Week 9: Lecture B

Optimization II

Wednesday, March 5, 2025

Recap: Key Dates

- **Jan. 15** — **Select one paper to present**
- **Jan. 20** — No class (MLK Jr. Day)
- **Jan. 29** — Lab 1 due
- **Feb. 05** — Lab 2 due
- **Feb. 17** — No class (President's Day)
- **Feb 19** — Lab 3 due (deadline extended)
- **Feb. 26** — **5-minute project pitches**
- **Mar. 10 & 12** — No class (Spring Break)
- **Apr. 16 & 21** — **Final project presentations**

cs.utah.edu/~snagy/courses/cs5963/schedule

Part 2: Fuzzing Enhancements

Monday Meeting	Wednesday Meeting
Mar. 03 **Optimization I** ▶ Readings:	Mar. 05 **Optimization II** ▶ Readings:
Mar. 10 No Class (Spring Break)	Mar. 12 No Class (Spring Break)
Mar. 17 **Hybrid Fuzzing I** ▶ Readings:	Mar. 19 **Hybrid Fuzzing II** ▶ Readings:
Mar. 24 **Directed Fuzzing** ▶ Readings:	Mar. 26 ⚠ In-class Project Workday

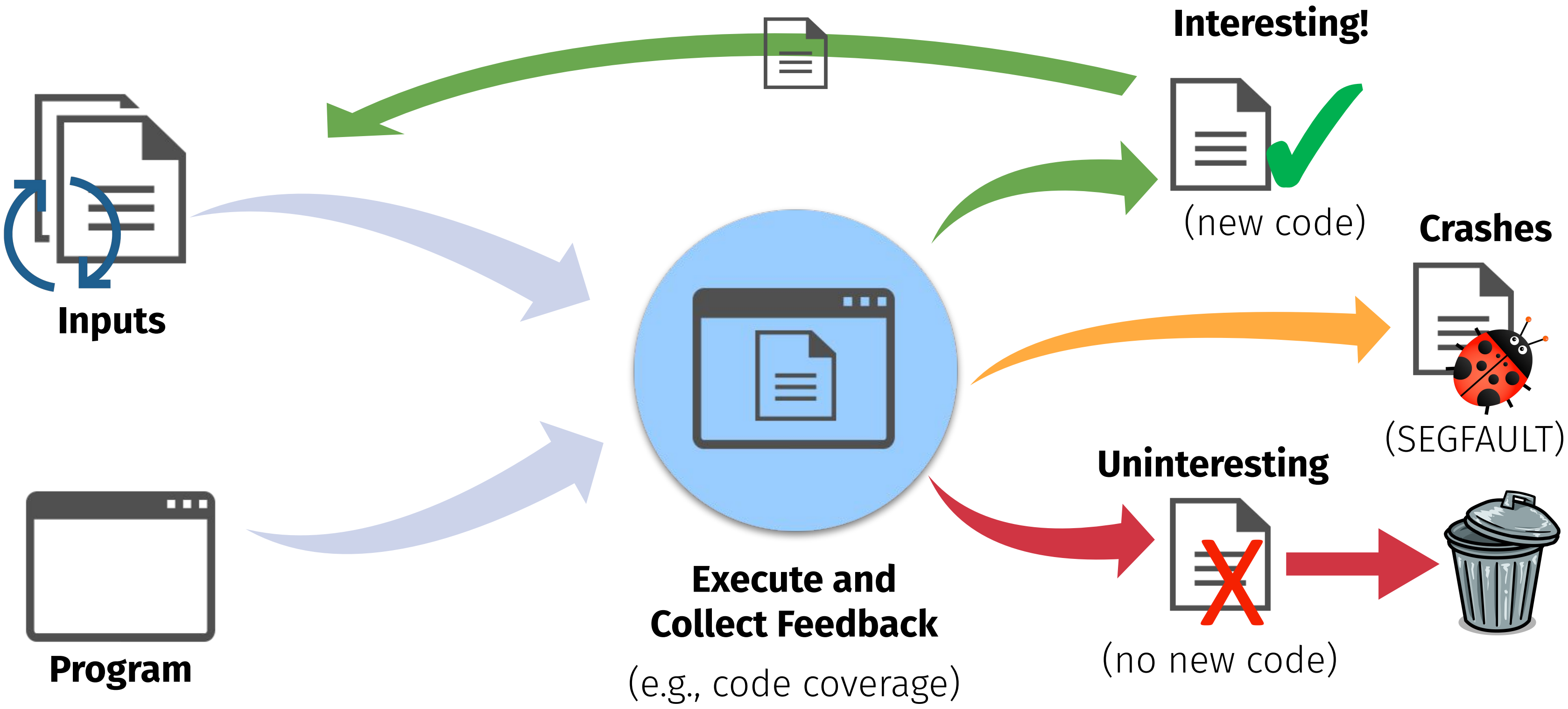
Part 3: New Frontiers in Fuzzing

Monday Meeting	Wednesday Meeting
Mar. 31 **Kernel Fuzzing** ▶ Readings:	Apr. 02 **LLM-assisted Fuzzing** ▶ Readings:
Apr. 07 **Compiler Fuzzing** ▶ Readings:	Apr. 09 **Hardware Fuzzing** ▶ Readings:
Apr. 14 **Fuzzing Configurable Software** ▶ Readings:	Apr. 16 ⚠ Final Presentations (Day 1)
Apr. 21 ⚠ Final Presentations (Day 2) ⚠ Final Reports due Tuesday by 11:59pm via Canvas	Apr. 23 No Class (Reading Day)

Questions?

Fuzzing (even) Faster

Recap: Coverage-guided Fuzzing



What affects fuzzing speed?

- **Process execution**
  - Performed on every input
- **Runtime instrumentation**
  - Code coverage tracing
- **Information post-processing**
  - Data structure writing/reading
  - Other essential computation

What affects fuzzing speed?

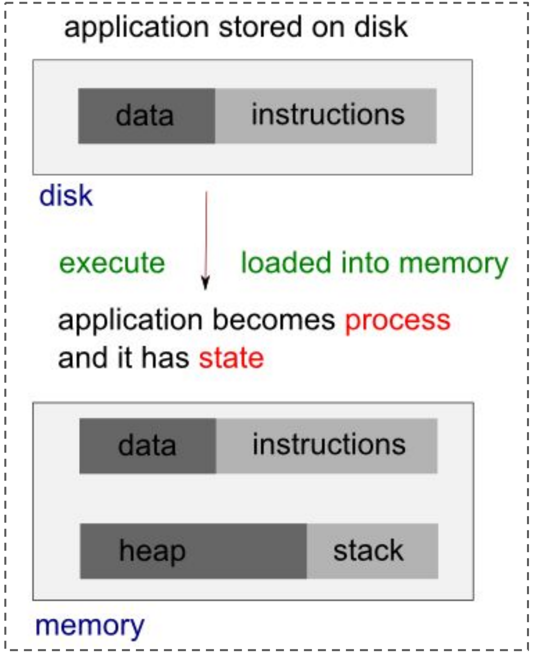
- **Process execution**
  - Performed on every input
- **Runtime instrumentation**
  - Code coverage tracing
- **Information post-processing**
  - Data structure writing/reading
  - Other essential computation

What is execution?

- Double-clicking a shortcut on your desktop
- Tapping an app icon on your smartphone
- “Hey Siri, play Midnights on Spotify”

What *really* is execution?

- **Load a program image into memory**
  - Data
  - Instructions
- **Initialize its process state**
  - Stack
  - Heap
  - Registers
  - Other data
- **Transfer control to it and execute it**
  - Clean things up when done



https://www.bogotobogo.com/Linux/linux_process_and_signals.php

How does execution impact fuzzing?

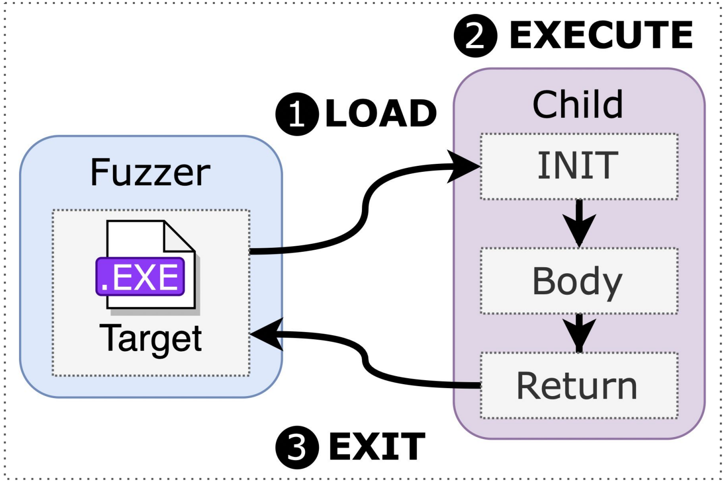
- Many **execution mechanisms** to choose from
  - Process creation
  - Forkserver-based process cloning
  - In-memory process looping
  - Kernel-based snapshotting

- **Fundamentally different behaviors**
  - Time spent within the target program
  - Underlying OS-level machinery
  - Post-execution cleanup steps
  - Support for arbitrary programs

Fuzzing Execution Mechanisms

In the early days...

- **Process Creation:**
  1. Load target image into child process
  2. Initialize child and begin executing it
  3. On exit:
     - Free child's resources
     - Wait for next test case
     - Return to step 1



Trade-offs

- **Easily the most portable mechanism**
  - Every OS has its **primitives** for this
    - POSIX: **???**

Trade-offs

- **Easily the most portable mechanism**
  - Every OS has its **primitives** for this
    - POSIX: `fork()` + `exec()`
    - Windows: **???**

Trade-offs

- **Easily the most portable mechanism**
  - Every OS has its **primitives** for this
    - POSIX: `fork()` + `exec()`
    - Windows: `CreateProcess()`

Trade-offs

- **Easily the most portable mechanism**
  - Every OS has its **primitives** for this
    - POSIX: `fork()` + `exec()`
    - Windows: `CreateProcess()`
- **By far fuzzing's slowest execution**
  - Repeatedly covers program startup code
    - E.g., Library initialization
  - Lots of underlying OS machinery
    - Process ID assignment
    - Updating kernel structures
    - *And more!*

Not all primitives are alike

- Windows: `CreateProcess()`
  - Initialize process completely from scratch
    - Expensive when done **per test case**
  - Higher cost from other kernel operations
    - **Why?** No one knows (closed-source)

	Fork()	CreateProcess
Supports PE files?		✔
Copy-on-Write?		✗
Speed (exec/sec)		91.9

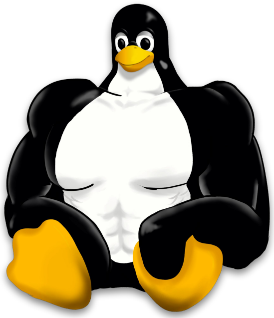
Source: "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning"

Not all primitives are alike

- Windows: `CreateProcess()`
  - Initialize process completely from scratch
    - Expensive when done **per test case**
  - Higher cost from other kernel operations
    - **Why?** No one knows (closed-source)

Fork()	CreateProcess	Linux
Supports PE files?	✔	✗
Copy-on-Write?	✗	✔
Speed (exec/sec)	91.9	4907.5

- POSIX: `fork()` + `exec()`
  - Faster from **copy-on-write process cloning**
    - Child cheaply inherits copy of parent

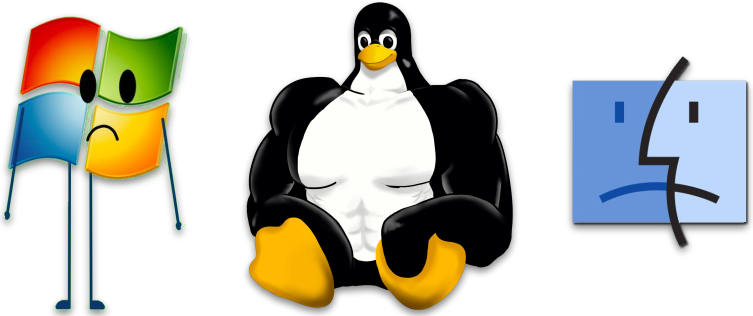


Source: "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning"

Not all primitives are alike

- Windows: `CreateProcess()`
  - Initialize process completely from scratch
    - Expensive when done **per test case**
  - Higher cost from other kernel operations
    - **Why?** No one knows (closed-source)
- POSIX: `fork()` + `exec()`
  - Faster from **copy-on-write process cloning**
    - Child cheaply inherits copy of parent
  - Somehow really slow on **MacOS**
    - **Why?** No one knows (closed-source)

Fork()	CreateProcess	Linux
Supports PE files?	✔	✗
Copy-on-Write?	✗	✔
Speed (exec/sec)	91.9	4907.5



Source: “WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning”

Can we skip target initialization entirely?

- **Idea:** fork at a **pre-initialized** state
  - After library initialization
  - After GUI initialization
  - After program-specific startup code
  - **At the program's `main()`**

- 2014: AFL's **fork-server** is born
  - By far the most popular execution mechanism used in fuzzing since

This is a personal blog. My other stuff: book | home page | Twitter | Substack

lcamtuf's old blog

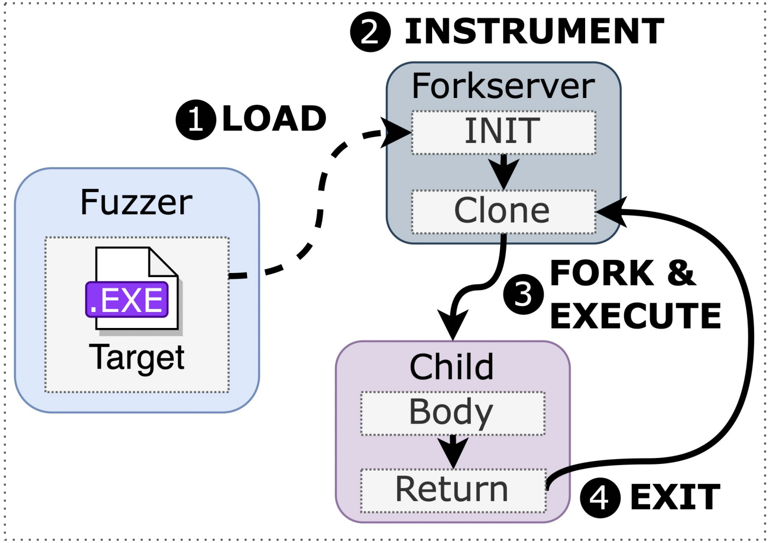
October 14, 2014

Fuzzing random programs without execve()

The most common way to fuzz data parsing libraries is to find a simple binary that exercises the interesting functionality, and then simply keep executing it over and over again - of course, with slightly different, randomly mutated inputs in each run. In such a setup, testing for evident memory corruption bugs in the library can be as simple as doing *waitpid()* on the child process and checking if it ever dies with *SIGSEGV*, *SIGABRT*, or something equivalent.

Forkserver-based Process Cloning

- General workflow:
  1. Load target by calling `fork()` + `exec()`
  2. Hook a post-initialization target routine
     - E.g., `init()` or `main()`
  3. From here, call `fork()` + `exec()` again
     - Child begins executing directly from **our hooked target routine**
     - Never repeat initialization again
  4. On exit, kill child process and repeat



Trade-offs

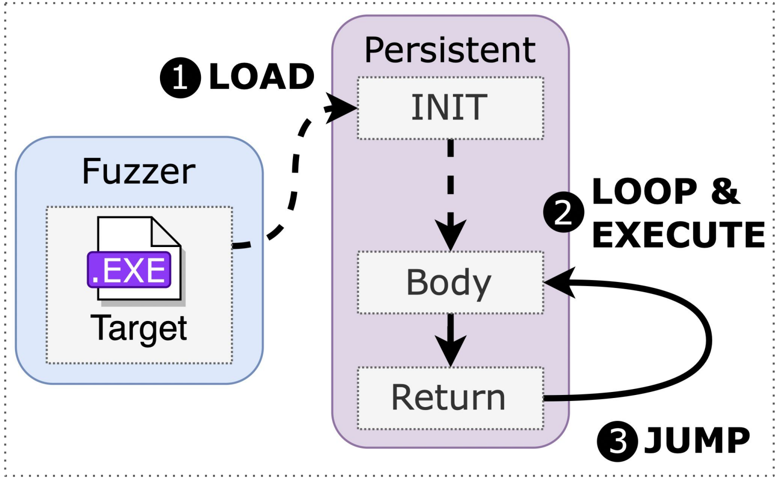
- **Skipping initialization over 100x faster**
  - Far more lightweight than process creation
  - Easy deployable via basic instrumentation
    - Both compilers and binary alteration

Trade-offs

- **Skipping initialization over 100x faster**
  - Far more lightweight than process creation
  - Easy deployable via basic instrumentation
    - Both compilers and binary alteration
- **Restricted to POSIX systems only**
  - Windows has no copy-on-write primitives
  - Stuck with Linux and MacOS
    - Yet MacOS is absurdly slow

What if we just never exited the target?

- In-memory looping ("persistent" mode):
  1. Load target by calling `fork()` + `exec()`
  2. Execute the core function you want to test
     - E.g., `main()`
     - E.g., `LLVMFuzzerTestOneInput()`
  3. Loop back to the function and repeat
     - One loop iteration per test case
     - **Never exit the program**

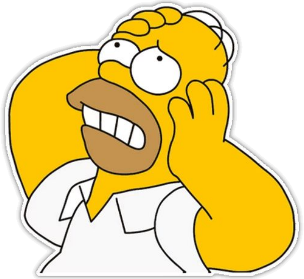


Trade-offs

- **Over 100x faster than forkserver-based cloning**
  - Avoids the cleanup cost of process teardown
  - Avoids memory duplication cost of forking

Trade-offs

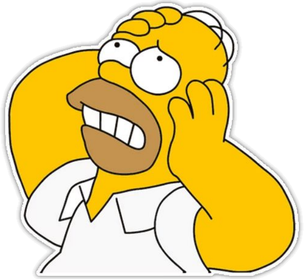
- **Over 100x faster than forkserver-based cloning**
  - Avoids the cleanup cost of process teardown
  - Avoids memory duplication cost of forking
- **No cleanup leads to corrupted process state**
  - Failure to reset global variables, heap memory, etc.
  - **Effects:** spurious and false positive crashes, leaks



Trade-offs

- **Over 100x faster than forkserver-based cloning**
  - Avoids the cleanup cost of process teardown
  - Avoids memory duplication cost of forking

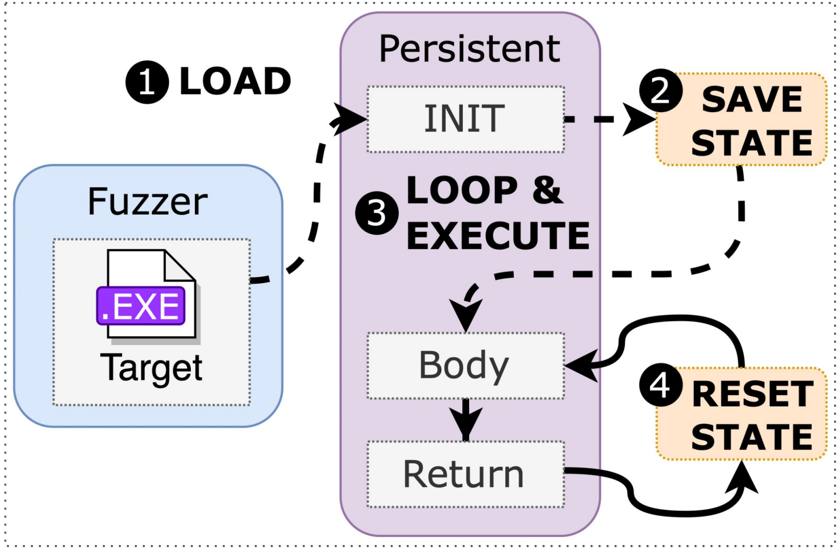
- **No cleanup leads to corrupted process state**
  - Failure to reset global variables, heap memory, etc.
  - **Effects:** spurious and false positive crashes, leaks



- **Requires significant reconnaissance of target**
  - For binaries, must choose exact addresses to loop on
  - Becomes a **harnessing** problem

Why don’t we just write better primitives?

- Kernel-based Process Snapshotting:
  0. Rewrite kernel with our faster primitives
  1. Load target process into memory
  2. Invoke our snapshot() to save its state
     - Global state
     - Register state
     - Stack and heap state
  3. Loop (same as in-memory looping)
  4. Before preparing for next test case, recover target to our snapshotted state



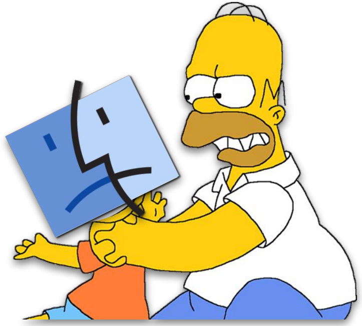
Trade-offs

- **Among the fastest execution mechanisms**
  - Comparable speed to in-memory looping
  - Without corruption of process state

Trade-offs

- **Among the fastest execution mechanisms**
  - Comparable speed to in-memory looping
  - Without corruption of process state

- **Achievable only by modifying the kernel**
  - Cannot be ported to **closed-source** kernels
    - Good luck convincing Microsoft and Apple…
    - As of now, completely restricted to Linux



Other Considerations

Does execution mechanism speed always matter?

- Profile average time spent on **target program** vs. **execution mechanism**

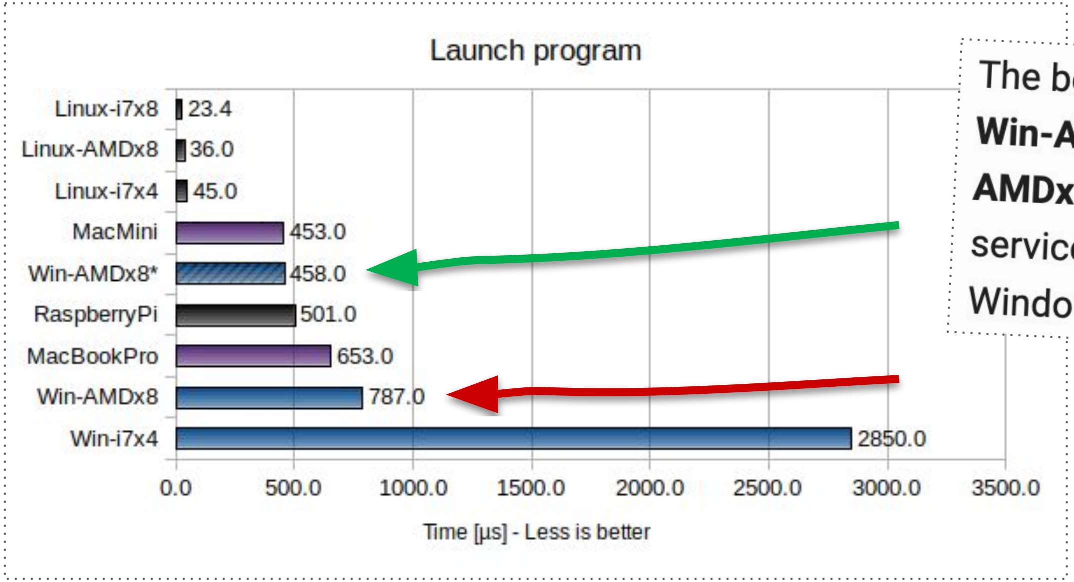
Avg. Target Time / input	Avg. Execution Time / input	Prop. spent on Execution
2 ms	1—10 ms	**33.3—83.3%**
300 ms	1—10 ms	**0.0—3.2%**

  - **Short-running** test cases = execution speed matters **more**
  - **Long-running** test cases = execution matters **less** (and **coverage tracing** matters more)

- As usual, this phenomena is **target-dependent**

Anti-virus software (and other bloatware)

- Fuzzing can be slowed by **default-on services and apps**
  - **Turn them off!**



> The best results on Windows were achieved by **Win-AMDx8***, which is the same system as **Win-AMDx8** but with most performance hogging services completely disabled (including Windows Defender and search indexing).

https://www.bitsnbites.eu/benchmarking-os-primitives/

Squeezing a few more execs/sec

- Use a **RAM disk** for even faster speeds
  - Eliminates fragmentation
  - Linux: **tempfs** or **ramfs**

- Find ways to pass **avoid file input/output**
  - Target must support reading of “streamed” data
  - libFuzzer exclusively operates this way
    - Must stitch together the requisite API calls

Questions?