Designing New Operating Primitives to Improve Fuzzing Performance

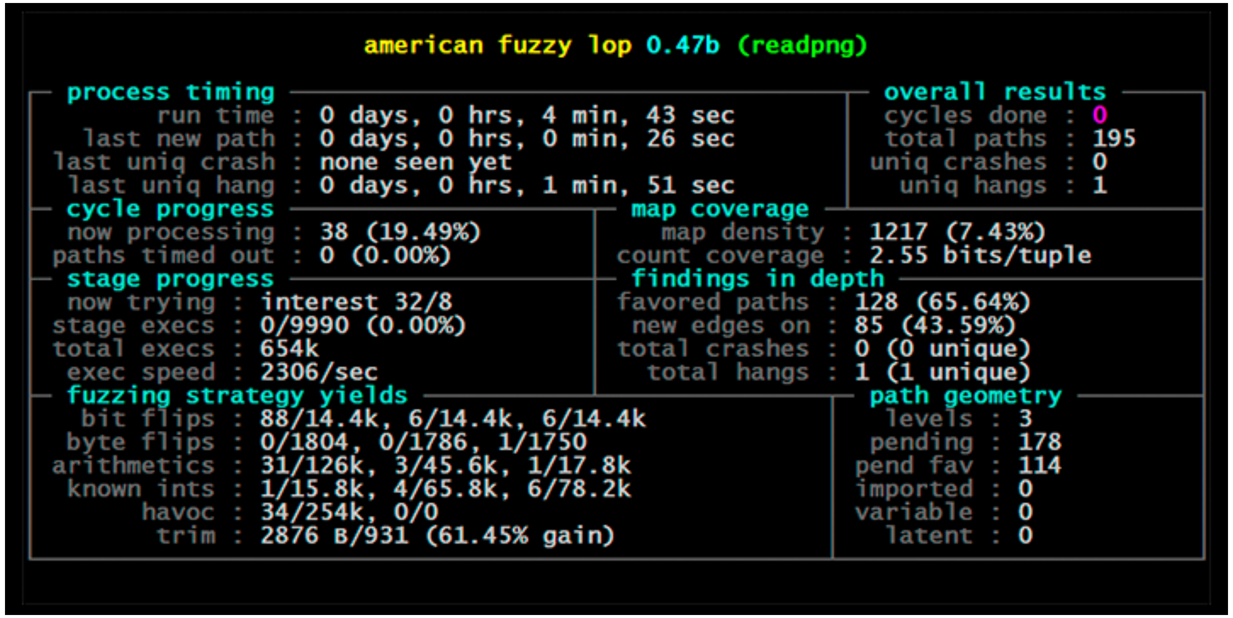
Wen Xu, Sanidhya Kashyap, Changwoo Min*, and Taesoo Kim

Georgia Tech, Virginia Tech*

ACM CCS 2017

11.02.2017

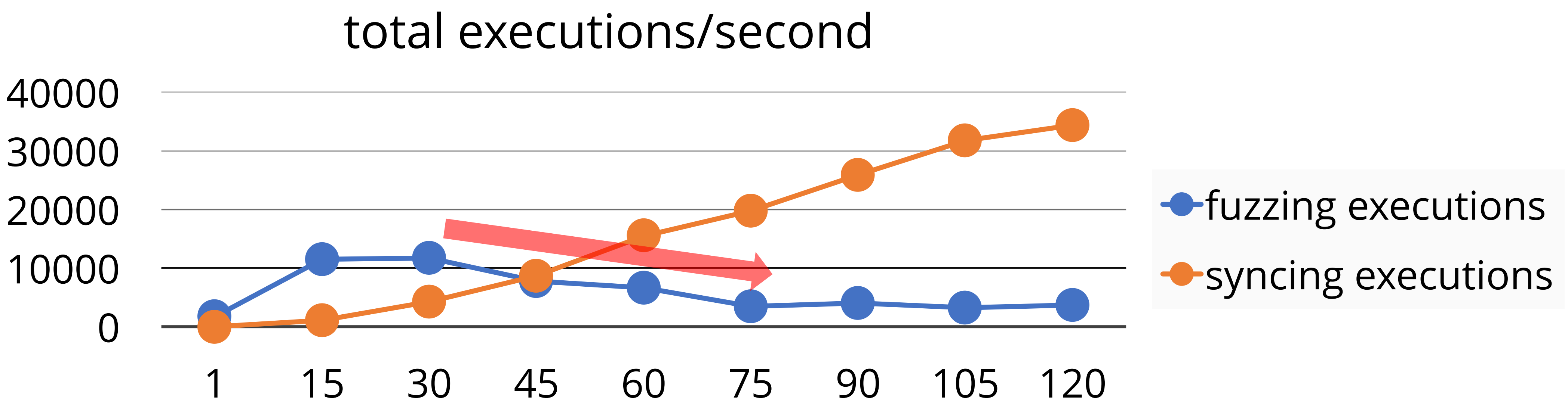
Fuzzing becomes popular

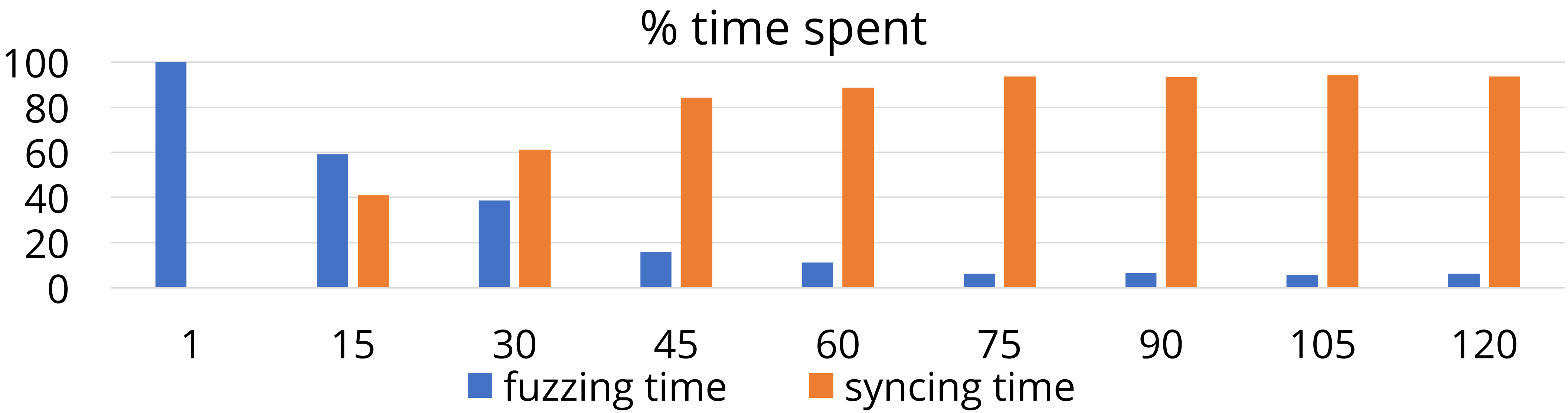


The dilemma of fuzzing

- How to produce an input that is more likely to trigger a vulnerability? (*fuzzing strategy*)
- *Our work:* How to execute more inputs within a given time? (*fuzzing performance*)
  - Save huge cost on computing resources in parallel fuzzing
  - No change in applied fuzzing strategies

Poor scalability of AFL

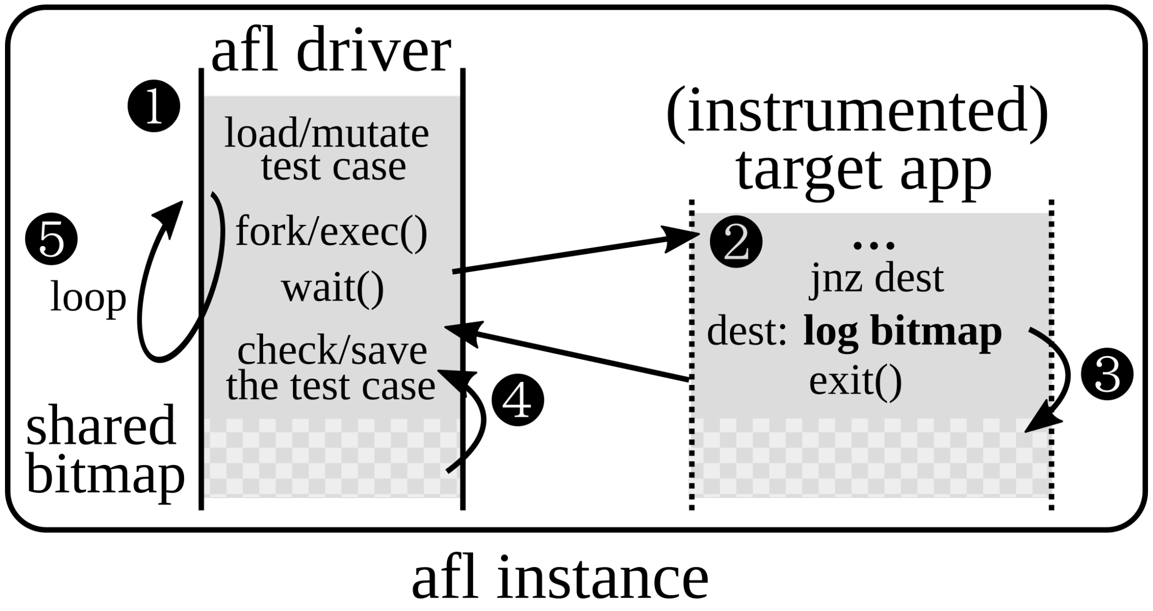




AFL explained – single instance

Repeating

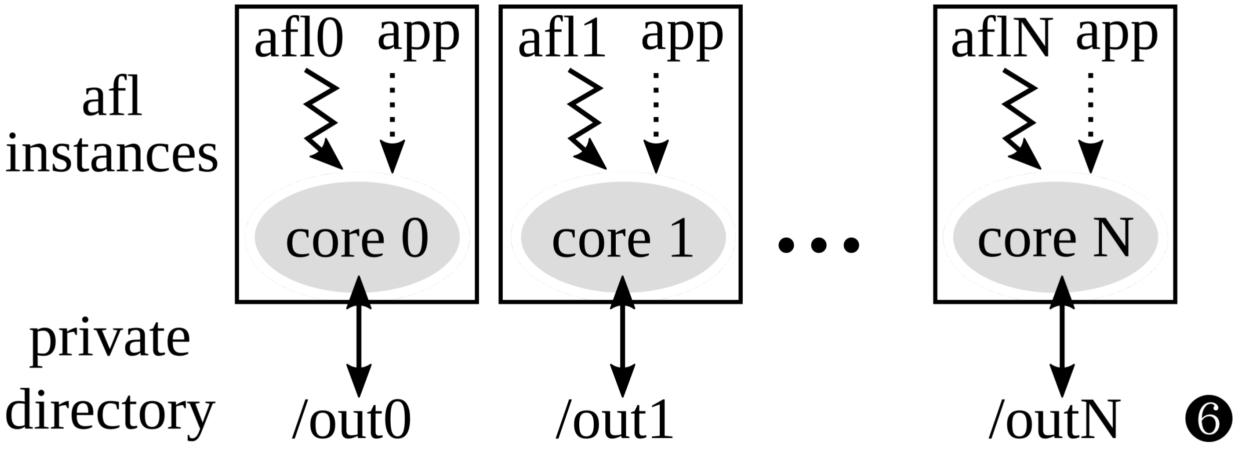
(1) Reading and mutating inputs
(2) Launching the target application
(3) Executing and recording runtime coverage
(4) Bookkeeping results



AFL explained – parallel fuzzing

Syncing phase

(1) Scanning the private directories of other fuzzer instances
(2) Executing unseen test cases
(3) Copying to own directory if interesting



Fuzzers rely on non-scalable OS primitives

- Launching the target application
  - fork()
- Reading or writing test cases on the disk
  - typical disk file system operations on small files
- Syncing test cases from other fuzzer instances
  - Directory scanning
  - fork() for test case re-executions

I. fork() to clone new target instances

- fork() is generally designed to duplicate the state of any running process
- In terms of fuzzing on multicores, fork() involves

Redundant operations

- Duplicating virtual memory space
- Duplicating files, sockets, credentials

…

Non-scalable operations

- Updating the reverse mapping
- Stressing the global memory allocator
- Scheduling the new task

…

II. Managing test cases through the disk file system

open/creat

generate test cases on the disk

↓

creating files in a private directory

write

flush interesting test cases to the disk

↓

writing a small files in a private directory

heavy modifications on the file system metadata in critical sections which are **not scalable**

III. Syncing test cases from other instances

Directory enumeration

- non-linearly increase

  number of fuzzers × number of test cases

- interfere with the running fuzzer

  Directory read and write cannot be performed concurrently

Test case re-execution

- redundant

  The runtime coverage information was achieved before

Solutions

- *General* operating primitives specialized for fuzzers
  - Snapshot() system call
    - *A lightweight, scalable fork() substitute for fuzzing*
  - Dual file system service
  - Shared in-memory test case log

Snapshot() system call

Before fuzzing,

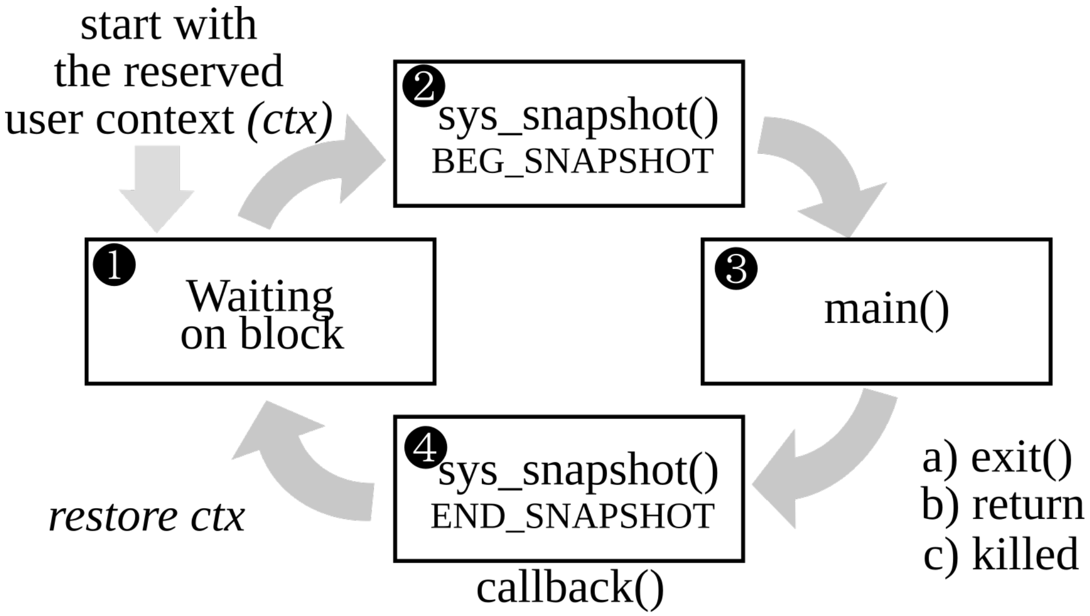
- Saving memory and file information
- Manipulating PTE of rw pages to ro (CoW)

During fuzzing,

- Demanding page copy when memory write triggers page fault handler

After fuzzing,

- Recovering memory and file information
- Recovering copied page data
- Returning to the starting point



Snapshot() system call

- Compared with fork()
  - No copies of numerous kernel data structures
  - No new stack area allocation for the new process
  - No stress on the kernel memory allocator
  - No scheduling cost

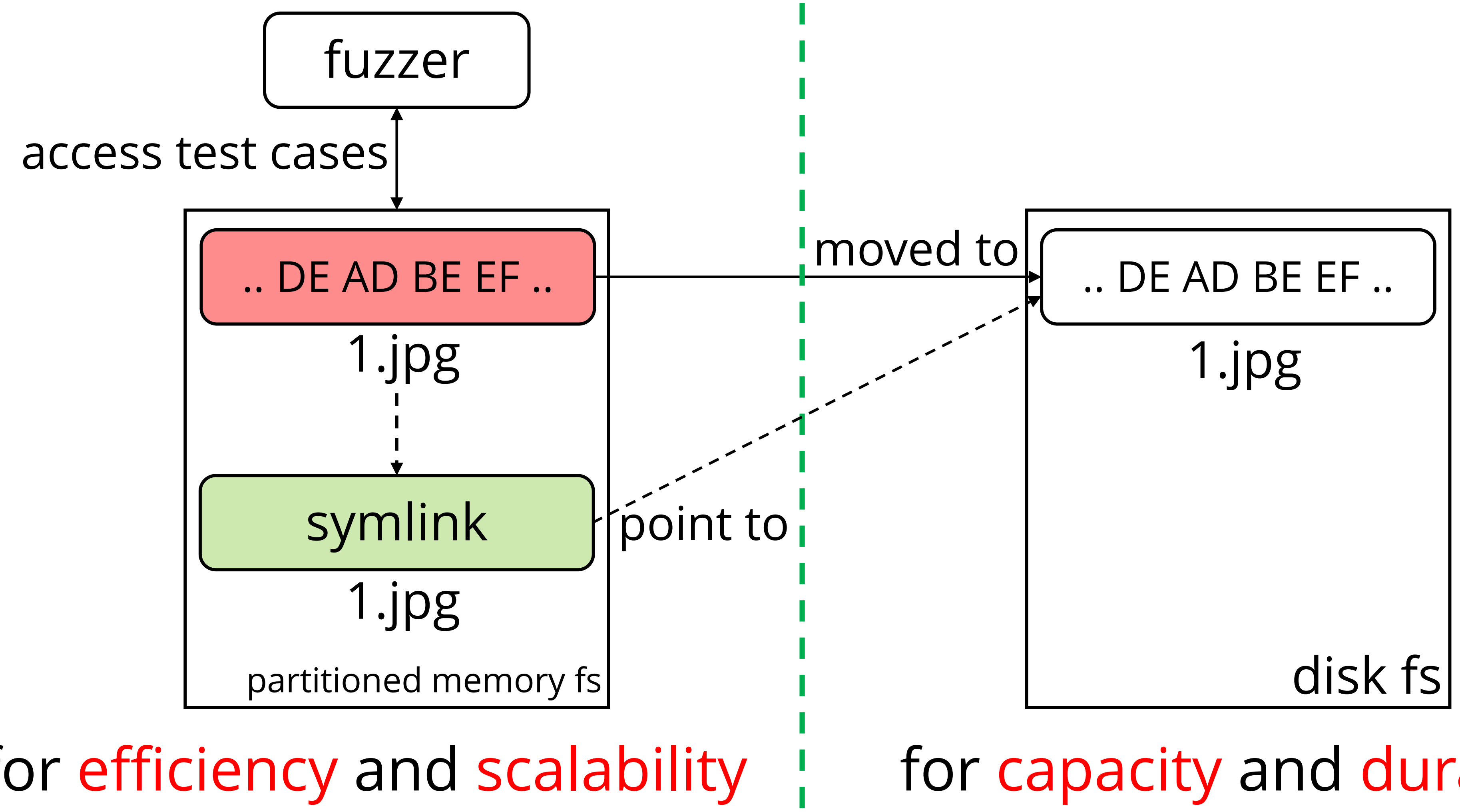
Solutions

- *General* operating primitives specialized for fuzzers
  - Snapshot() system call
  - Dual file system service
    - *A two-level tiering of file systems ensuring efficiency and deferred durability*
  - Shared in-memory test case log

Dual file system service

- *Observation:* neither the fuzzer instance nor the target instances requires strong consistency provided by the disk file system
  - Fuzzers can always reproduce lost test cases upon unexpected failures
  - We introduce memory file system and trade off between *consistent storage* and *fuzzing performance*

Dual file system service



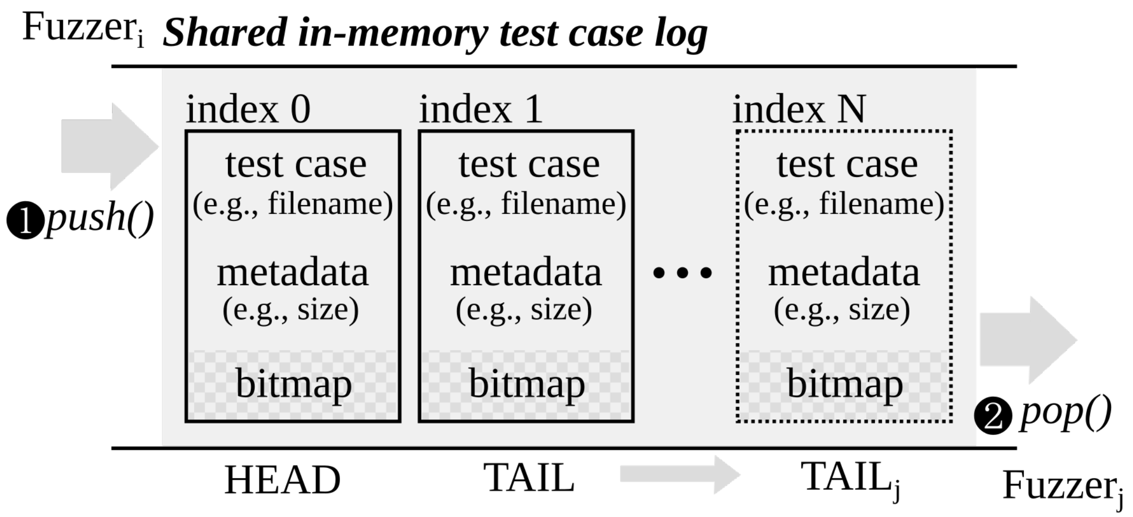
for efficiency and scalability

for capacity and durability

Solutions

- *General* operating primitives specialized for fuzzers
  - Snapshot() system call
  - Dual file system service
  - Shared in-memory test case log
    - *A circular queue for efficient collaborative fuzzing*

Shared in-memory test case log



- **No directory enumeration:** pop() to examine test cases from neighbors
- **No test case re-execution:** direct reference on the bitmap
- **No contention:** a lock-free design

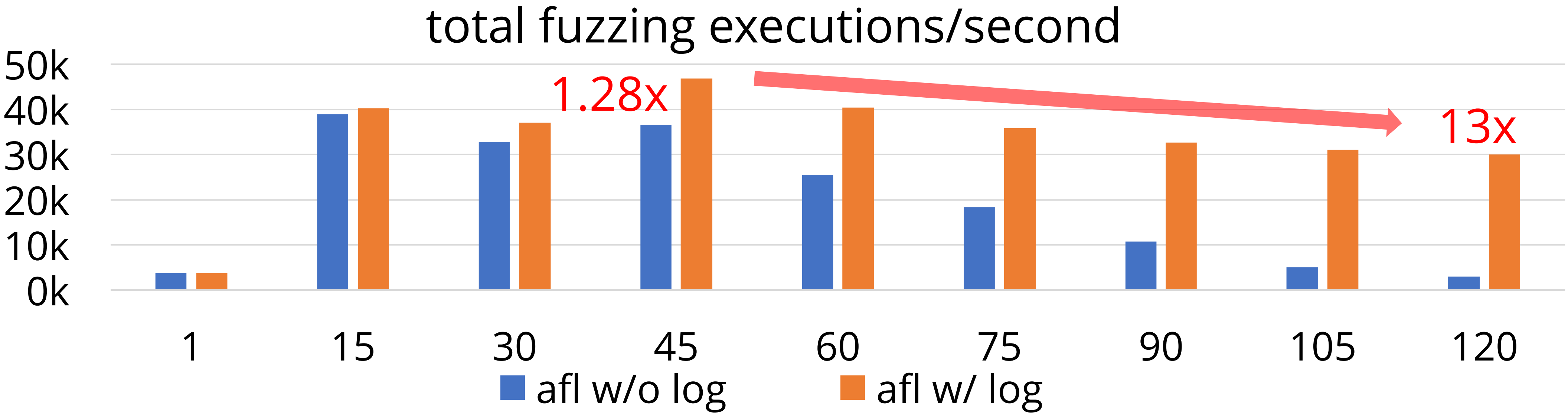
Applicability of techniques

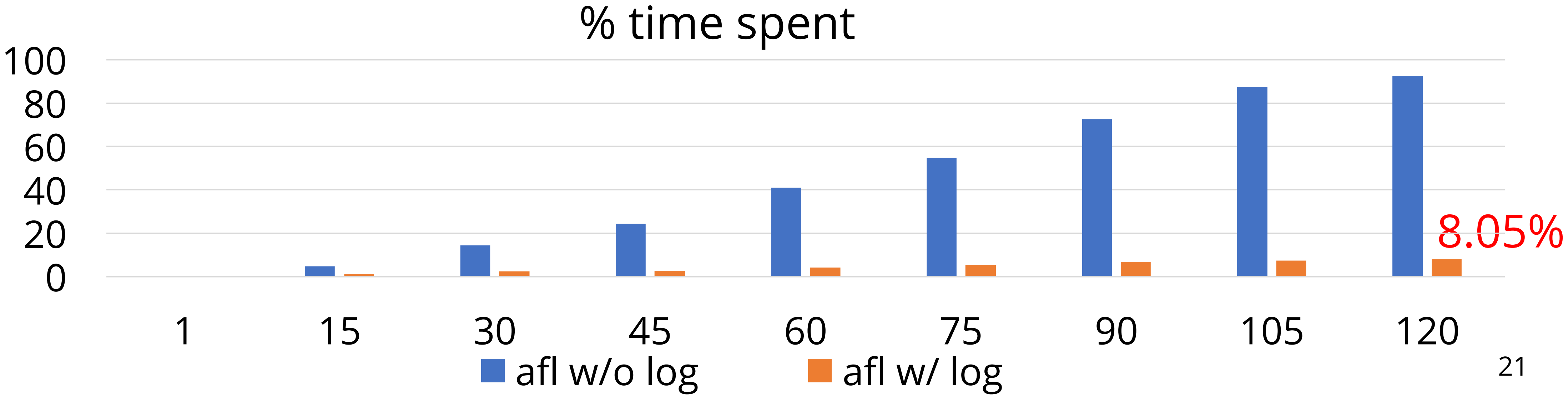
Fuzzers	Snapshot	Dual FS	In-memory log
AFL	✓	✓	✓
AFLFast	✓	✓	✓
Driller	✓	✓	✓
LibFuzzer	-	✓	✓
Honggfuzz	-	✓	✓
VUzzer	✓	✓	✓
Choronzon	✓	✓	✓
IFuzzer	✓	✓	✓
jsfunfuzz	✓	✓	-
zzuf	✓	-	-

Implementation

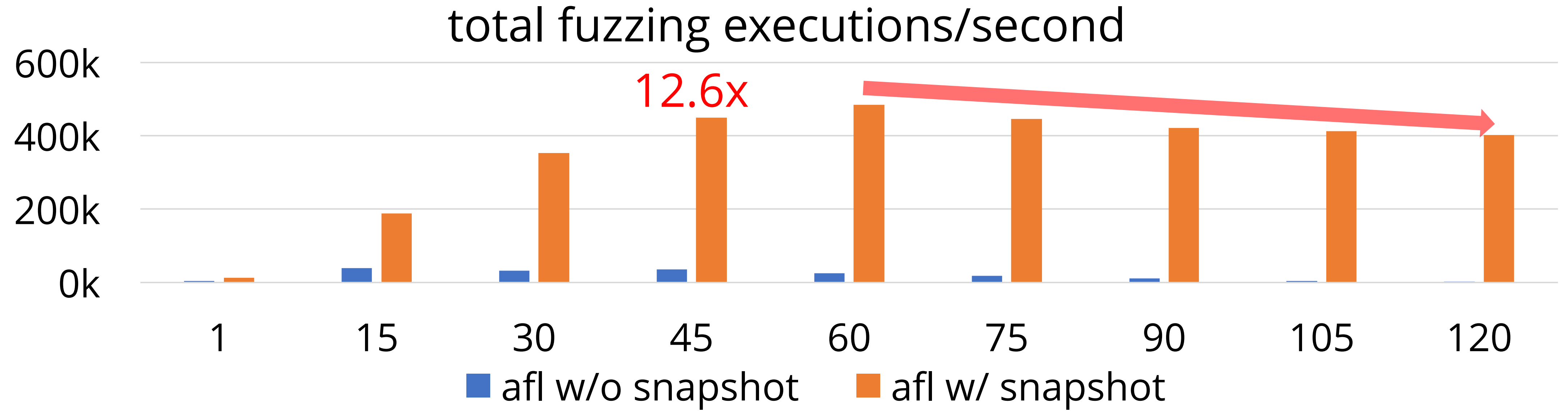
- A new x86_64 system call snapshot() (750 LoC)
- A library for the shared in-memory test case log (100 LoC)
- A dual file system service daemon (100 LoC)
- We applied our new primitives to AFL (400 LoC) and LibFuzzer (200LoC)

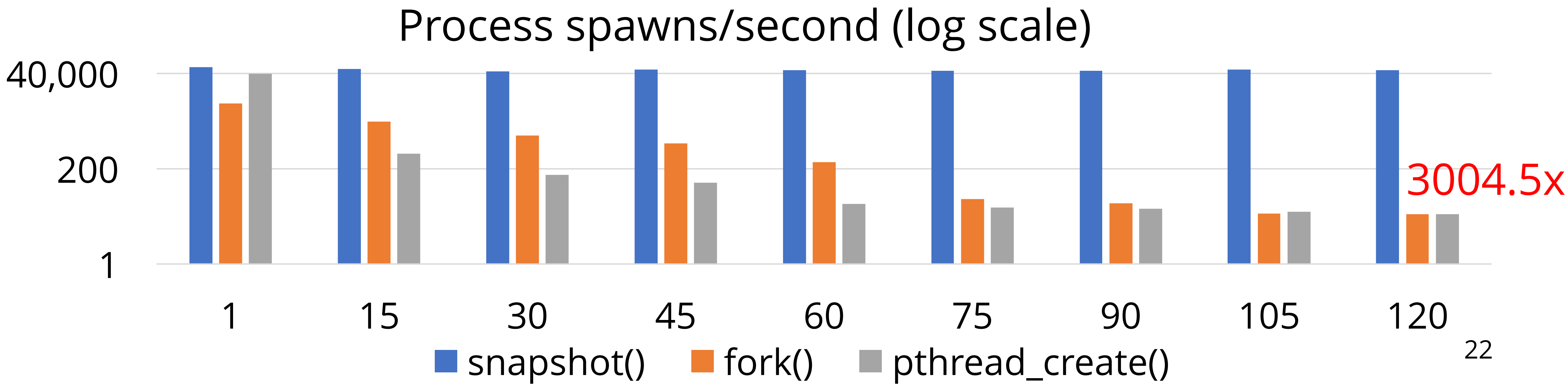
Evaluation – shared test case log



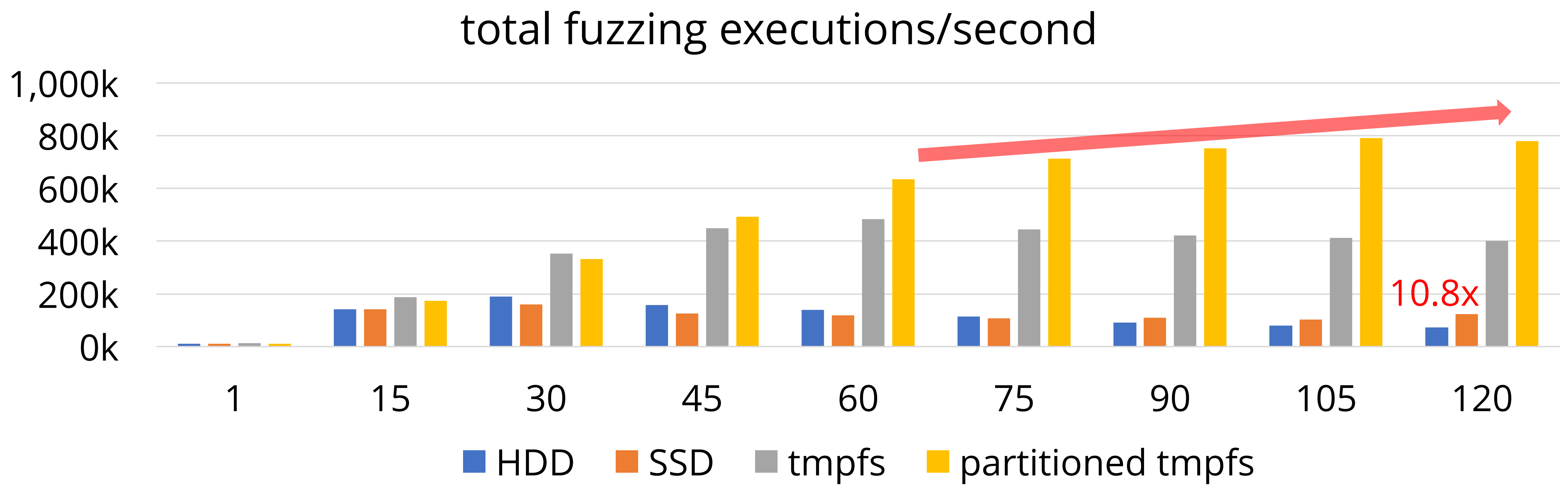


Evaluation – snapshot() system call

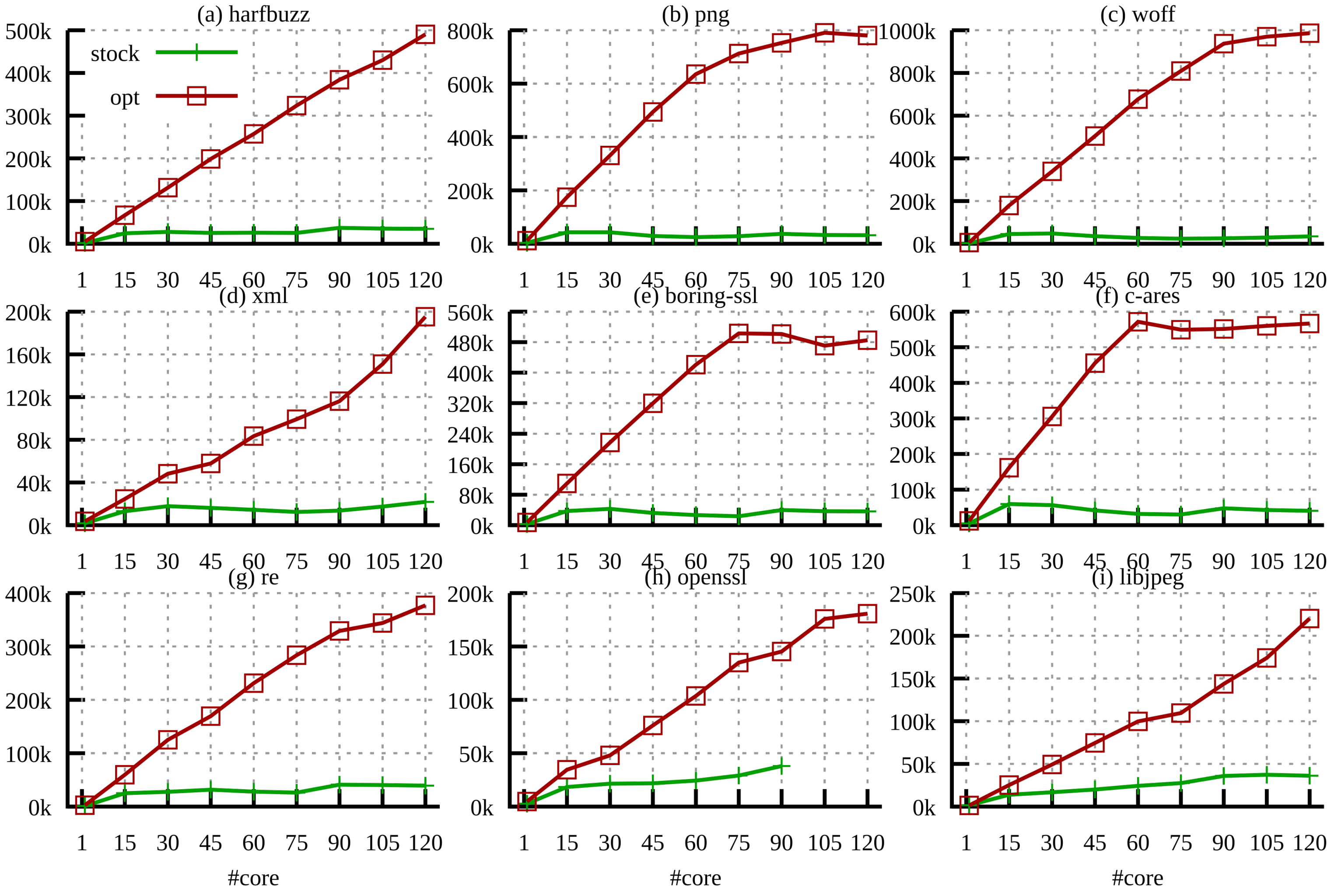




Evaluation – file system service in fuzzing



Evaluation - AFL

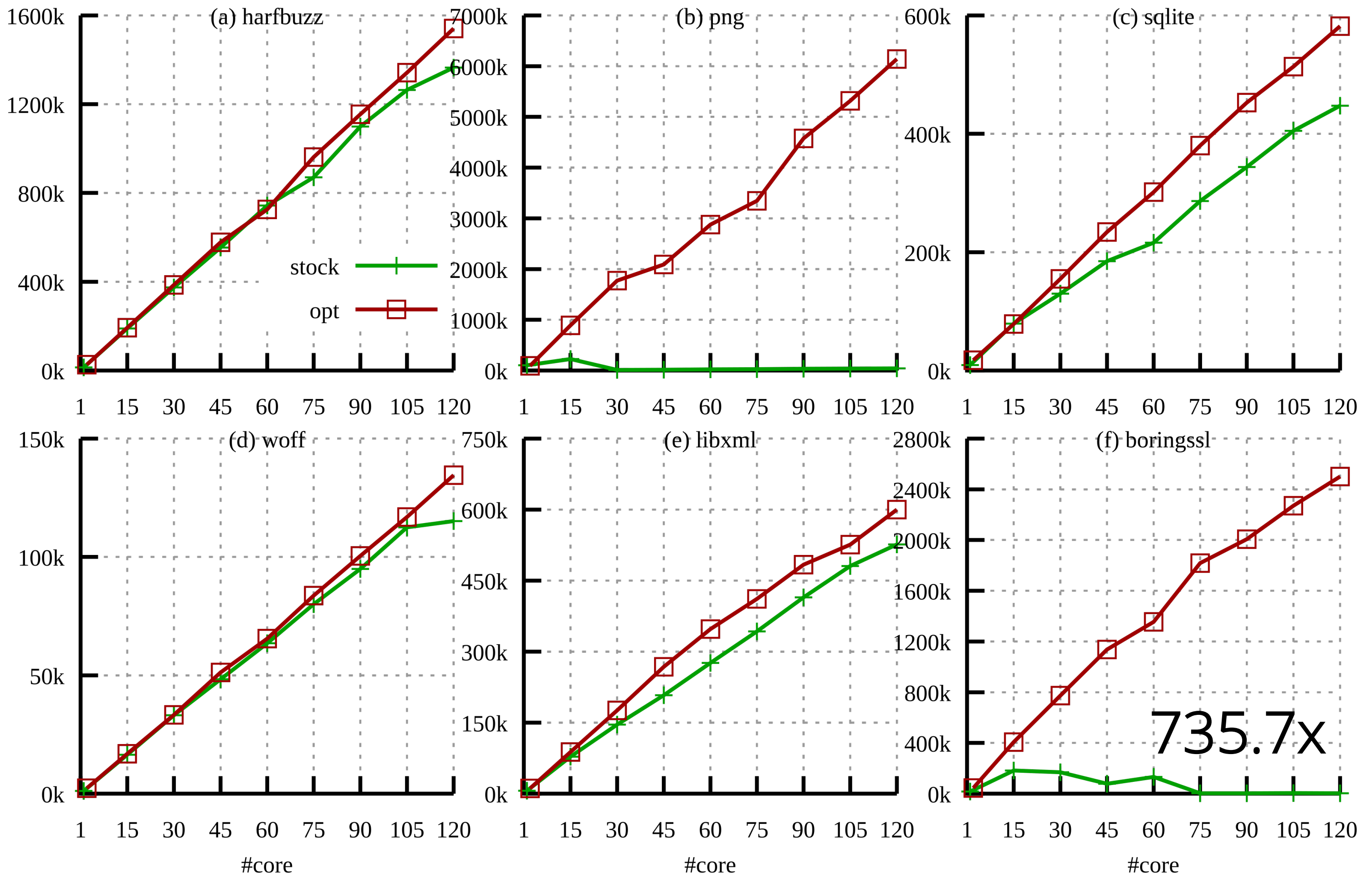


7.7x on 30 cores

25.9x on 60 cores

28.9x on 120 cores

Evaluation - LibFuzzer



Conclusion

- Current fuzzers are not at all scalable on modern OSes with manycore architectures.
- The underlying system components heavily relied on by the fuzzer degrade its scalability.
- New operating primitives specially designed for fuzzing can largely improve the performance and scalability for the state-of-the-art fuzzers.

Open source at https://github.com/sslab-gatech/perf-fuzz

Supported by moz://a

Thanks for listening!

Wen Xu (wen.xu@gatech.edu)