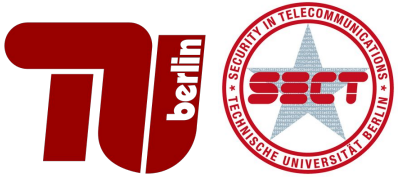
WAFL

Keno Haßler & Dominik Maier

Binary-Only WebAssembly Fuzzing with Fast Snapshots



Reversing and Offensive-oriented Trends Symposium 2021 (ROOTS)

Overview

1. WebAssembly Intro
2. Fuzzing Intro
3. Evolution of WAFL
4. Evaluation
5. Demo

WebAssembly

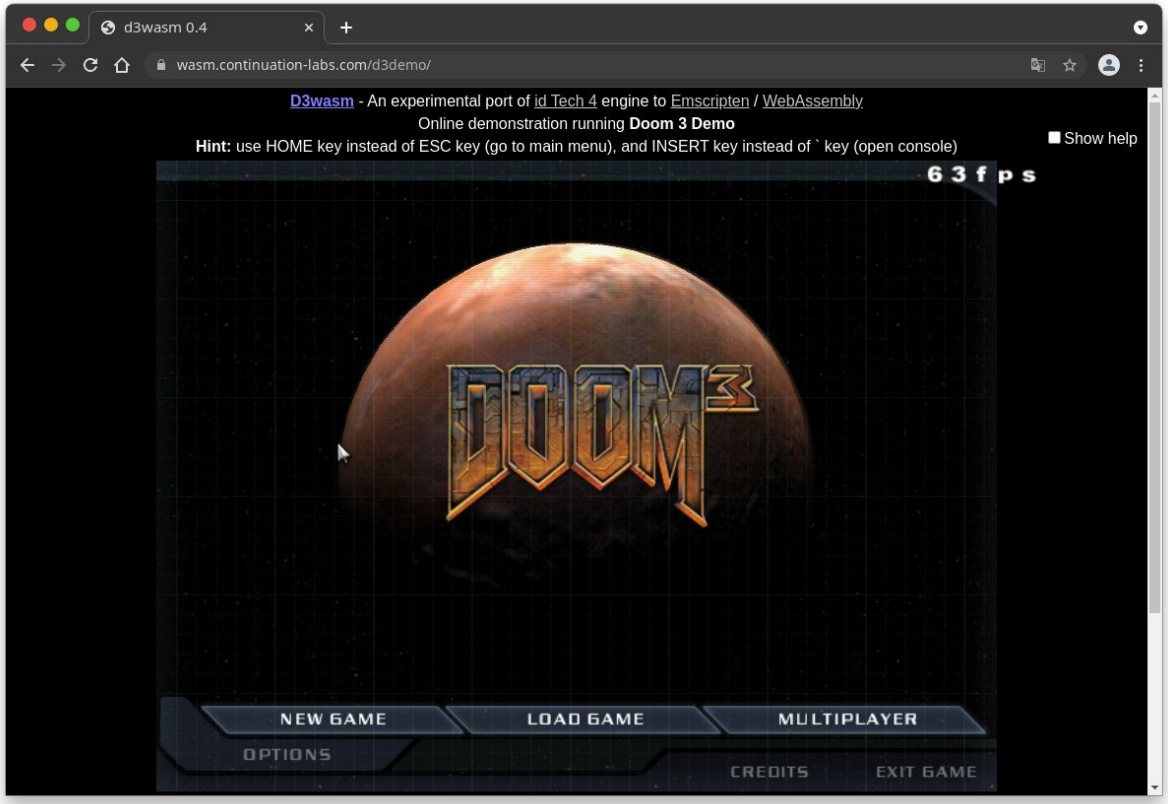
WebAssembly

JS is slow
⇒ computation intensive workloads
(encoding/decoding, crypto, …) are slow

Solution:

- Write stuff in low-level languages
- Compile for a safe, portable, performant VM
- Profit

Does it run Doom?



Yes, it does!

(quite well, actually)

Ported by Continuation Labs

(https://www.continuation-labs.com/projects/d3wasm/)

WebAssembly System Interface

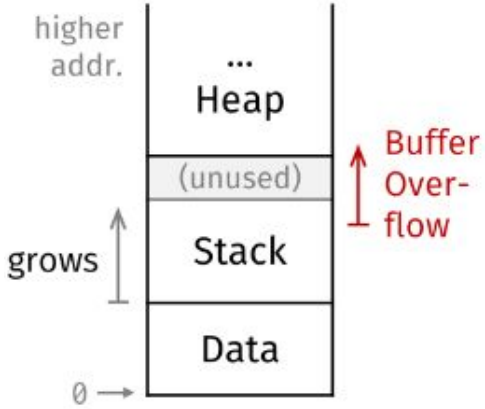
Standalone binaries possible (think Java):

- standardized system interface
- wasi-libc: POSIX-like API
- compile standard CLI tools to WebAssembly

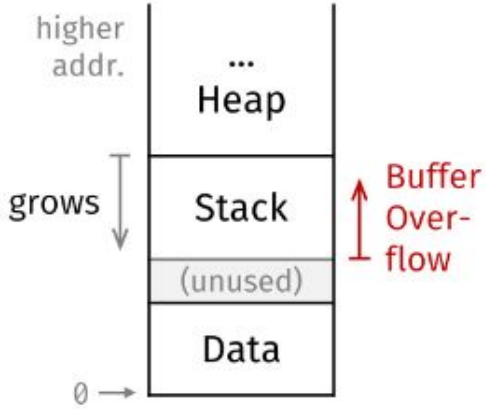
WebAssembly | Technical Overview

- Stack machine
  - No registers
  - Globals, locals, linear memory
- Compilers create stack, heap, data sections
  - Unmanaged, no VM-based protections
  - Returns are safe

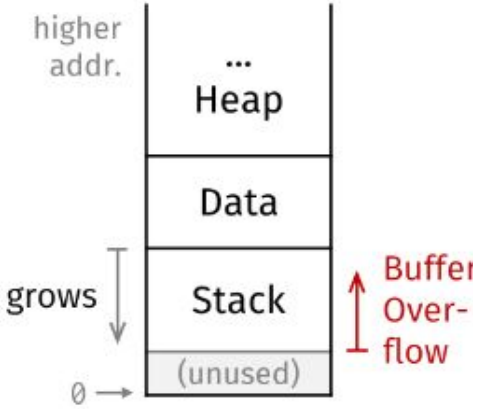
WebAssembly | Memory Organization



(a) emcc 1.39.7 (*fastcomp* backend, deprecated).



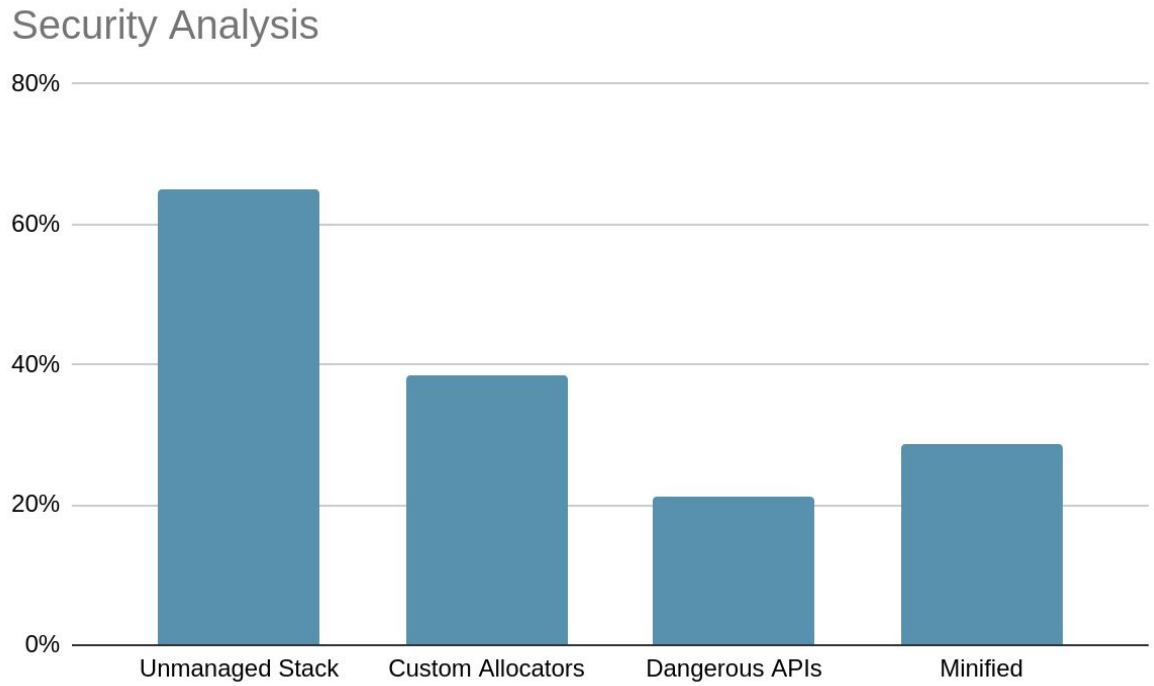
(b) emcc 1.39.7 (*upstream* backend), clang 9 (WASI).



(c) clang 9 (WASI with `stack-first`), rustc 1.41 (WASI).

Image Source: Lehmann et al. *Everything Old is new Again* (2020)

WebAssembly | In the wild



Source: Hilbig et al. *An Empirical Study of Real-World WebAssembly Binaries* (2021)

Motivation

- WebAssembly widely used
  - riddled with bugs from source langs
- Minimal binary security tooling
- We need a Fuzzer for unknown binaries!

Fuzzing

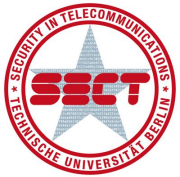
TL;DR: Throw corner-case input at a program until it breaks.

Fuzzing | Intro

- Fuzzing idea: random input may trigger corner cases
- Greybox fuzzing: use **coverage feedback** to reach more code
- Our focus it on AFL++

Fuzzing | AFL 101

- Instrument source code with compiler
  - insert feedback mechanism, shared mem
- execute program
- repeat



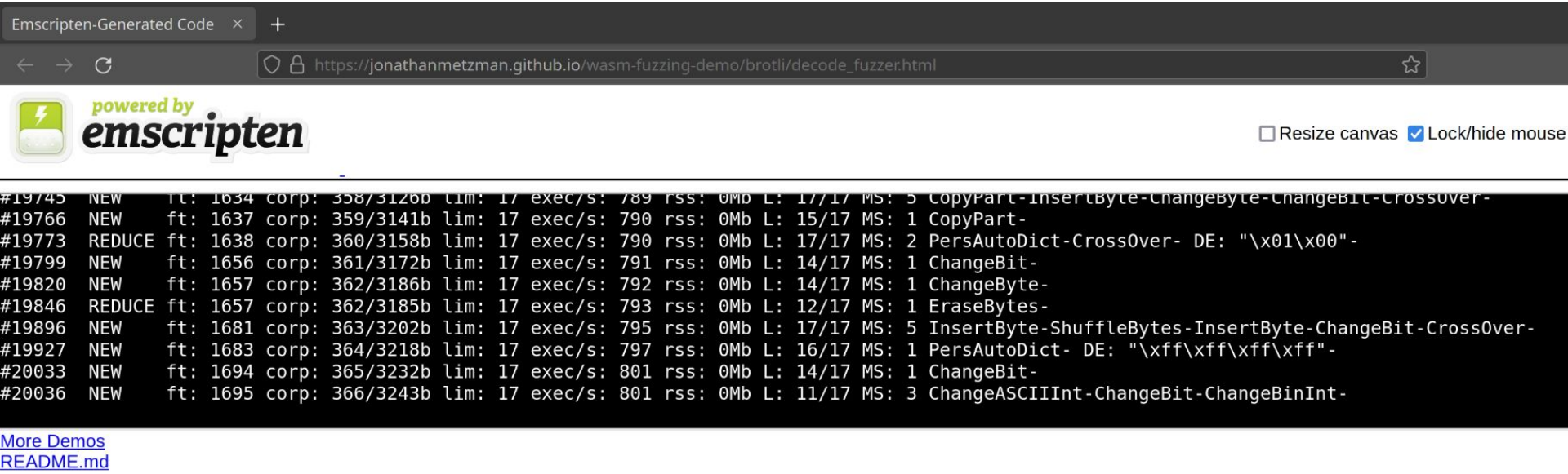
Fuzzing | AFL's Improvement Strategies

- Instrumentation: less edges (InsTrim, SanCov)
  - improved hard case handling (e.g. comparisons)
- fork server instead of execve
- *Persistent mode* instead of fork

Fuzzing | WebAssembly SOTA

- Metzman: instrument source code using libFuzzer
- Result: OSS-Fuzz programs fuzzing in the browser
- Drawback: needs source

Fuzzing | WebAssembly SOTA



Motivation

- We want to fuzz real-world Wasm binaries
  - without source code access
  - at a reasonable performance level

WAFL

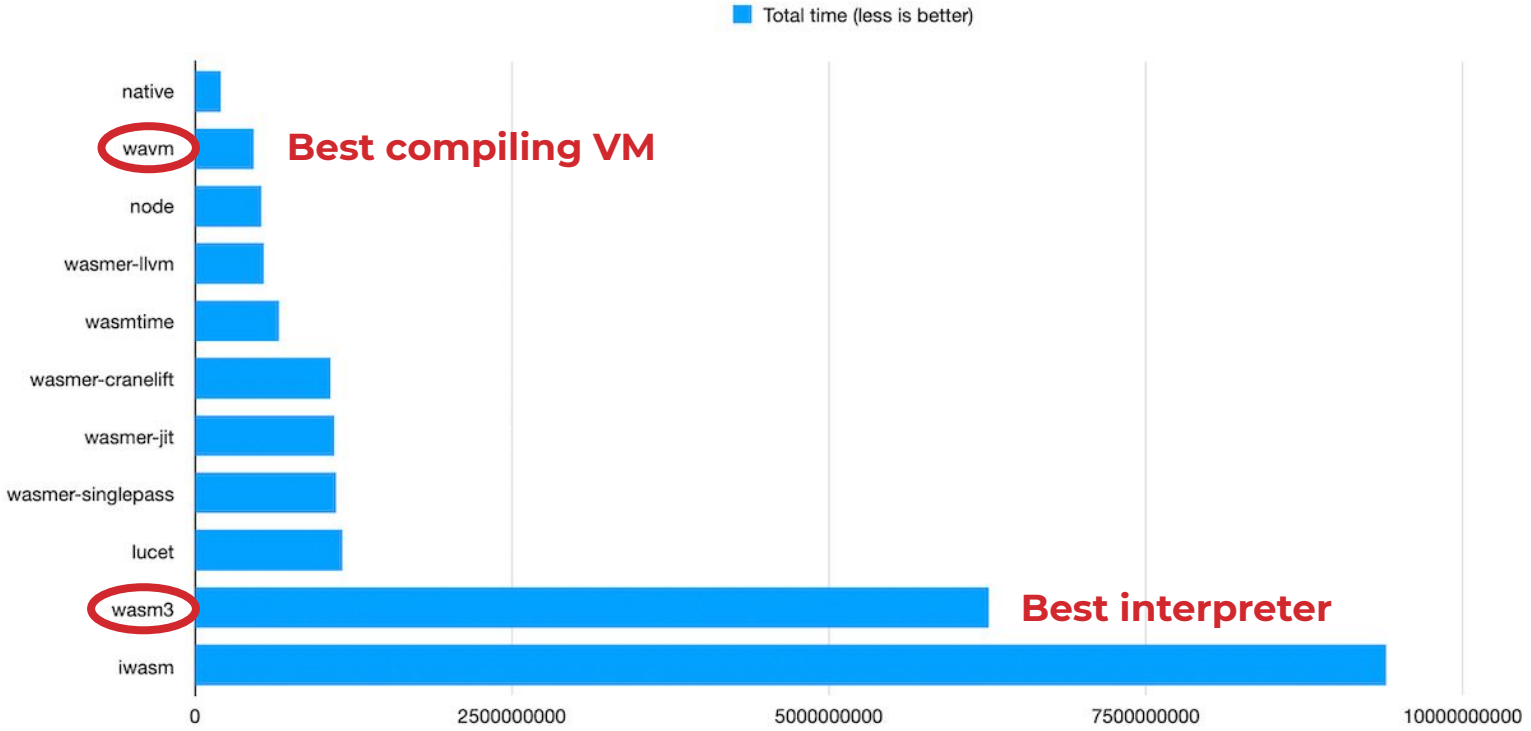
Binary-only WebAssembly Fuzzing with Fast Snapshots

First Steps

We need

1. a performant Wasm VM
2. to insert instrumentation code
3. to communicate with AFL

WebAssembly VM performance



Graph & Data by Frank Denis (https://00f.net/2021/02/22/webassembly-runtimes-benchmarks/)

Inserting Instrumentation 1

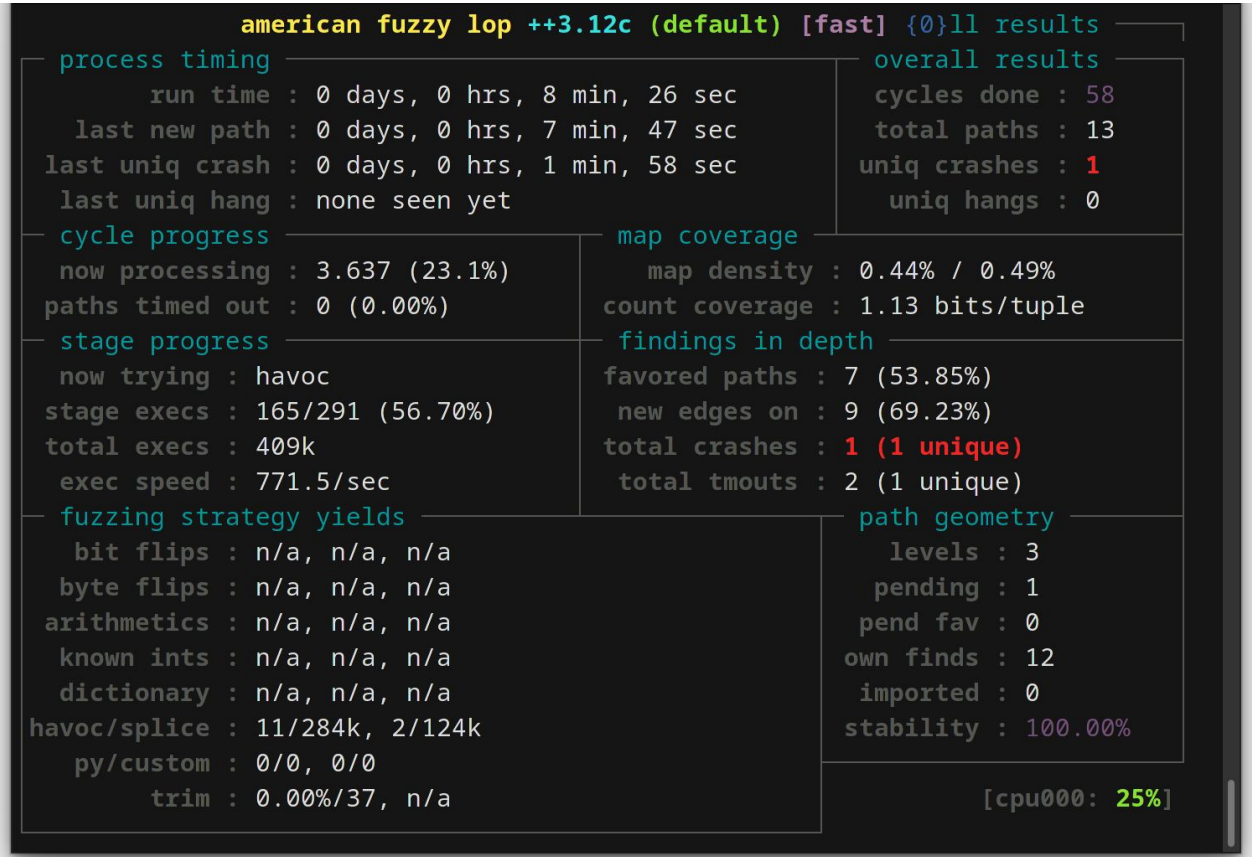
- Straightforward for interpreted VMs
  - Modify implementation for *control instructions*
  - Create coverage information

```
if (condition) {
    afl_maybe_log(branch); // log the branch target address
    return jumpOp (branch);
} else {
    afl_maybe_log(_pc + 1); // log the next operation
    nextOp ();
}
```

“Hello, AFL”

- Insert into VM startup:
  - Function to map AFL’s shared memory
  - *Fork server* code

It works!


american fuzzy lop ++3.12c (default) [fast] {0}ll results
process timing
run time : 0 days, 0 hrs, 8 min, 26 sec
last new path : 0 days, 0 hrs, 7 min, 47 sec
last uniq crash : 0 days, 0 hrs, 1 min, 58 sec
last uniq hang : none seen yet
overall results
cycles done : 58
total paths : 13
uniq crashes : 1
uniq hangs : 0
cycle progress
now processing : 3.637 (23.1%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.44% / 0.49%
count coverage : 1.13 bits/tuple
stage progress
now trying : havoc
stage execs : 165/291 (56.70%)
total execs : 409k
exec speed : 771.5/sec
findings in depth
favored paths : 7 (53.85%)
new edges on : 9 (69.23%)
total crashes : 1 (1 unique)
total tmouts : 2 (1 unique)
fuzzing strategy yields
bit flips : n/a, n/a, n/a
byte flips : n/a, n/a, n/a
arithmetics : n/a, n/a, n/a
known ints : n/a, n/a, n/a
dictionary : n/a, n/a, n/a
havoc/splice : 11/284k, 2/124k
py/custom : 0/0, 0/0
trim : 0.00%/37, n/a
path geometry
levels : 3
pending : 1
pend fav : 0
own finds : 12
imported : 0
stability : 100.00%
[cpu000: 25%]

Can we do better?

Status quo:

- Interpreted runtime
- Fork syscalls
- Inputs passed via FS

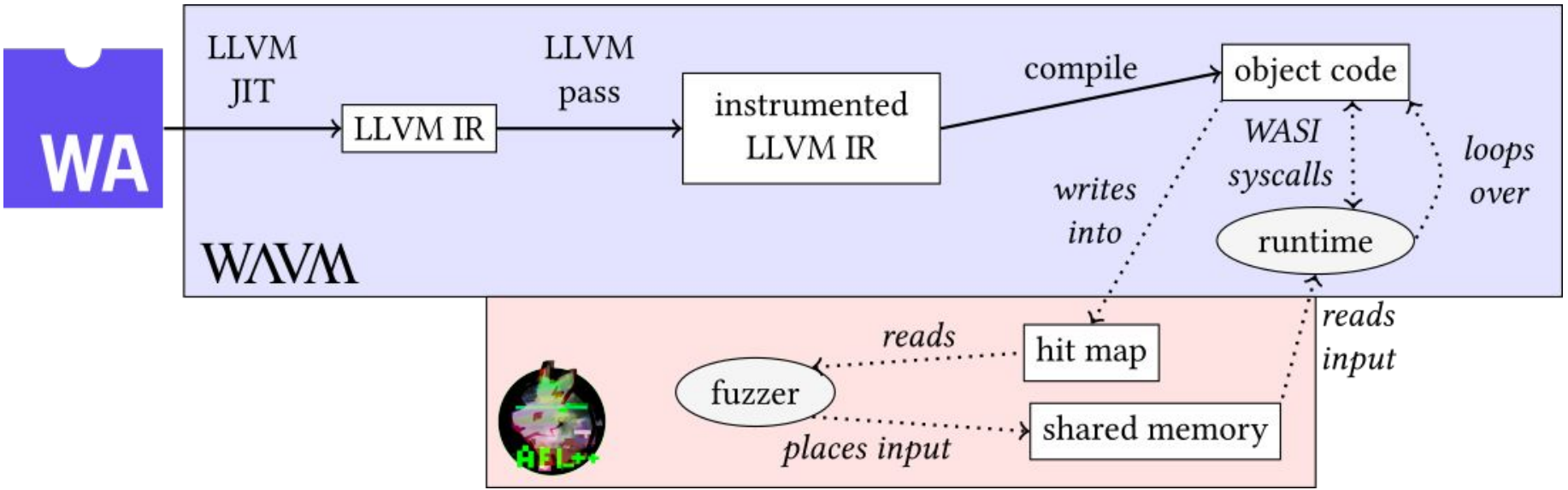
WAVM internals

- LLVM-JIT based, currently fastest runtime
- Ahead-of-time compilation:
- WebAssembly -> LLVM IR -> native code
  - Still needs the WAVM runtime, not standalone

Using WAVM

- Re-use the startup and fork server code
- How to instrument?
  - Runtime uses LLVM optimizer passes
  - AFL compilers use LLVM optimizer passes

WAFL: high-level overview



Inserting Instrumentation 2

- Can use standard afl-llvm-pass!
  - run it after optimization passes
- Need to provide data structures (shared map)
- works!

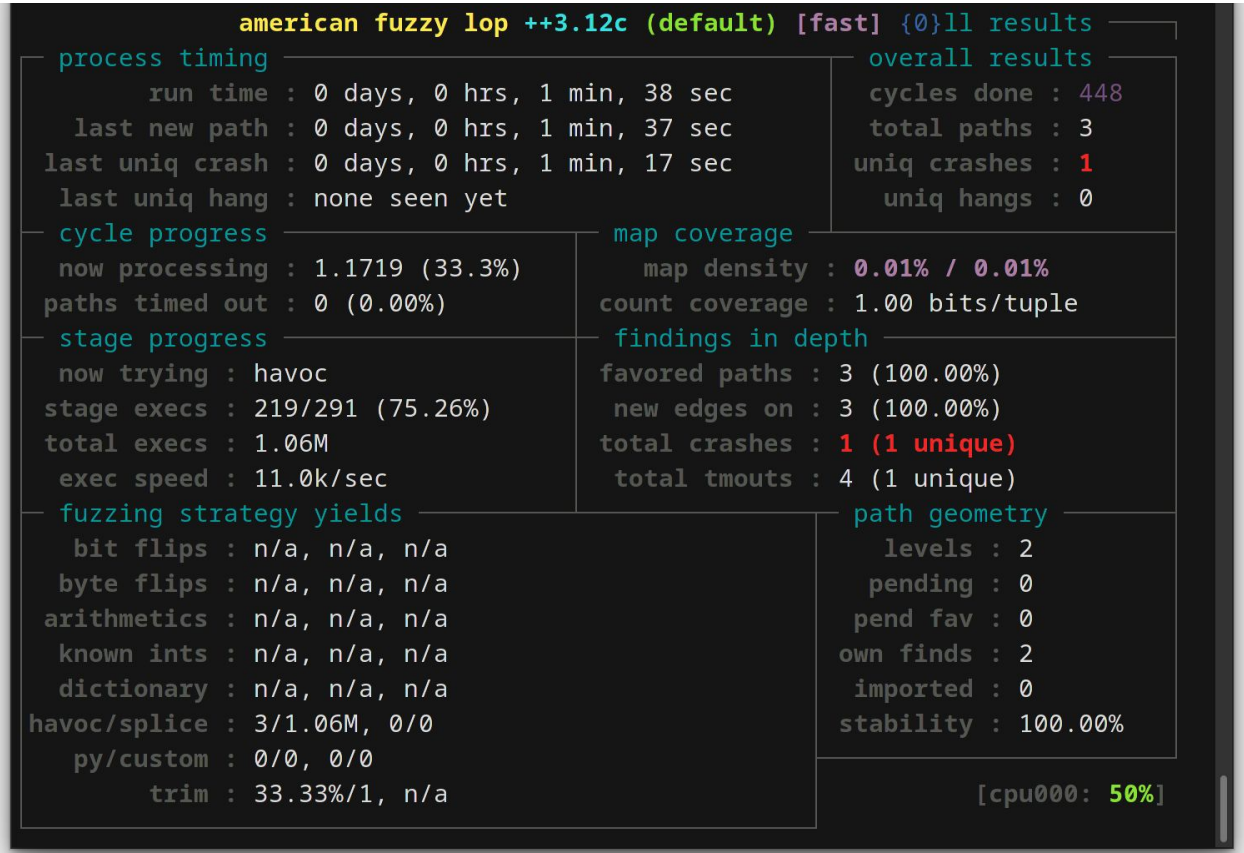
Improving Performance

- We need to get rid of forking
- AFL *persistent mode* fits our application
  - Loop over interesting code region
  - Reset the runtime

Snapshot & Reset

- Hard part: proper VM reset
- Could just reinitialize -> too costly!
- sufficient to reset only WebAssembly Memory
  - backup after initialization
  - restore on every loop iteration

Boom!


american fuzzy lop ++3.12c (default) [fast] {0}ll results
process timing
run time : 0 days, 0 hrs, 1 min, 38 sec
last new path : 0 days, 0 hrs, 1 min, 37 sec
last uniq crash : 0 days, 0 hrs, 1 min, 17 sec
last uniq hang : none seen yet
overall results
cycles done : 448
total paths : 3
uniq crashes : 1
uniq hangs : 0
cycle progress
now processing : 1.1719 (33.3%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.01% / 0.01%
count coverage : 1.00 bits/tuple
stage progress
now trying : havoc
stage execs : 219/291 (75.26%)
total execs : 1.06M
exec speed : 11.0k/sec
findings in depth
favored paths : 3 (100.00%)
new edges on : 3 (100.00%)
total crashes : 1 (1 unique)
total tmouts : 4 (1 unique)
fuzzing strategy yields
bit flips : n/a, n/a, n/a
byte flips : n/a, n/a, n/a
arithmetics : n/a, n/a, n/a
known ints : n/a, n/a, n/a
dictionary : n/a, n/a, n/a
havoc/splice : 3/1.06M, 0/0
py/custom : 0/0, 0/0
trim : 33.33%/1, n/a
path geometry
levels : 2
pending : 0
pend fav : 0
own finds : 2
imported : 0
stability : 100.00%
[cpu000: 50%]

More improvement 1

- Input is still passed via FS -> Kernel
- Want to use the shared memory
  - Modify the readv WASI implementation
  - if (fd == 0) then read_from_shmem

More improvement 2

- Other AFL passes preferred upstream
- We can integrate InsTrim
- AFL SanitizerCoverage not working
  - Vanilla LLVM version works with a small hack*

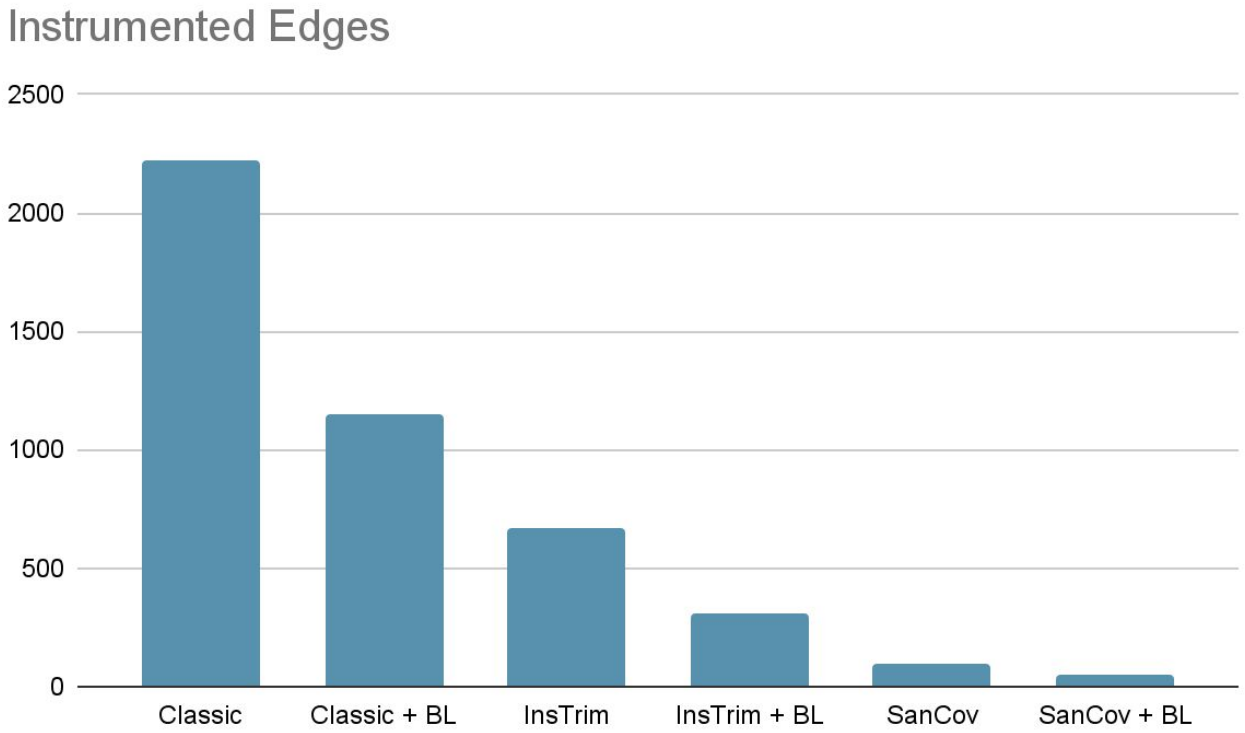
*may cost some performance

More improvement 3

- We instrument libc, it’s part of the binary
  - probably not main interest
- Non-stripped binaries: can identify function names
  - we provide a blocklist*

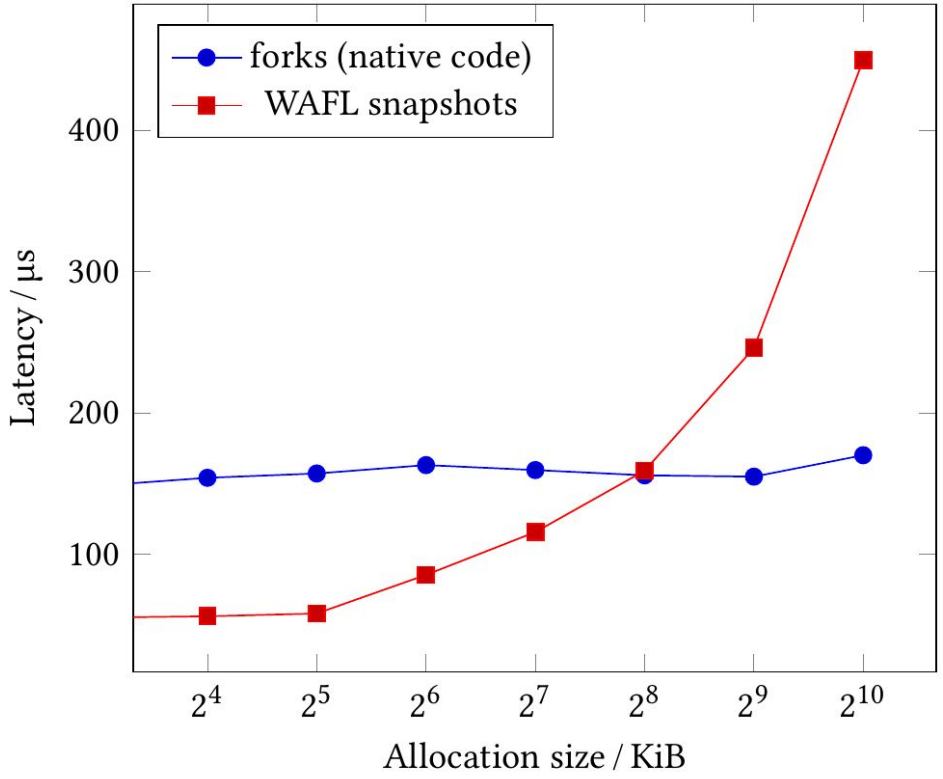
*derived from wasi-libc symbol list

Result (brotli)



Evaluation

Snapshot efficiency

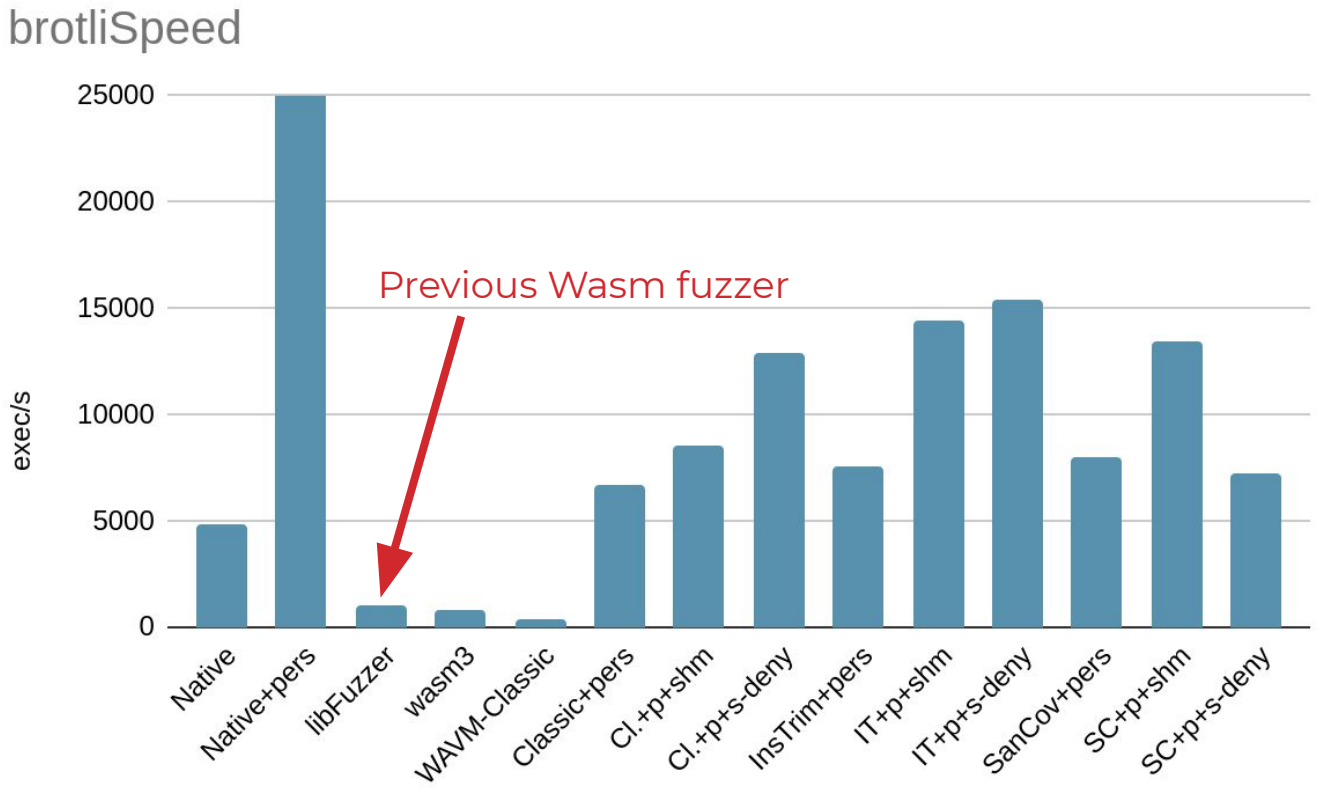


- depends on allocation size (memory copy)
- if < 256KiB:
  - faster than fork()

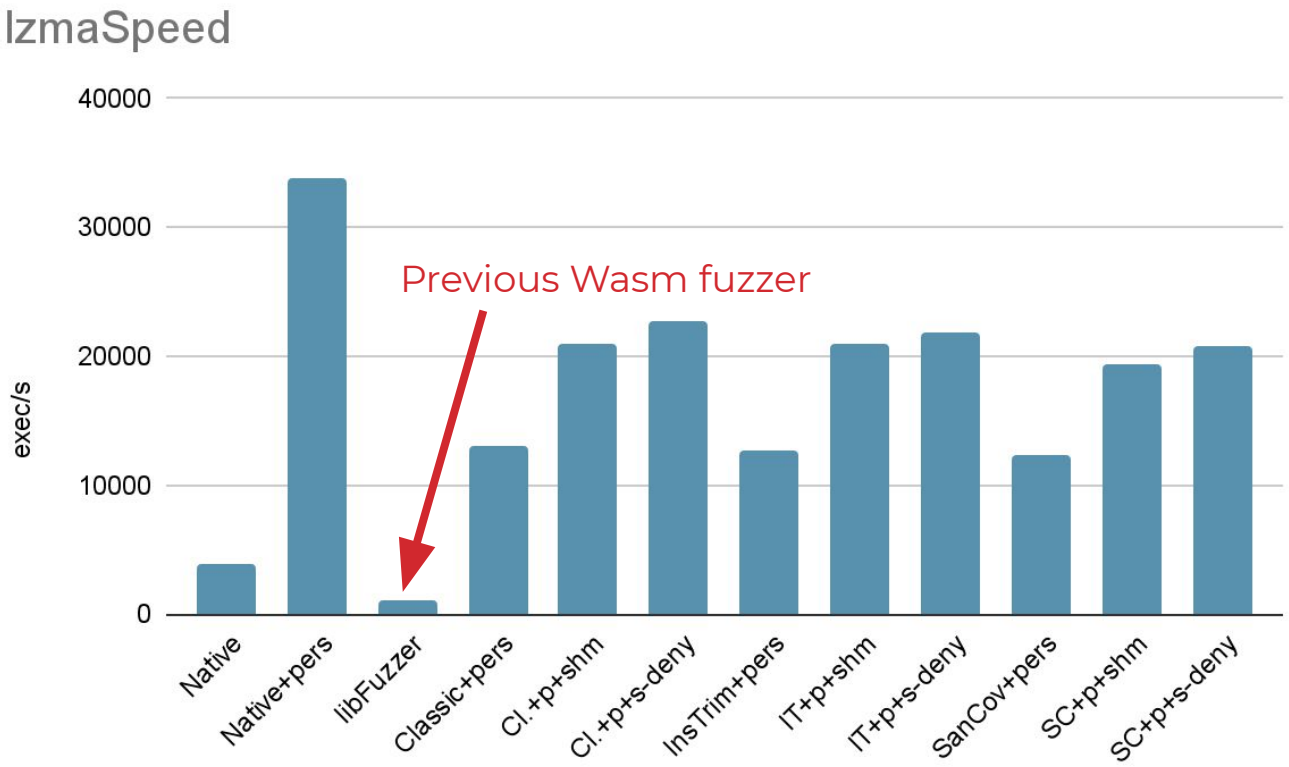
Test setup

- We test lots of configurations
  - Persistent, shmem, BL, different instrumentation
  - Also, native binaries for comparison
- No changes to the fuzzer
  - **Short** fuzzing runs, evaluating raw **speed**

Speed Test 1


brotliSpeed
exec/s
25000
20000
15000
10000
5000
0
Previous Wasm fuzzer
Native
Native+pers
libFuzzer
wasm3
WAVM-Classic
Classic+pers
Cl.+p+shm
Cl.+p+s-deny
InsTrim+pers
IT+p+shm
IT+p+s-deny
SanCov+pers
SC+p+shm
SC+p+s-deny

Speed Test 2


lzmaSpeed
40000
30000
20000
10000
0
exec/s
Previous Wasm fuzzer
Native
Native+pers
libFuzzer
Classic+pers
Cl.+p+shm
Cl.+p+s-deny
InsTrim+pers
IT+p+shm
IT+p+s-deny
SanCov+pers
SC+p+shm
SC+p+s-deny

Bottom Line

- **WAFL >> native** (forking harness)
- **Snapshots** are a must-have
- SanitizerCoverage underperforms
  - Is preferred upstream -> our problem
  - Potential for optimization

Conclusion

- First binary-only WebAssembly fuzzer
- Fast snapshots make high performance
- Can profit from further AFL++ optimizations

- Open-source code available at

  https://github.com/fgsect/WAFL

```
while (questions());

char buf[16];
strncpy(buf, ""
    "Thank you for your attention."
    "\n", sizeof(buf));
printf("%s", buf);
```