with the `wasm2c` strategy described in Section 3.1.

Fuzzer	Type	Version
LibFuzzer	Native*	LLVM 16
libafl_libfuzzer	Native*	035c01b4
AFL++	Native*	4.08c
WAFL	WASM	a95a2bf
WasmFuzz	WASM	fuzzing24-prelim

3 Preliminary Evaluation and Plans

To answer the research questions outlined in the introduction, we conduct two experiments as a litmus test for the evaluation of our prototype implementation: How does our fuzzer compare to other WebAssembly fuzzing tools, and can we compare the code coverage results to native fuzzers?

Since there are few existing WASM module fuzzers, we present a straightforward approach to run native fuzzers on WebAssembly harnesses (see Section 3.1) and include them in our experiments. All our experiments were run on an AMD EPYC 9654 with two sockets and 768 GB RAM. We follow the recommendations of Klees et al. [24] and Schloegel et al. [39] during our experiments. We plan to publicly release our prototype, the harness suite, and all artifact data at https://github.com/CISPA-SysSec/wasmfuzz.

3.1 Fuzzers

We compare WasmFuzz to the four other fuzzers shown in Table 1. We include AFL++ [18] and LibFuzzer [12] as baselines for native fuzzers. As we build on LibAFL's mutators, we also include libafl_libfuzzer [11] in the evaluation for a more direct comparison. In addition, WAFL [22] is our basis for WASM module fuzzing. We would have liked to include FuzzM [27] in this evaluation. Unfortunately, however, we encountered a compatibility problem: The self-implemented WASM module parser of FuzzM does not seem to support the WebAssembly 1.0 specification properly and fails when parsing our harness modules.

Additional WASM fuzzer baselines via `wasm2c`. The `wasm2c` tool offers an interesting comparison option here, as it enables the translation of WASM modules into C source code, which we can then pass through the standard fuzzing setup for source-level fuzzers. This is a general and convenient way to retrofit WASM module support for existing fuzzers, as shown in Figure 3.

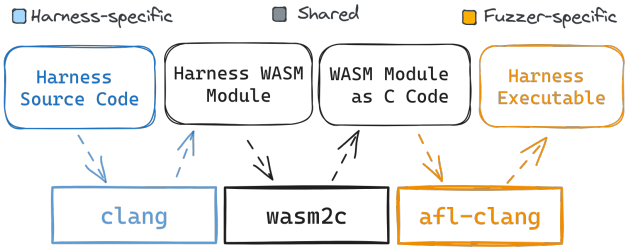


Figure 3: A generic approach to WASM module fuzzing.

The approach works as follows:

- Take (or produce) a WebAssembly module.
- Run `wasm2c` to obtain a WASM module as C source code.
- Add a thin wrapper for system interaction (stub WASI API).
- Re-compile and instrument with a source-level fuzzer.

Notably, this approach addresses the project × fuzzer compatibility issue: The intermediate C module produced from WebAssembly via `wasm2c` is fully self-contained and can be compiled with any C99-compatible instrumenting compiler, even if the WASM module was produced from other programming languages.

RLBox [35] follows a similar approach for sandboxing potentially insecure libraries. The authors evaluate the performance overhead of their approach and find an increase in wallclock time of about 25 % for image decoding benchmarks. This is consistent with our preliminary experience, although the performance impact varies widely, and the `wasm2c`-based version sometimes even outperforms the native version. We expect further performance differences in fuzzing to result from lost instrumentation capabilities and plan to investigate this further in the full paper.

We apply this compatibility hack to LibFuzzer, AFL++, and libafl_libfuzzer in order to establish a consistent performance baseline. Each of these configurations performs, on average, better than WAFL in our preliminary experiments, which significantly improves the starting point for WASM module-based fuzzers.

3.2 Targets

Currently, there is no dataset of fuzzing harnesses compatible with WebAssembly. While tools like FuzzBench [32] are nice for standardized fuzzer comparisons, the build systems of FuzzBench targets are quite heterogeneous (C/C++, plain Makefiles, autotools, CMake, etc.). This implies that compiling FuzzBench harnesses to WebAssembly requires manual effort for each project. Rust projects, on the other hand, exist in a more homogenous build environment: Each project's fuzzers are based on a similar structure and are built with Rust's build system `cargo`. This allows us to build many Rust projects as-is and experiment with many targets without additional work. We follow two approaches to build the targets under test:

C/C++ Harnesses. We randomly select targets from the FuzzBench [32] and Magma [23] harness suites to test the feasibility of compiling common fuzzing harnesses to WebAssembly. Most of them require some customization to the build setup as well as additional build flags. Some targets require explicit source code patches, e.g., to disable interaction with the platform or to add support for the WASI platform. We have encountered an issue when compiling `libpng` to WebAssembly: `libpng` implements error handling using the `setjmp` and `longjmp` mechanisms to handle exceptions. WebAssembly 1.0 does not support exception handling, and `setjmp` cannot be emulated without it.

For this experiment, we build harnesses for ten different projects. This requires an average of 32 lines of build scripts (11 to 64) and 35 lines of `.patch` files (0 to 212) per project.

Rust Harnesses. The Rust fuzzing harnesses in `oss-fuzz` are based on `cargo-fuzz`, which is recommended in the Rust Fuzz Book [2]. As mentioned in Section 2.2, adding WASM-harness support to `cargo-fuzz` allows us to build and evaluate a large set of Rust programs without any target-specific modification. Our setup is based on a small patch to the `libfuzzer-sys` crate and builds targets with `cargo build --target wasm32-wasip1`. We have collected 58 projects with fuzzing harnesses from `oss-fuzz` and the "100 most downloaded crates list". Of these, 49 (84 %) were compatible out of the box with our approach.

Notably, while the C/C++ harnesses in FuzzBench are generally large targets, many Rust projects include tightly-scoped fuzzers that function more like unit tests than integration tests. These harnesses are generally not as challenging to explore and saturate quickly, so they are not fully representative of the usual fuzzing harnesses used in fuzzer benchmarking.

3.3 Experiment 1: Different WASM Fuzzers

In this experiment, we compare the performance of multiple WASM fuzzers (see Table 1) with some Rust fuzzing harnesses. We run each fuzzer for 8 hours in 10 trials to account for the statistical nature of fuzzing. The `wasm2c` variants are described in Section 3.1. Alongside each fuzzer, we run an instance of WasmFuzz that collects coverage statistics on the saved inputs. Since each fuzzer runs on the same WASM module, evaluating the code coverage in our implementation does not bias the results. Note that this assumes that each fuzzer implementation is sensitive to edge coverage, which is the case for the fuzzers we evaluated.

The results are depicted in Figure 4, demonstrating that our prototype tool, WasmFuzz, can access code that other WASM fuzzers cannot reach. We also see that our additional `wasm2c`-based baselines beat WAFL as the previous WASM-fuzzer baseline. `fuzz_target_qcms.wasm` stands out here because WAFL has an edge coverage of almost zero during the entire run. This shows the main shortcoming of WAFL: The implementation does not support comparison logging and is, therefore, unable to solve some of the crucial initial edges of the harnesses. All other fuzzers tested use a form of input-to-state mutation in the style of RedQueen [6].

We show the `qcms` target specifically as it includes a *coverage tarpit* which can be solved by additional instrumentation: The parser covered by this harness includes a coding pattern where errors are flagged during parsing and the flag is checked afterward at a single location. This means that simple edge coverage feedback is not effective at solving these parsing errors since all error paths need to be *avoided* for a successful parse.

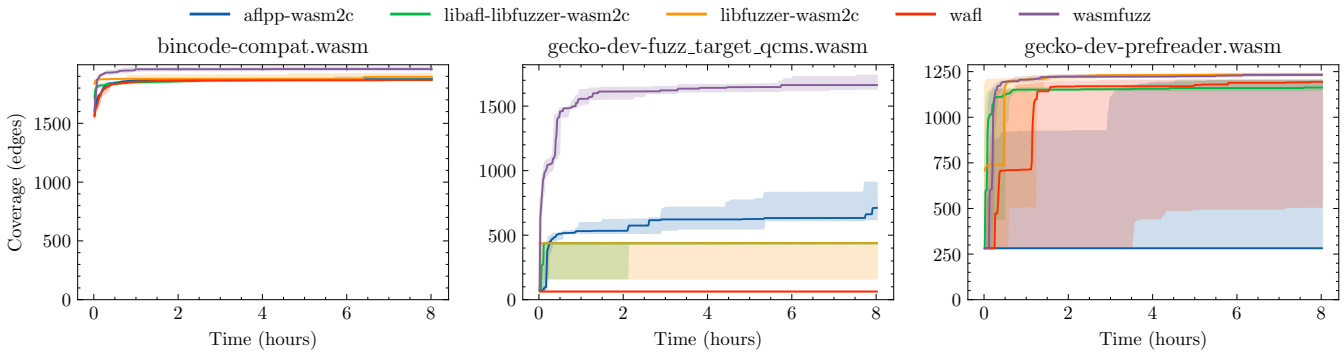


Figure 4: Coverage over time for WASM-based fuzzers. We show the median and interquartile range of 10 trials over 8 hours.

3.4 Experiment 2: Does WebAssembly Coverage Translate to Native Coverage?

To support our proposal to use WebAssembly as a fuzzing target, we compare the fuzzing performance between existing native fuzzers and WasmFuzz. Our Rust fuzzing harnesses are based on the `cargo-fuzz` ecosystem, which uses LibFuzzer under the hood. With our patches, we can compile these harnesses in both WASM and `x86_64` and evaluate both versions at the same time.

For this experiment, we present two Rust harnesses and perform ten trials of eight hours each with WasmFuzz and LibFuzzer. We would also like to compare with AFL++, but `cargo-fuzz`, unfortunately, only supports LibFuzzer. In our previous experiment, we worked around this compatibility issue with the "wasm2c" strategy described in Section 3.1, but supporting AFL++ for native Rust targets requires additional compiler passes that are not available in Rust's custom LLVM version. Since we are comparing fuzzing runs on different target platforms (`x86_64` and WASM), we need to make sure we have a valid comparison. Showing the coverage reported by WasmFuzz would favor WASM-based fuzzers, so we collect the coverage for both the WASM module and LibFuzzer binary for each trial by replaying the fuzzer's corpus on both platforms. This strategy allows us to compare WASM coverage as a proxy measure for native coverage.

Figure 5 shows two examples from this experiment: In the first example, the WasmFuzz-reported coverage curves closely match the curves reported by LibFuzzer (the shape of the coverage matches, although the two Y-axes are not *inherently* comparable). The second example shows that both coverage metrics can be incompatible when used for evaluation: Relying on WASM-based code coverage would rank WasmFuzz first, while LibFuzzer's metric has LibFuzzer in first place.

In our small, preliminary experiment, we found that there is no clear winner between the native LibFuzzer workflow and our WasmFuzz tool, even when only the coverage metric of LibFuzzer is evaluated. This suggests that the proposed approach may be worthwhile as a complement to run alongside native fuzzers.

3.5 Planned Evaluation

In the introduction, we outlined the following research questions: (RQ-1) Is the WebAssembly ISA a good fit for instrumentation? (RQ-2) Is compatibility an issue?

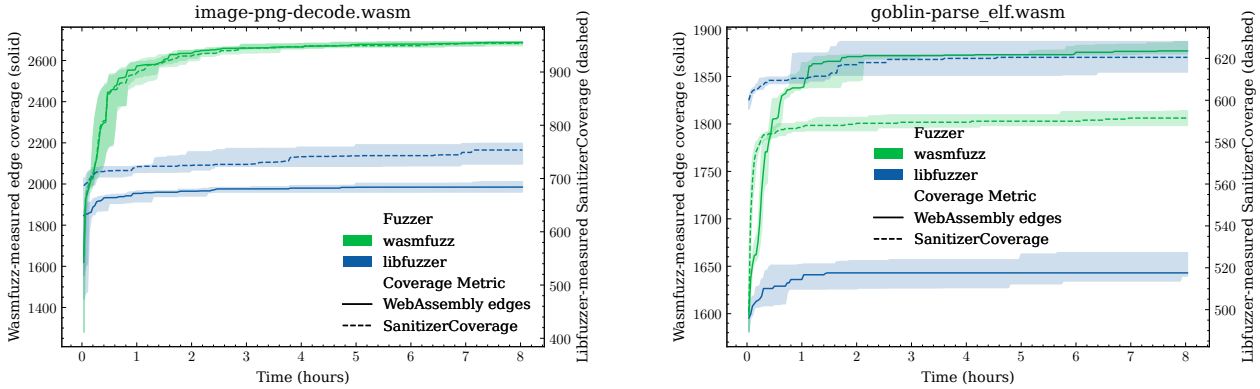
Our preliminary evaluation provides two first results but no conclusive experiments. For the full evaluation, we will perform code coverage experiments on a larger set of harnesses and evaluate the results in a statistically sound manner, as recommended by Klees et al. [24] and Schloegel et al. [39]. More specifically, we plan to perform the following experiments to answer the three research questions:

Adapt Standard Suite of C/C++Targets. Our preliminary set of targets described in Section 3.2 is mostly Rust-based and not necessarily representative of standard fuzzing workloads. To address this, we plan to attempt ports for all targets from the FuzzBench and Magma [23, 32] suites. This will give quantitative insights into the compatibility of standard targets with our approach and will allow future WebAssembly-based fuzzers to evaluate more thoroughly.

Compare Against Native Fuzzers. Our preliminary experiment, shown in Figure 5, is based on Rust targets and, through `cargo-fuzz`, uses LibFuzzer as its fuzzing engine. We plan to compare our approach with the native AFL++ fuzzer on our C/C++ harnesses as well. This requires modifications to the build setup for C/C++ targets since we are currently only targeting the WASM platform.

Measure Bug Discovery. In our preliminary WASM fuzzer experiment, we encountered several crashes (aborts, panics, access to unmapped linear memory). For the extended version, we will investigate some of these crashes in a "time-to-bug" experiment and determine whether the crashes are related to platform differences or represent *real* bugs that are reproducible with native fuzzing harnesses. For example, one of the bugs we encountered was caused by a memory corruption issue in `wasi-libc`, which we include in our C/C++ targets, so it only reproduced on WebAssembly builds. We reported and patched the issue upstream [5].

Evaluate Instrumentation Passes. Our prototype, WasmFuzz, implements several different instrumentation passes. While basic passes such as code and edge coverage are well understood, we also implement more exotic techniques such as PerfFuzz [28] and



(a) Example of a close coverage match

(b) Coverage metrics disagree in ranking

Figure 5: WasmFuzz–LibFuzzer coverage agreement: Shows a native and WASM fuzzer in both fuzzer's coverage metrics (twin Y axis). The covered area denotes the interquartile range. Eight hours per trial, ten trials per fuzzer.

LibFuzzer's `-use_value_profile=1`. We implement these passes to make sure WebAssembly modules are rich enough for useful instrumentation but have not investigated the effectiveness of these passes yet. We plan to perform feedback-specific ablation studies to measure the performance and efficacy of these passes empirically. We will also provide a lines-of-code-based measure for the implementation complexity of these passes to gain insight into the trade-off between instrumenting WebAssembly and other instruction sets.

4 Limitations and Future Work

In the following, we discuss potential limitations of our approach and sketch ideas for future work in this area.

Harness Compatibility. We discuss several platform differences in Section 3.2 that lead to harnesses that are not compatible with current WebAssembly toolchains. While many targets can be ported with little effort, some platform features, such as exception handling or threading, are not supported in the current WebAssembly release (and our tool). Still, many real-world projects can target WebAssembly (in part due to efforts like RLBox [35]), which uses WebAssembly for library sandboxing. Some of WebAssembly's intentional restrictions, such as the omission of stack unwinding, focus on single-threaded modules, and restriction to 4 GB of memory, are being lifted in future releases with the `exception-handling`, `threads`, and `memory64` proposals.

For Rust projects, most of the fuzzing harnesses we collected were compatible out-of-the-box. We aim to quantify harness compatibility for C/C++ projects as outlined in Section 3.5.

Bug Detection. Compiler-assisted fuzzers benefit from mature sanitizers [43] like Address Sanitizer [42] for fault detection. Even without sanitizers, memory corruption usually leads to a program crash. Unfortunately, WebAssembly modules hide many of these issues due to the linear memory design. Lehmann et al. [26] surveyed standard binary exploitation strategies and found that many old attacks work on WASM modules due to missing mitigations, which, in the case of WASM, lead to silent corruption. Therefore, for targets that are prone to memory corruption (C/C++), WASM-based fuzzers should be run together with a natively sanitized build to detect WASM-oblivious defects.

Compiling to WebAssembly can also affect bug detection due to logic differences between native and WASM builds in the platform support code, standard library, or target program. The main difference we observe is related to machine word size: Native builds usually target 64-bit architectures, while WebAssembly is a 32-bit platform.

These differences can manifest as both false positives and false negatives when looking at crashes compared to the native harness counterpart. False positives, in this case, may be valid bugs that only matter on 32-bit platforms, while false negatives might be bugs that do not crash under WASM's linear memory model.

Lack of a Standard WASM Harness suite. Our initial selection of fuzzing harnesses is a somewhat arbitrary mix of Rust and C/C++ targets (see Section 3.2). We would like to run experiments with a common set of targets on which other fuzzers have been evaluated, but unfortunately, all existing collections of fuzzing harnesses require manual porting to be compiled into WebAssembly.

We use Rust harnesses because the Rust toolchain provides good support for cross-compilation and allows us to fuzz many targets without manually porting each one. However, these targets are not necessarily representative of fuzzing benchmark suites such as FuzzBench [32] or Magma [23], as many Rust harnesses process structured input using, for example, the `arbitrary` library.

Coverage Evaluation Alone Provides Limited Insight. Our prototype is largely based on a custom implementation of the fuzzing lifecycle together with the mutators provided by LibAFL. This makes meaningful comparisons difficult as we cannot easily attribute them to specific changes. We plan to perform ablation studies to investigate the effectiveness of our instrumentation design choices and feedback types.

Dynamic Instrumentation. Our on-demand instrumentation enables approaches that dynamically apply feedback strategies. We imagine that scheduling instrumentation passes in a multi-armed

bandit fashion could improve fuzzing efficiency, as different targets benefit from different feedback types. Furthermore, applying different instrumentation to a selection of code locations might yield good results. We could either choose to periodically instrument a random subset of code locations, apply heuristics, or learn strategies that suggest effective instrumentation strategies for each function or code block. We plan to investigate such strategies as part of the full evaluation of this work.

Directed Fuzzing. On-demand instrumentation is well suited for optimized directed fuzzing approaches like *tripwiring* (SieveFuzz [44]). As part of the full evaluation, we plan to apply different coverage conditions to our harness automatically. Besides "reaches a certain edge" à la SieveFuzz, we plan to also explore runs that avoid a certain edge, require a certain value condition, or match a restricted runtime limit (e.g., execution trace length, input length, memory size limit).

5 Related Work

This paper forms the basis for further research into dynamic instrumentation for fuzzing, and we now discuss how our approach relates to prior work in this area.

5.1 Fuzzing and WebAssembly

With its first stable release in 2019, WebAssembly has not yet been explored a lot in the fuzzing literature. Nevertheless, several projects have experimented with WASM modules for fuzzing.

In late 2019, Metzman presented a tech demo that performs fuzzing in the browser ([31]): By combining LibFuzzer and a target harness in an object file, LLVM can generate a single WASM module that includes the LibFuzzer driver and the coverage feedback of the harness module. This approach demonstrates a rudimentary browser-based fuzzing workflow, although running in the browser provides little benefit: Without multiprocessing or threads, we cannot scale fuzzing in the browser, and corpus management is omitted from the demo as it traditionally requires access to the file system.

Later, in 2022, Haßler and Maier published WAFL [22], an extension of AFL that enables the fuzzing of WASM modules. The authors motivate WASM modules with an *extended target surface*, as the closed-source binaries of WebAssembly were not compatible with previous fuzzing tools. FuzzM [27] addresses the same goals as WAFL and seems to be concurrent work. Notably, neither work proposes to use WASM in the context of source-based native fuzzing. Several other tools target WASM modules in a binary-only setting: LibAFL [19] supports compilation to WebAssembly, similar to the browser fuzzing demo of Metzman, and Manticore [33] is a symbolic execution engine that supports WebAssembly.

In this work, we argue that fuzzing WebAssembly modules can be worthwhile even in a source-available setting where we usually compile to native code.

5.2 Execution Environments Tailored for Fuzzing

Several other fuzzing and binary analysis tools provide a deterministic and isolated execution environment. A notable example is the Cyber Grand Challenge (CGC) DECREE platform [47], which is designed for fully automated software exploitation and hardening. DECREE achieves deterministic execution—and thus stable reproduction of vulnerabilities—through a custom, simplified system call interface. While DECREE replaces the syscall interface, it still runs on a normal Intel x86 platform and thus must take into account the effects of platform-dependent instructions such as `cpuid`. In fact, one of the CGC contest submissions exploits these discrepancies to mislead other participants [45]. Although DECREE was developed before the advent of WebAssembly, a WASM-based platform would have been a suitable choice for their needs to avoid the complexity of the Intel x86 instruction set architecture.

6 Conclusion

In this work, we introduced and prototyped WasmFuzz as an alternative to the standard source-available fuzzing workflow. We discussed several differences compared to traditional toolchain-inserted instrumentation and highlightd the stability and portability of WebAssembly harness artifacts w.r.t. compatibility between fuzzers and target projects. This work poses two research questions which our preliminary evaluation provides first answers to:

RQ-1: Is the WebAssembly ISA a good fit for instrumentation? In Section 2.2, we discuss WebAssembly with respect to instrumentation opportunities. While WASM modules lose some high-level information, all data required for common feedback types remains available. To investigate fuzzing effectiveness, we compare WebAssembly module fuzzing and native fuzzing: To enable a fair comparison, we evaluate the corpus of each platform against the code coverage metric of the other platform. This shows that the differences between the platforms do not have a drastic impact on the fuzzing campaign and that the fuzzing of WASM modules can be used as a proxy for a native campaign—and even outperform `cargo-fuzz` in one of our preliminary experiments.

RQ-2: Is compatibility an issue? We collect a set of WebAssembly fuzzing harnesses in Section 3.2. For C/C++ targets, this required manual effort for each project (e.g., due to different build systems or projects not set up for cross-compiling), but only one harness turned out to be incompatible with WebAssembly (`libpng` requires `setjmp/longjmp`). For Rust targets, we were able to build 84 % of the 58 projects we collected without any modifications. We expect the percentage of incompatible projects to decrease in the future, both because WebAssembly is gradually becoming a common compilation target and because future versions of WebAssembly and WASI will implement missing APIs and functions.

In summary, we hope this work will form the basis for further research in these directions. We plan to extend the evaluation to address the research questions and explore the idea of dynamic instrumentation in the future.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669). F. Bauckholt carried out this work as a member of the Saarbrücken Graduate School of Computer Science.

References

[1] [n. d.]. OSS-Fuzz - Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz.

[2] [n. d.]. Rust Fuzz Book. https://github.com/rust-fuzz/book.

[3] 2016. CGC: Cyber Grand Challenge. https://www.darpa.mil/program/cyber-grand-challenge.

[4] 2020. [GitHub Issue] QSYM results are inaccurate, does not work on modern kernel. https://github.com/google/fuzzbench/issues/131

[5] 2024. [GitHub PR: wasi-libc] iconv/wctomb: Fix memory corruption related to CURRENT_UTF8. https://github.com/WebAssembly/wasi-libc/pull/511

[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*.

[7] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATEC)*.

[8] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *ACM European Software Engineering Conference / Foundations of Software Engineering (ESEC-FSE)*.

[9] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*.

[10] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*.

[11] Addison Crump, Andrea Fioraldi, Dominik Christian Maier, and Dongjia Zhang. 2023. LIBAFL LIBFUZZER: LIBFUZZER on Top of LIBAFL. In *IEEE Workshop on Search-Based and Fuzz Testing (SBFT)*.

[12] LLVM Developers. 2015. LibFuzzer - A Library for Coverage-guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.

[13] LLVM Developers. 2017. UBSan: Undefined Behaviour Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[14] Wasmtime Developers. [n. d.]. Cranelift Code Generator. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (S&P)*.

[16] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3385972

[17] Alessandro Di Federico. 2016. [Qemu-devel] Support for using TCG frontend as a library. https://qemu-devel.nongnu.narkive.com/T3szFzTw/support-for-using-tcg-frontend-as-a-library.

[18] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[19] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*.

[20] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security Symposium*.

[21] Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. 2023. Systematic Assessment of Fuzzers using Mutation Analysis. In *USENIX Security Symposium*.

[22] Keno Haßler and Dominik Maier. 2022. WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots. In *Reversing and Offensive-Oriented Trends Symposium (ROOTS) (ROOTS'21)*.

[23] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*.

[25] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *ACM International Symposium on Code Generation and Optimization (CGO)*.

[26] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *USENIX Security Symposium*.

[27] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. 2021. FuzzM: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly. *Computing Research Repository (CoRR)* (2021). arXiv:2110.15433

[28] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[29] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[30] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*.

[31] Jonathan Metzman. 2019. Your Browser is my Fuzzer: Fuzzing Native Applications in Web Browsers. https://github.com/jonathanmetzman/wasm-fuzzing-demo/blob/master/meetup-Fuzzing-Native-Applications-in-Browsers-With-WASM.pdf.

[32] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*.

[33] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[34] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing. In *USENIX Security Symposium*.

[35] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *USENIX Security Symposium*. USENIX Association.

[36] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *IEEE Symposium on Security and Privacy (S&P)*.

[37] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *USENIX Security Symposium*.

[38] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Symposium on Network and Distributed System Security (NDSS)*.

[39] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*.

[40] Eric Schulte, Michael D. Brown, and Vlad Folts. 2022. A Broad Comparative Evaluation of x86-64 Binary Rewriters. In *Cyber Security Experimentation and Test Workshop (CSET)*. https://doi.org/10.1145/3546096.3546112

[41] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*.

[43] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *IEEE Symposium on Security and Privacy (S&P)*.

[44] Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. 2022. One Fuzz Doesn't Fit All: Optimizing Directed Fuzzing via Target-tailored Program State Restriction. In *Annual Computer Security Applications Conference (ACSAC)*. https://doi.org/10.1145/3564625.3564643

[45] Michael F. Thompson. 2018. Effects of a Honeypot on the Cyber Grand Challenge Final Event. In *IEEE Symposium on Security and Privacy (S&P)*.

[46] WebAssembly Community Group. 2020. WebAssembly System Interface. https://github.com/WebAssembly/WASI/blob/d8b286c697364d8bc4daf1820b25a9159de364a3/phases/snapshot/docs.md

[47] Lok K. Yan, Benjamin Price, Michael Zhivich, Brian Caswell, Christopher Eagle, Michael Frantzen, Holt Sorenson, Michael Thompson, Timothy Vidas, Jason Wright, Vernon Rivet, Samuel Colt VanWinkle, and Clark Wood. 2018. BP: DECREE: A Platform for Repeatable and Reproducible Security Experiments. In *IEEE Cybersecurity Development (SecDev)*.

[48] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*.

[49] Michał Zalewski. 2013. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[50] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2023. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/

Received 2024-06-21; accepted 2024-07-22