WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots

Keno Haßler
keno.hassler@campus.tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Dominik Maier
dmaier@sect.tu-berlin.de
Technische Universität Berlin
Berlin, Germany

ABSTRACT

WebAssembly, the open standard for binary code, is quickly gaining adoption on the web and beyond. As the binaries are often written in low-level languages, like C and C++, they are riddled with the same bugs as their traditional counterparts. Minimal tooling to uncover these bugs on WebAssembly binaries exists. In this paper we present WAFL, a fuzzer for WebAssembly binaries. WAFL adds a set of patches to the WAVM WebAssembly runtime to generate coverage data for the popular AFL++ fuzzer. Thanks to the underlying ahead-of-time (AOT) compiling WAVM, WAFL is already very performant. WAFL adds lightweight VM snapshots. By replacing forks, traditionally used in AFL++ harnesses, with WAFL's snapshots, WAFL harnesses can even outperform native harnesses with compile-time instrumentation in raw fuzzing performance. To the best of our knowledge, WAFL is the first coverage-guided fuzzer for binary-only WebAssembly, without the need for source.

CCS CONCEPTS

• **Security and privacy** → *Web application security*; Software reverse engineering.

KEYWORDS

Fuzzing, WebAssembly, AFL, Binary-Only

ACM Reference Format:
Keno Haßler and Dominik Maier. 2021. WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots. In *Reversing and Offensive-oriented Trends Symposium (ROOTS'21), November 18–19, 2021, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3503921.3503924

1 INTRODUCTION

The web has evolved from a passive hypertext-reader to a platform for highly interactive client-side applications. Even though JavaScript performance has improved drastically in modern just-in-time (JIT) engines, most high-performance frameworks for imaging and gaming are written in native programming languages. To bring these to the web and to get speeds of complex tasks closer to native, WebAssembly was introduced [3]. WebAssembly is an open binary standard adopted by modern browsers, used in various other use-cases, and is supported as a compilation target for many compiled programming languages. New browser frameworks like Yew [8] and Blazor [13] even side-step JavaScript for web development completely. Developers can write web applications in languages like Rust and C# directly, the frameworks then target WebAssembly to execute the respective language.

Taking the idea of portability one step further, the open WASI standard [4] allows standalone WebAssembly programs that even run outside the browser. The goal is to create a truly universal binary platform. The infrastructure around WASI is still young but starting to grow, for example, through the WebAssembly Package Manager (wapm) [23]. Using wapm, users can download WebAssembly binaries that run on WebAssembly System Interface (WASI)-enabled VMs. The programs, distributed as WebAssembly binaries, run on every platform for which a runtime is available.

As it is a compilation target like any other, memory vulnerabilities in unsafe source languages, like C, are translated to WebAssembly, and remain potentially vulnerable. While the platform was developed with security in mind and supports modern mitigations, bugs may still be exploitable and lead to code execution, as Lehman et al. have shown [9]. Until now, the tooling to uncover memory corruptions in WebAssembly binaries is limited.

In this paper, we present WAFL, a fuzzer for WebAssembly binaries with good throughput. WAFL uses the well-known AFL++ fuzzer for input generation and lightweight VM snapshots for performance. Building on top of WebAssembly Virtual Machine (WAVM), we can fuzz WebAssembly binaries without source code access. We evaluate the resulting fuzzing speed and show that, thanks to its snapshotting mechanism, WAFL even outperforms naive harnesses compiled from source for x86-64.

Contributions

- We develop WAFL, an open-source, binary-only WebAssembly fuzzer.
- We implement and benchmark multiple improvements on top of an initial wasm3-based implementation.
- In its final form, the AOT compiling, WAVM-based snapshot fuzzer even outperforms traditional compiled, *native code* AFL harnesses that use the slow `fork` syscall.

2 BACKGROUND

Fuzzing, or fuzz testing, is a dynamic analysis technique that feeds random input to programs and observes their behavior. American Fuzzy Lop (AFL) is an open-source coverage-guided fuzzing engine. Created in 2014, it became a standard tool. After development stalled in 2017, the community-based fork AFL++ [2] continued to integrate improvements from scientific research, based on AFL.

2.1 Coverage Measurement

AFL aims to increase the test surface covered by fuzzing inputs. The coverage is measured by the number of visited edges in a program's

ROOTS'21, November 18–19, 2021, Vienna, Austria
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9602-8/21/11…$15.00
https://doi.org/10.1145/3503921.3503924

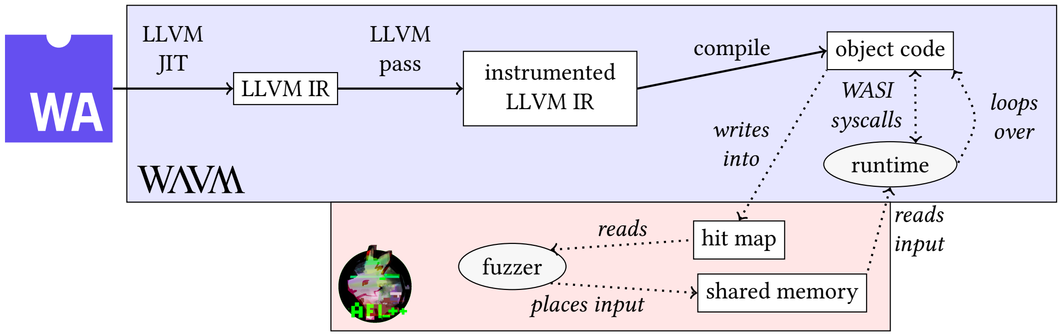

Figure 1: Schematic overview of WAFL. WebAssembly is lifted to LLVM intermediate representation (IR) by WAVM. WAFL then instruments the IR with one of the available passes and compiles it to object code. This code runs in a loop, leaving hit counts in the shared map. The fuzzer evaluates the map and writes new input into shared memory, which is read by the runtime and returned to native code through the WASI syscall interface.

control-flow graph (CFG). An input is considered interesting if it generates new coverage in the target during execution. For feedback gathering, AFL's instrumentation injects a shared memory map into the program under test. During execution, the instrumentation assigns a hash or unique ID to each basic block, and increases a counter in AFL's map at the respective location. After a run finishes, the fuzzer checks for new entries in the map. If new entries were found during execution, AFL keeps the input for subsequent mutations [29].

Programs with available source code get instrumented using a wrapper around existing compilers, `afl-cc`. The tool injects the needed compiler passes to gcc or clang.

The InsTrim pass [6] included in AFL++[1] improves instrumentation speed by analyzing the Control Flow Graph. It only marks a subset of blocks necessary to distinguish paths (around 20%, according to the authors). As a downside of the block ID hashing technique used in InsTrim and the traditional afl-clang pass, the algorithms are likely to produce index collisions when the number of instrumented blocks grows.

An even more sophisticated approach is taken by the LLVM SanitizerCoverage [21] pass. It assigns a *guard* variable to each edge and inserts a callback function using the variable as a parameter. Initialization of the guards is performed in a second callback function, so each can be set to a unique number, making the hashing obsolete. AFL++ uses this pass as default.

2.2 WebAssembly

Web-based applications are becoming more popular and complex, the main programming language of which is JavaScript. With the release of Emscripten in 2011, a solution to compile high-level languages like C to JavaScript for use on the web existed [28]. With it, native code could be ported to the web.

However, JavaScript is not an ideal compilation target. It lacks a compact representation. Also, by being a high-level scripting language, all code has to get interpreted or JIT compiled at runtime, leading to poor performance and increased startup times. WebAssembly got introduced to provide a proper compile target for native applications on the web [3].

It is a low-level bytecode for a stack machine, typically JIT or AOT compiled to machine code in the web browser and became Emscripten's new default target. Besides web applications calling into WebAssembly from JavaScript to off-load performance-critical tasks, Emscripten can also create standalone binaries. To this end, it defines an Application Binary Interface (ABI) that can be implemented by WebAssembly runtimes and includes a standard C library compiled against it.

As an attempt to standardize this ABI, WASI was created [4]. Although similar in their approach, these ABIs are not fully compatible. A WebAssembly backend has been included in Low-Level Virtual Machine (LLVM) and can be used in conjunction with `wasi-libc` [26] to compile C (and to some extent, C++) to WASI. Rust also supports WASI as a Tier 2 target.

2.3 WebAssembly Runtimes

In mid-2021, most WebAssembly runtimes are still in an early stage of development or don't support the full specification of WASI [1]. This work focuses on two of the more fully-featured WebAssembly runtimes: *WAVM* and *wasm3*. The latter is a lightweight runtime written in C that provides fast startup and small memory footprint. It is based on an optimized tail-call interpreter design dubbed *M3* [10]. WAVM is written in C++ and uses LLVM's JIT engine to compile the WebAssembly binary into native code AOT. While translating the WebAssembly code to a native binary ahead of time, it links the code to the WAVM runtime library, providing built-in WebAssembly and WASI functions. Thereby, it trades a notably higher startup latency for higher performance after initialization.

A benchmark by Denis [1] evaluates the execution speed of eight popular standalone WebAssembly runtimes using the cryptography library *libsodium*. LLVM-based (AOT) runtimes achieve the best results, with WAVM running 15% faster than the next best, wasmer [24], and only 2*x* slower than native code. Restricting the test field to interpreters, wasm3 scores best, albeit with a 30*x* slowdown compared to native code.

[1]Up to version ++3.12c

3 RELATED WORK

While WebAssembly is still niche, we will highlight some research in the field in this section. In the second part of this section, we will go on to discuss VM snapshots further.

3.1 WebAssembly Analysis

Metzman [11] demonstrated source-code based in-browser fuzzing based on libFuzzer, compiled with Emscripten. After building an instrumented WebAssembly binary from source, the resulting fuzzer can run and fuzz on any WebAssembly platform. The target harnesses (*brotli*, *lzma* and *sqlite*) were adapted from OSS-Fuzz. Notably, Metzman et al. also created a framework for fuzzer evaluation, with similar samples to those of OSS-Fuzz, Fuzzbench [12].

Fuzzcoin [7], a crowd computing network for fuzzing, expands the same idea of an in-browser fuzzer for WebAssembly to 30 OSS-Fuzz projects. A user-friendly web interface invites visitors to donate their CPU time for virtual coins.

As opposed to fuzzing the WebAssembly binaries, Watt, as well as Perényi and Midtgaard, propose methods to verify the WebAssembly VMs using fuzzing. For it, binaries are randomly generated as input to the runtime under test, either by translating from C [25] or by directly generating WebAssembly [15]. Runtimes are fuzzed regularly: wasm3 is an OSS-Fuzz project itself, WAVM provides multiple targets for libFuzzer.

Lehmann et al. [9] find that WebAssembly binaries lack many mitigations common in traditional binaries. They show vulnerabilities in memory-unsafe code can translate into WebAssembly and, combined into exploit chains, can compromise the host system. Building on their work, Hilbig et al. [5] collect a real-world dataset of WebAssembly binaries and scan it for vulnerabilities. They find that many binaries use the unmanaged stack, custom allocators or dangerous APIs (65%, 38.6% and 21.2%, respectively), which are the potential weak points identified by Lehmann et al.

3.2 Snapshot Fuzzing

The idea to snapshot the memory space of an application or VM is nothing new in fuzzing. Even a basic AFL harness, using the `fork` syscall, technically uses snapshots for fuzzing. Of course, over the years, more advanced snapshot mechanisms got proposed. Newsham and Jesse [14] adapt the traditional AFL forks for full-system fuzzing by forking a system-mode QEMU instance. After uncovering the suboptimal scaling and speed of the `fork` syscall, Xu et al. [27] propose a syscall that's better-suited for fuzzing.

Instead of snapshotting a userspace process, VMs can snapshot a complete guest state for fast resets of a full system. Notably, Agamotto by Dokyung et al. [20] adds a quick VM snapshotting mechanism to QEMU KVM to then fuzz kernel drivers. Similarly, Nyx by Schumilo et al. [18] uses quick VM snapshots to fuzz kernel and even userspace targets with high throughput.

4 DESIGN

The high-level objective of WAFL is to enable coverage-guided fuzzing for binary-only WebAssembly programs using AFL++, and to allow fast VM resets after each execution with lightweight snapshots. As mentioned in subsection 2.2, WebAssembly binaries can target different ABIs. In the context of this work, we focus on standalone binaries targeting WASI and, in part, Emscripten.

4.1 WAFL-wasm3

We will discuss different ways to implement WebAssembly fuzzing during this paper. We based a simple baseline implementation of WAFL on wasm3, a fully interpreted VM. For this interpreter-based WebAssembly VM, we inserted the feedback mechanism for AFL directly into the wasm3 runtime. The patched WAFL-wasm3 interpreter runs instrumentation code at every WebAssembly Control Instruction during execution (see [16]).

This takes place in the `m3_exec` component. When executing a control instruction, we take note of the target address (i.e., the address of the next block being executed) and relay it to a feedback function. It calculates a hash from the given address and the one it has previously seen. These hashes correspond to edges in the program's Control Flow Graph. Their Least Significant Bits are then used as an index into the AFL shared map, where the corresponding field is incremented.

Upon runtime startup, a setup routine needs to be run that maps this shared memory region provided by AFL++ into the runtime address space.

The second core component is the forkserver, [2] a component normally inserted into binaries under test by AFL++. Before the actual program starts, it performs a handshake with the parent process and drops into a forking loop. Child processes exit the loop, continue normal execution through the program under test and terminate. Placing this component as late into the runtime startup as possible provides an opportunity for speed enhancements because child processes can thereby skip the initialization phase. If an error occurs at runtime, the child process is aborted, and the forkserver tells AFL++ about the crash.

With these modifications (and switching off the binary check), AFL++ is able to fuzz WebAssembly binaries through wasm3. However, performance-wise, this leaves a lot to be desired.

4.2 WAVM-WAFL

Recalling the vast performance differences between WebAssembly runtimes (subsection 2.3), fuzzing solutions based on an interpreter (like wasm3) cannot be expected to deliver the highest execution speeds. Instead, we implemented the main WAFL version, based on a fast AOT compiling runtime. We chose WAVM, which has good performance according to benchmarks [1]. However, it requires a different approach for instrumentation.

Upon loading a WebAssembly file, WAVM compiles the binary to platform-native code using LLVM-JIT and executes optimization passes on the generated code. This is convenient since AFL++ uses LLVM optimizer passes to insert its instrumentation code. Consequently, the first step was linking the classic AFL LLVM pass to WAVM (requiring a one-line modification to the pass) and setting it up to run last.

The setup code for the shared map and forkserver is the same as WAFL-wasm3.

4.3 Lightweight VM Snapshots and Resets

In AFL++ terms, fuzzing in persistent mode [2] means reusing one child process for multiple iterations. It allows replacing time-intensive `fork()` syscalls by looping over relevant code regions in the child process. Persistent mode fits well with our application, where the interesting code is the pre-compiled target. However, a target may accumulate state during execution or even leak memory, rendering persistent fuzzing unstable. Hence, if we want to fuzz without forks, we must reset the target state after each execution. Ideally, we want to do this without patching the WebAssembly binary or instrumenting it further, especially in a binary-only scenario.

WebAssembly defines three kinds of stateful objects which might be altered by the target program: globals, tables, and memories [3]. Currently, compilers only use one memory, an array of bytes. In it, they create a familiar layout comprised of Stack, Heap, and Data sections [9]. Based on this observation, we can implement VM snapshots and restores: we intercept the runtime shortly before the first call into the target code. At this point, the linear memory has already been initialized by the runtime. We create a snapshot of its content and size. When control flow returns to the runtime after each loop iteration, we shrink the memory to its initial size and write back the snapshot.

Using this minimalistic snapshot engine removes the considerable overhead of re-initializing the runtime, promising a corresponding performance increase.

4.4 Improved Instrumentation

AFL++ provides multiple options for instrumentation besides the classic pass. An early improvement called InsTrim [6] analyzes the programs Control Flow Graph to reduce the number of instrumentation points, thereby increasing fuzzing performance. Integration of InsTrim into WAVM required a small Makefile change for AFL++ and the same one-line patch as for the classic pass.

As additional options for the discussed instrumentation techniques, AFL++ offers Context-sensitive and N-Gram Branch Coverage presented by [22]. Both modify the calculation of shared map indices when an instrumentation point gets hit. The former incorporates the calling context, represented by a simplified call stack by XORing it with the normal branch ID. The latter stores the last N branch IDs in a tuple and hashes them together (using XOR and bit-shifting) to calculate an index. To switch between all these options, an environment variable (`AFL_LLVM_INSTRUMENT`) can be set for WAVM analogous to afl-clang-fast.

Currently, the default AFL++ instrumentation is based on SanitizerCoverage [21], LLVM's code coverage instrumentation. For SanitizerCoverage, an address is used to calculate a unique index into the shared map:

```
idx = ((&offset - &guard) >> 2) & (MAP_SIZE - 1)
```

The instrumentation subtracts an offset pointer to calculate reproducible indices across runs, despite randomly remapped guard variables. Since all guard variables are 32-bit unsigned integers placed continuously in memory, shifting their addresses two bits to the right makes indices consecutive. Lastly, masking ensures that all indices are in $[0, \texttt{MAP\_SIZE} - 1]$ (so they will not collide as long as $\texttt{MAP\_SIZE} > \#guards$). We encountered an issue with AFL++'s SanitizerCoverage implementation that kept it from working for WAFL. The user (us) needs to provide implementations for two callback functions. However, in WAFL, the first callback is never called, leaving the *guards* (indices into the shared map) uninitialized, Instead, WAFL now relies on LLVM's own SanitizerCoverage implementation directly.

4.5 Shared Memory Fuzzing

AFL usually passes test cases to target programs via the file system, inducing overhead. Either AFL passes the input file name to the program as an argument, or it opens the file itself and pipes the contents to the target via standard input. When fuzzing in persistent mode, AFL++ can instead pass input through a shared memory buffer similar to the shared map for hit counts.

Assuming that most targets will be able to read from standard input, the goal was making reads from it independent of the file system. To this end, we added a check to the WAVM runtime implementation of the WASI `readv()` syscall: If the file number corresponds to standard input, a custom routine reads from the shared buffer instead. Every iteration, the persistent loop re-opens the buffer with the correct length using `fmemopen()`.

4.6 Blocking libc Instrumentation for Speed

There is usually no dynamic linking in WebAssembly. Everything comes bundled in one statically linked file. WebAssembly programs are commonly written in high-level languages (C or Rust). These languages provide a set of standard functions that need to be incorporated in the binary, which means they get instrumented by WAFL as well.

Given that, in a white-box fuzzing scenario, one would probably not instrument library code, the respective functions in WebAssembly binaries should also be excluded. For this, we leverage AFL++'s blocklist functionality. For WAFL, we only needed to modify WAVM slightly to support it: Function names are usually not passed from the input file into the generated code. After adding this feature, we were able to read function names in the LLVM passes.

We adapted a list of symbols in the WASI C library [26] [2] to the LLVM format, by prefixing each name with `fun:`. Additionally, we added `printf_core` and `pop_arg`, two function names appearing commonly in WASI binaries. Using this blocklist, we reduce the number of instrumented code blocks considerably, as shown in Figure 2. With small targets, like brotli and lzma2dec, instrumented edge counts can be cut by 63% (1697 vs. 628 for lzma2dec with the Classic pass). For the LLVM pass, we added a small check to WAVM that reads the `AFL_LLVM_{ALLOW,DENY}LIST` environment variables and passes the indicated file names to LLVM if it supports this feature (version 11 and up).

5 EVALUATION

In the following, we will evaluate the WAFL development steps discussed in section 4. Moreover, we compare WAFL to native instrumentation, to a binary compiled using AFL++'s LLVM backend with the same simple harness. As set out in section 3, there is not much prior work to compare WAFL to, other than the in-browser solution presented by Metzman [11]. Unfortunately, their binaries

[2] expected/wasm32-wasi/defined-symbols.txt

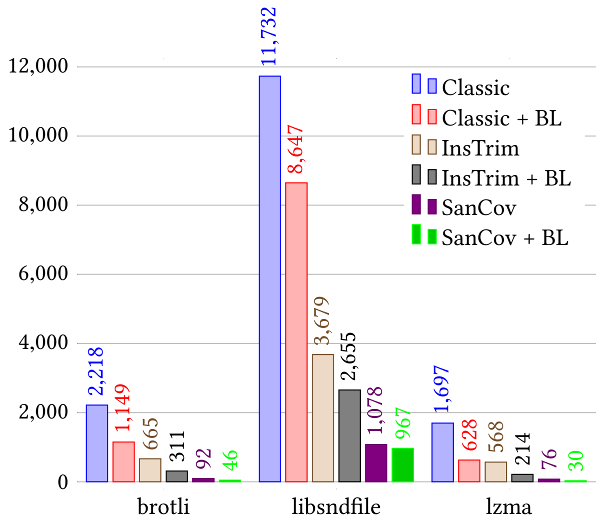

Figure 2: Number of instrumented edges in three WebAssembly binaries using Classic, InsTrim and SanitizerCoverage, with and without the blocklist.

do not run in WAVM or wasm3. Therefore, they were compiled from source with a custom standalone runner that reads a buffer from `stdin` and passes it to `LLVMFuzzerTestOneInput()`. This worked well for brotli and lzma, two compression libraries, but failed for sqlite. Additionally, we include libsndfile, an audio library, in this evaluation, and describe an ad-hoc test against binary-only WebAssembly blobs.

5.1 Test Setup

The platform used is a server with two AMD EPYC 7281 16-core processors and 112 GiB of RAM running Ubuntu 20.04 (Linux 5.4). All tests were run in parallel using LLVM / Clang version 12 and AFL++ 3.15a. We took the fuzz targets and harnesses from OSS-Fuzz [19]: After compiling them according to the provided build scripts, we linked against a custom standalone runner that reads from standard input.

For the native binary, we used `afl-clang-fast`. We compiled the WASI binaries using Clang with the wasm32-wasi backend and a sysroot from wasi-libc 12 [26]. Additionally, `-mthread-model single` had to be passed to the compiler because WASI does not support threads. We evaluated WAFL's performance in a variety of configurations enumerated in Table 1. We ran every configuration for 24 hours, averaging over five runs, to find out how many times the target gets executed per second and how many branches the fuzzer discovers.

As a comparison, the fuzzers for brotli and lzma2dec created in [11] were run for a multi-hour fuzz run in Firefox 91. The maximum execution speed reported during that time is used here to have an optimistic estimation of the in-browser performance. However, this value might still under-estimate the solution's full potential, as Metzman provides an optimized sqlite-fast variant that does not draw on the browser canvas, substantially outperforming sqlite by about 10x. For brotli and lzma2dec, there is no such fast binary available.

5.2 Brotli Harness

The brotli library provides a fuzz target for its decoder; the produced WebAssembly binary weights 201 KiB. As input for AFL++, the included fuzzer seed corpus was used. The results can be seen in Figure 3. While the non-persistent WAVM setup (174 exec/s) is vastly outperformed by the native binary (1878 exec/s), enabling persistent mode changes the picture. Here, all three instrumentation options (Classic, InsTrim and SanitizerCoverage) run faster than the native binary (with 2532, 5008 and 2753 exec/s, respectively). Additionally, enabling shared memory fuzzing increases performance to 3328 for Classic and 3376 for SanitizerCoverage, but slightly decreases it for InsTrim (4715 exec/s).

In the final stage, including shared memory fuzzing and blocklisting C library functions, the performance of a SanitizerCoverage based instrumentation drops below the version with just persistent mode enabled (2644 exec/s), although the other two configurations achieve their best values (Classic with 4077 exec/s and InsTrim with 7443 exec/s), the latter even *outperforming* the optimized native binary (6917 exec/s). The browser-based libFuzzer achieved a maximum of 1081 exec/s.

5.3 Lzma Harness

The lzma project in OSS-Fuzz contains fuzz targets for widely-used compression algorithms such as 7z, lzma, lzma2 and xz. Binaries are similar in size, and the lzma2 decode fuzzer (96 KiB) was chosen arbitrarily for evaluation. Seed files are provided within the project, so the lzma2dec corpus was used.

The highest speeds are achieved with the InsTrim-based instrumentation in all three configurations: with snapshots and shared memory (9655 exec/s), closely followed by the same configuration with blocklists (9171 exec/s) and with only snapshots (8859 exec/s). The classic AFL instrumentation scores best with blocklists enabled (7053 exec/s), outperforming the configurations without blocklists (6372 exec/s) and with only snapshots (4902 exec/s). SanitizerCoverage behaves the same, performing best with all options enabled (6339 exec/s), slightly worse without blocklists (5988 exec/s) and notably slower without shared memory input (4571 exec/s).

These numbers fall between the performance of our non-optimized native target (1780 exec/s) and an optimized target compiled with AFL++'s libFuzzer driver, including persistent mode and shared memory fuzzing (9186 exec/s), with the best configuration again *outperforming* the optimized native version. The maximum speed achieved by libFuzzer in-browser was 1076 exec/s.

5.4 Libsndfile Harness

Sndfile is a library for converting sound files and formats. It is considerably larger than brotli (1.2 MiB as WebAssembly) and AFL++'s forkserver initialization timeout had to be increased due to the AOT compilation overhead. Short of a proper seed corpus, `testfile.mp3` from the liblame repository was used as input to AFL++. Execution time for different configurations are shown in Figure 3. All configurations are rather slow, owing to the large binary. Running without snapshots performs well on first sight (414 exec/s vs. 754 in persistent mode), but cannot be recommended because virtually no paths are discovered. Enabling blocklists roughly cuts the performance

					Brotli		Lzma		Sndfile	
Instrumentation	Pass	📷[i]	💾[ii]	📝[iii]	exec/s	paths	exec/s	paths	exec/s	paths
afl-clang	PCGuard	×	×	—	1878± 301	2002± 73	1780± 53	16± 0	—	—
afl-clang	PCGuard	✓[iv]	✓	—	6917± 3647	2107± 69	9186± 3186	12± 0	—	—
libFuzzer (web)	—	✓[v]	—	—	1081	—	1076	—	—	—
WAFL-WAVM	Classic	×	×	×	174± 27	2202± 55	179± 30	1151± 71	414± 48	59± 8
WAFL-WAVM	Classic	✓	×	×	2532± 648	2477± 54	4902± 336	1330± 37	754± 10	2101± 2298
WAFL-WAVM	Classic	✓	✓	×	3328± 2114	2401± 150	6372± 949	1300± 83	715± 122	2978± 1654
WAFL-WAVM	Classic	✓	✓	✓	4077± 1686	2230± 81	7053± 953	1245± 40	290± 12	2099± 398
WAFL-WAVM	InsTrim	✓	×	×	5008± 2409	3031± 154	8859± 2459	1061± 43	701± 31	1977± 1775
WAFL-WAVM	InsTrim	✓	✓	×	4715± 2354	2983± 80	9655± 2712	1047± 36	717± 11	3428± 1908
WAFL-WAVM	InsTrim	✓	✓	✓	7443± 1569	2532± 139	9171± 1376	915± 32	301± 126	1574± 525
WAFL-WAVM	SanCov	✓	×	×	2753± 1292	2044± 108	4571± 370	946± 26	669± 18	1804± 466
WAFL-WAVM	SanCov	✓	✓	×	3376± 1166	2066± 97	5988± 1059	911± 33	636± 47	3778± 2733
WAFL-WAVM	SanCov	✓	✓	✓	2644± 2067	1897± 96	6339± 903	844± 28	444± 26	1724± 1463

[i] Snapshots [ii] Shared Memory [iii] Blocklist [iv] Persistent loop [v] In-process

Table 1: Fuzzer executions per second and paths found for WAFL with different configurations, native binaries instrumented with afl-clang and the in-browser libFuzzer WebAssembly solution from [11].

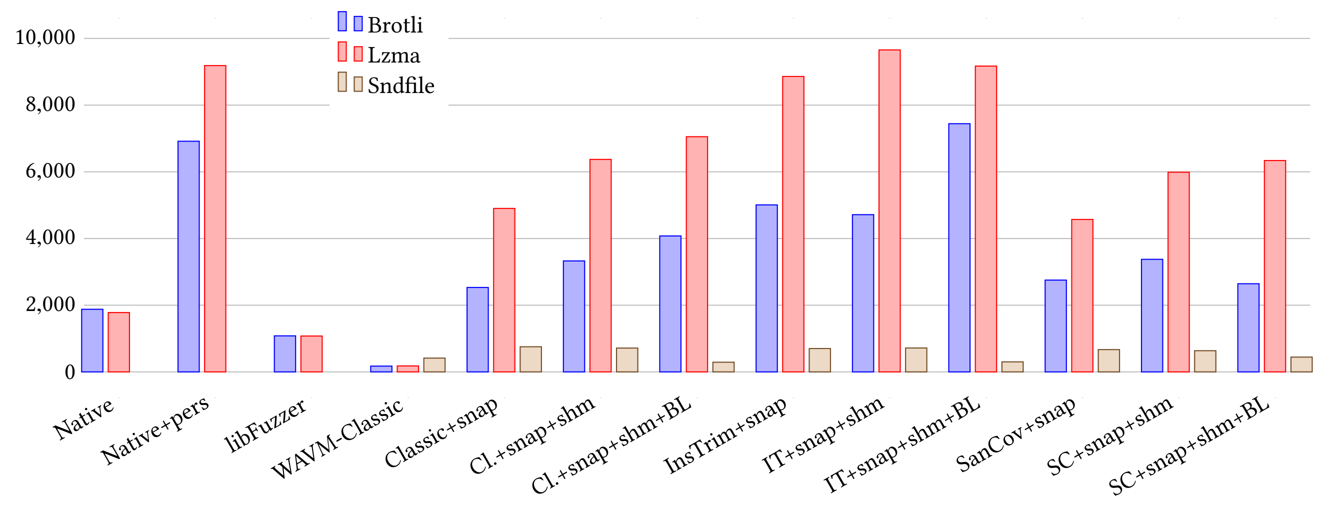

Figure 3: Fuzzer executions per second across all benchmark configurations.

in half for this program, with all three instrumentation algorithms performing on par. However, looking at the discovered paths without blocklists, SanitizerCoverage scores best at 3778 paths, with InsTrim (3428) and Classic (2978) behind.

5.5 WAFL Snapshot Performance

In the benchmarks described above, WAFL with persistent mode performs better than a native binary without persistent mode and, in some cases, comparable to an optimized native binary. To evaluate the speed of our lightweight VM memory snapshots, we tested a small C program that allocates a chunk of memory and writes into it. The results are depicted in Figure 4. For small allocation sizes, WAFL needs 55 µs for one execution, while the native binary takes 150 µs. Only at an allocation size of 256 KiB and larger fork outperforms WAFL snapshots, which need 160 µs at that point. Notably, this measurement also takes overhead during execution into account, where most time is spent according to our evaluation. Moreover, we benchmarked on a single core. We assume WAFL snapshots will scale a lot better than kernel interactions and page table walks, needed by the slow `fork` syscall [27].

5.6 Binary-Only Fuzzing of Real-World WebAssembly Applications

There are already a number of real-world applications powered by WebAssembly [5]. We queried wapm [23] to find suitable test candidates. The amount of packages in wapm is still low. However, we found two packages that were straightforward to execute and fuzz, namely `cowsay` and `qr2text`.

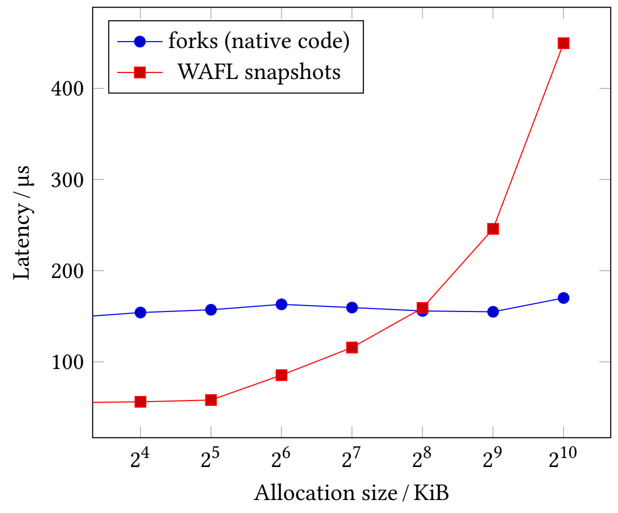

Figure 4: Execution latency over allocation size. WAFL snapshots *outperform* native forked harnesses that touch less than 256 KiB per execution.

After running WAVM against `qr2text`, a program that, contrary to its name, creates QR codes from text input did not find any bugs. After a starting phase, fuzzing speed stabilized around 400 exec/s. No crashes were triggered during six hours.

In the wapm version of cowsay, AFL++ immediately uncovered crashes. After manual analysis, we concluded that this rust version of cowsay available in the package manager would indeed panic on any non-UTF8 input. While this proves the applicability of WAFL in a real-world scenario, the bug wouldn't cause any real harm, leading to a DoS against a network-based cowsay in the worst case.

5.7 Discussion

As expected, the interpreter-based WAFL-wasm3 and WAFL-WAVM without persistent mode are slow, much slower than the in-browser fuzzer baseline. However, wasm3 is, surprisingly, faster than WAVM in this scenario. One possible explanation is the smaller runtime that may be beneficial when forking. As soon as snapshot mode is enabled, performance surpasses the native binary, regardless of the instrumentation used. This shows the use of snapshot mode in practice.

Additionally, enabling shared memory fuzzing has a positive effect for the brotli and lzma2dec targets. Overall, the result is positive, indicating that shared memory fuzzing should be used if available (i.e., input is passed via stdin instead of files). We expect that the result benefits from scaling to multiple cores, as shared memory input does not need to go through the kernel.

Regarding blocklists, there is a mixed picture: In some configurations (Brotli with SanitizerCoverage, Lzma with InsTrim), it seems that performance decreases with blocklists. This might be due to harder (and more relevant) paths being found by the fuzzer. Figure 2 shows high absolute numbers of instrumented blocks saved for the Classic instrumentation, but almost none for SanitizerCoverage. Due to this, the positive effect on SanitizerCoverage may be small. Overall, we think that blocklists help with fuzzing, but the measured effect is small.

Comparing the different instrumentation passes, SanitizerCoverage yields disappointing results, with InsTrim being stronger than both others. The results might be target-dependent, but in our test set, InsTrim can be recommended as the best choice for instrumentation.

6 FUTURE WORK

SanitizerCoverage has the lowest number of instrumented blocks and is favored over InsTrim by upstream AFL++. In our measurements, its integration in WAFL-WAVM performs worse in comparison. We suspect that the callback design, especially the initialization of guards during runtime, can be improved further. Ultimately, it should be possible to reach faster speeds than InsTrim. If the initialization problem can be solved, either in WAVM or by editing the SanitizerCoverage pass, inlining the callbacks comes at no cost using the `Inline8bitCounters` option.

There are further instrumentation options in AFL++ not explored in this paper. AFL++ has Link Time Optimization (LTO) LLVM passes and recommends using them. Since, in our case, the input is already in one big translation unit, it is unlikely that they would improve results, but this remains to be verified.

More promising are feedback improvements. CompCov [2] splits multi-byte compare instructions into single-byte compares, making it easier for the fuzzer to traverse them. Also, by hooking compares, using heuristics similar to the libc instrumentation blocklist, CmpLog support can be added, feeding feedback to AFL++'s RedQueen mutator. Both options trade in some execution speed for better coverage but will require further modifications in the WAVM runtime.

7 CONCLUSION

WAFL instruments unmodified WebAssembly binaries during the AOT compilation step of WAVM, by applying existing AFL++ LLVM passes. We integrate AFL++ optimizations, like shared-memory fuzzing, into the runtime to profit from them, even though the application under test is running in a sandboxed scenario.

Our evaluation shows that the performance is excellent. Compared with native targets built from source, we discovered surprising results: For small targets the lightweight WAVM snapshots **outperform** native AFL x86-64 harnesses compiled from source, if they rely on the slow `fork` syscall. This is true for any non-persistent-mode target.

WAFL solves a problem so far unsolved: how to uncover bugs in compiled WebAssembly binaries. The need for a binary-only fuzzer is evident, as prior work has shown that a majority of binaries in the wild are potentially vulnerable [5]. WAFL is the first tool to fuzz binary-only WebAssembly targets.

Availability

WAFL is available open-source at https://github.com/fgsect/WAFL.

ACKNOWLEDGMENTS

We would like to thank our shepherd Marcus Niemietz for his valuable feedback and assistance.

REFERENCES

[1] Frank Denis. 2021. Benchmark of WebAssembly runtimes - 2021 Q1. https://00f.net/2021/02/22/webassembly-runtimes-benchmarks/

[2] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[3] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Barcelona Spain, 185–200. https://doi.org/10.1145/3062341.3062363

[4] Pat Hickey, Jakub Konka, Dan Gohman, Sam Clegg, Andrew Brown, Alex Crichton, Lin Clark, Colin Ihrig, Peter Huene, YAMAMOTO Yuji, Denis Vasilik, Josh Triplett, Sergey Rubanov, Syrus Akbary, Mike Frysinger, Aaron Turner, Alon Zakai, Andrew Mackenzie, Benjamin Brittain, Casper Beyer, David McKay, Leon Wang, Marcin Mielniczuk, Mendy Berger, PTrottier, Piotr Sikora, Till Schneidereit, Katelyn Martin, and Nasso. 2020. WebAssembly/WASI: snapshot-01. https://doi.org/10.5281/ZENODO.4323447

[5] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021*. ACM, Ljubljana Slovenia, 2696–2708. https://doi.org/10.1145/3442381.3450138

[6] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *Proceedings 2018 Workshop on Binary Analysis Research*. Internet Society, San Diego, CA. https://doi.org/10.14722/bar.2018.23014

[7] Daehee Jang and Ammar Askar. 2020. FuzzCoin: A Digital Currency with Fuzzing as a Proof-of- Work. https://fuzzcoin.kr

[8] Denis Kolodin, Henry Zimmerman, Justin Starry, and Simon. 2021. Yew: Rust / Wasm framework for building client web apps. https://github.com/yewstack/yew

[9] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[10] Steven Massey and Volodymyr Shymanskyy. 2021. wasm3: The fastest WebAssembly interpreter. https://github.com/wasm3/wasm3

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

[12] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1393–1403.

[13] Microsoft. 2021. Blazor: Build client web apps with C#. https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor

[14] Tim Newsham and Jesse Hertz. 2016. Project Triforce: Run AFL on Everything! https://web.archive.org/web/20160627201122/https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/ archived from the original.

[15] Árpád Perényi and Jan Midtgaard. 2020. Stack-Driven Program Generation of WebAssembly. In *Programming Languages and Systems*, Bruno C. d. S. Oliveira (Ed.). Vol. 12470. Springer International Publishing, Cham, 209–230. https://doi.org/10.1007/978-3-030-64437-6_11 Series Title: Lecture Notes in Computer Science.

[16] Andreas Rossberg. 2019. *WebAssembly Core Specification*. Technical Report. W3C. https://www.w3.org/TR/wasm-core-1/

[17] Andrew Scheidecker. 2021. WebAssembly Virtual Machine. https://github.com/WAVM/WAVM

[18] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2597–2614. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[19] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[20] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2541–2557. https://www.usenix.org/conference/usenixsecurity20/presentation/song

[21] The Clang Team. 2021. LLVM SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html

[22] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[23] Wasmer, Inc. 2019. wapm is the WebAssembly Package Manager. https://wapm.io/

[24] Wasmer, Inc. 2021. wasmer: The leading WebAssembly Runtime. https://github.com/wasmerio/wasmer

[25] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Los Angeles CA USA, 53–65. https://doi.org/10.1145/3167082

[26] WebAssembly Community Group. 2020. wasi-libc: WASI libc implemenation for WebAssembly. https://github.com/WebAssembly/wasi-libc

[27] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Dallas Texas USA, 2313–2328. https://doi.org/10.1145/3133956.3134046

[28] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '11*. ACM Press, Portland, Oregon, USA, 301. https://doi.org/10.1145/2048147.2048224

[29] Michał Zalewski. 2017. American Fuzzy Lop. Technical Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt