

# Fuzzing and how to evaluate it

**Michael Hicks**

The University of Maryland



ISSISP 2018



Joint work with George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei

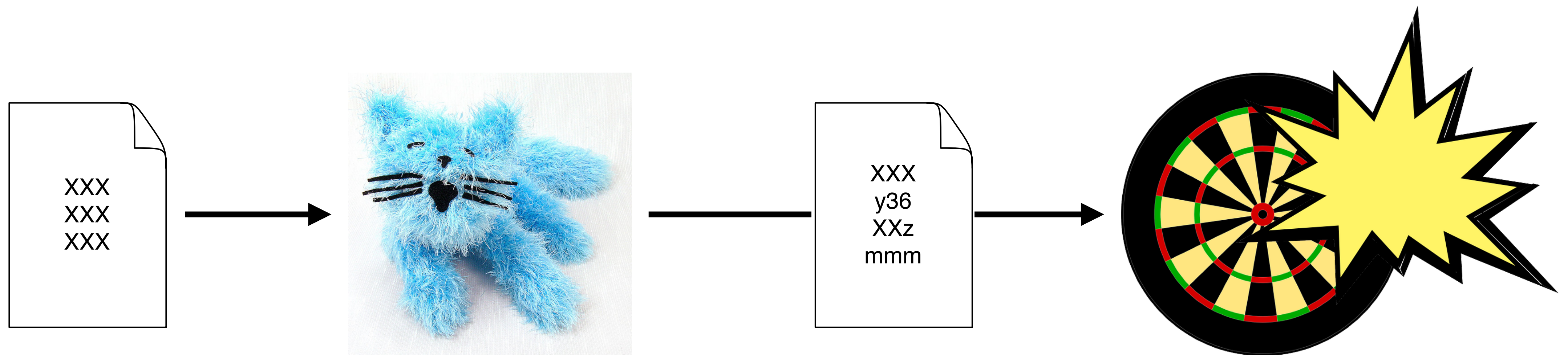


# What is fuzzing?

- A kind of **random testing**
- **Goal**: make sure certain **bad things don't happen, no matter what**
  - **Crashes, thrown exceptions, non-termination**
  - All of these things can be the foundation of security vulnerabilities
- **Complements functional testing**
  - Test features (and lack of misfeatures) directly
  - Normal tests can be starting points for fuzz tests

# File-based fuzzing

- **Mutate** or **generate** inputs
- **Run the target program** with them
- See **what happens**
- Repeat



# Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based, black box* fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++((5- + 3)
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 +4 + 3)
% echo ... | radamsa --seed 12 -n 4 | bc -l
```

- **Blab** generates inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

```
% blab -e '([wrstp][aeiouy]{1,2}){1,4} 32){5} 10'
soty wypisi tisyro to patu
```

# Ex: American Fuzzy Lop (AFL)

- It is a *mutation-based*, “gray-box” fuzzer. Process:
  - **Instrument target** to gather tuple of **<ID of current code location, ID last code location>**
    - On Linux, the optional QEMU mode allows black-box binaries to be fuzzed
  - Retain test input to create a new one *if coverage profile updated*
    - New tuple seen, or existing one a substantially increased number of times
    - Mutations include bit flips, arithmetic, other standard stuff

```
% afl-gcc -c ... -o target
% afl-fuzz -i inputs -o outputs target
afl-fuzz 0.23b (Sep 28 2014 19:39:32) by <lcamtuf@google.com>
[*] Verifying test case 'inputs/sample.txt'...
[+] Done: 0 bits set, 32768 remaining in the bitmap. ...
_____
Queue cycle: 1n time : 0 days, 0 hrs, 0 min, 0.53 sec ...
```

<http://lcamtuf.coredump.cx/afl/>



## american fuzzy lop 0.47b (readpng)

### process timing

run time : 0 days, 0 hrs, 4 min, 43 sec  
last new path : 0 days, 0 hrs, 0 min, 26 sec  
last uniq crash : none seen yet  
last uniq hang : 0 days, 0 hrs, 1 min, 51 sec

### cycle progress

now processing : 38 (19.49%)  
paths timed out : 0 (0.00%)

### stage progress

now trying : interest 32/8  
stage execs : 0/9990 (0.00%)  
total execs : 654k  
exec speed : 2306/sec

### fuzzing strategy yields

bit flips : 88/14.4k, 6/14.4k, 6/14.4k  
byte flips : 0/1804, 0/1786, 1/1750  
arithmetics : 31/126k, 3/45.6k, 1/17.8k  
known ints : 1/15.8k, 4/65.8k, 6/78.2k  
havoc : 34/254k, 0/0  
trim : 2876 B/931 (61.45% gain)

### overall results

cycles done : 0  
total paths : 195  
uniq crashes : 0  
uniq hangs : 1

### map coverage

map density : 1217 (7.43%)  
count coverage : 2.55 bits/tuple

### findings in depth

favorable paths : 128 (65.64%)  
new edges on : 85 (43.59%)  
total crashes : 0 (0 unique)  
total hangs : 1 (1 unique)

### path geometry

levels : 3  
pending : 178  
pend fav : 114  
imported : 0  
variable : 0  
latent : 0

# Other fuzzers

- **Black box:** CERT Basic Fuzzing Framework (BFF), Zzuf, ...
- **Gray box:** VUzzer, Driller, Fairfuzz, T-Fuzz, Angorra, ...
- White box: KLEE, angr, SAGE, Mayhem, ...

There are many more ...

# Evaluating Fuzzing

an adventure in the scientific method



# Assessing Progress

- **Fuzzing is an active area**
  - 2-4 papers per top security conference per year
  - Many fuzzers now in use
- So things are getting better, right?
- To know, **claims must be supported by empirical evidence**
  - I.e., that a new fuzzer is more effective at finding vulnerabilities than a baseline on a realistic workload
  - Is the **evidence reliable?**

# Fuzzing Evaluation Recipe

Requires for Advanced Fuzzer (call it A)

- A compelling **baseline** fuzzer B to compare against
- A **sample of target programs** (benchmark suite)
  - Representative of larger population
- A **performance metric**
  - Ideally, the number of bugs found (else a proxy)
- A meaningful set of configuration parameters
  - Notably, justifiable **seed file(s)**, **timeout**
- A sufficient **number of trials** to judge performance
  - Comparison with baseline using a **statistical test**



# Assessing Progress

- We looked at **32 published papers** and compared their evaluation to our template
  - What **target programs, seeds and timeouts** did they choose and how did they justify them?
  - Against what **baseline** did they compare?
  - How did they measure (or approximate) **performance**?
  - How many **trials** did they perform, and what **statistical test**?
- We found that **most papers did some things right**, but **none were perfect**
  - Raises questions about the strength of published results

# Measuring Effects

- Failure to follow the template may not mean reported results are wrong
  - *Potential* for wrong conclusions, not certainty
- **We carried out experiments** to start to assess this potential
  - Goal is to get a sense of whether the evaluation problem is real
- Short answer: **There are problems**
  - So we provide some recommended mitigations

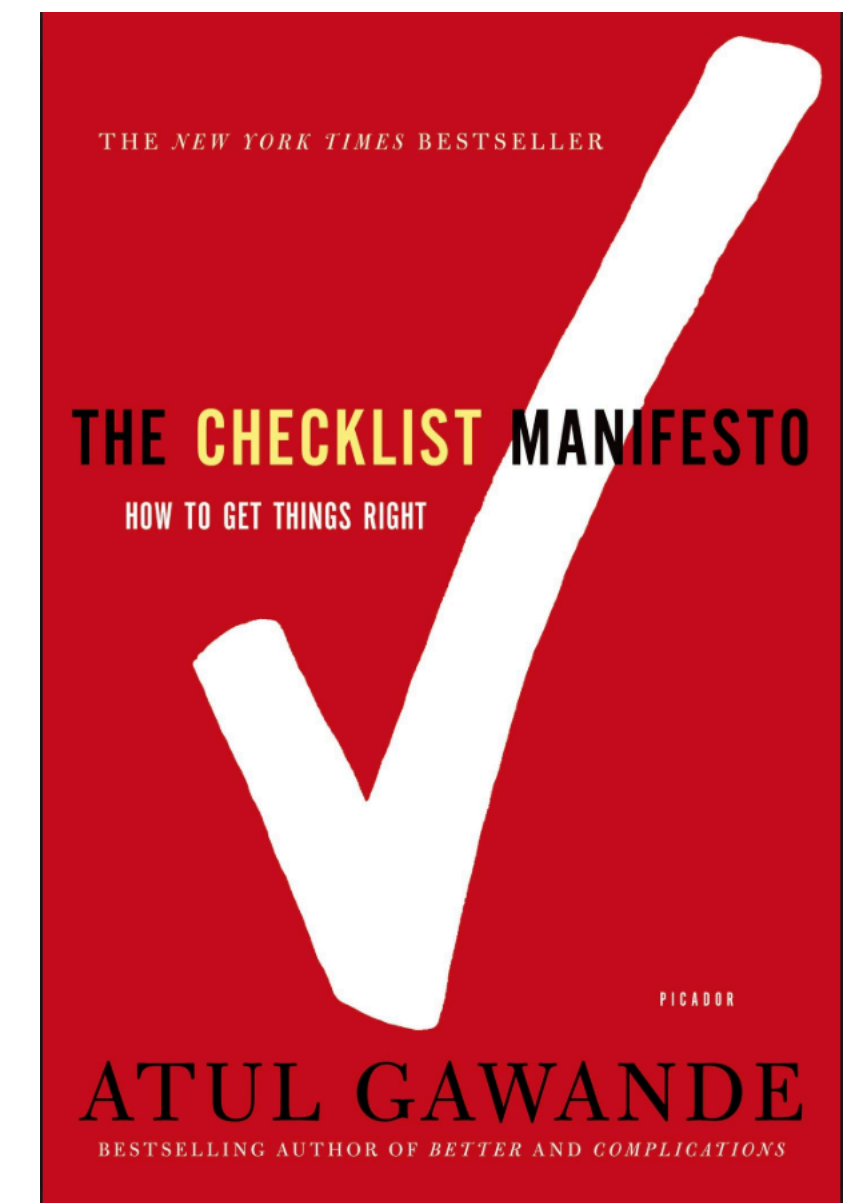
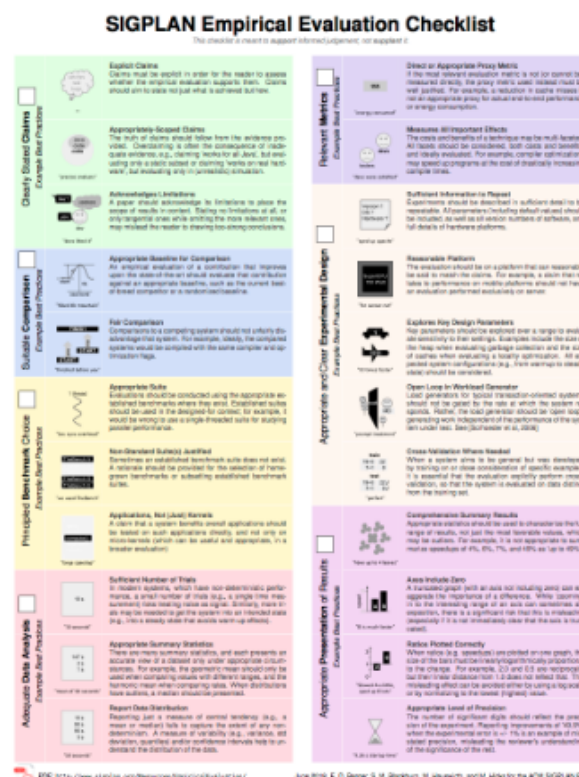


# Summary of Results

- **Few papers measure multiple runs**
  - And yet fuzzer *performance can vary substantially across runs*
- Papers often choose **small number of target programs**, with a **small common set**
  - And yet they target the same population
  - And *performance can vary substantially*
- **Few papers justify the choice of seeds or timeouts**
  - Yet *seeds strongly influence performance*,
  - And *trends can change over time*
- Many papers use **heuristics to relate crashing inputs to bugs**
  - Yet these *heuristics have not been evaluated*
  - One experiment shows they ***dramatically overcount bugs***

# Don't Researchers Know Better?

- **Yes**, many do. Even so, experts forget or are nudged away from best practice by culture and circumstance
  - Especially when best practice is more effort
- **Solution:** List of recommendations
  - And identification of open problems
- Inspiration for effort to provide checklist broadly
  - SIGPLAN Empirical Evaluation Guidelines
  - <http://sigplan.org/Resources/EmpiricalEvaluation/>





# Outline

- Preliminaries
  - Papers we looked at
  - Categories we considered
  - Experimental setup
- Results by category, with recommendations
  - Statistical Soundness
  - Seed selection
  - Timeouts
  - Performance metric
  - Benchmark choice
- Future Work

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[7]	R(29)				G	?	V	-
FuzzSim[44]	R(101)	B	100	C	S		R/M	10D
Dowser[18]	R(7)	O	?		O		V	8H
COVERSET[38]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[8]	R(8)	A, B, Z			S		M	1H
MutaGen[23]	R(8)	R, Z			S	L	V	24H
SDF[28]	R(1)	Z, O			O		V	5D
Driller[41]	C(126)	A			G	L, E	V	24H
QuickFuzz-1[16]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[43]	R(5)	O			M	O	G, R	2H
[46]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[37]	C(63), L, R(10)	A			G, S, O		V	6H, 24H
SlowFuzz[35]	R(10)	O	100		-		V	
Steelix[26]	C(17), L, R(5)	A, V, O			C, G	L, E, M	V	5H
Skyfire[42]	R(4)	O			?	L, M	R, G	LONG
kAFL[39]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[11]	R(7)	O			G*		G	5H
Orthrus[40]	G, R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[22]	R(1)	O			G*		G	-
VDF[21]	R(18)				C	E	V	30D
QuickFuzz-2[17]	R(?)	O	10		G*		G, M	
IMF[19]	R(105)	O			G*	O	G	24H
[48]	S(?)	O	5		G		G	24H
NEZHA[34]	R(6)	A, L, O			O		R	
[45]	G	A, L					V	5M
S2F[47]	L, R(8)	A, O			G	O	V	5H, 24H
FairFuzz[25]	R(9)	A	20	C	C	E	V/M	24H
Angora[9]	L, R(8)	A, V, O	5		G, C	L, E	V	5H
T-Fuzz[33]	C(296), L, R(4)	A, O			C, G*		V	24H
MEDS[20]	S(2), R(12)	O	10		C		V	6H

- **32 papers** (2012-2018)
  - Started from 10 high-impact papers, and chased references
  - Plus: Keyword search
- **Disparate goals**
  - Improve initial seed selection
  - Smarter mutation (e.g., based on taint data)
  - Different observations (e.g., running time)
  - Faster execution times, parallelism
  - Etc.



# Experimental Setup

- Advanced Fuzzer: **AFLFast** (CCS'16), Baseline: **AFL**
- Five target programs used by previous fuzzers
  - Three binutils programs: **cxxfilt**, **nm**, **objdump** (AFLFast)
  - Two image processing ones: **gif2png** (VUzzer), **FFmpeg** (fuzzsim)
- **30 trials** (more or less) at **24 hours** per run
  - Empty seed, sampled seed, others
  - Mann Whitney U test
- Experiments on **de-duplication effectiveness**

# Why AFL, AFLFast?

- AFL is popular (14/32 papers used it as baseline)
- AFLFast is open source, easy build instructions, and easy experiments to reproduce and extend
  - Thanks to the authors for their help!
- Issues that we found not unique to AFLFast
  - Other papers do worse
  - Other fuzzers have same core structure as AFL/AFLFast
- Issues may not undermine results
  - But conclusions are probably weakened, caveated
  - The point: We need stronger evaluations to see

# Statistical Soundness



# Fuzzing is a Random Process

- The mutation of the **input is chosen randomly** by the fuzzer, and the target may make random choices
- Each **fuzzing run is a sample** of the random process
  - Question: Did it find a crash or not?
- **Samples** can be used to **approximate the distribution**
  - More samples give greater certainty
- Is A better than B at fuzzing? Need to **compare distributions** to make a statement

# Analogy: Biased Dice

- We want to **compare the “performance” of two dice**
  - Die A is better than die B if it tends to land on higher numbers more often (biased!)
- Suppose rolling A and B yields 6 and 1. **Is A better?**
  - **Maybe**. But we don't have enough information. One trial is not enough to characterize a random process.



# Multiple Trials

- What if I roll A and B five times each and get
  - **A**: 6, 6, 1, 1, 6
  - **B**: 4, 4, 4, 4, 4
  - *Is A better?*
- Could **compare average measures**
  - $\text{median}(A) = 6$ ,  $\text{median}(B) = 4$
  - $\text{mean}(A) = 4$ ,  $\text{mean}(B) = 4$
  - The first suggests A is better, but the second does not
  - And there is **still uncertainty** that these comparisons hold up after more trials



# Statistical Tests

- A mechanism for quantitatively accepting or rejecting a **hypothesis** about a **process**
- In our case, the process is **fuzz testing** and the hypothesis is that fuzz tester A (a “random variable”) is better than B at finding bugs in a particular program, e.g., that  **$\text{median}(\mathbf{A}) - \text{median}(\mathbf{B}) \geq 0$**  for that program
- The **confidence** of our judgment is captured in the ***p-value***
  - It is the probability that the outcome of the test is wrong
  - Convention:  **$\mathbf{p\text{-}value} \leq 0.05$**  is a sufficient level of confidence



# A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering

Andrea Arcuri  
Simula Research Laboratory  
P.O. Box 134, 1325 Lysaker, Norway  
arcuri@simula.no

Lionel Briand  
Simula Research Laboratory and  
University of Oslo  
P.O. Box 134, 1325 Lysaker, Norway  
briand@simula.no

## ABSTRACT

Randomized algorithms have been used to successfully address many different types of software engineering problems. This type of algorithms employ a degree of randomness as part of their logic. Randomized algorithms are useful for difficult problems where a precise solution cannot be derived in a deterministic way within reasonable time. However, randomized algorithms produce different results on every run when applied to the same problem instance. It is hence important to assess the effectiveness of randomized algorithms by collecting data from a large enough number of runs. The use of rigorous statistical tests is then essential to provide support to the conclusions derived by analyzing such data. In this paper, we provide a systematic review of the use of randomized algorithms in selected software engineering venues in 2009. Its goal is not to perform a complete survey but to get a representative snapshot of current practice in software engineering research. We show that randomized algorithms are used in a significant percentage of papers but that, in most cases, randomness is not properly accounted for. This casts doubts on the validity of most empirical results assessing randomized algorithms. There are numerous statistical tests, based on different assumptions, and it is not always clear when and how to use these tests. We hence provide practical guidelines to support empirical research on randomized algorithms in software engineering.

## Categories and Subject Descriptors

D.2.0 [Software Engineering]: General;  
I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods,

## 1. INTRODUCTION

Many problems in software engineering can be alleviated through automated support. For example, automated techniques exist to generate test cases that satisfy some desired coverage criteria on the system under test, such as for example branch [26] and path coverage [22]. Because often these problems are undecidable, deterministic algorithms that are able to provide optimal solutions in reasonable time do not exist. The use of randomized algorithms [44] is hence necessary to address this type of problems.

The most well-known example of randomized algorithm in software engineering is perhaps *random testing* [13, 6]. Techniques that use random testing are of course randomized, as for example DART [22] (which combines random testing with symbolic execution). Furthermore, there is a large body of work on the application of *search algorithms* in software engineering [25], as for example Genetic Algorithms. Since practically all search algorithms are randomized and numerous software engineering problems can be addressed with search algorithms, randomized algorithms therefore play an increasingly important role. Applications of search algorithms include software testing [41], requirement engineering [8], project planning and cost estimation [2], bug fixing [7], automated maintenance [43], service-oriented software engineering [9], compiler optimisation [11] and quality assessment [32].

A randomized algorithm may be strongly affected by chance. It may find an optimal solution in a very short time or may never converge towards an acceptable solution. Running a randomized algorithm twice on the same instance of a software engineering problem usually produces different results. Hence, researchers in software engineering that develop novel techniques based on ran-

- Use the *Student T test* ?
  - Meets the right form for the test
  - But assumes that samples (fuzz test inputs) drawn from a normal distribution. Certainly not true
- Arcuri & Brian advice: Use the **Mann Whitney U Test**
- No assumption of distribution normality



paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[7]	R(29)				G	?	V	-
FuzzSim[44]	R(101)	B	100	C	S		R/M	10D
Dowser[18]	R(7)	O	?		O		V	8H
COVERSET[38]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[8]	R(8)	A, B, Z			S		M	1H
MutaGen[23]	R(8)	R, Z			S	L	V	24H
SDF[28]	R(1)	Z, O			O		V	5D
Driller[41]	C(126)	A			G	L, E	V	24H
QuickFuzz-1[16]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[43]	R(5)	O			M	O	G, R	2H
[46]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[37]	C(63), L, R(10)	A			G, S, O		V	6H, 24H
SlowFuzz[35]	R(10)	O	100		-		V	
Steelix[26]	C(17), L, R(5)	A, V, O			C, G	L, E, M	V	5H
Skyfire[42]	R(4)	O			?	L, M	R, G	LONG
kAFL[39]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[11]	R(7)	O			G*		G	5H
Orthrus[40]	G, R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[22]	R(1)	O			G*		G	-
VDF[21]	R(18)				C	E	V	30D
QuickFuzz-2[17]	R(?)	O	10		G*		G, M	
IMF[19]	R(105)	O			G*	O	G	24H
[48]	S(?)	O	5		G		G	24H
NEZHA[34]	R(6)	A, L, O			O		R	
[45]	G	A, L					V	5M
S2F[47]	L, R(8)	A, O			G	O	V	5H, 24H
FairFuzz[25]	R(9)	A	20	C	C	E	V/M	24H
Angora[9]	L, R(8)	A, V, O	5		G, C	L, E	V	5H
T-Fuzz[33]	C(296), L, R(4)	A, O			C, G*		V	24H
MEDS[20]	S(2), R(12)	O	10		C		V	6H

# Evaluations

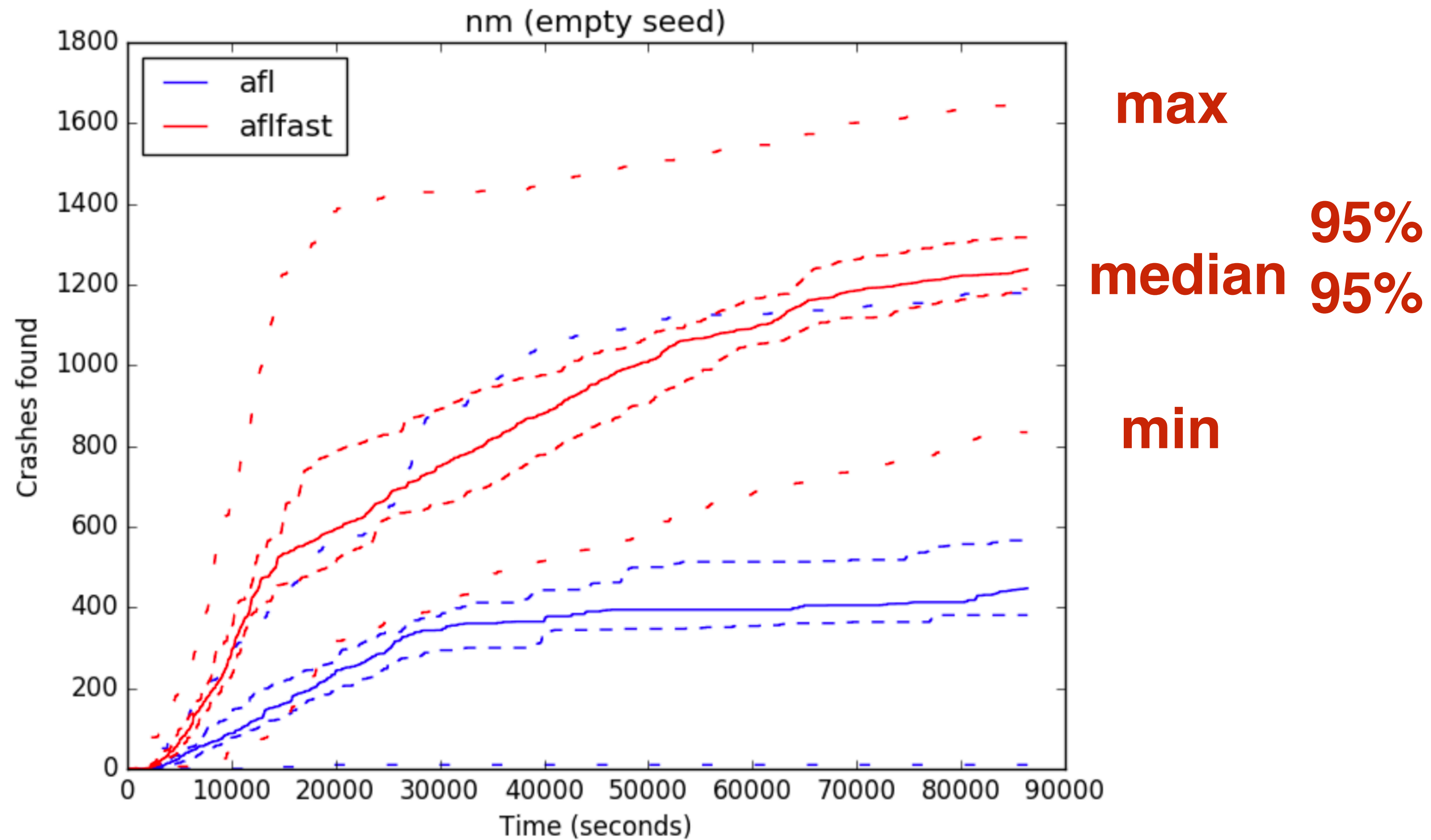
- 19/32 papers said nothing about multiple trials
  - Assume 1
- 13/32 papers said multiple trials
  - Varying number; one case not specified
- 3/13 papers characterized variance across runs
- 0 papers performed a statistical test



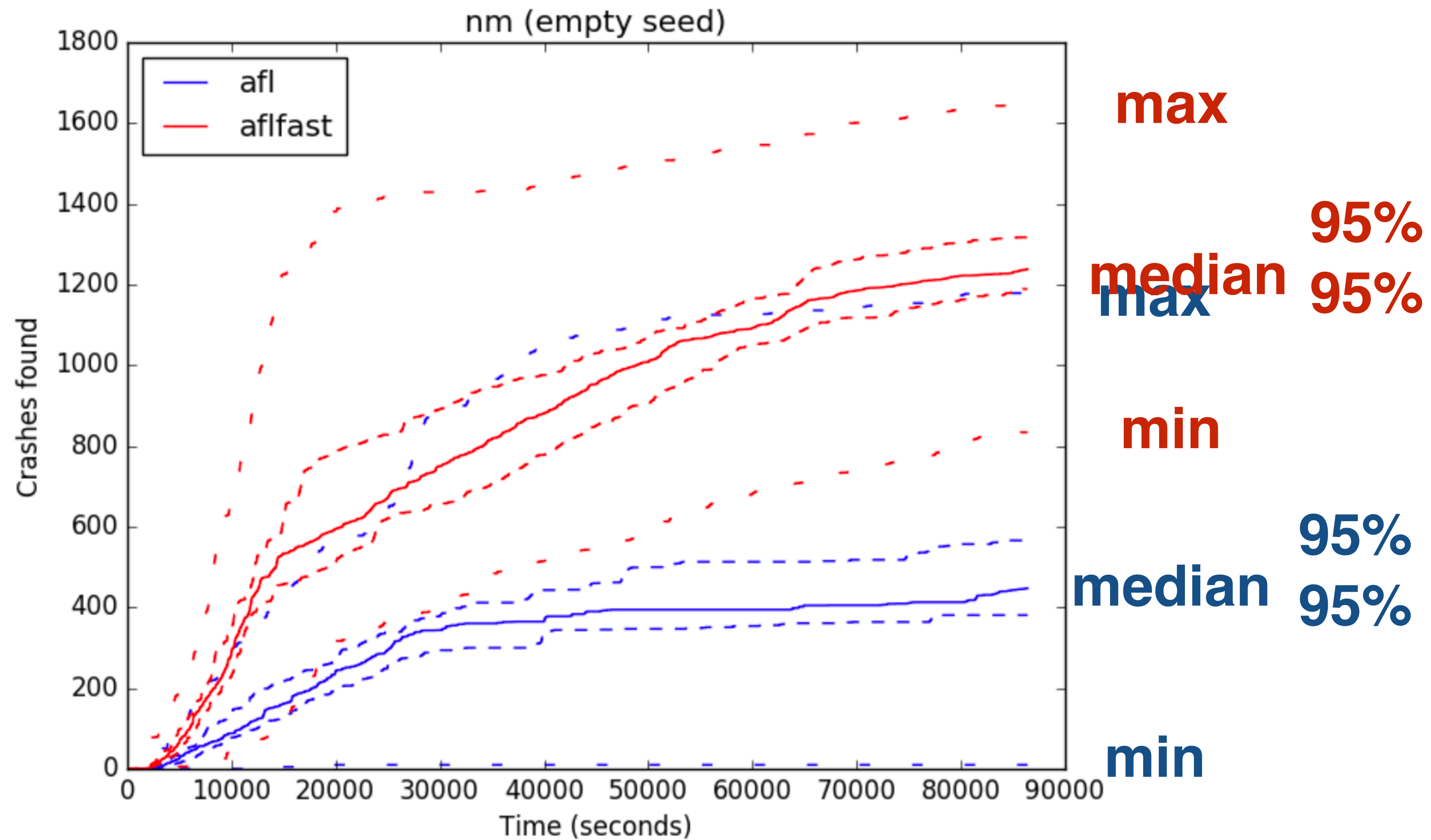
# Practical Impact?

- Fuzzers run for a long time, conducting potentially millions of individual tests over many hours
  - If we consider our biased die: Perhaps **no statistical test** is needed (just the mean/median) **if we have a lot of trials?**
- Problem: **Fuzzing is a *stateful* search process**
  - **Each test is not independent**, as in a die roll
    - Rather, it is influenced by the outcome of previous tests
  - The **search space is vast**; covering it all is difficult
- Therefore, we should **consider each run as a trial**, and consider **many trials**
  - Experimental results show **potentially high per-trial variance**

# Performance Plot

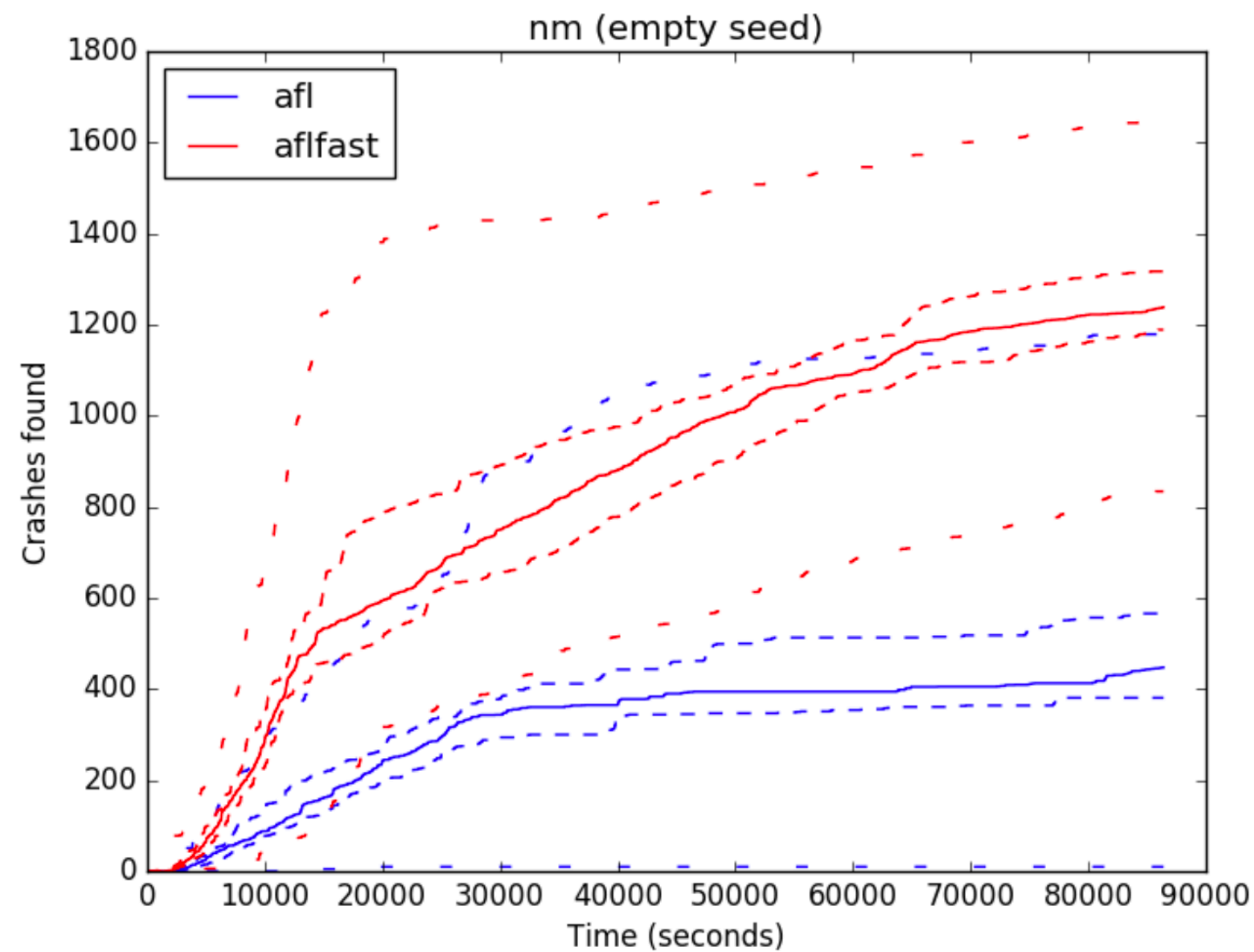


# Performance Plot



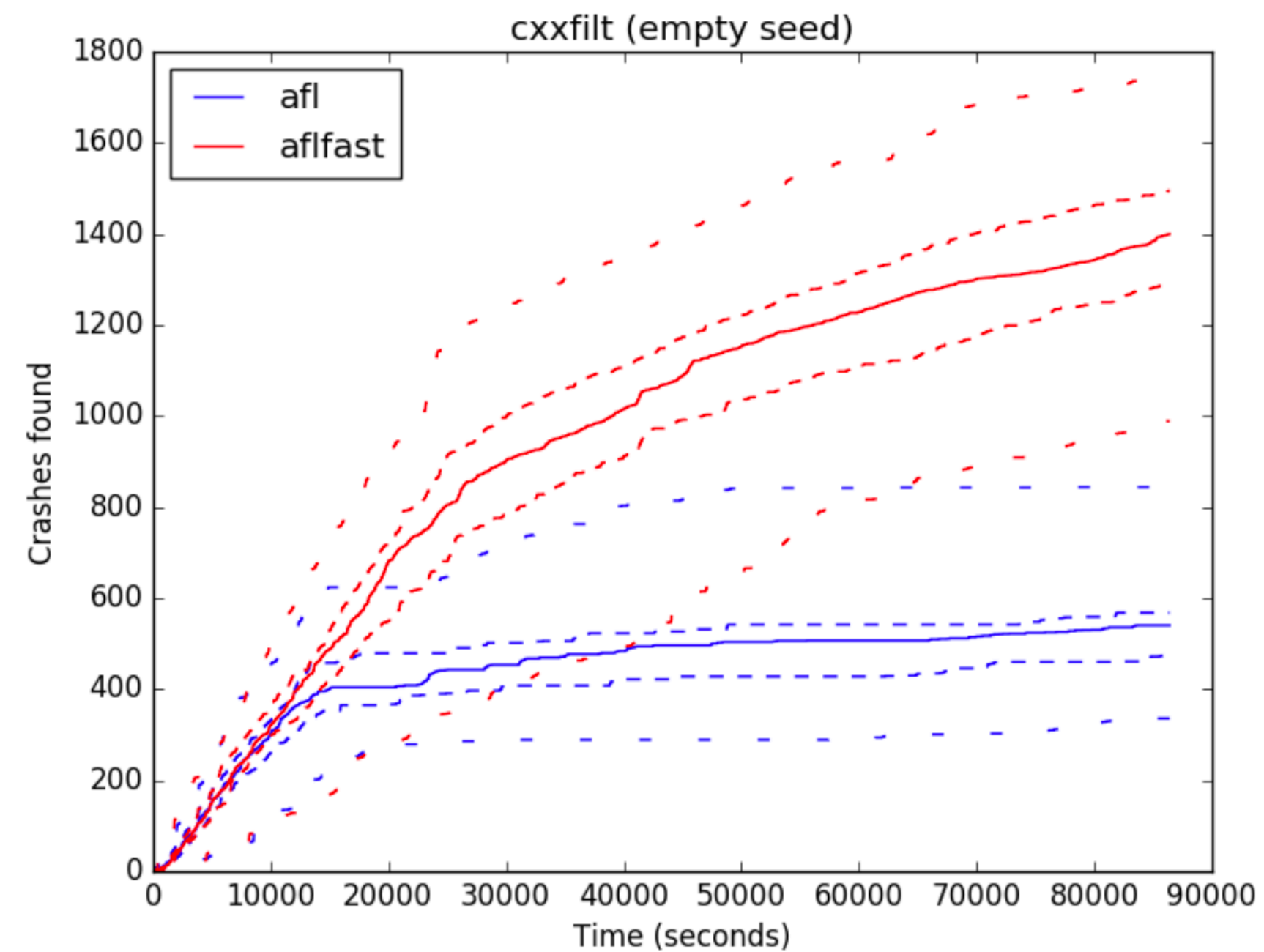


# Statistically Significant



$$p < 10^{-13}$$

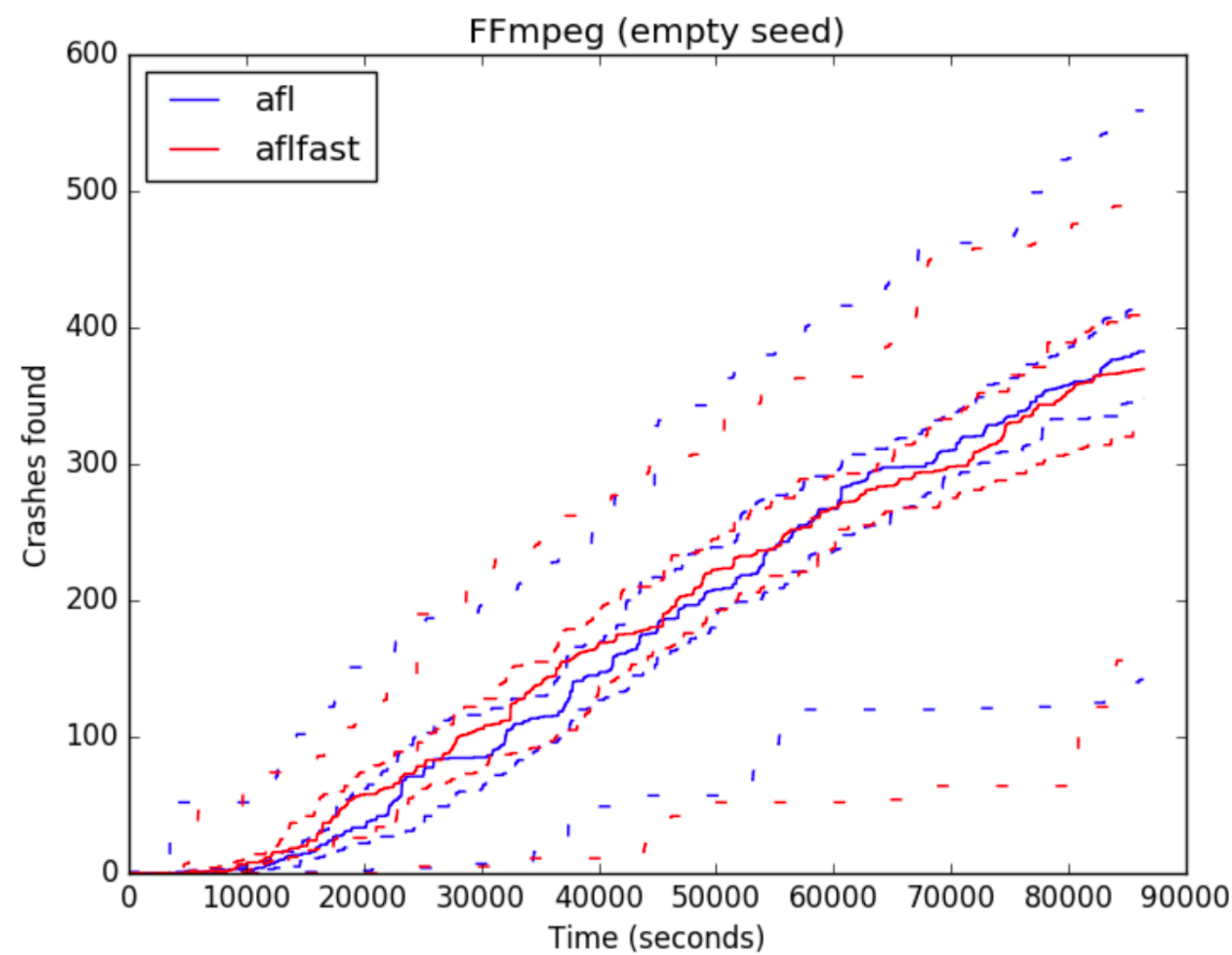
**significant variance**  
in performance



$$p < 10^{-10}$$

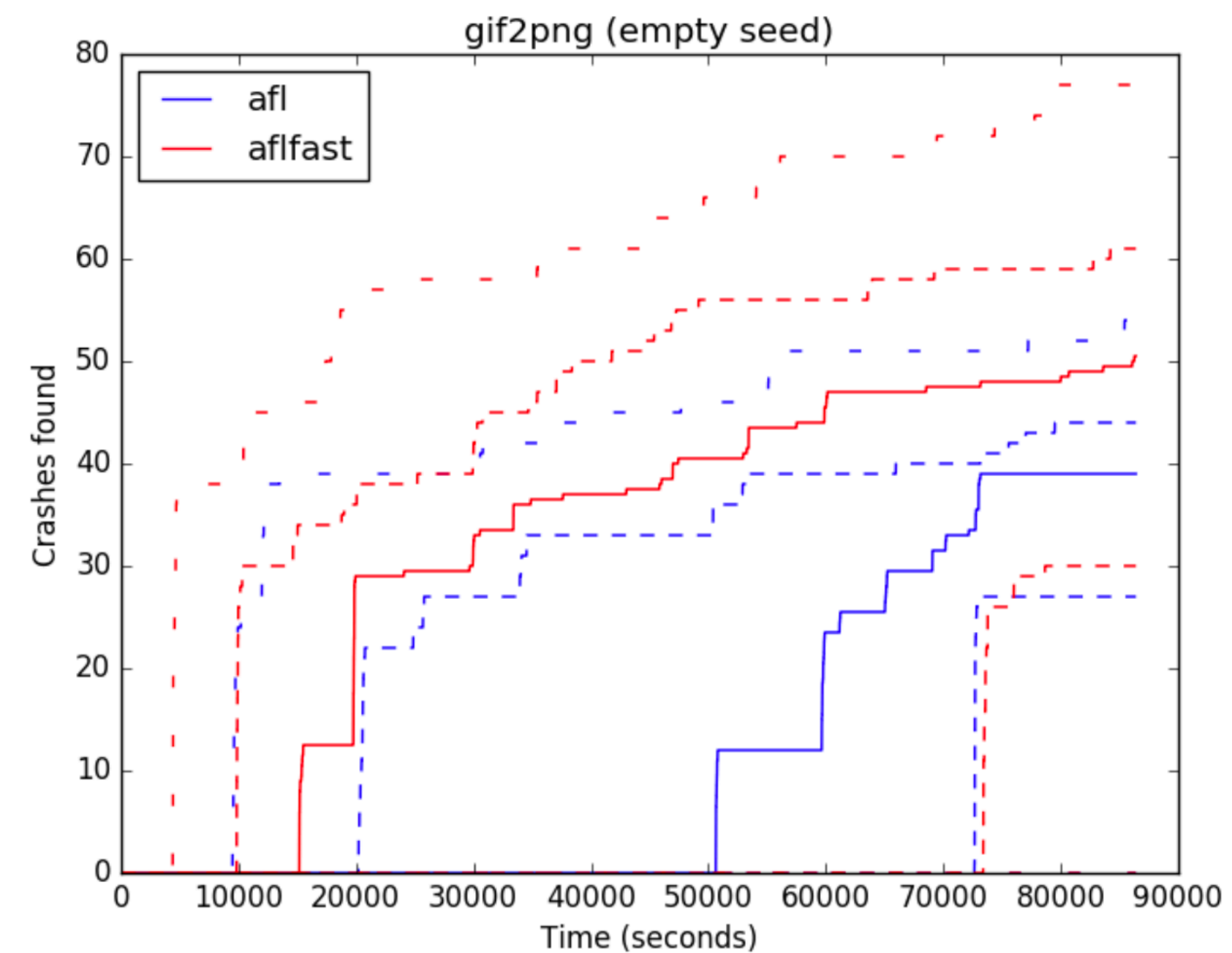
**Higher median**  
clearly better

# Statistically Insignificant



$p = 0.379$

Max **AFL** = 550  
Min **AFLFast** = 150



$p = 0.0676$

**Higher median**  
does *not* meet bar  
for significance

# I Want **You**



*to run multiple trials*

and

*use a statistical test to  
compare distributions!*

# Seed Selection



# Seed Corpus

- Mutation-based fuzzers require an **initial seed** (or seeds) to start the process
- **Conventional wisdom: Valid input, but small**
  - Valid, to drive the program into its “main” logic
  - Small, to complete test more quickly
- Some studies on how to choose seeds
  - Applied to black box fuzzer; relevant to gray box?
- How might seed choices matter?

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[7]	R(29)				G	?	V	-
FuzzSim[44]	R(101)	B	100	C	S		R/M	10D
Dowser[18]	R(7)	O	?		O		V	8H
COVERSET[38]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[8]	R(8)	A, B, Z			S		M	1H
MutaGen[23]	R(8)	R, Z			S	L	V	24H
SDF[28]	R(1)	Z, O			O		V	5D
Driller[41]	C(126)	A			G	L, E	V	24H
QuickFuzz-1[16]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[43]	R(5)	O			M	O	G, R	2H
[46]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[37]	C(63), L, R(10)	A			G, S, O		V	6H, 24H
SlowFuzz[35]	R(10)	O	100		-		V	
Steelix[26]	C(17), L, R(5)	A, V, O			C, G	L, E, M	V	5H
Skyfire[42]	R(4)	O			?	L, M	R, G	LONG
kAFL[39]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[11]	R(7)	O			G*		G	5H
Orthrus[40]	G, R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[22]	R(1)	O			G*		G	-
VDF[21]	R(18)				C	E	V	30D
QuickFuzz-2[17]	R(?)	O	10		G*		G, M	
IMF[19]	R(105)	O			G*	O	G	24H
[48]	S(?)	O	5		G		G	24H
NEZHA[34]	R(6)	A, L, O			O		R	
[45]	G	A, L					V	5M
S2F[47]	L, R(8)	A, O			G	O	V	5H, 24H
FairFuzz[25]	R(9)	A	20	C	C	E	V/M	24H
Angora[9]	L, R(8)	A, V, O	5		G, C	L, E	V	5H
T-Fuzz[33]	C(296), L, R(4)	A, O			C, G*		V	24H
MEDS[20]	S(2), R(12)	O	10		C		V	6H

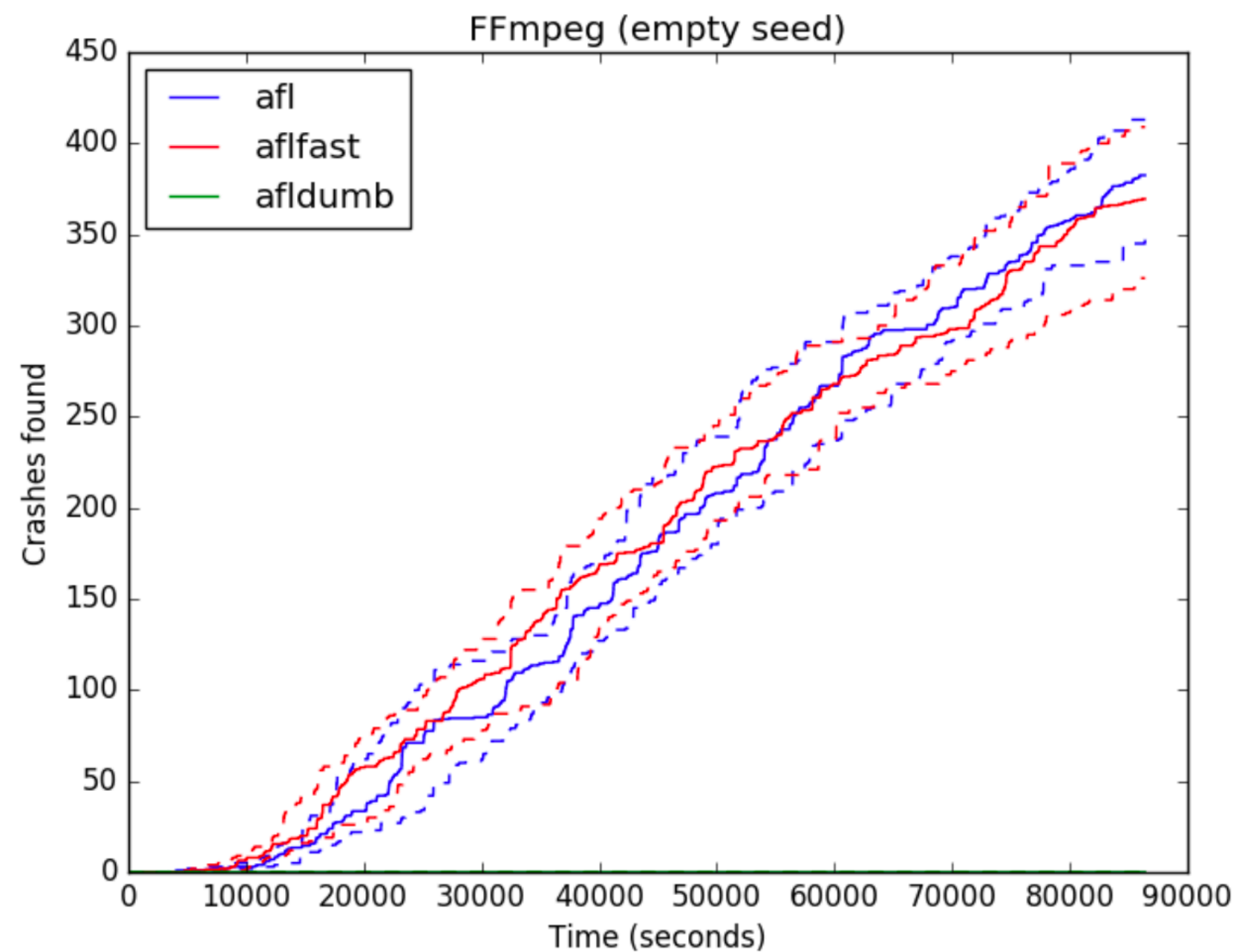
# Evaluations

- 16/32 papers skipped particulars of seed choice
  - “Valid” seed (V)
- 2/32 papers used the empty (E) file (eg. AFLFast)
  - Surprising contradiction to conventional wisdom
- Question: Practical impact?

# Experiments

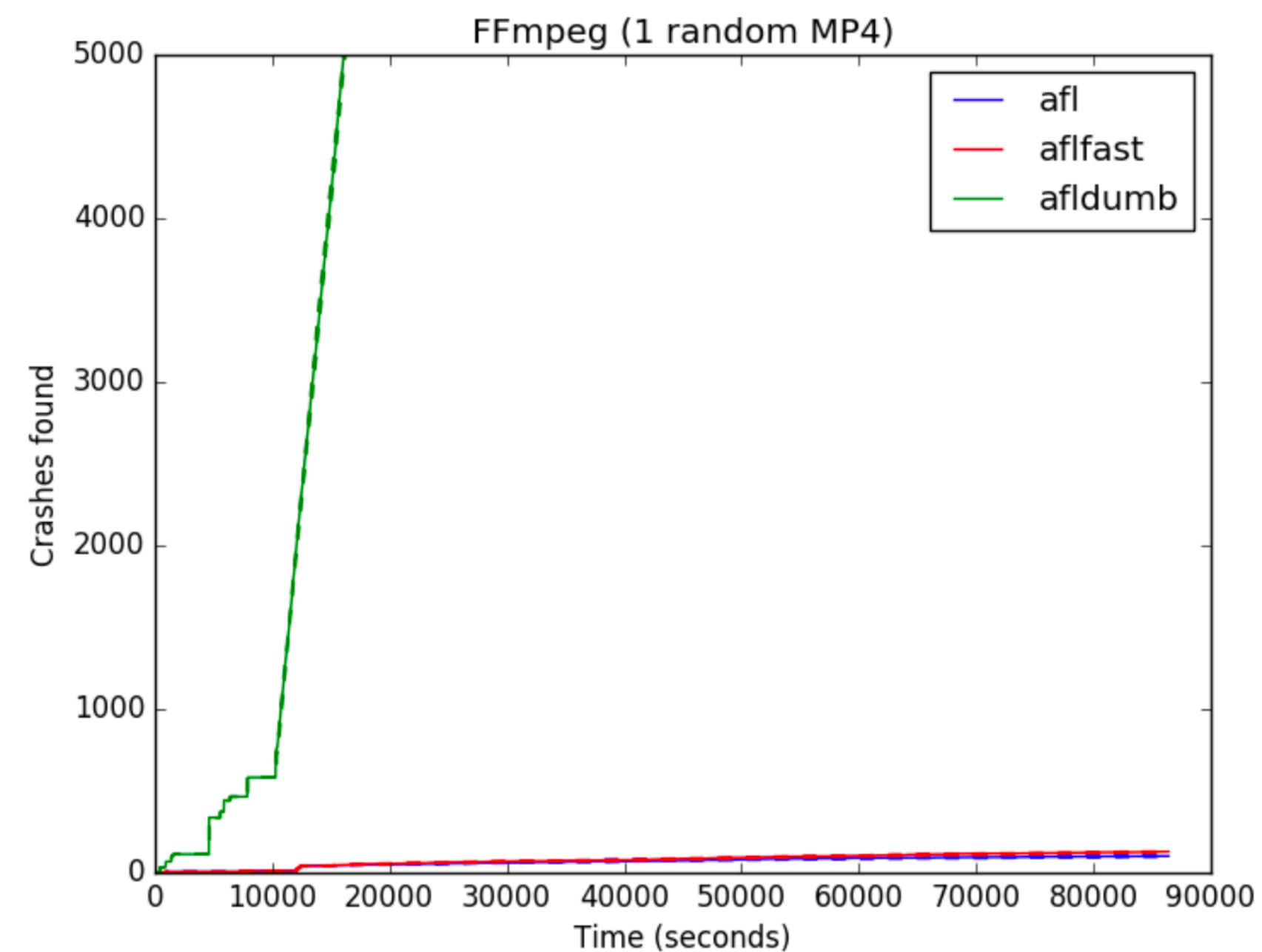
- **Empty** seed
- **Sampled** from FFmpeg site (<http://samples.mpeg.org>)
  - All less than 1 MB
  - Picked smallest one
- **Made** with FFmpeg itself (using videogen and audiogen programs)
- Also sampled and made object files for nm and objdump, and text for cxxfilt

# FFMpeg: Empty vs. Handmade



empty seed

(AFLFast vs. AFL)  $p1 = 0.379$   
(AFLDumb vs. AFL)  $p2 < 10^{-15}$

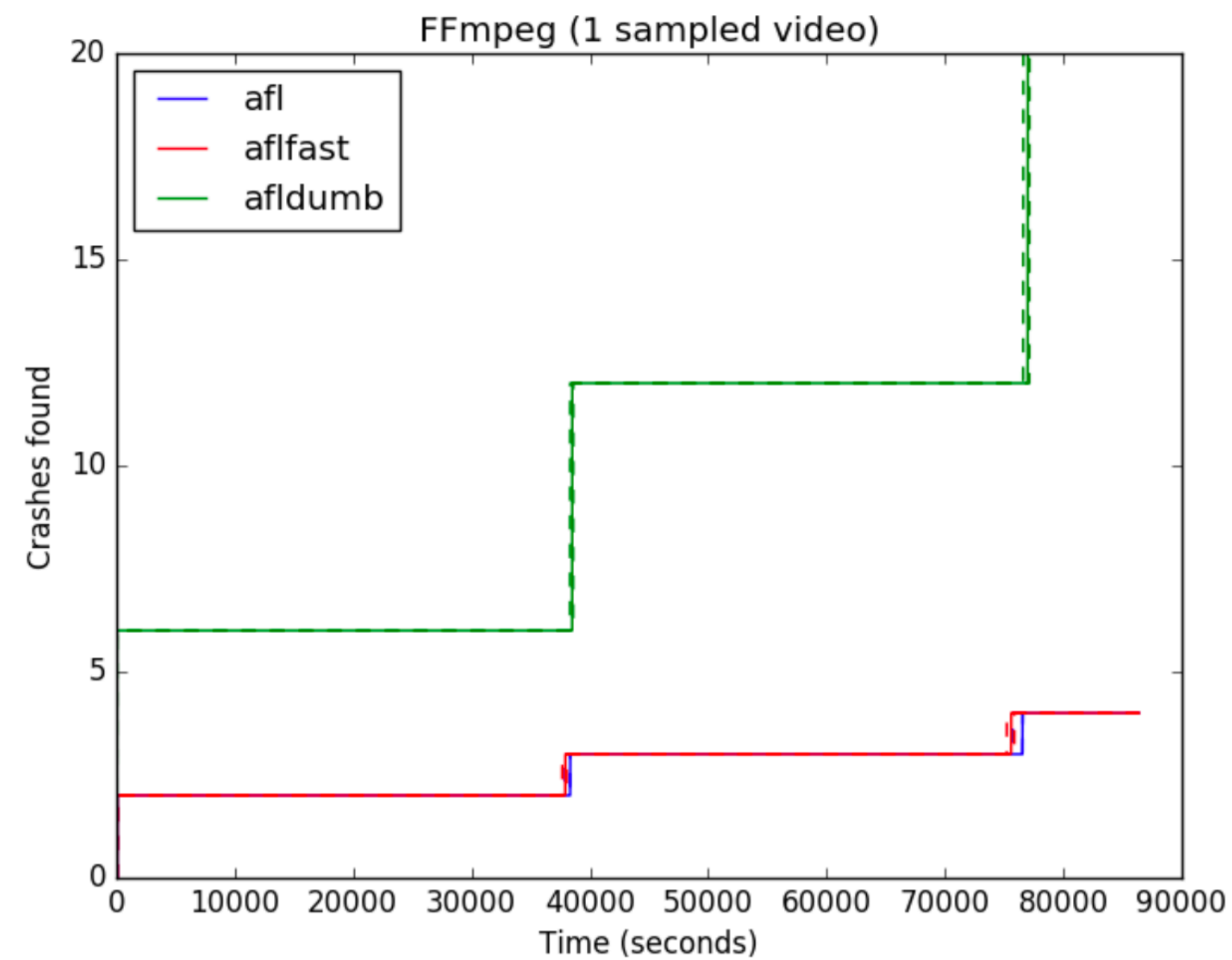


1-made

$p1 = 0.048$   
 $p2 < 10^{-11}$



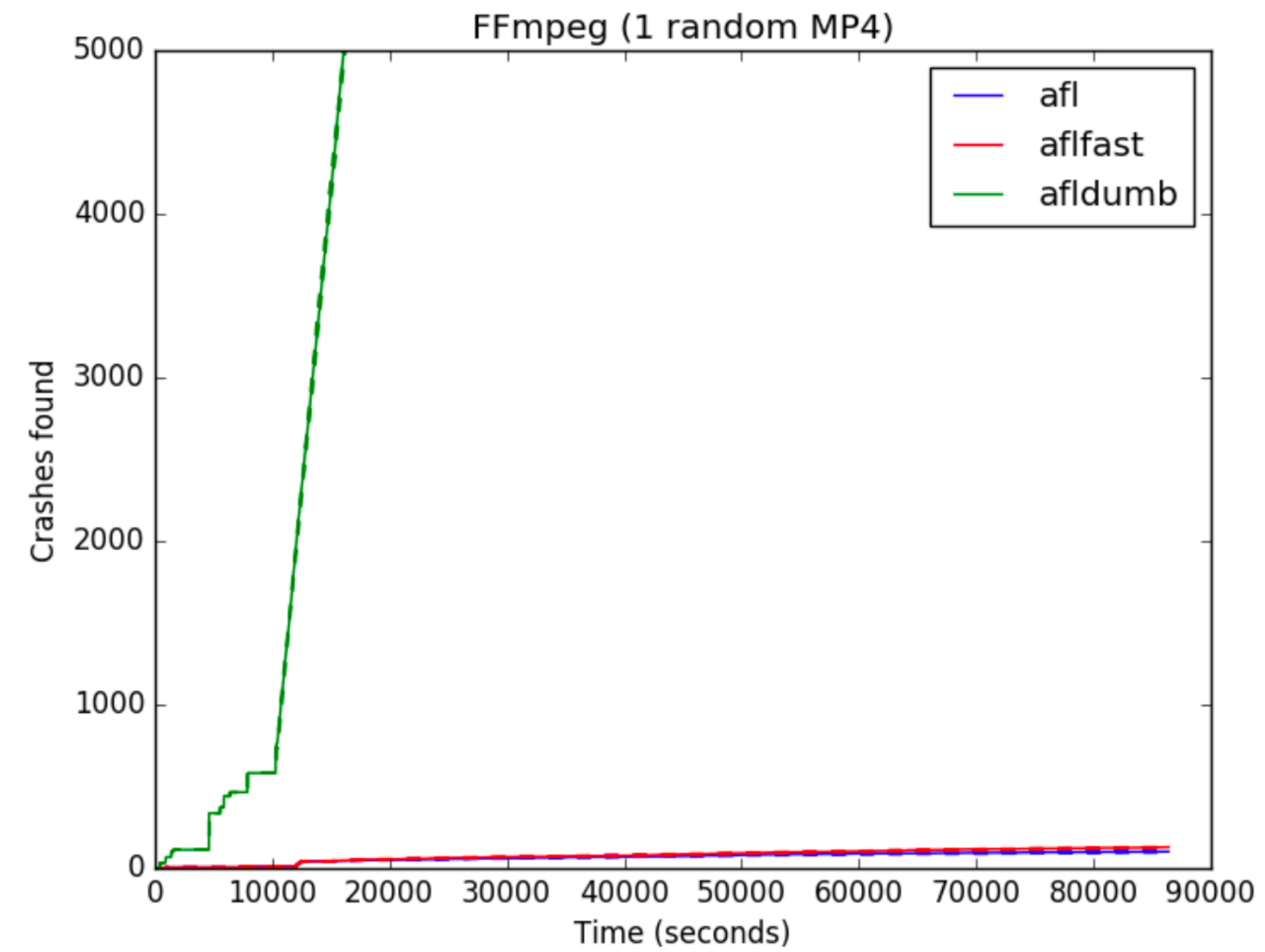
# FFMpeg: Sampled vs. Handmade



1-sampled

$p1 > 0.05$

$p2 < 10^{-5}$



1-made

$p1 = 0.048$

$p2 < 10^{-11}$

# Summary, More Programs

	<b>empty</b>	<b>1-made</b>
FFmpeg, AFLDumb	0 ( $< 10^{-15}$ )	5000 ( $< 10^{-11}$ )
FFmpeg, AFL	382.5	102
FFmpeg, AFLFast	369.5 (= 0.379)	129 ( $< 0.05$ )
nm, AFL	448	23
nm, AFLFast	1239 ( $< 10^{-13}$ )	24 (= 0.830)
objdump, AFL	6.5	5
objdump, AFLFast	29 ( $< 10^{-3}$ )	6 ( $< 10^{-2}$ )
cxxfilt, AFL	540.5	572.5
cxxfilt, AFLFast	1400 ( $< 10^{-10}$ )	1364 ( $< 10^{-10}$ )

**median** **p-value**  
relative to AFL

# Seed Corpus: Recommendations

- Performance with different seeds varies dramatically
  - **Not all “valid” seeds are the same**
- The **empty seed can perform well**
  - Contrary to conventional wisdom
- Evaluations should
  - Clearly **document seed choices**
  - **Evaluate on several seeds** to assess performance difference
    - But how to say something comprehensive is not easy

# Timeouts

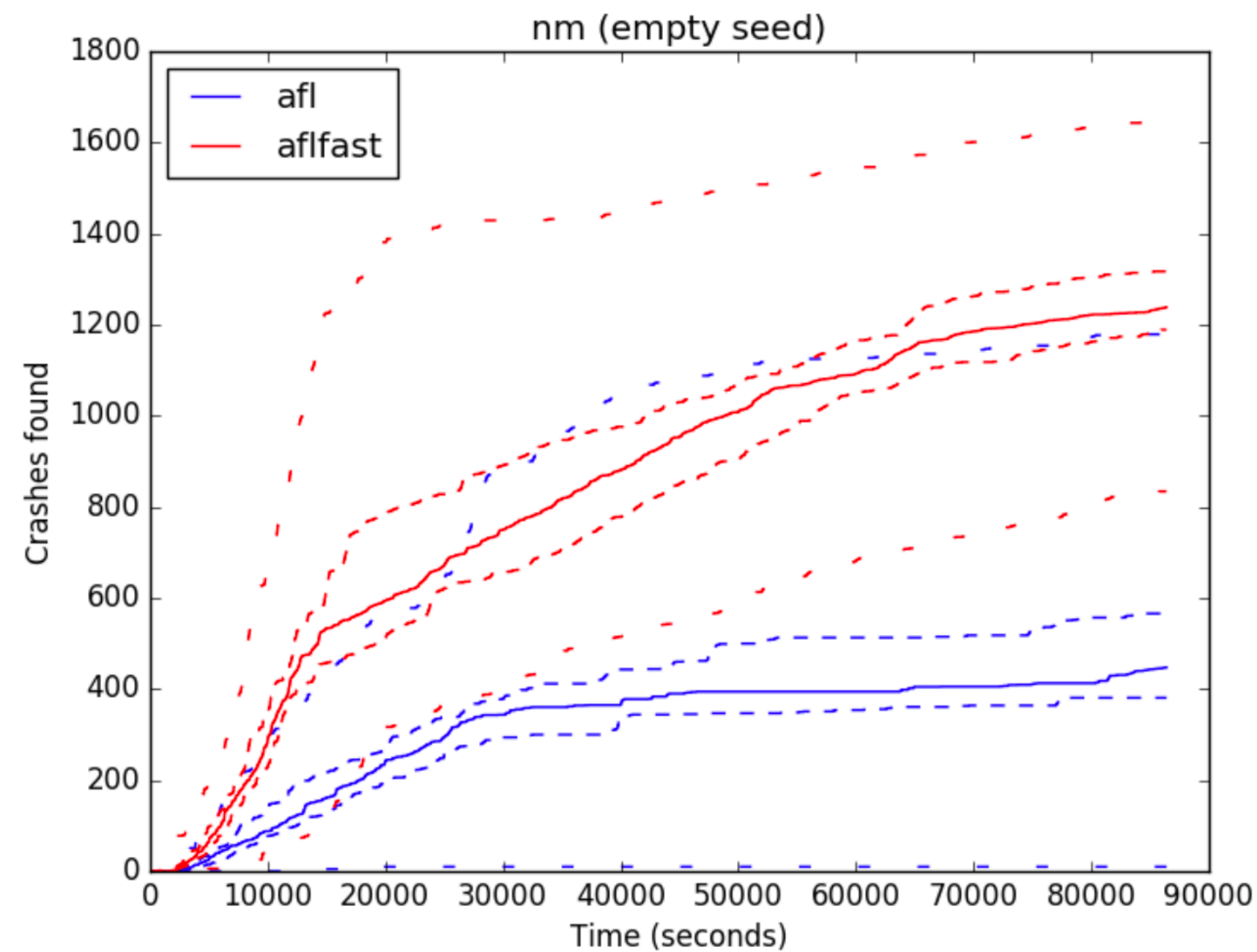


paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[7]	R(29)				G	?	V	-
FuzzSim[44]	R(101)	B	100	C	S		R/M	10D
Dowser[18]	R(7)	O	?		O		V	8H
COVERSET[38]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[8]	R(8)	A, B, Z			S		M	1H
MutaGen[23]	R(8)	R, Z			S	L	V	24H
SDF[28]	R(1)	Z, O			O		V	5D
Driller[41]	C(126)	A			G	L, E	V	24H
QuickFuzz-1[16]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[43]	R(5)	O			M	O	G, R	2H
[46]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[37]	C(63), L, R(10)	A			G, S, O		V	6H, 24H
SlowFuzz[35]	R(10)	O	100		-		V	
Steelix[26]	C(17), L, R(5)	A, V, O			C, G	L, E, M	V	5H
Skyfire[42]	R(4)	O			?	L, M	R, G	LONG
kAFL[39]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[11]	R(7)	O			G*		G	5H
Orthrus[40]	G, R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[22]	R(1)	O			G*		G	-
VDF[21]	R(18)				C	E	V	30D
QuickFuzz-2[17]	R(?)	O	10		G*		G, M	
IMF[19]	R(105)	O			G*	O	G	24H
[48]	S(?)	O	5		G		G	24H
NEZHA[34]	R(6)	A, L, O			O		R	
[45]	G	A, L					V	5M
S2F[47]	L, R(8)	A, O			G	O	V	5H, 24H
FairFuzz[25]	R(9)	A	20	C	C	E	V/M	24H
Angora[9]	L, R(8)	A, V, O	5		G, C	L, E	V	5H
T-Fuzz[33]	C(296), L, R(4)	A, O			C, G*		V	24H
MEDS[20]	S(2), R(12)	O	10		C		V	6H

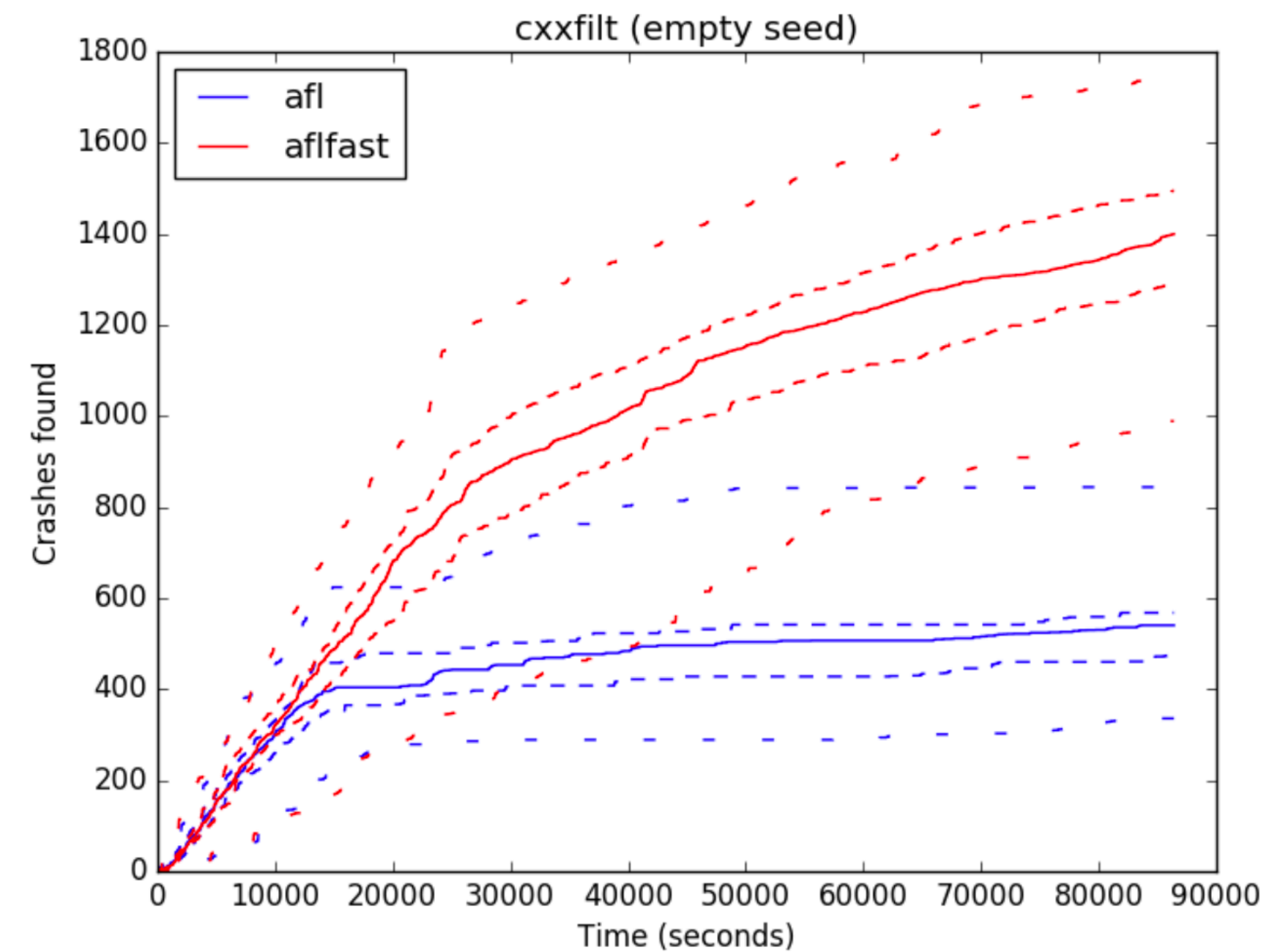
# Evaluations

- 10/32 papers ran 24 hours
- 7/32 papers ran 5 or 6 hours
- Others less, or much more
  - Minutes ... or months!
- Question: How much does this choice matter?

# Trends can be Stable



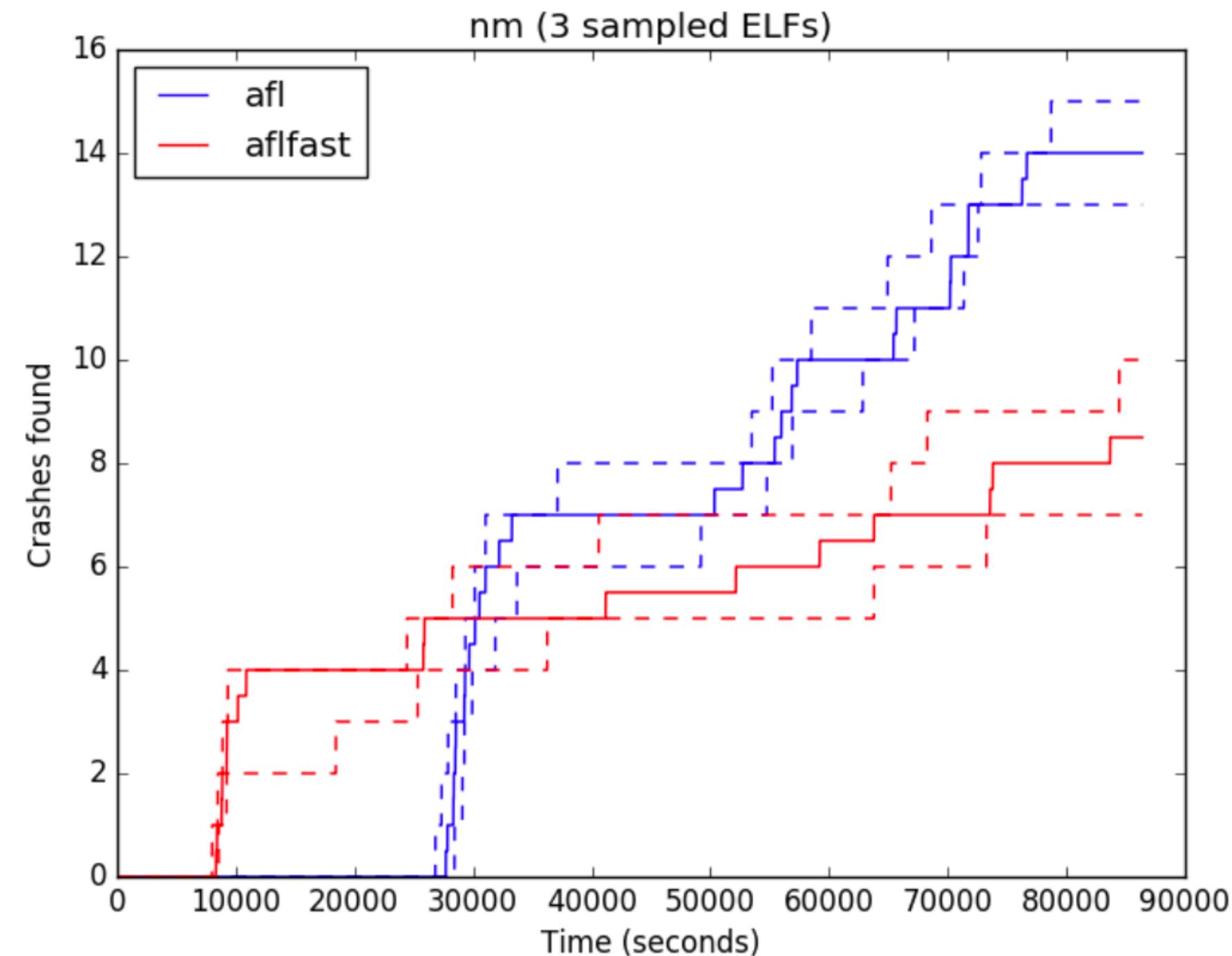
$p < 10^{-13}$



$p < 10^{-10}$

**AFLFast** better at  
**5, 8, 24 hours**

# Trends can Change



Can take time for  
fuzzing to “warm up”

3-sampled

**6 hours:**  $p < 10^{-13}$

**AFLFast** is better

**24 hours:**  $p = 0.000105$

**AFL** is better

# Timeouts: Recommendations

- **Longer timeouts are better** because they subsume shorter ones
  - Using plots like ones we've shown earlier, **performance** can be compared **at different points in time**
- But there is a **practical limit to long timeouts**
  - Hard to work on substantial program corpus over weeks or months
- **24 hours seems like a good target**
  - Ecologically relevant
    - But longer would be even better!
  - Subsumes common 5 and 8 hour limits



# Assessing Performance

# Performance Metrics

- Ultimate “**ground truth**”: **Bugs**
  - Finding lots of different inputs whose root cause is the same bug is not that useful (maybe, harmful!)
- **Some benchmarks** designed with **known bugs**
  - Crash has telltale sign
- For others: *Which crash signals which bug?*
- *Heuristics*: **Stack hash** and coverage (**AFL CMIN**)

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[7]	R(29)				G	?	V	-
FuzzSim[44]	R(101)	B	100	C	S		R/M	10D
Dowser[18]	R(7)	O	?		O		V	8H
COVERSET[38]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[8]	R(8)	A, B, Z			S		M	1H
MutaGen[23]	R(8)	R, Z			S	L	V	24H
SDF[28]	R(1)	Z, O			O		V	5D
Driller[41]	C(126)	A			G	L, E	V	24H
QuickFuzz-1[16]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[43]	R(5)	O			M	O	G, R	2H
[46]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[37]	C(63), L, R(10)	A			G, S, O		V	6H, 24H
SlowFuzz[35]	R(10)	O	100		-		V	
Steelix[26]	C(17), L, R(5)	A, V, O			C, G	L, E, M	V	5H
Skyfire[42]	R(4)	O			?	L, M	R, G	LONG
kAFL[39]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[11]	R(7)	O			G*		G	5H
Orthrus[40]	G, R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[22]	R(1)	O			G*		G	-
VDF[21]	R(18)				C	E	V	30D
QuickFuzz-2[17]	R(?)	O	10		G*		G, M	
IMF[19]	R(105)	O			G*	O	G	24H
[48]	S(?)	O	5		G		G	24H
NEZHA[34]	R(6)	A, L, O			O		R	
[45]	G	A, L					V	5M
S2F[47]	L, R(8)	A, O			G	O	V	5H, 24H
FairFuzz[25]	R(9)	A	20	C	C	E	V/M	24H
Angora[9]	L, R(8)	A, V, O	5		G, C	L, E	V	5H
T-Fuzz[33]	C(296), L, R(4)	A, O			C, G*		V	24H
MEDS[20]	S(2), R(12)	O	10		C		V	6H

# Evaluations

- 8 used AFL CMIN (“unique crashes”) (C)
- 7 used stack hashes (S)
- 7 assessed ground truth perfectly (G)
  - 8 others did, in part (“case study”, G\*)
- For C and S: How effective at predicting G?

# AFL CMIN

- A crashing input is considered “unique” if either
  - the **coverage profile includes an edge (“tuple”) not seen** in any of the previous crashes
  - the profile is **missing a tuple always present in earlier faults**
- AFL calls this *CMIN*. Docs justify it by saying:
  - Just using the faulting location will result in false negatives
    - Might be a common sink for distinct bugs
  - Hashing a stack trace will inflate counts (false positives)
    - if the crash site can be reached through a number of different, possibly recursive code paths
- But **CMIN may suffer from inflated counts**, too



# False Positives

```
int main(int argc, char* argv[]) {  
    if (argc >= 2) {  
        char b = argv [1][0];  
        if (b == 'a')    crash( );  
        else            crash( );  
    }  
    return 0;  
}
```

- Bug is in `crash( )`
- But different inputs that lead to `crash( )` will be treated as distinct
- They have different control-flow edges

# (Fuzzy) Stack Hashes

- Idea: **Identify bug** according to the **stack at the time of the crash** (return addresses)
  - Or: Limit attention to the top N frames (where N is between 3 and 5 in most papers)
- Rationale: Faulting location highly indicative of source of bug
  - Stack provides necessary context (i.e., when faulting function given a input, only from certain caller)
  - But some “context” may be superfluous
    - Assume: frames closer to bug more relevant

# False Positives and Negatives

```
void f() { ... format(s1); ... }
void g() { ... format(s2); ... }
void format(char *s) {
    //bug: corrupt s
    prepare(s);
}
void prepare(char *s) {
    output(s);
}
void output(char *s) {
    //failure manifests
}
```

- With N=3, distinct calls to `format` from `f` and `g` will be conflated, properly
- But with N=5, calling `format` from `f` and `g` are made distinct
  - Overcounting
- With N=2, a bug in a different caller to `prepare` that corrupts its argument will be conflated with the `format` bug
  - Undercounting

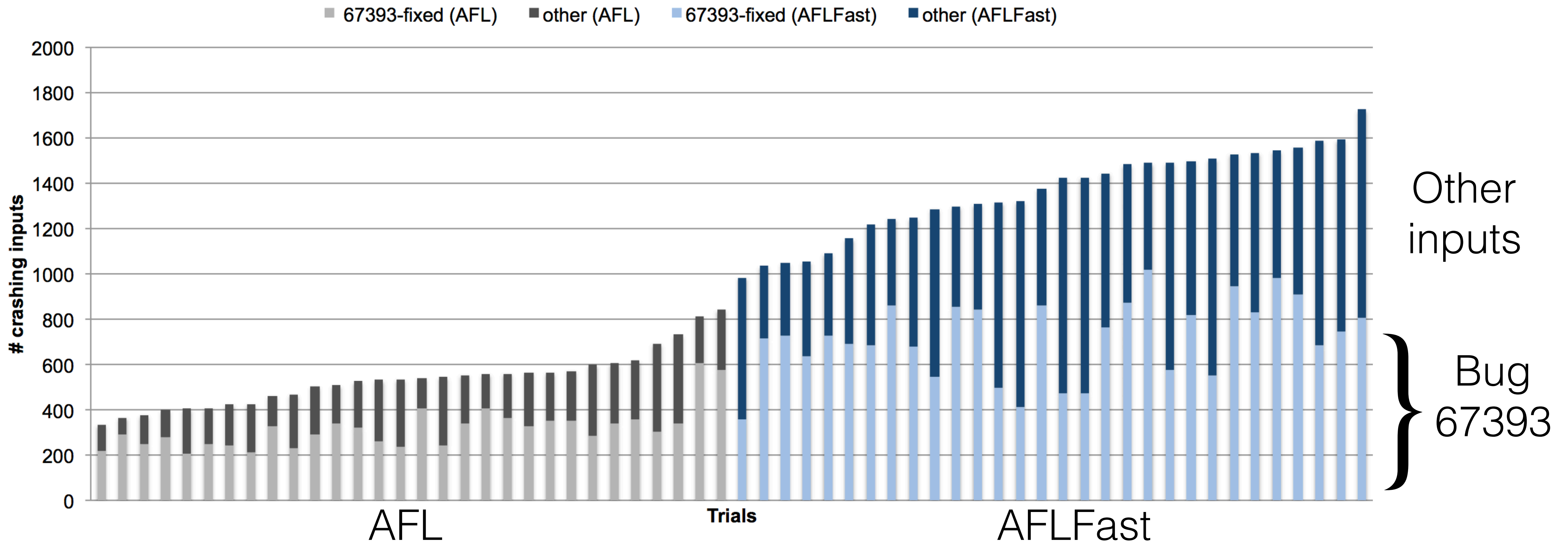
# Assessing Heuristics

<pre># Line 419 static struct demangle_component *d_sour 423 424 static long d_number (struct d_info *); 425 426 static struct demangle_component *d_identifier (struct d_info *, int); 427 428 static struct demangle_component *d_operator_name (struct d_info *); 429 # Line 715 d_dump (struct demangle_component *dc, i 719 case DEMANGLE_COMPONENT_FIXED_TYPE: 720 printf ("fixed-point type, accum? %d, sat? %d\n", 721 dc-&gt;u.s_fixed.accum, dc-&gt;u.s_fixed.sat); 722 d_dump (dc-&gt;u.s_fixed.length, indent + 2) 723 break; 724 case DEMANGLE_COMPONENT_ARGLIST: 725 printf ("argument list\n"); # Line 1656 d_number_component (struct d_info *di) 1660 /* identifier ::= &lt;(unqualified source code identifier)&gt; */ 1661 1662 static struct demangle_component * 1663 d_identifier (struct d_info *di, int len) 1664 { 1665 const char *name; 1666 # Line 1677 d_identifier (struct d_info *di, int len 1681 /* Look for something which looks like a gcc encoding of an 1682 anonymous namespace, and replace it with a more user friendly 1683 name. */ 1684 if (len &gt;= (int) ANONYMOUS_NAMESPACE_PREFIX_LEN + 2 1685 &amp;&amp; memcmp (name, ANONYMOUS_NAMESPACE_PREFIX, 1686 ANONYMOUS_NAMESPACE_PREFIX_LEN) == 0) 1687 {</pre>	<pre>Line 423 static struct demangle_component *d_sour static long d_number (struct d_info *); static struct demangle_component *d_identifier (struct d_info *, long); static struct demangle_component *d_operator_name (struct d_info *); Line 719 d_dump (struct demangle_component *dc, i case DEMANGLE_COMPONENT_FIXED_TYPE: printf ("fixed-point type, accum? %d, sat? %d\n", dc-&gt;u.s_fixed.accum, dc-&gt;u.s_fixed.sat); d_dump (dc-&gt;u.s_fixed.length, indent + 2); break; case DEMANGLE_COMPONENT_ARGLIST: printf ("argument list\n"); Line 1660 d_number_component (struct d_info *di) /* identifier ::= &lt;(unqualified source code identifier)&gt; */ static struct demangle_component * d_identifier (struct d_info *di, long len) { const char *name; Line 1681 d_identifier (struct d_info *di, int len /* Look for something which looks like a gcc encoding of an anonymous namespace, and replace it with a more user friendly name. */ if (len &gt;= (long) ANONYMOUS_NAMESPACE_PREFIX_LEN + 2 &amp;&amp; memcmp (name, ANONYMOUS_NAMESPACE_PREFIX, ANONYMOUS_NAMESPACE_PREFIX_LEN) == 0) {</pre>
--	--

- Used bug tracker to find patches since fuzzed version
- Picked 67393 that fixed an integer overflow
- Applied just that fix to the baseline and re-ran against all 57,000 crashing inputs (post-CMIN)
- Those that no longer crash are due to this bug
- Re-run must account for non-determinism
- Used valgrind: “non crash” only if it found no issue



# CMIN Results



*31 124 total inputs found for this bug*

# Stack Hash Results

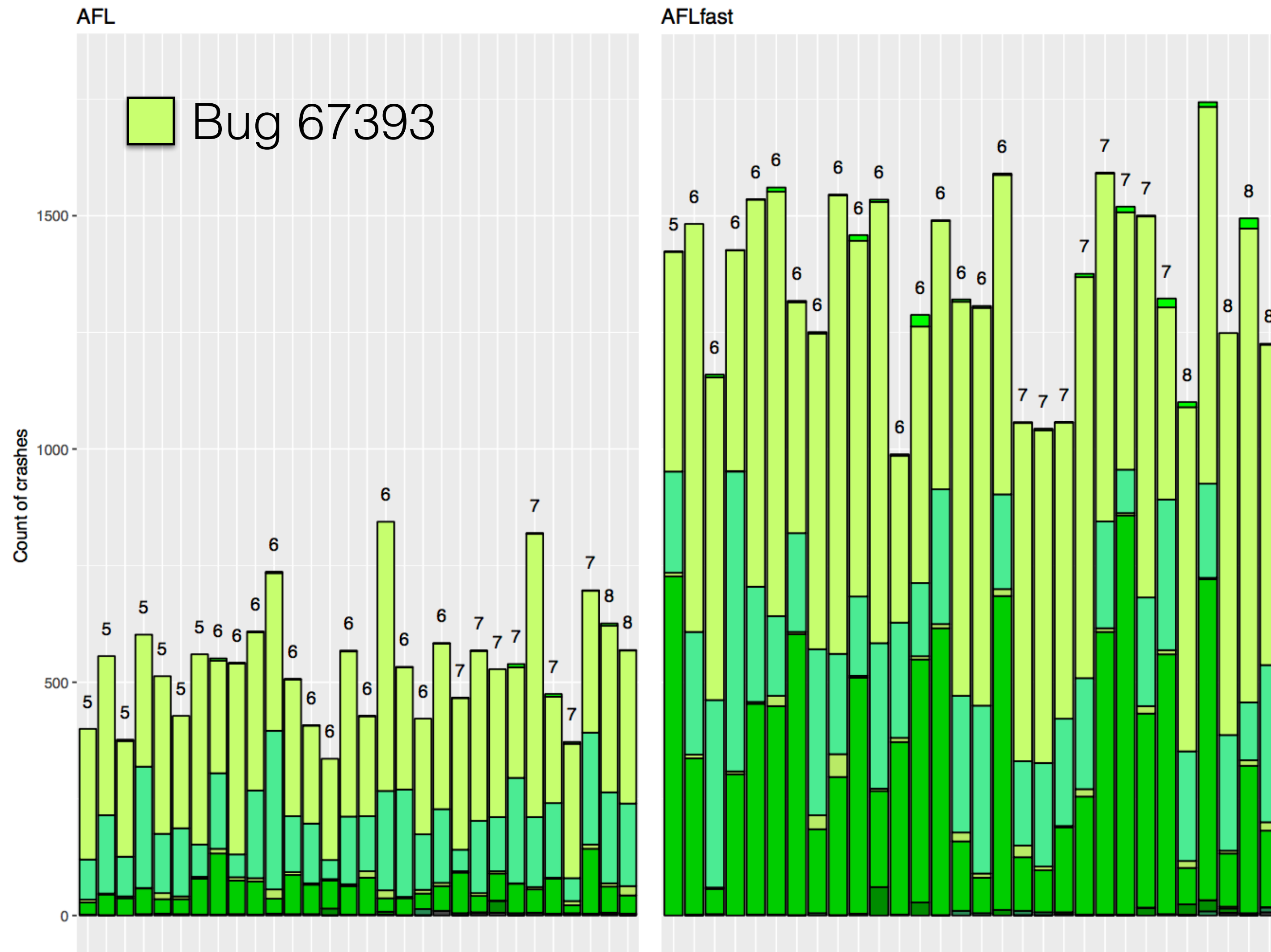
Bug fix to <i>cxxfilt</i>	Distinct Hashes	Matches	False Matches
Bug 67393	336	311	25

- Computed stack hashes (N=5) for all 31 124 inputs corresponding to bug
  - 336 distinct stack hashes
  - or 12 out of 500 CMIN (average on a per-trial basis)
    - Much better!
- But: only 311 distinct: 25 also matched another bug
  - False negatives; might mean missed bugs!

# Full Triage for Cxxfilt

- We considered all Git commits from the version of cxxfilt with tested on until the present
- We applied each commit and retested each input
  - Those that now passed were grouped with that commit
- We examined commits to see if they should be considered multiple bug fixes, rather than just one
  - Split a big commit into 5 smaller ones — part of an *en masse* merge of trunks (includes 67393)
- No results for stack hashes as yet

# cxxfilt: AFL CMIN vs. Bugs



- 13 total bugs
- No trial found more than 8
- 3 bugs account for most crashing inputs
- Bug 67393 the most inputs
- Number of crashing inputs correlates with number of bugs, but only loosely
- Mann Whitney p-value is .091 for AFLFast *bugs* > AFL bugs
- vs.  $10^{-10}$  for “unique” crashes



# What is a (single) Bug?

- All of the previous discussion assumes that we can **identify one bug as distinct from another**
  - Maybe we didn't split patches as much as we should have, and so heuristics better than we've said
- But it turns out that “bug” is a slippery concept
- Proposal: A **bug is a code fragment** (or lack thereof) that contributes to one or more failures. By “contributes to”, we mean that the **buggy code fragment** is **executed** (or **should have been**, but was missing) when the failure happens
  - <http://www.pl-enthusiast.net/2015/09/08/what-is-a-bug/>

# Metrics Summary

- This is just one program and set of fuzzing results, but it shows the potential for heuristics to
  - **Massively overcount bugs** (false positives)
  - **Miss bugs** (false negatives)
  - The good news is that the situation seems tilted toward the former
- As such, it seems prudent to attempt to **measure ground truth directly**
  - Use benchmarks with known bugs
  - Might still use other programs, to avoid overfitting

# Q: Better Heuristic?

- If CMIN and Stack Hashes are poor, perhaps there's room to do better, even if not perfectly
- Relies on (at least partially) answering the “what is a single bug?” question
- We are starting to explore some ideas here

# Q: Improve the Search?

- Our results overall show that there can be a fair bit of variance in performance from run to run
  - esp. when counting crashes
- Indeed, no cxxfilt run found all 13 bugs
  - Found a few common in common but then varied a fair bit on the rare ones
- Perhaps the fuzzing search is hitting a local minima, and so “rebooting” helps
  - A similar observation underpins search in SAT solvers today



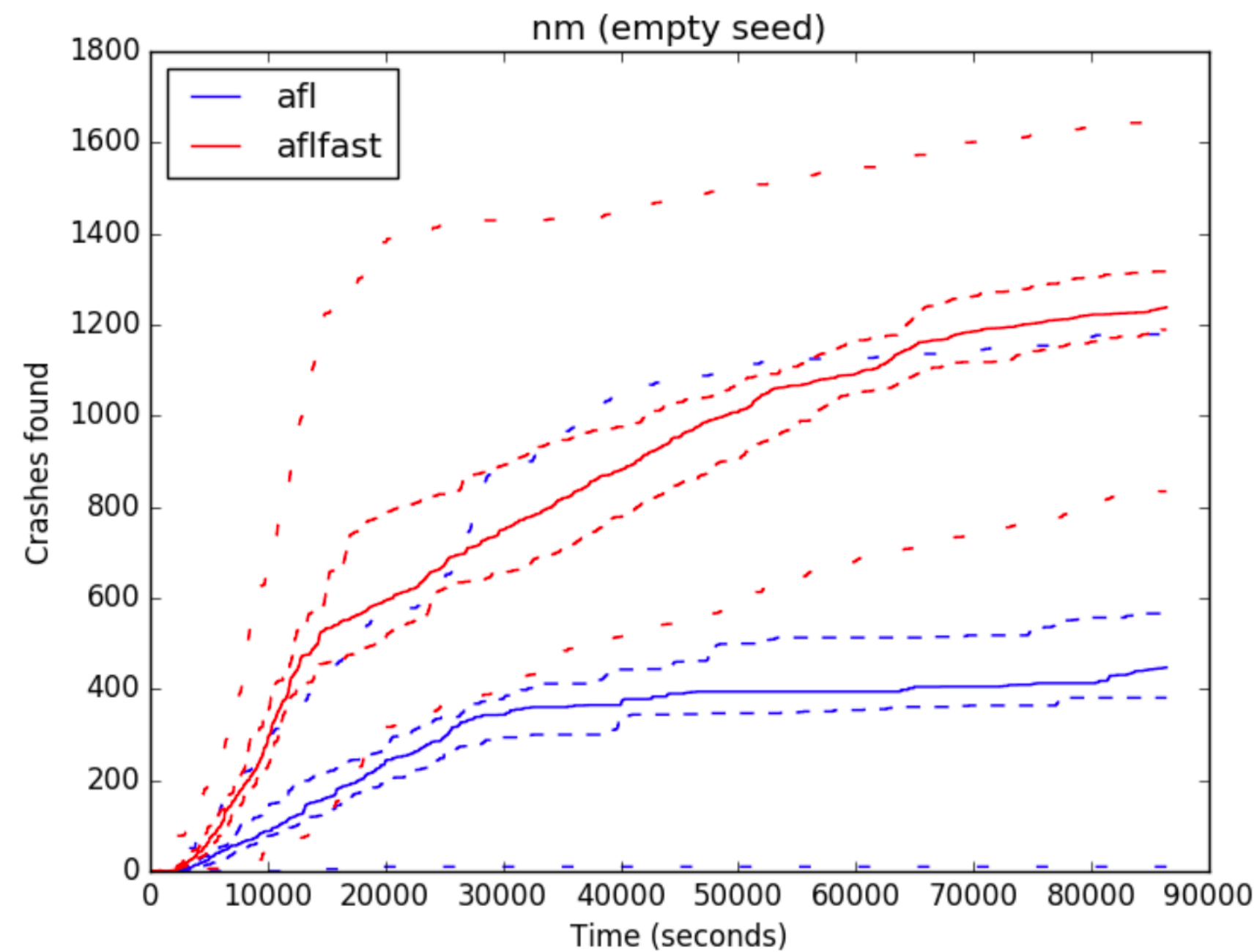
# Target Programs

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[7]	R(29)				G	?	V	-
FuzzSim[44]	R(101)	B	100	C	S		R/M	10D
Dowser[18]	R(7)	O	?		O		V	8H
COVERSET[38]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[8]	R(8)	A, B, Z			S		M	1H
MutaGen[23]	R(8)	R, Z			S	L	V	24H
SDF[28]	R(1)	Z, O			O		V	5D
Driller[41]	C(126)	A			G	L, E	V	24H
QuickFuzz-1[16]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[43]	R(5)	O			M	O	G, R	2H
[46]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[37]	C(63), L, R(10)	A			G, S, O		V	6H, 24H
SlowFuzz[35]	R(10)	O	100		-		V	
Steelix[26]	C(17), L, R(5)	A, V, O			C, G	L, E, M	V	5H
Skyfire[42]	R(4)	O			?	L, M	R, G	LONG
kAFL[39]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[11]	R(7)	O			G*		G	5H
Orthrus[40]	G, R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[22]	R(1)	O			G*		G	-
VDF[21]	R(18)				C	E	V	30D
QuickFuzz-2[17]	R(?)	O	10		G*		G, M	
IMF[19]	R(105)	O			G*	O	G	24H
[48]	S(?)	O	5		G		G	24H
NEZHA[34]	R(6)	A, L, O			O		R	
[45]	G	A, L					V	5M
S2F[47]	L, R(8)	A, O			G	O	V	5H, 24H
FairFuzz[25]	R(9)	A	20	C	C	E	V/M	24H
Angora[9]	L, R(8)	A, V, O	5		G, C	L, E	V	5H
T-Fuzz[33]	C(296), L, R(4)	A, O			C, G*		V	24H
MEDS[20]	S(2), R(12)	O	10		C		V	6H

# Evaluations

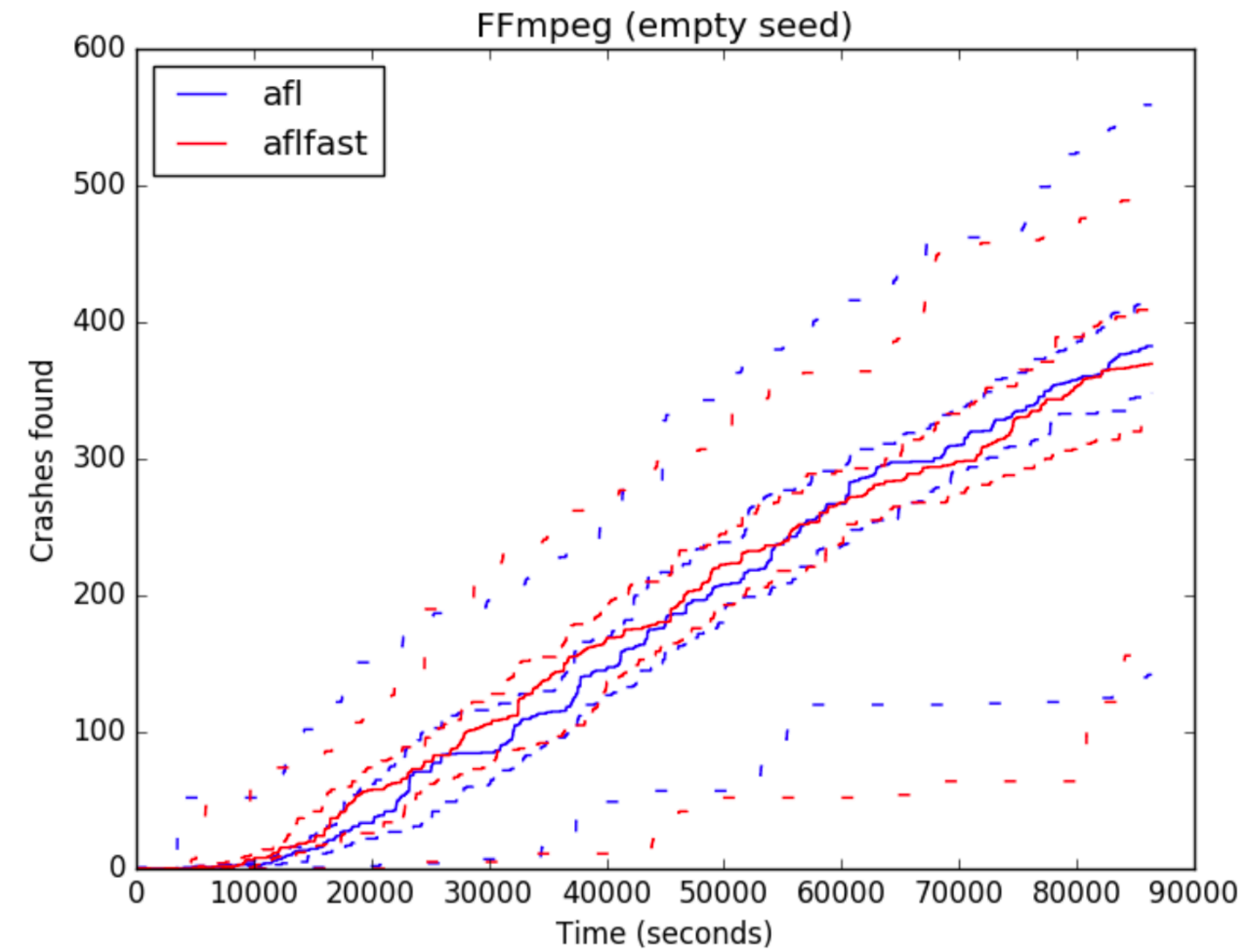
- 30/32 used real programs
  - Typically 5-10, as many as 100, but vary a fair bit across papers
  - 2/32 use Google Test Suite
  - Fair/sufficient sample?
- 8/32 purposely-vulnerable programs (or injected bugs)
  - 5/32 use LAVA-M
  - 4/32 use CGC
  - Ecological validity?

# Binutils vs. Image proc.



$p < 10^{-13}$

From AFLFast paper



$p = 0.379$

From VUzzer paper

# Google Fuzz Test Suite

- <https://github.com/google/fuzzer-test-suite>
- **24 programs** and libraries with **known bugs**
  - OpenSSL, PCRE, SQLite, libpng, libxml2, libarchive, ...
- Comes with harness to connect to AFL and libfuzzer
  - And confirm when a bug is discovered
- This is a sort of regression suite, so its **generality is not entirely clear**
- Also, Google OSS-Fuzz project
  - <https://github.com/google/oss-fuzz>



# Cyber Grand Challenge

- CGC is a suite of 296 programs constructed for DARPA's Cyber Grand Challenge
  - **Intended to be ecologically valid**, but also intended to be challenging (gamification)
  - **Validity not tested**
  - And subset in many papers
- Good feature: **Known ground truth** (telltale sign when bug is triggered)
- <https://github.com/trailofbits/cb-multios>

# LAVA-M

- **LAVA** is a **bug injection** methodology that adds “magic number checks” to inputs that otherwise do not affect control flow (much)
- LAVA-M is the result of using it to inject bugs in four open-source programs (**base64**, **md5sum**, **uniq**, and **who**)
  - 2000+ bugs injected in who (!)
- *“A significant chunk of future work for LAVA involves making the generated corpora look more like the bugs that are found in real programs.”*
- <http://moyix.blogspot.com/2016/10/the-lava-synthetic-bug-corpora.html>

# A Fuzzing Benchmark?

- A substantial (large) sample of relevant programs (look at the breadth of existing fuzzing papers)
  - Some justification for ecological validity
- Should know ground truth
- Fuzzers should not overfit to the benchmark
  - Perhaps run a sample from a larger population
  - May want to include non-benchmark programs too, despite not necessarily having ground truth
- Google Fuzz, CGC, LAVA-M, current papers may be good starting points






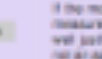
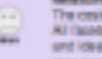
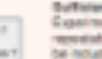



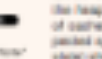

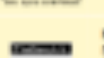
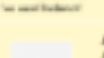




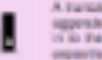

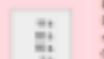
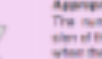
# Summary: Do's and Don'ts

- Do assess a random process **using multiple trials and a statistical test**
  - Don't run just one trial
  - Don't compute just the mean/median
- **Don't use heuristics** as only **performance measure**
  - Some results should be based on ground truth
- Do **clarify choice of seed**
  - Evaluate choices and understand which is best
- Do use **longer timeout** and measure performance over time





# General advice: SIGPLAN guidelines!

SIGPLAN Empirical Evaluation Checklist	
This checklist is meant to support informed judgement; not a substitute for it	
<div><input type="checkbox"/> <b>Clearly Stated Claims</b> <i>Example Best Practice</i></div> <div></div> <div><b>Explicit Claims</b> Claims that are explicit in order for the reader to assess whether the empirical evaluation supports them. Claims should aim to state not just what is achieved but how.</div> <div></div> <div><b>Appropriately-Scaled Claims</b> The main or claims should follow from the evidence provided. Overstating is often the consequence of inadequate evidence, e.g., claiming works for all Java, but evaluating only a single subset of running worlds (small test-world), but evaluating only in a particular situation.</div> <div></div> <div><b>Acknowledges Limitations</b> A paper should acknowledge its limitations to place the scope of results in context. Stating no limitations at all, or only tangential ones while omitting the more relevant ones, may mislead the reader in drawing too-strong conclusions.</div>	<div><input type="checkbox"/> <b>Relevant Metrics</b> <i>Example Best Practice</i></div> <div></div> <div><b>Select an Appropriate Proxy Metric</b> If the most relevant evaluation metric is not (or cannot be) measured directly, the proxy metric used should be well justified. For example, a reduction in cache misses is not an appropriate proxy for reduction in peak performance or energy consumption.</div> <div></div> <div><b>Measures All Important Effects</b> The most performance of a technique may be multi-faceted. All results should be considered, both costs and benefits, and clearly evaluated. For example, compiler optimizations may speed programs at the cost of drastically increasing compile time.</div> <div></div> <div><b>Sufficient Information to Repeat</b> Experiments should be described in sufficient detail to be repeatable. All parameters (including default values) should be included as well as all version numbers of software, and full details of hardware platforms.</div>
<div><input type="checkbox"/> <b>Suitable Comparison</b> <i>Example Best Practice</i></div> <div></div> <div><b>Appropriate Baseline for Comparison</b> An empirical evaluation of a contribution that improves upon the state-of-the-art should evaluate that contribution against an appropriate baseline, such as the current best or best competitor or a random baseline.</div> <div></div> <div><b>Fair Comparison</b> Comparisons to a competing system should not unfairly disadvantage that system. For example, clearly, the compared systems would be compiled with the same compiler and optimization flags.</div>	<div><input type="checkbox"/> <b>Appropriate and Clear Experimental Design</b> <i>Example Best Practice</i></div> <div></div> <div><b>Reusable Platform</b> The evaluation should be on a platform that can reasonably be said to match the claims. For example, a claim that a task is performed on multi-processor should not have an evaluation performed exclusively on a server.</div> <div></div> <div><b>Explores Key Design Parameters</b> Key parameters should be explored over a range to establish sensitivity to their settings. Examples include the size of the heap when evaluating garbage collection and the size of caches when evaluating a memory optimization. All essential system configurations (e.g., host setup) to which a claim should be considered.</div> <div></div> <div><b>Open Loop or Workload Generator</b> Load generators for system transaction-oriented systems should not be used to generate the data for the system response. Rather, the load generator should be open loop, generating work independent of the performance of the system under test. See [Schwartz et al., 2004].</div>
<div><input type="checkbox"/> <b>Principal Benchmark Choice</b> <i>Example Best Practice</i></div> <div></div> <div><b>Appropriate Suite</b> Evaluations should be conducted using the appropriate established benchmarks where they exist. Established suites should be used if the designer for context, for example, it would be wrong to use a single-threaded suite for studying parallel performance.</div> <div></div> <div><b>Non-Standard Suites Justified</b> Sometimes an established benchmark suite does not exist. A rationale should be provided for the selection of a non-standard benchmark or a subset of established benchmark suites.</div>	<div></div> <div><b>Cross-Validation Where Needed</b> When a system state is by general but was developed by training on or close consideration of specific examples, it is essential that the evaluation explicitly perform cross-validation, so that the system is evaluated on data distinct from the training set.</div> <div></div> <div><b>Comprehensive Summary Results</b> Appropriate results should be used to characterize the full range of results, not just the most favorable values, which may be outliers. For example, it is not appropriate to summarize as speedups of 40%, 60%, 70%, and 80% as up to 80%.</div>
<div><input type="checkbox"/> <b>Adequate Data Analysis</b> <i>Example Best Practice</i></div> <div></div> <div><b>Sufficient Number of Trials</b> In modern systems, which have non-deterministic performance, a small number of runs (e.g., a single run measurement) may be noisy. Similarly, there is an issue with measuring the system in a non-steady state (e.g., this is a steady state that avoids warm-up effects).</div> <div></div> <div><b>Appropriate Summary Statistics</b> There are many summary statistics, and each presents an accurate view of a dataset only under appropriate circumstances. For example, the geometric mean should only be used when comparing results with different ranges, and the harmonic mean when comparing rates. When distributions have outliers, a median should also be presented.</div>	<div><input type="checkbox"/> <b>Appropriate Presentation of Results</b> <i>Example Best Practice</i></div> <div></div> <div><b>Axes Include Zero</b> A bar chart (with or without including zero) can exaggerate the importance of a difference. When comparing in the interesting range of an axis can sometimes add perspective, there is a significant risk that this is misleading, especially if it is not immediately clear that the axis is that scaled.</div> <div></div> <div><b>Ranking Plotted Correctly</b> When rates (e.g., speedups) are plotted on the graph, the use of the bar height to represent the magnitude of the change. For example, 2.0 and 3.0 are not plotted, but their true distance from 1.0 is not plotted. This misleading effect can be avoided either by using a logarithmic or by normalizing to the lowest (rightmost) value.</div>
<div></div> <div><b>Report Data Distribution</b> Reporting just a measure of central tendency (e.g., a mean or median) fails to capture the extent of any non-determinism. A measure of variability (e.g., variance, and deviation) quantified under confidence intervals help to understand the distribution of the data.</div>	<div></div> <div><b>Appropriate Level of Precision</b> The number of significant digits should reflect the precision of the experiment. Reporting measurements of 0.0001 when the experimental error is <math>\approx 1\%</math> is an example of this. The number of significant digits should reflect the precision of the experiment.</div>

PDF: <http://www.sigplan.org/Resources/EmpiricalEvaluation/>

June 2016: E. G. Berger, S. M. Blackburn, M. Hauswirth, and M. Hicks for the ACM SIGPLAN 2016

<http://sigplan.org/Resources/EmpiricalEvaluation/>