Fuzzing
and how to evaluate it

Michael Hicks
The University of Maryland

MARYLAND CYBERSECURITY CENTER

PLUM

HotSOS 2020

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

Presented at ACM CCS Conference, October 2018

Won NSA Best Scientific Cybersecurity Paper Award, Sep. 2019

Evaluating Fuzz Testing

George Klees, Andrew Ruef, Benji Cooper
University of Maryland

Shiyi Wei
University of Texas at Dallas

Michael Hicks
University of Maryland

ABSTRACT

Fuzz testing has enjoyed great success at discovering security critical bugs in real software. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Such new ideas are primarily evaluated experimentally so an important question is: What experimental setup is needed to produce trustworthy results? We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments. We conclude with some guidelines that we hope will help improve experimental evaluations of fuzz testing algorithms, making reported results more robust.

CCS CONCEPTS

• **Security and privacy → Software and application security;**

KEYWORDS

fuzzing, evaluation, security

ACM Reference Format:
George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243804

Why do we think fuzzers work? While inspiration for new ideas may be drawn from mathematical analysis, fuzzers are primarily evaluated experimentally. When a researcher develops a new fuzzer algorithm (call it *A*), they must empirically demonstrate that it provides an advantage over the status quo. To do this, they must choose:

- a compelling *baseline* fuzzer *B* to compare against;
- a sample of target programs—the *benchmark suite*;
- a *performance metric* to measure when *A* and *B* are run on the benchmark suite; ideally, this is the number of (possibly exploitable) bugs identified by crashing inputs;
- a meaningful set of *configuration parameters*, e.g., the *seed file* (or files) to start fuzzing with, and the *timeout* (i.e., the duration) of a fuzzing run.

An evaluation should also account for the fundamentally random nature of fuzzing: Each fuzzing run on a target program may produce different results than the last due to the use of randomness. As such, an evaluation should measure *sufficiently many trials* to sample the overall distribution that represents the fuzzer's performance, using a *statistical test* [38] to determine that *A*'s measured improvement over *B* is real, rather than due to chance.

Failure to perform one of these steps, or failing to follow recommended practice when carrying it out, could lead to misleading or incorrect conclusions. Such conclusions waste time for practitioners, who might profit more from using alternative methods or configurations. They also waste the time of researchers, who make overly strong assumptions based on an arbitrary tuning of

- The paper looks critically at the how potential advances in fuzz testing are evaluated scientifically
- This talk will cover the content in that paper, which is still relevant today

How should we scientifically evaluate potential advances in randomized testing (*fuzzing*) technology?

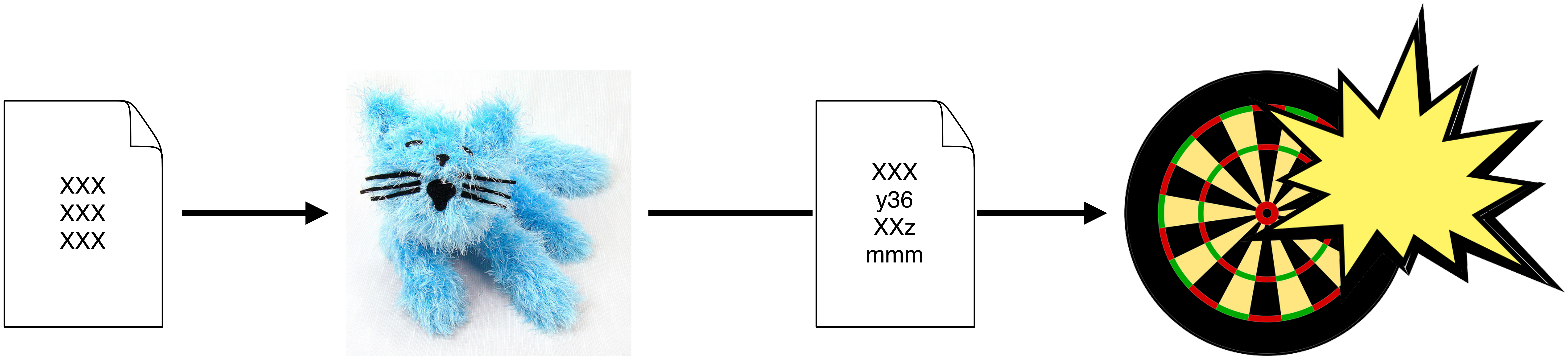
- I will also briefly look at the impact of the paper since it was published — have things changed?

What is fuzzing?

- A kind of **testing** based on **random input generation**
- **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 (e.g., according to a grammar)
- **Run the target program** with them
- See **what happens**
- Repeat



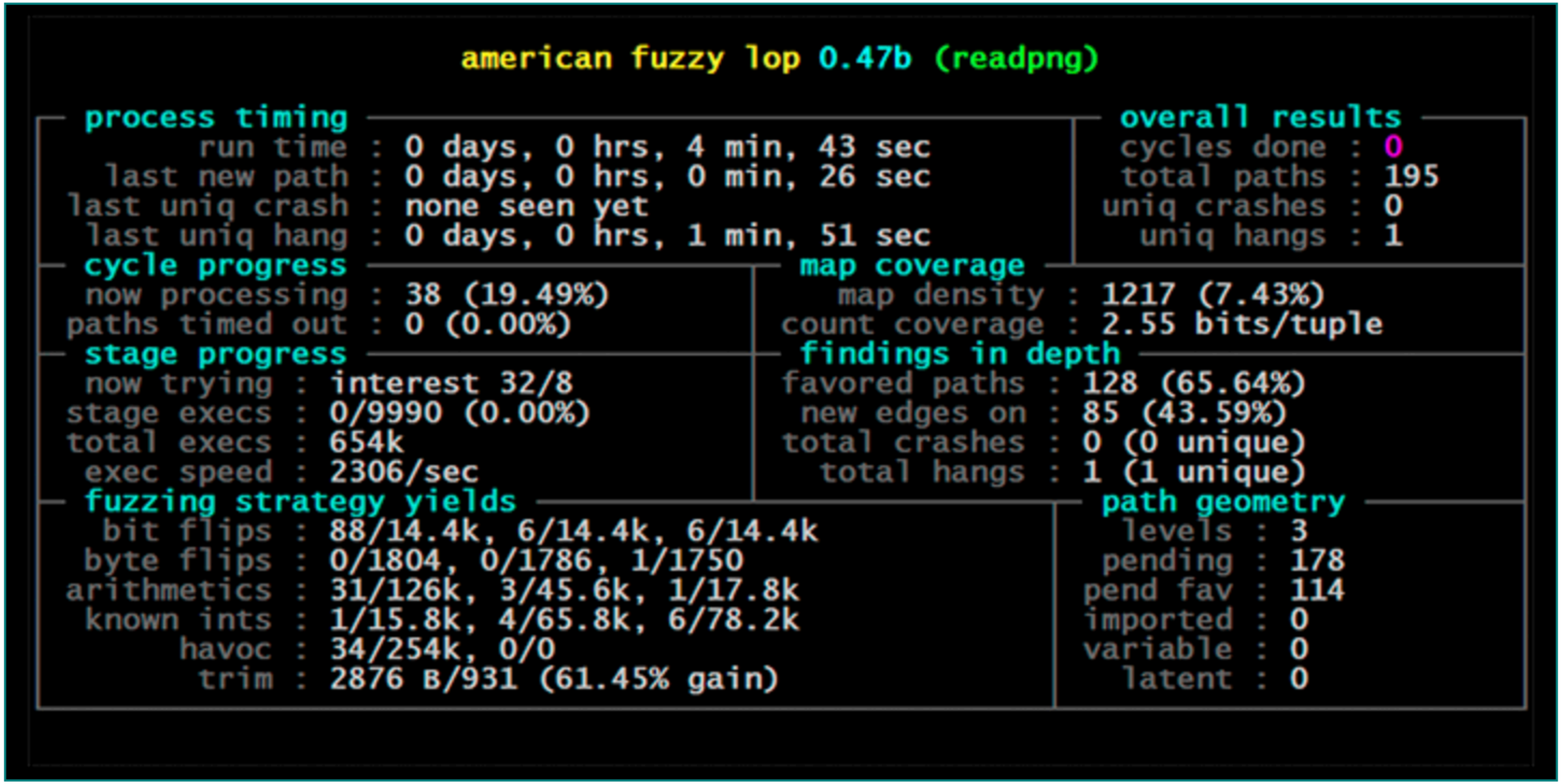
American Fuzzy Lop (AFL)

- AFL is a ***mutation**-based*, ***gray-box** fuzzer* - de-facto standard
  - **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
overall results
cycles done : 0
total paths : 195
uniq crashes : 0
uniq hangs : 1
cycle progress
now processing : 38 (19.49%)
paths timed out : 0 (0.00%)
map coverage
map density : 1217 (7.43%)
count coverage : 2.55 bits/tuple
stage progress
now trying : interest 32/8
stage execs : 0/9990 (0.00%)
total execs : 654k
exec speed : 2306/sec
findings in depth
favored paths : 128 (65.64%)
new edges on : 85 (43.59%)
total crashes : 0 (0 unique)
total hangs : 1 (1 unique)
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)
path geometry
levels : 3
pending : 178
pend fav : 114
imported : 0
variable : 0
latent : 0

Active Area of Research

- **Black box**: CERT Basic Fuzzing Framework (BFF), Zzuf, …
- **Gray box:** VUzzer, Fairfuzz, T-Fuzz, AFLFast, Angorra, Parmesan, Zest, EcoFuzz, GREYONE,…
- White box: KLEE, angr, SAGE, Mayhem, …
- **Hybrid**: Pangolin, QSYM, Driller, …

There are many more …

Evaluating Fuzzing

an adventure in the scientific method

Assessing Progress

- Since fuzzing is an active area, we can assume the technology is 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

for Advanced Fuzzer (call it A)

Requires

- 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, justifable **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** from 2012-2018 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

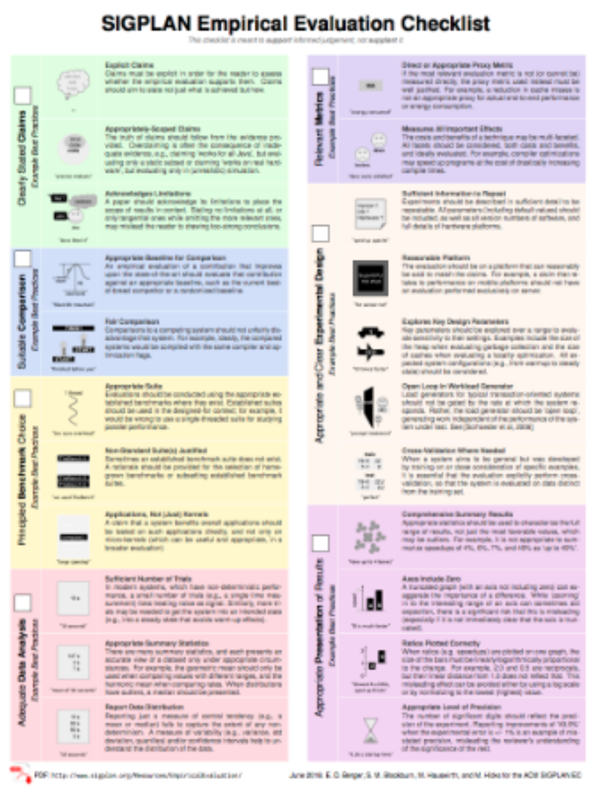
- Failure to follow the template may not mean reported results are wrong
  - *Potential* for wrong conclusions
- **We carried out experiments** to start to assess this potential
  - Goal is to get a sense of whether the evaluation problem is real
- Short answer: **There are problems**
  - So we provide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

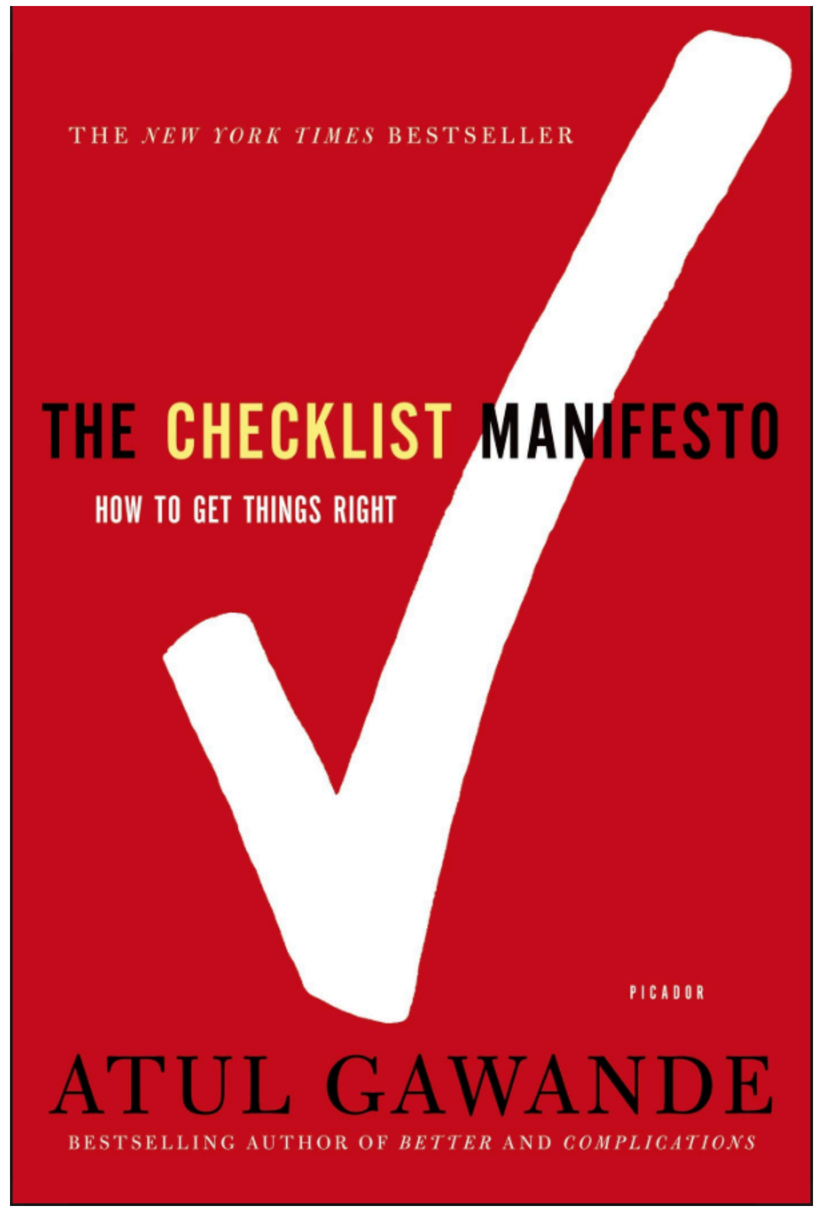
Summary of Results

- **Less than half of papers measure multiple runs**
  - Fewer still consider variance across runs
  - And yet fuzzer *performance can vary substantially from run to run*
- 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*
  - We find that 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
- Updates throughout, based on where we are in 2020!

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[8]	R(29)				G	?	N	-
FuzzSim[55]	R(101)	B	100	C	S		R/M	10D
Dowser[22]	R(7)	O	?		O		N	8H
COVERSET[45]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[9]	R(8)	A, B, Z			S		M	1H
MutaGen[29]	R(8)	R, Z			S	L	V	24H
SDF[35]	R(1)	Z, O			O		V	5D
Driller[50]	C(126)	A			G	L, E	N	24H
QuickFuzz-1[20]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[54]	R(5)	O			M	O	G, R	2H
[57]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[44]	C(63), L, R(10)	A			G, S, O		N	6H, 24H
SlowFuzz[41]	R(10)	O	100		-		N	
Steelix[33]	C(17), L, R(5)	A, V, O			C, G	L, E, M	N	5H
Skyfire[53]	R(4)	O			?	L, M	R, G	LONG
kAFL[47]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[13]	R(7)	O			G*		G	5H
Orthrus[49]	G(4), R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[27]	R(1)	O			G*		G	-
VDF[25]	R(18)				C	E	V	30D
QuickFuzz-2[21]	R(?)	O	10		G*		G, M	
IMF[23]	R(1)	O			G*	O	G	24H
[59]	S(?)	O	5		G		G	24H
NEZHA[40]	R(6)	A, L, O	100		O		R	
[56]	G(10)	A, L					V	5M
S2F[58]	L, R(8)	A, O			G	O	N	5H, 24H
FairFuzz[32]	R(9)	A	20	C		E	V/M	24H
Angora[10]	L, R(8)	A, V, O	5		G, C	L, E	N	5H
T-Fuzz[39]	C(296), L, R(4)	A, O	3		C, G*		N	24H
MEDS[24]	S(2), R(12)	O	10		C		N	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**, **objump** (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**

Since 2018

- To prepare this talk, I looked at 15 fuzzer papers published in top conferences in 2018-2020
  - USENIX Security, IEEE S&P, ISSTA, ICSE, ACSAC
  - Not comprehensive (ran out of time), but hopefully not far off
- Compared them against the same criteria. Are they doing things as we recommended?

Since 2018

Paper	Where	When	Benchmarks	Baseline	Trials	Variance	Crash	Coverage	Seed	Timeout
DIE (JS)	S&P	2020	R(3)	Superion, CA	5	C	G*, C*	E (path)	R	24 H
Ijon	S&P	2020	C*, R*	A	3		G	E (path)	M	24 H
Pangolin	S&P	2020	R(9), L	A, AF, Q, D, Angora, T-Fuzz	10	M-W	G, C		M*	24 H
Retrowrite	S&P	2020	L	AFL in various modes	5	M-W	G		V	24 H
SAVIOR	S&P	2020	L, R(8)	A, AG, TFuzz, Angora, Driller, Q	5	M-W	G, S-UBSAN(1)*	L		24x3 H
EcoFuzz	Sec	2020	R(14), G	A, AF, FairFuzz, …	5	Yes	G*, C	E (path)	?	24 H
EcoFuzz 2	Sec	2020	L	Angora, VUzzer	5	?	G			5 H
FiFUZZ	Sec	2020	R(9) R(5-binutils)	A, AF, AS, FairFuzz	3	?	G, O	E	?	24 H
GreyOne	Sec	2020	L, R(19)	A, V, Angora, CollAFL, Honggfuzz, Q	5		G, C*	E (path)	R (10)	60 H
Montage (JS)	Sec	2020	R(1)	CA, JSF, JFuzzer	5	M-W	G*, S		R, G, V	72x88 H
ParmeSan	Sec	2020	G	Angorra	30	M-W	G	E	V	48 H
Superion	ICSE	2019	R(4)	A, JSF			G*, C	L, M		“100 cycles” (3 months)
Zest	ISSTA	2019	R(5)	A, QC-junit	20	Yes	G	E	V(1)	3 H
UnTracer	S&P	2019	R(8)	AFL in various modes	8	M-W, A12	-	-	?	24 H
EnFuzz	Sec	2019	L, G, R(15)	A, AF, FairFuzz, Q, libFuzzer, R	10	“within 5%”	G, S-ASAN(1)	E (path)	V	24*4 H
TIFF	ACSAC	2018	L, R(9)	AF, VUzzer	3	“marginal statistical variations”	G(*), S	L	R(4)	12 H

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**
  - median(A) = 6, median(B) = 4
  - mean(A) = 4, 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 **median(A) - median(B) ≥ 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: **p-value ≤ 0.05** is a sufficient level of confidence

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

ICSE 2011

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 & Briand advice: Use the **Mann Whitney U Test**
  - No assumption of distribution normality

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[8]	R(29)				G	?	N	-
FuzzSim[55]	R(101)	B	100	C	S		R/M	10D
Dowser[22]	R(7)	O	?		O		N	8H
COVERSET[45]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[9]	R(8)	A, B, Z			S		M	1H
MutaGen[29]	R(8)	R, Z			S	L	V	24H
SDF[35]	R(1)	Z, O			O		V	5D
Driller[50]	C(126)	A			G	L, E	N	24H
QuickFuzz-1[20]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[54]	R(5)	O			M	O	G, R	2H
[57]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[44]	C(63), L, R(10)	A			G, S, O		N	6H, 24H
SlowFuzz[41]	R(10)	O	100		-		N	
Steelix[33]	C(17), L, R(5)	A, V, O			C, G	L, E, M	N	5H
Skyfire[53]	R(4)	O			?	L, M	R, G	LONG
kAFL[47]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[13]	R(7)	O			G*		G	5H
Orthrus[49]	G(4), R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[27]	R(1)	O			G*		G	-
VDF[25]	R(18)				C	E	V	30D
QuickFuzz-2[21]	R(?)	O	10		G*		G, M	
IMF[23]	R(1)	O			G*	O	G	24H
[59]	S(?)	O	5		G		G	24H
NEZHA[40]	R(6)	A, L, O	100		O		R	
[56]	G(10)	A, L					V	5M
S2F[58]	L, R(8)	A, O			G	O	N	5H, 24H
FairFuzz[32]	R(9)	A	20	C		E	V/M	24H
Angora[10]	L, R(8)	A, V, O	5		G, C	L, E	N	5H
T-Fuzz[39]	C(296), L, R(4)	A, O	3		C, G*		N	24H
MEDS[24]	S(2), R(12)	O	10		C		N	6H

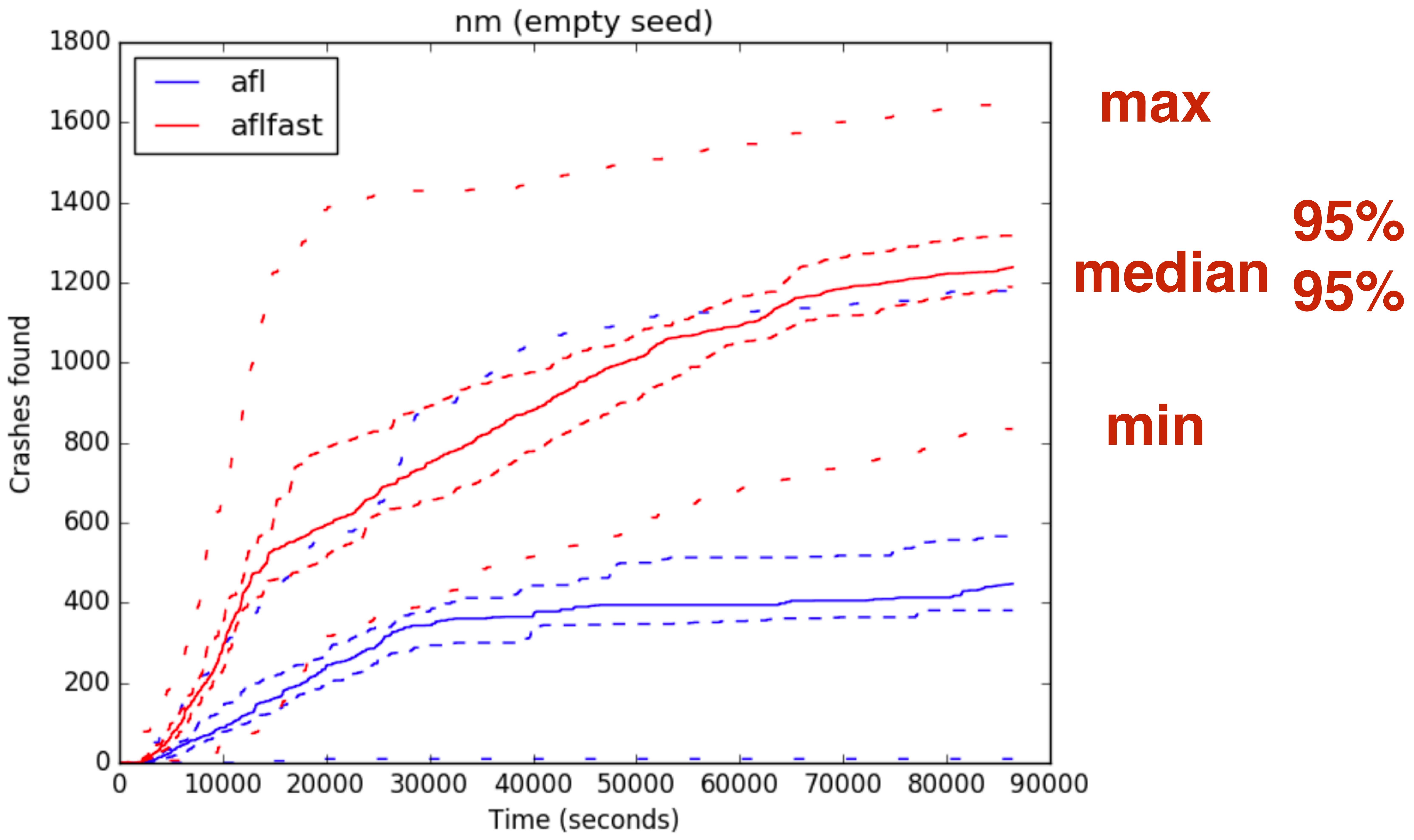
Evaluations

- 17/32 papers said nothing about multiple trials
  - Assume 1
- 15/32 papers said multiple trials
  - Varying number; one case not specified
- 3/13 papers characterized variance across runs
- 0 papers performed a statistical test

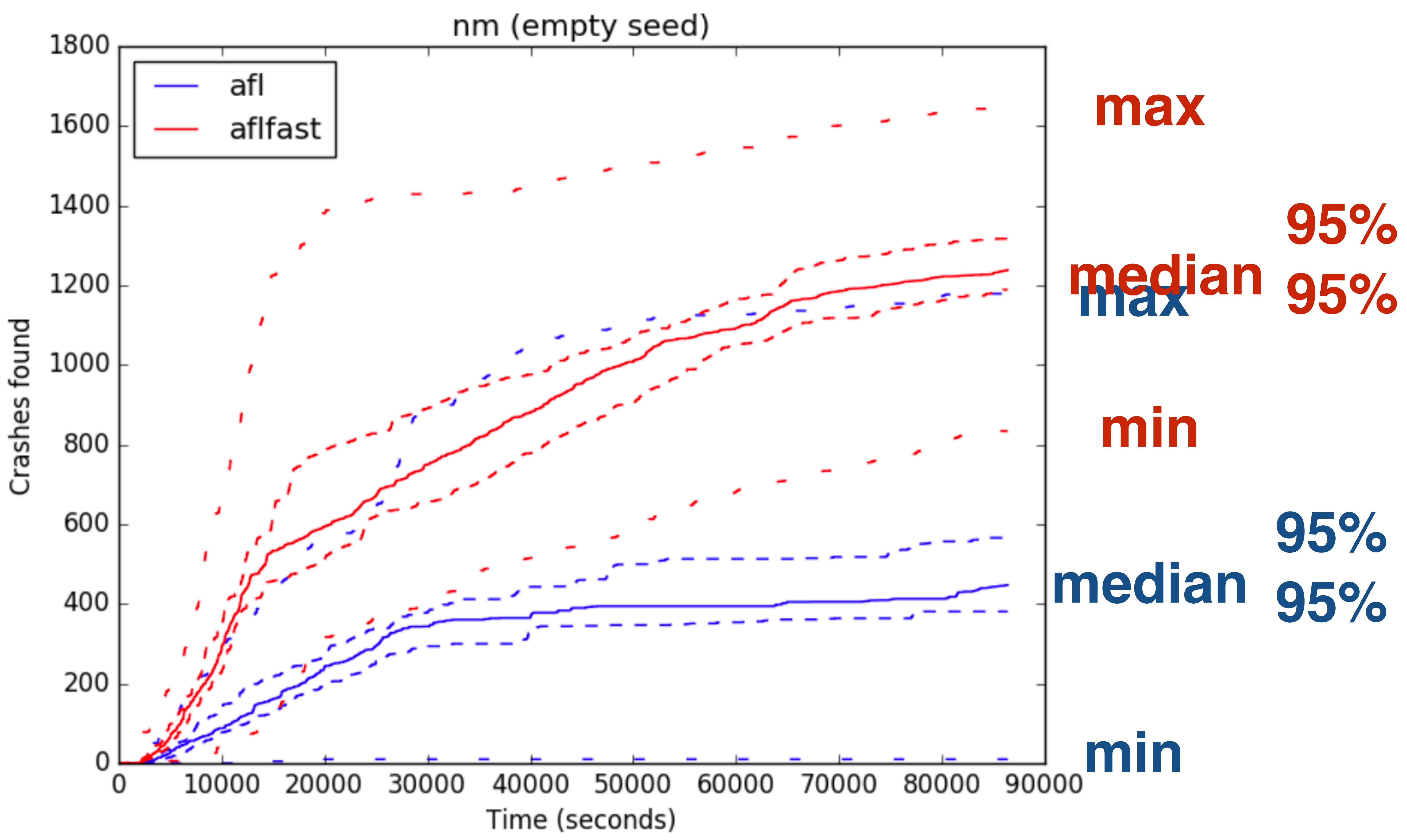
Practical Impact?

- Fuzzers run for a long time, conducting potentially millions of individual tests over many hours
  - If we consider our biased die: Perhaps no statistical comparison is needed (just the mean/median) **if we have *a lot* of tests?**
- 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**

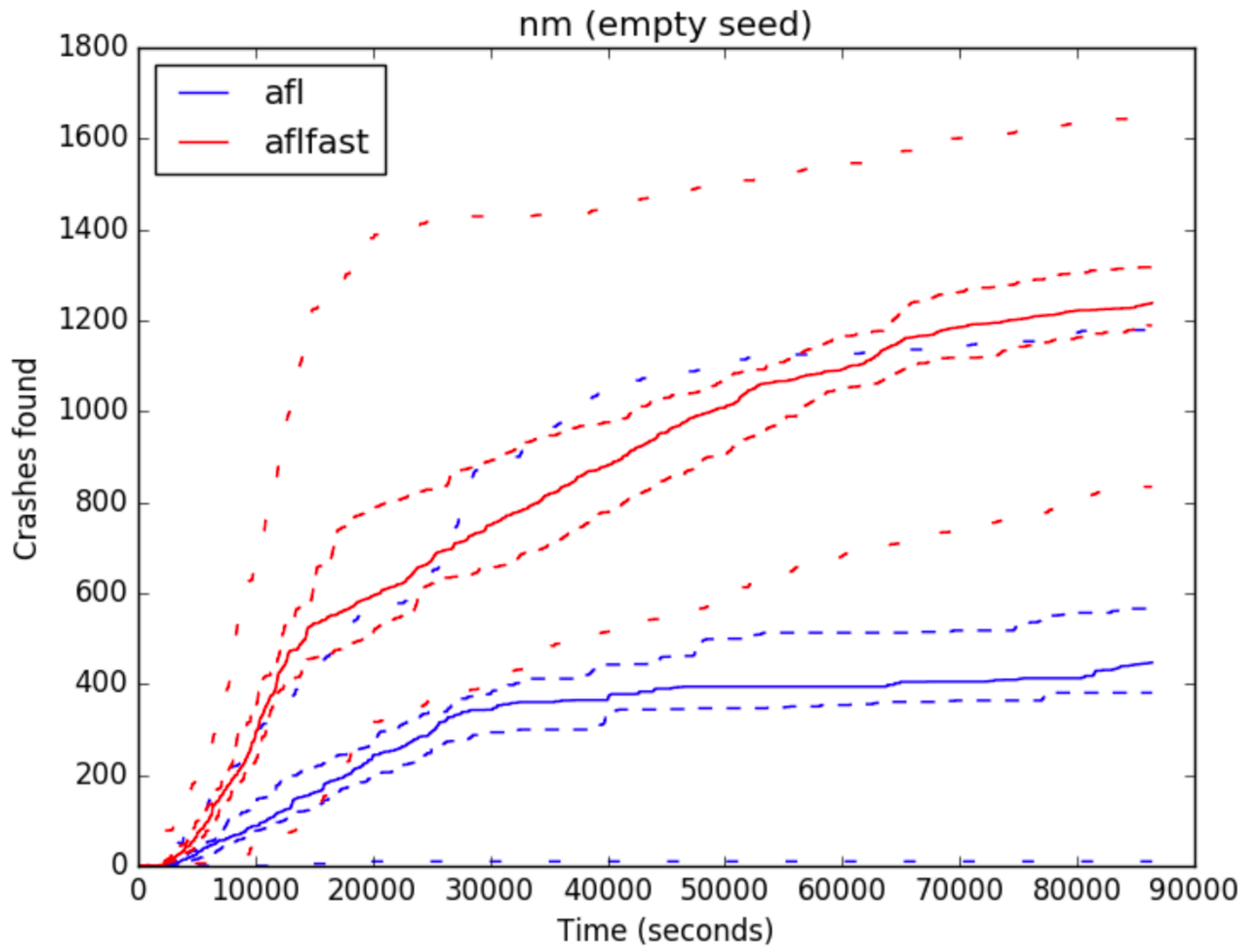
Performance Plot



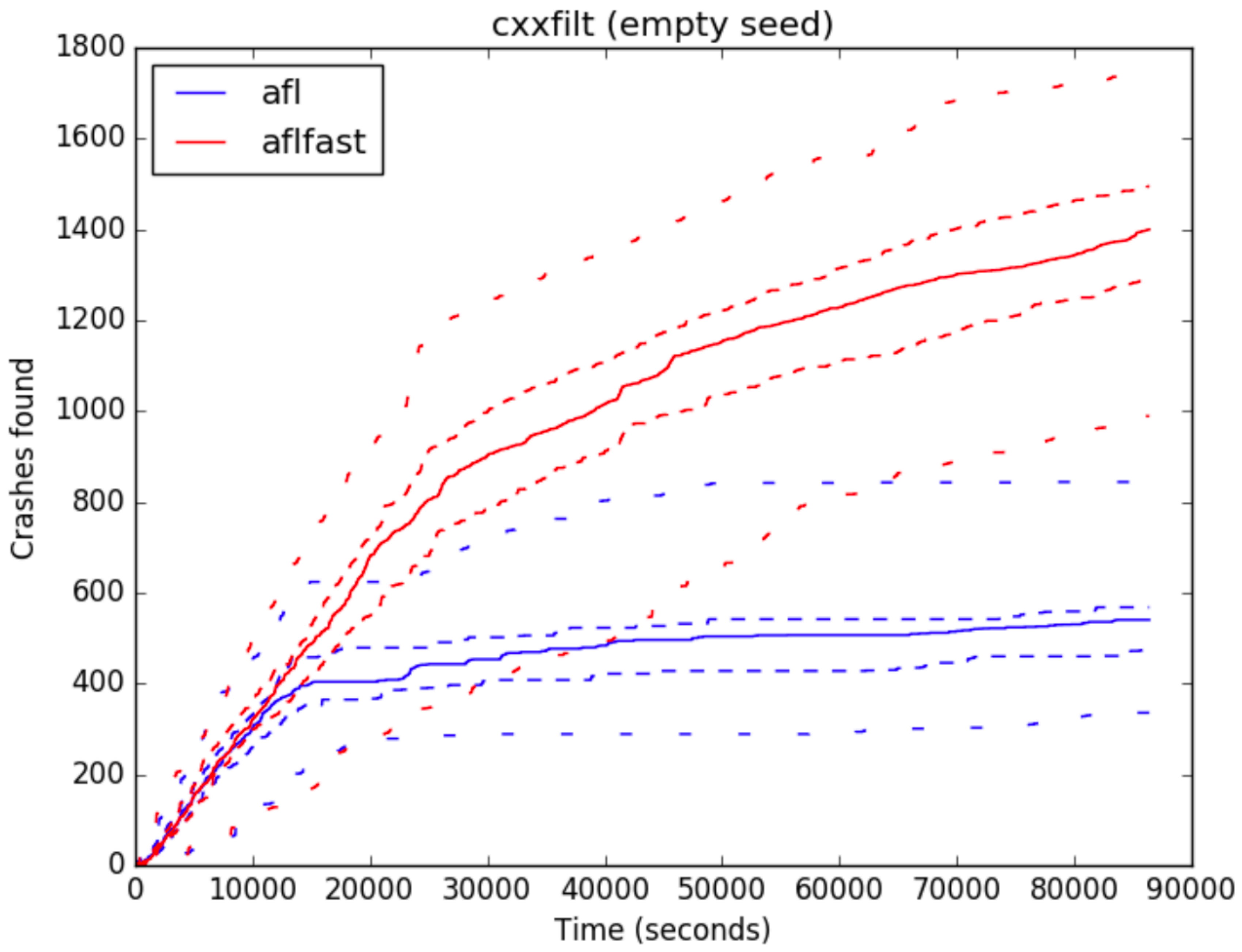
Performance Plot



Statistically Significant



$p < 10^{-13}$

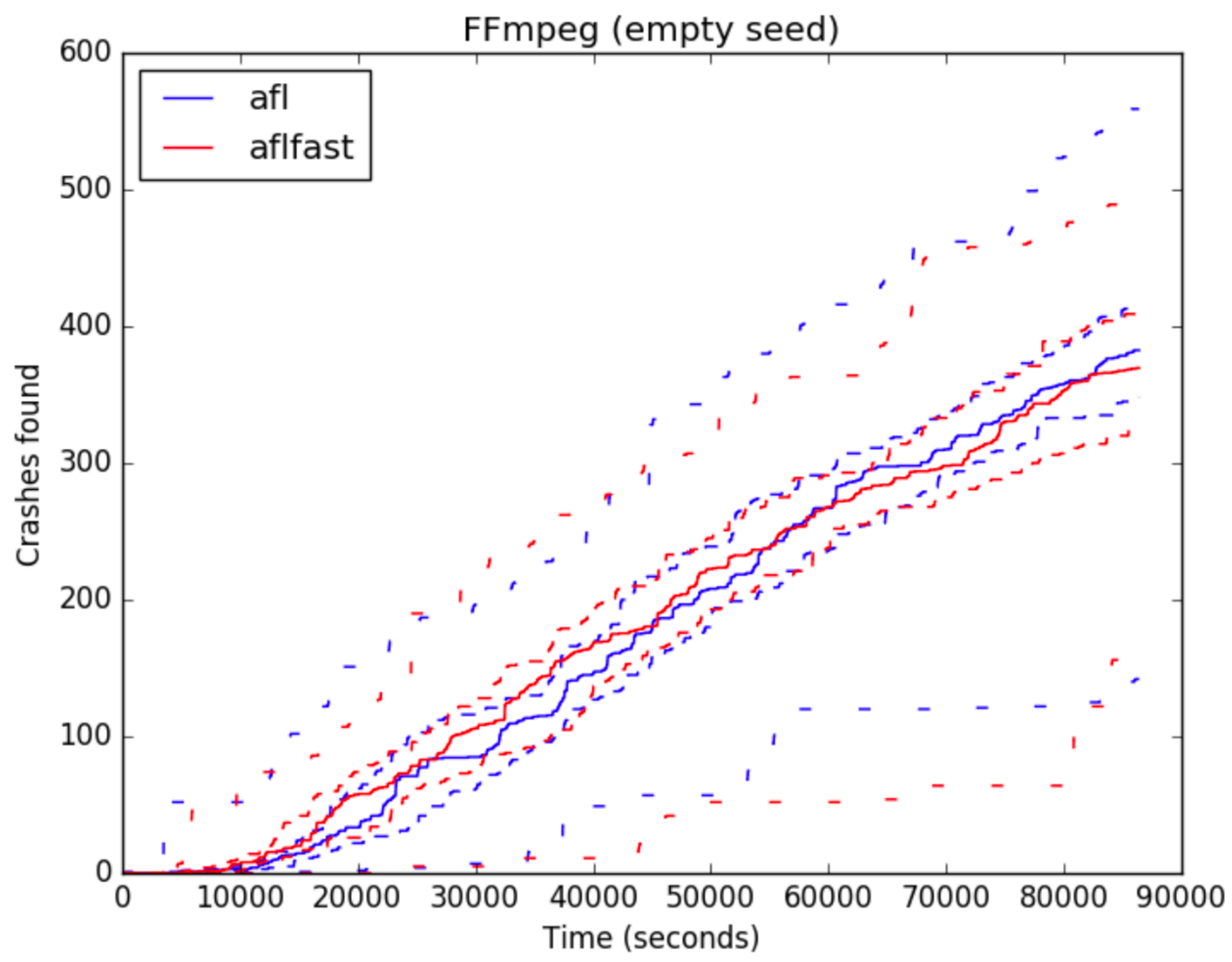


$p < 10^{-10}$

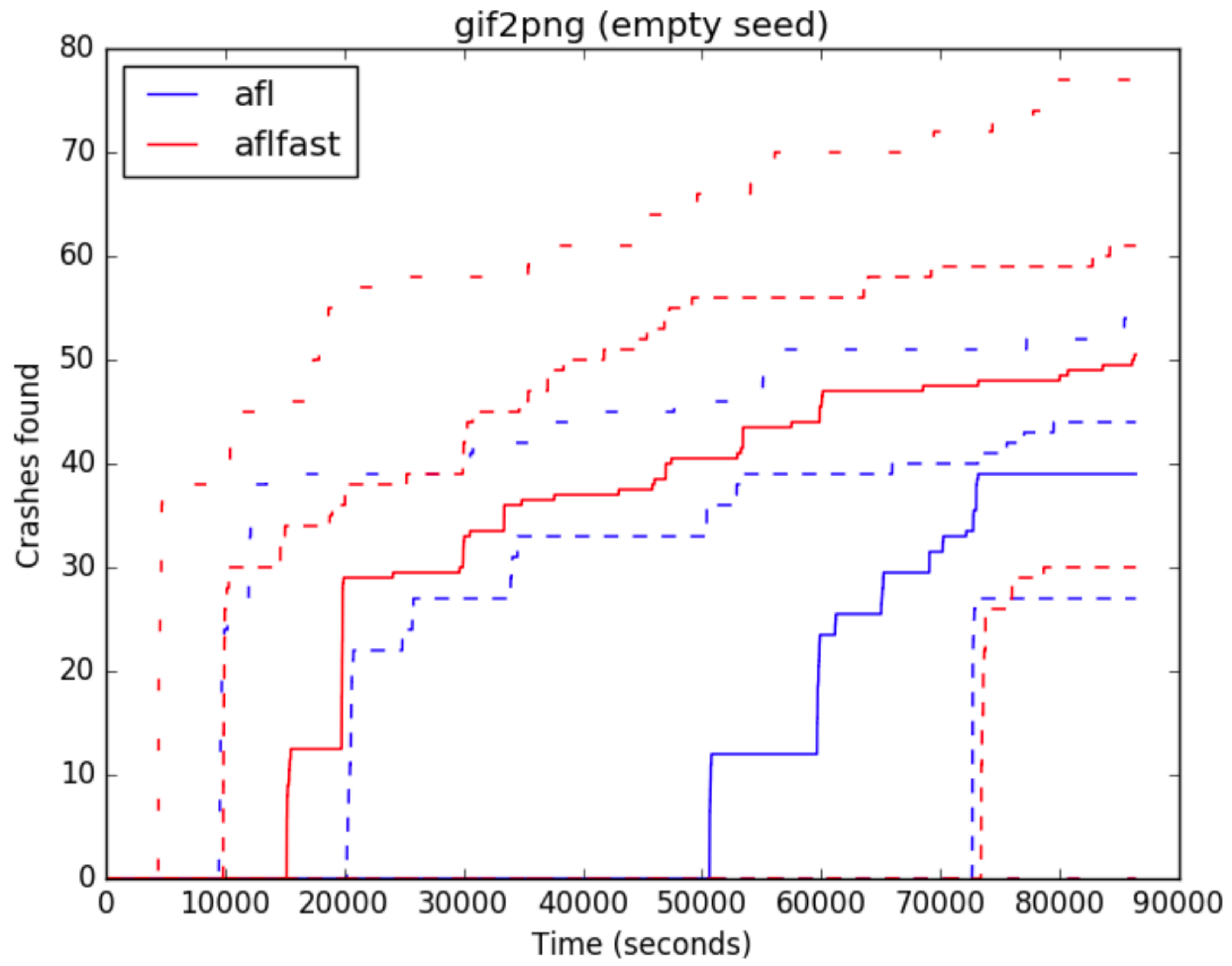
significant variance
in performance

Higher median
clearly better

Statistically Insignificant



p = 0.379



p = 0.0676

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

Higher median
does *not* meet bar
for significance

I Want **You**

to run multiple trials

and

use a statistical test to compare distributions!

How are things in late 2020? Better!

Paper	Where	When	Benchmarks	Baseline	Trials	Variance	Crash	Coverage	Seed	Timeout
DIE (JS)	S&P	2020	R(3)	Superion, CA	5	C	G*, C*	E (path)	R	24 H
Ijon	S&P	2020	C*, R*	A	3		G	E (path)	M	24 H
Pangolin	S&P	2020	R(9), L	A, AF, Q, D, Angora, T-Fuzz	10	M-W	G, C		M*	24 H
Retrowrite	S&P	2020	L	AFL in various modes	5	M-W	G		V	24 H
SAVIOR	S&P	2020	L, R(8)	A, AG, TFuzz, Angora, Driller, Q	5	M-W	G, S-UBSAN(1)*	L		24x3 H
EcoFuzz	Sec	2020	R(14), G	A, AF, FairFuzz, …	5	Yes	G*, C	E (path)	?	24 H
EcoFuzz 2	Sec	2020	L	Angora, VUzzer	5	?	G			5 H
FiFUZZ	Sec	2020	R(9) R(5-binutils)	A, AF, AS, FairFuzz	3	?	G, O	E	?	24 H
GreyOne	Sec	2020	L, R(19)	A, V, Angora, CollAFL, Honggfuzz, Q	5		G, C*	E (path)	R (10)	60 H
Montage (JS)	Sec	2020	R(1)	CA, JSF, JFuzzer	5	M-W	G*, S		R, G, V	72x88 H
ParmeSan	Sec	2020	G	Angorra	30	M-W	G	E	V	48 H
Superion	ICSE	2019	R(4)	A, JSF			G*, C	L, M		"100 cycles" (3 months)
Zest	ISSTA	2019	R(5)	A, QC-junit	20	Yes	G	E	V(1)	3 H
UnTracer	S&P	2019	R(8)	AFL in various modes	8	M-W, A12	-	-	?	24 H
EnFuzz	Sec	2019	L, G, R(15)	A, AF, FairFuzz, Q, libFuzzer, R	10	"within 5%"	G, S-ASAN(1)	E (path)	V	24*4 H
TIFF	ACSAC	2018	L, R(9)	AF, VUzzer	3	"marginal statistical variations"	G(*), S	L	R(4)	12 H

- 14/15 had multiple trials
  - Varying number; 5 typical
- 7/14 papers performed a statistical test
  - Most use M-W U
    - One also used A12 effect size
  - 2 didn't say which test
- 3/7 said something about variance

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[8]	R(29)				G	?	N	-
FuzzSim[55]	R(101)	B	100	C	S		R/M	10D
Dowser[22]	R(7)	O	?		O		N	8H
COVERSET[45]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[9]	R(8)	A, B, Z			S		M	1H
MutaGen[29]	R(8)	R, Z			S	L	V	24H
SDF[35]	R(1)	Z, O			O		V	5D
Driller[50]	C(126)	A			G	L, E	N	24H
QuickFuzz-1[20]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[54]	R(5)	O			M	O	G, R	2H
[57]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[44]	C(63), L, R(10)	A			G, S, O		N	6H, 24H
SlowFuzz[41]	R(10)	O	100		-		N	
Steelix[33]	C(17), L, R(5)	A, V, O			C, G	L, E, M	N	5H
Skyfire[53]	R(4)	O			?	L, M	R, G	LONG
kAFL[47]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[13]	R(7)	O			G*		G	5H
Orthrus[49]	G(4), R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[27]	R(1)	O			G*		G	-
VDF[25]	R(18)				C	E	V	30D
QuickFuzz-2[21]	R(?)	O	10		G*		G, M	
IMF[23]	R(1)	O			G*	O	G	24H
[59]	S(?)	O	5		G		G	24H
NEZHA[40]	R(6)	A, L, O	100		O		R	
[56]	G(10)	A, L					V	5M
S2F[58]	L, R(8)	A, O			G	O	N	5H, 24H
FairFuzz[32]	R(9)	A	20	C		E	V/M	24H
Angora[10]	L, R(8)	A, V, O	5		G, C	L, E	N	5H
T-Fuzz[39]	C(296), L, R(4)	A, O	3		C, G*		N	24H
MEDS[24]	S(2), R(12)	O	10		C		N	6H

Evaluations

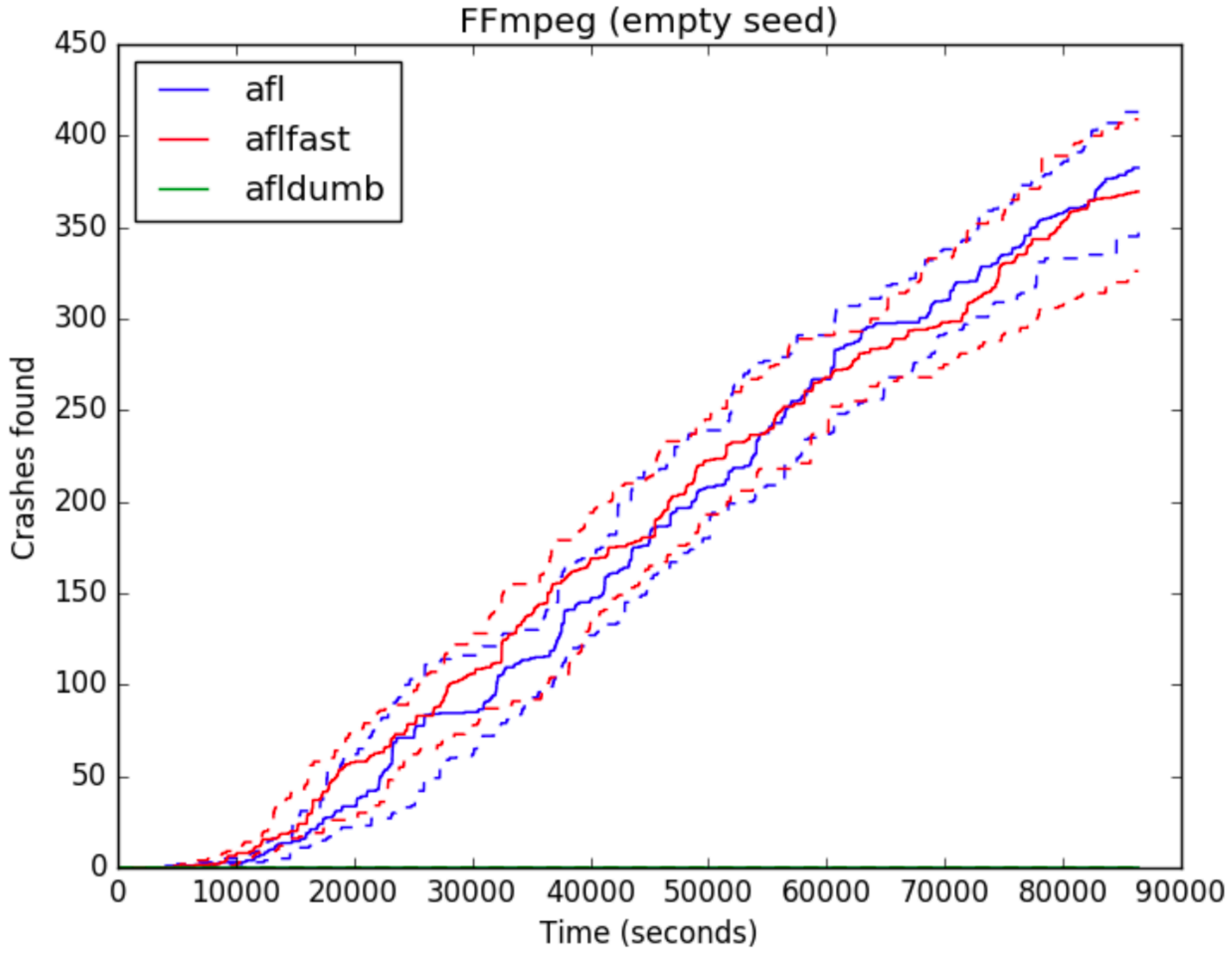
- 30/32 papers used non-empty seed
  - 10 say nothing else (N)
  - 9 used valid seed but no details (V)
- 2/32 papers used the empty (E) file (eg. AFLFast)
  - Good "default" choice in vast configuration space
  - But contrary to practice
- Question: Practical impact?

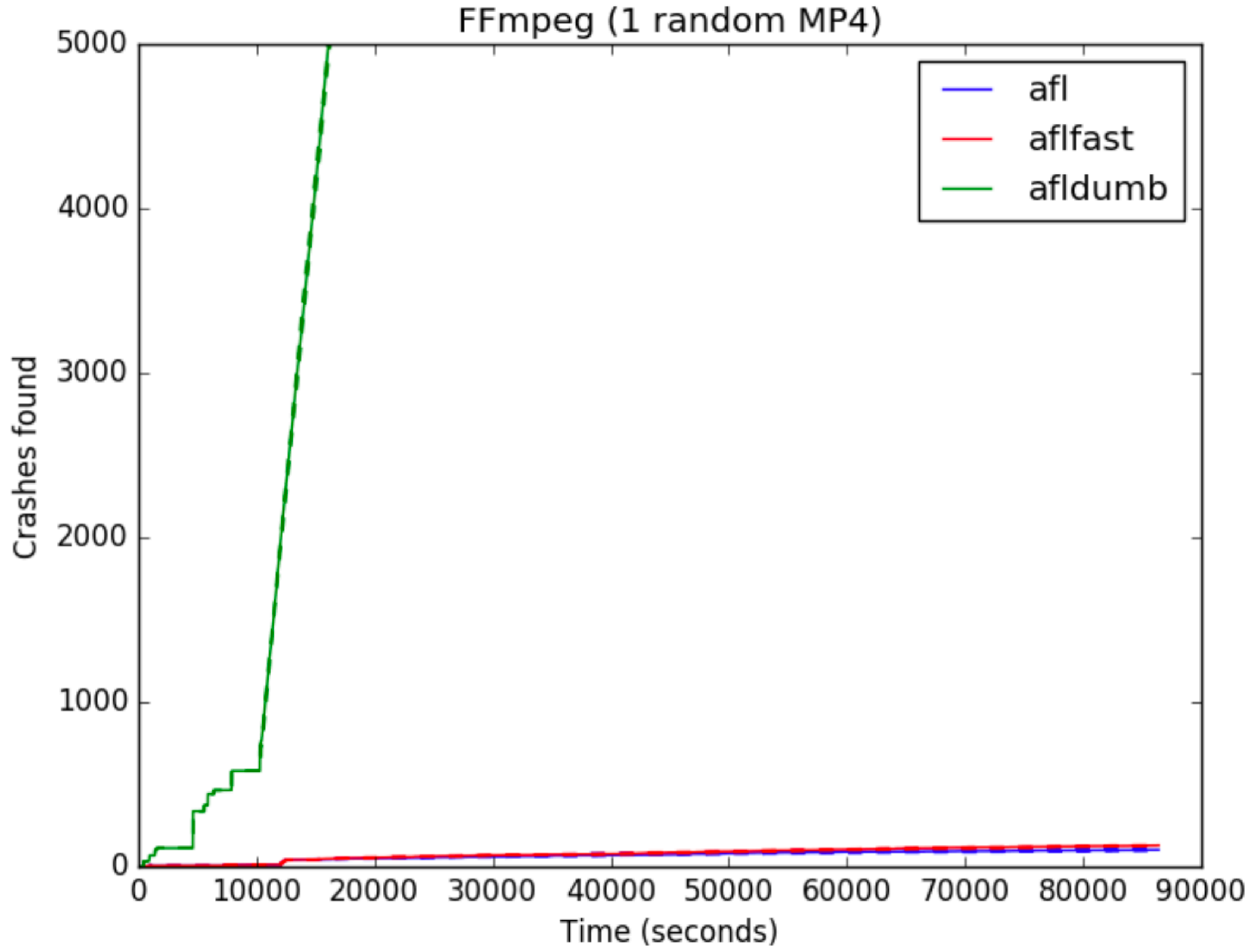
Experiments

- **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 (surprisingly) useful





<u>empty seed</u>

(AFLFast vs. AFL) $p = 0.379$

(AFLDumb vs. AFL) $p < 10^{-15}$

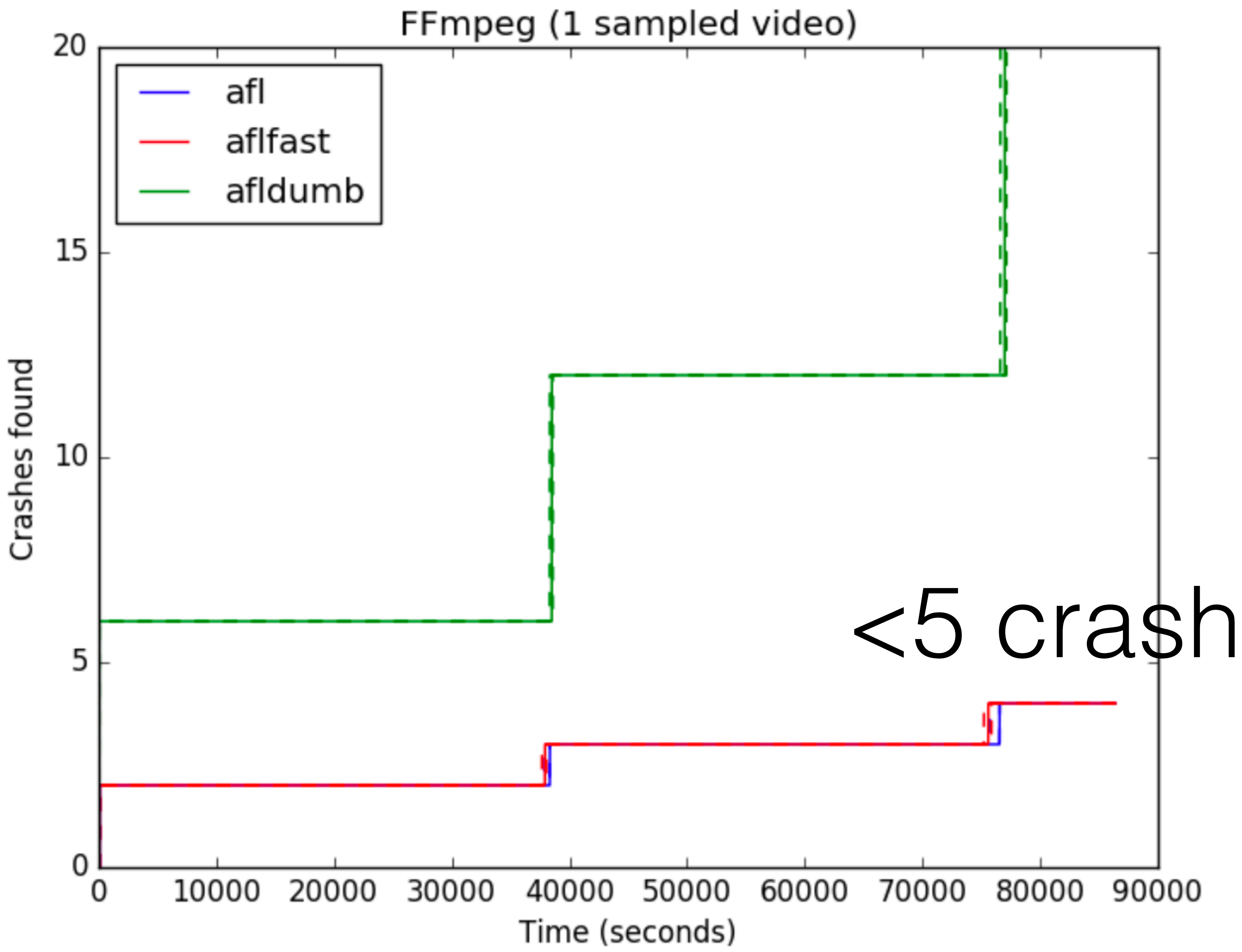
<u>1-made</u>

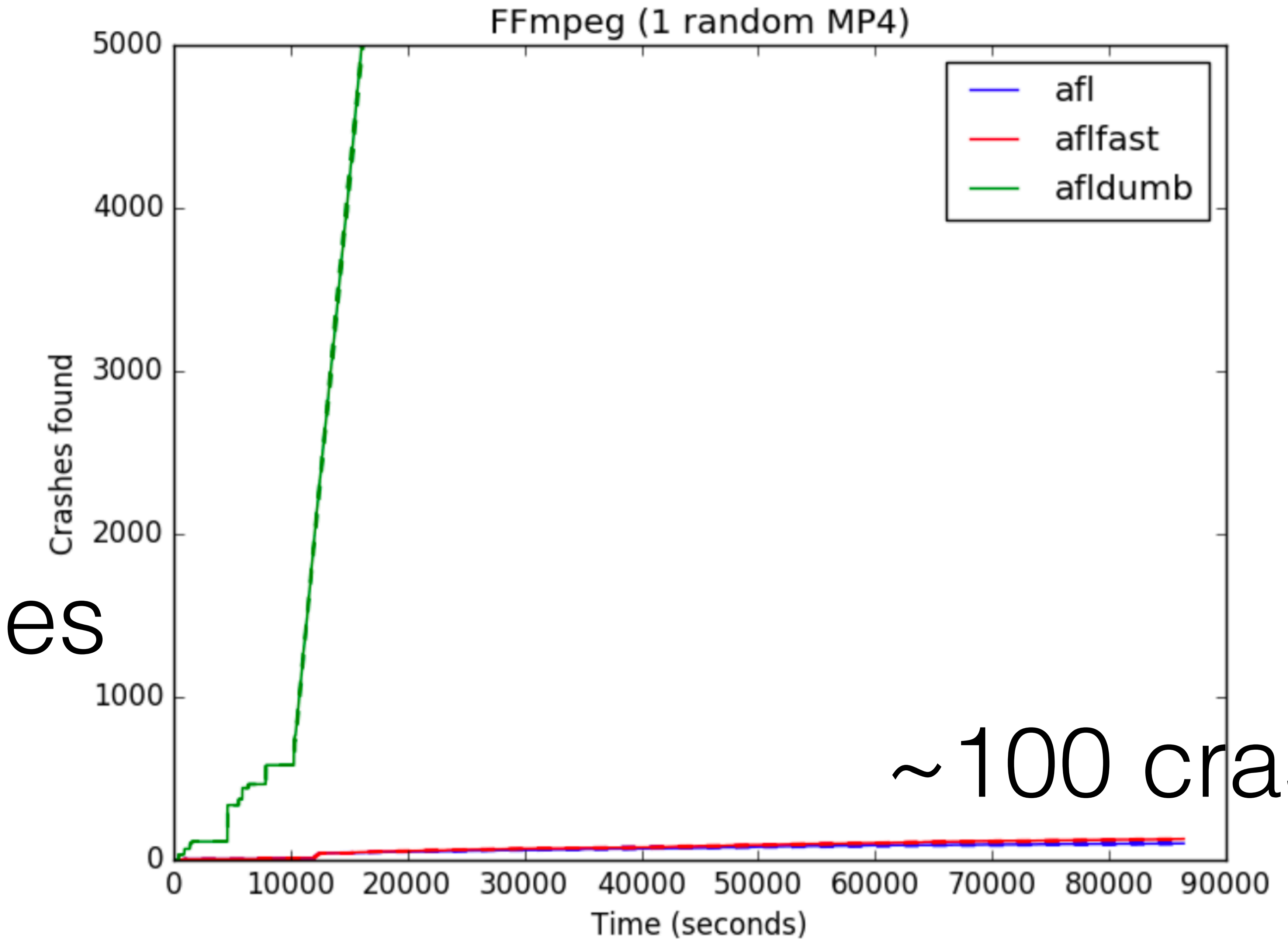
$p = 0.048$

$p < 10^{-11}$

FFMpeg: Sampled vs. Handmade

Both "valid", but very different performance





<u>1-sampled</u>

(AFLFast vs. AFL) $p > 0.05$

(AFLDumb vs. AFL) $p < 10^{-5}$

<u>1-made</u>

$p = 0.048$

$p < 10^{-11}$

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**
- Evaluations should **consider several seeds**, including **empty seed**
  - Multiple trials to sample large, random space; likewise, want to sample **large, disparate space of seeds**
  - Need **more research** to understand this better

How are things in late 2020? Same

Paper	Where	When	Benchmarks	Baseline	Trials	Variance	Crash	Coverage	Seed	Timeout
DIE (JS)	S&P	2020	R(3)	Superion, CA	5	C	G*, C*	E (path)	R	24 H
Ijon	S&P	2020	C*, R*	A	3		G	E (path)	M	24 H
Pangolin	S&P	2020	R(9), L	A, AF, Q, D, Angora, T-Fuzz	10	M-W	G, C		M*	24 H
Retrowrite	S&P	2020	L	AFL in various modes	5	M-W	G		V	24 H
SAVIOR	S&P	2020	L, R(8)	A, AG, TFuzz, Angora, Driller, Q	5	M-W	G, S-UBSAN(1)*	L		24x3 H
EcoFuzz	Sec	2020	R(14), G	A, AF, FairFuzz, …	5	Yes	G*, C	E (path)	?	24 H
EcoFuzz 2	Sec	2020	L	Angora, VUzzer	5	?	G			5 H
FiFUZZ	Sec	2020	R(9) R(5-binutils)	A, AF, AS, FairFuzz	3	?	G, O	E	?	24 H
GreyOne	Sec	2020	L, R(19)	A, V, Angora, CollAFL, Honggfuzz, Q	5		G, C*	E (path)	R (10)	60 H
Montage (JS)	Sec	2020	R(1)	CA, JSF, JFuzzer	5	M-W	G*, S		R, G, V	72x88 H
ParmeSan	Sec	2020	G	Angorra	30	M-W	G	E	V	48 H
Superion	ICSE	2019	R(4)	A, JSF			G*, C	L, M		"100 cycles" (3 months)
Zest	ISSTA	2019	R(5)	A, QC-junit	20	Yes	G	E	V(1)	3 H
UnTracer	S&P	2019	R(8)	AFL in various modes	8	M-W, A12	-	-	?	24 H
EnFuzz	Sec	2019	L, G, R(15)	A, AF, FairFuzz, Q, libFuzzer, R	10	"within 5%"	G, S-ASAN(1)	E (path)	V	24*4 H
TIFF	ACSAC	2018	L, R(9)	AF, VUzzer	3	"marginal statistical variations"	G(*), S	L	R(4)	12 H

- Very little said about the particulars of seed selection
  - Usually valid seeds were used,
    - sometimes mentioned how many,
    - sometimes mentioned how produced
  - No specific mention of the use of an empty seed

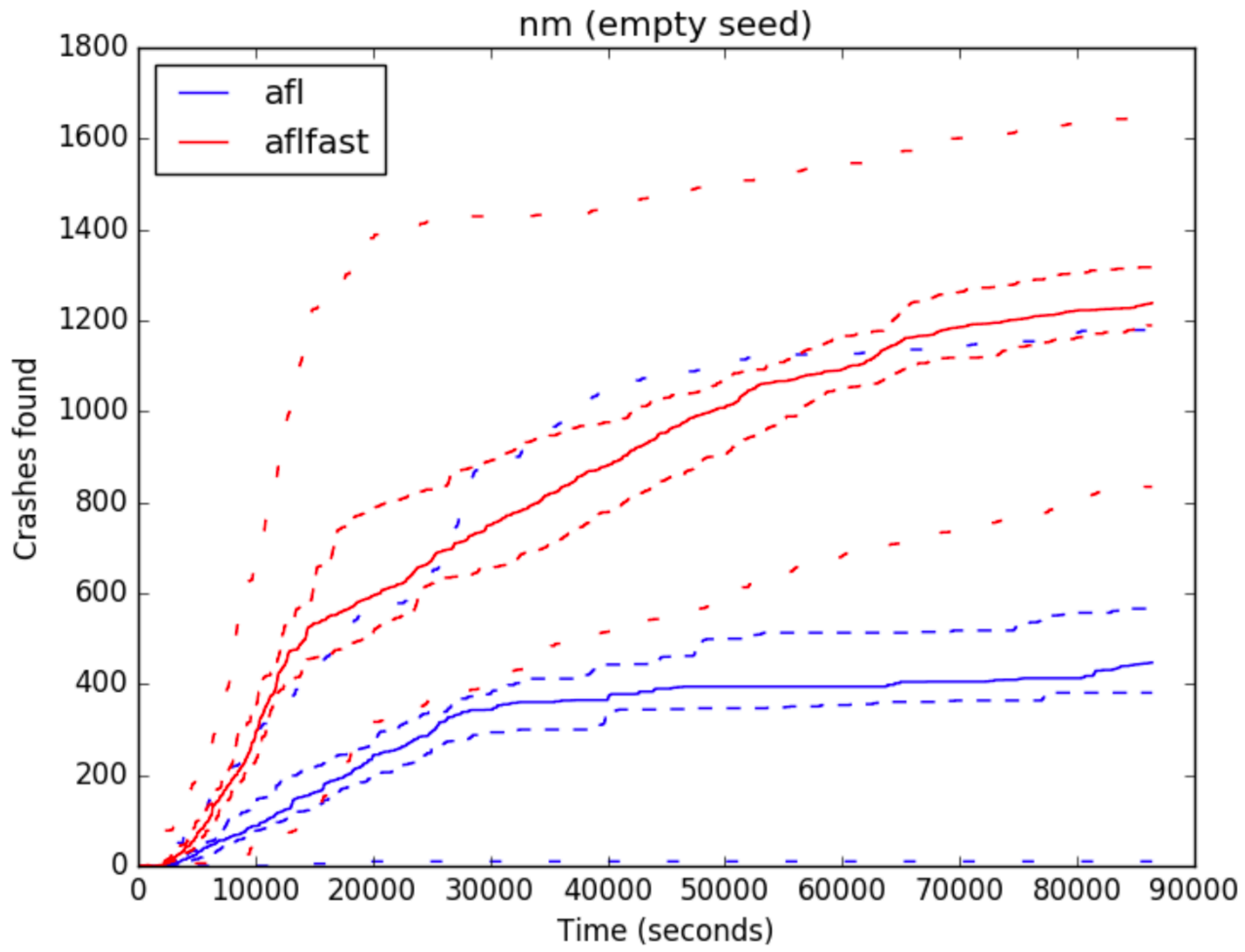
Timeouts

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[8]	R(29)				G	?	N	-
FuzzSim[55]	R(101)	B	100	C	S		R/M	10D
Dowser[22]	R(7)	O	?		O		N	8H
COVERSET[45]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[9]	R(8)	A, B, Z			S		M	1H
MutaGen[29]	R(8)	R, Z			S	L	V	24H
SDF[35]	R(1)	Z, O			O		V	5D
Driller[50]	C(126)	A			G	L, E	N	24H
QuickFuzz-1[20]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[54]	R(5)	O			M	O	G, R	2H
[57]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[44]	C(63), L, R(10)	A			G, S, O		N	6H, 24H
SlowFuzz[41]	R(10)	O	100		-		N	
Steelix[33]	C(17), L, R(5)	A, V, O			C, G	L, E, M	N	5H
Skyfire[53]	R(4)	O			?	L, M	R, G	LONG
kAFL[47]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[13]	R(7)	O			G*		G	5H
Orthrus[49]	G(4), R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[27]	R(1)	O			G*		G	-
VDF[25]	R(18)				C	E	V	30D
QuickFuzz-2[21]	R(?)	O	10		G*		G, M	
IMF[23]	R(1)	O			G*	O	G	24H
[59]	S(?)	O	5		G		G	24H
NEZHA[40]	R(6)	A, L, O	100		O		R	
[56]	G(10)	A, L					V	5M
S2F[58]	L, R(8)	A, O			G	O	N	5H, 24H
FairFuzz[32]	R(9)	A	20	C		E	V/M	24H
Angora[10]	L, R(8)	A, V, O	5		G, C	L, E	N	5H
T-Fuzz[39]	C(296), L, R(4)	A, O	3		C, G*		N	24H
MEDS[24]	S(2), R(12)	O	10		C		N	6H

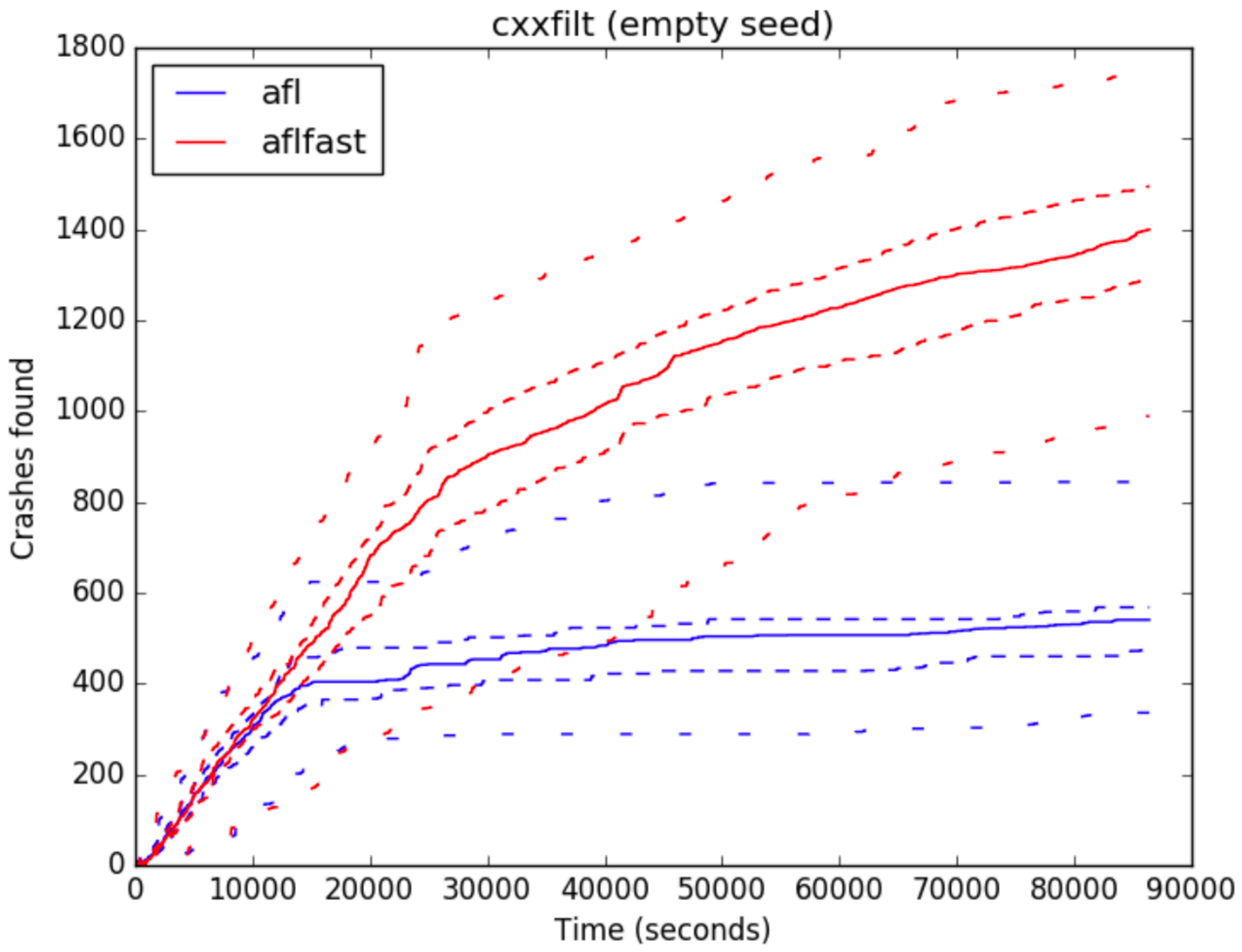
Evaluations

- 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



$p < 10^{-13}$

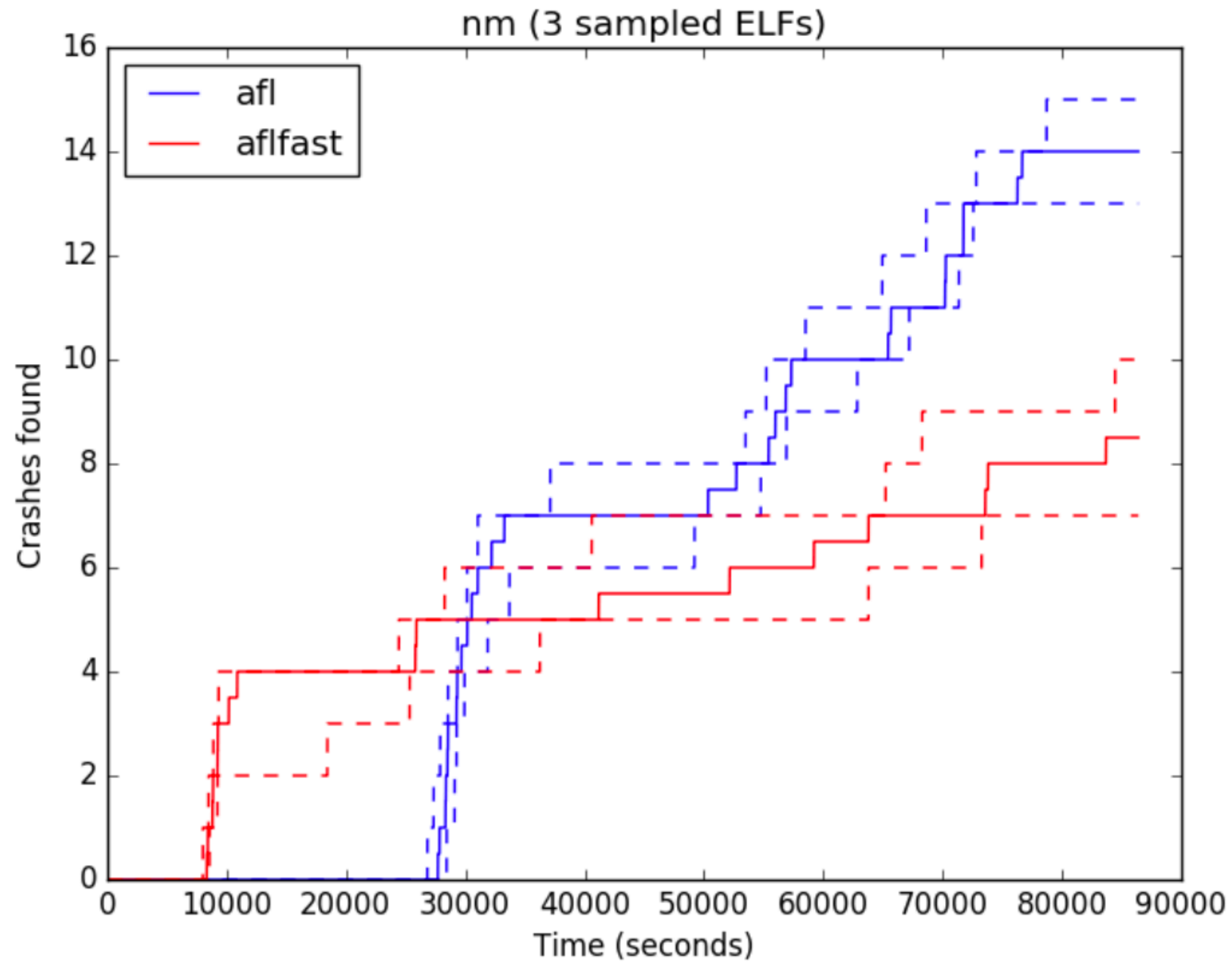


$p < 10^{-10}$

AFLFast better at

5, **8**, **24 hours**

Trends can Change



Can take time for fuzzing to “warm up”

<u>3-sampled</u>

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** … maybe?
  - Ecologically relevant
    - But longer would be even better!
  - Subsumes common 5 and 8 hour limits
  - Not great principles for choosing it

How are things in late 2020? Good

Paper	Where	When	Benchmarks	Baseline	Trials	Variance	Crash	Coverage	Seed	Timeout
DIE (JS)	S&P	2020	R(3)	Superion, CA	5	C	G*, C*	E (path)	R	24 H
Ijon	S&P	2020	C*, R*	A	3		G	E (path)	M	24 H
Pangolin	S&P	2020	R(9), L	A, AF, Q, D, Angora, T-Fuzz	10	M-W	G, C		M*	24 H
Retrowrite	S&P	2020	L	AFL in various modes	5	M-W	G		V	24 H
SAVIOR	S&P	2020	L, R(8)	A, AG, TFuzz, Angora, Driller, Q	5	M-W	G, S-UBSAN(1)*	L		24x3 H
EcoFuzz	Sec	2020	R(14), G	A, AF, FairFuzz, …	5	Yes	G*, C	E (path)	?	24 H
EcoFuzz 2	Sec	2020	L	Angora, VUzzer	5	?	G			5 H
FiFUZZ	Sec	2020	R(9) R(5-binutils)	A, AF, AS, FairFuzz	3	?	G, O	E	?	24 H
GreyOne	Sec	2020	L, R(19)	A, V, Angora, CollAFL, Honggfuzz, Q	5		G, C*	E (path)	R (10)	60 H
Montage (JS)	Sec	2020	R(1)	CA, JSF, JFuzzer	5	M-W	G*, S		R, G, V	72x88 H
ParmeSan	Sec	2020	G	Angorra	30	M-W	G	E	V	48 H
Superion	ICSE	2019	R(4)	A, JSF			G*, C	L, M		"100 cycles" (3 months)
Zest	ISSTA	2019	R(5)	A, QC-junit	20	Yes	G	E	V(1)	3 H
UnTracer	S&P	2019	R(8)	AFL in various modes	8	M-W, A12	-	-	?	24 H
EnFuzz	Sec	2019	L, G, R(15)	A, AF, FairFuzz, Q, libFuzzer, R	10	"within 5%"	G, S-ASAN(1)	E (path)	V	24*4 H
TIFF	ACSAC	2018	L, R(9)	AF, VUzzer	3	"marginal statistical variations"	G(*), S	L	R(4)	12 H

- 13/15 papers used 24 hours or more
- 2 papers fuzzed a *long* time
  - 72 hours on 88 processors in parallel
  - 3 months

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[8]	R(29)				G	?	N	-
FuzzSim[55]	R(101)	B	100	C	S		R/M	10D
Dowser[22]	R(7)	O	?		O		N	8H
COVERSET[45]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[9]	R(8)	A, B, Z			S		M	1H
MutaGen[29]	R(8)	R, Z			S	L	V	24H
SDF[35]	R(1)	Z, O			O		V	5D
Driller[50]	C(126)	A			G	L, E	N	24H
QuickFuzz-1[20]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[54]	R(5)	O			M	O	G, R	2H
[57]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[44]	C(63), L, R(10)	A			G, S, O		N	6H, 24H
SlowFuzz[41]	R(10)	O	100		-		N	
Steelix[33]	C(17), L, R(5)	A, V, O			C, G	L, E, M	N	5H
Skyfire[53]	R(4)	O			?	L, M	R, G	LONG
kAFL[47]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[13]	R(7)	O			G*		G	5H
Orthrus[49]	G(4), R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[27]	R(1)	O			G*		G	-
VDF[25]	R(18)				C	E	V	30D
QuickFuzz-2[21]	R(?)	O	10		G*		G, M	
IMF[23]	R(1)	O			G*	O	G	24H
[59]	S(?)	O	5		G		G	24H
NEZHA[40]	R(6)	A, L, O	100		O		R	
[56]	G(10)	A, L					V	5M
S2F[58]	L, R(8)	A, O			G	O	N	5H, 24H
FairFuzz[32]	R(9)	A	20	C		E	V/M	24H
Angora[10]	L, R(8)	A, V, O	5		G, C	L, E	N	5H
T-Fuzz[39]	C(296), L, R(4)	A, O	3		C, G*		N	24H
MEDS[24]	S(2), R(12)	O	10		C		N	6H

Evaluations

- 7 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?

(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 useful 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

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 mentioning the issues with stack hashes
- **CMIN may also suffer from inflated counts** (false positives)
  - Many superfluously different paths to the same fault-point are treated as distinct

Assessing Heuristics: cxxfilt

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

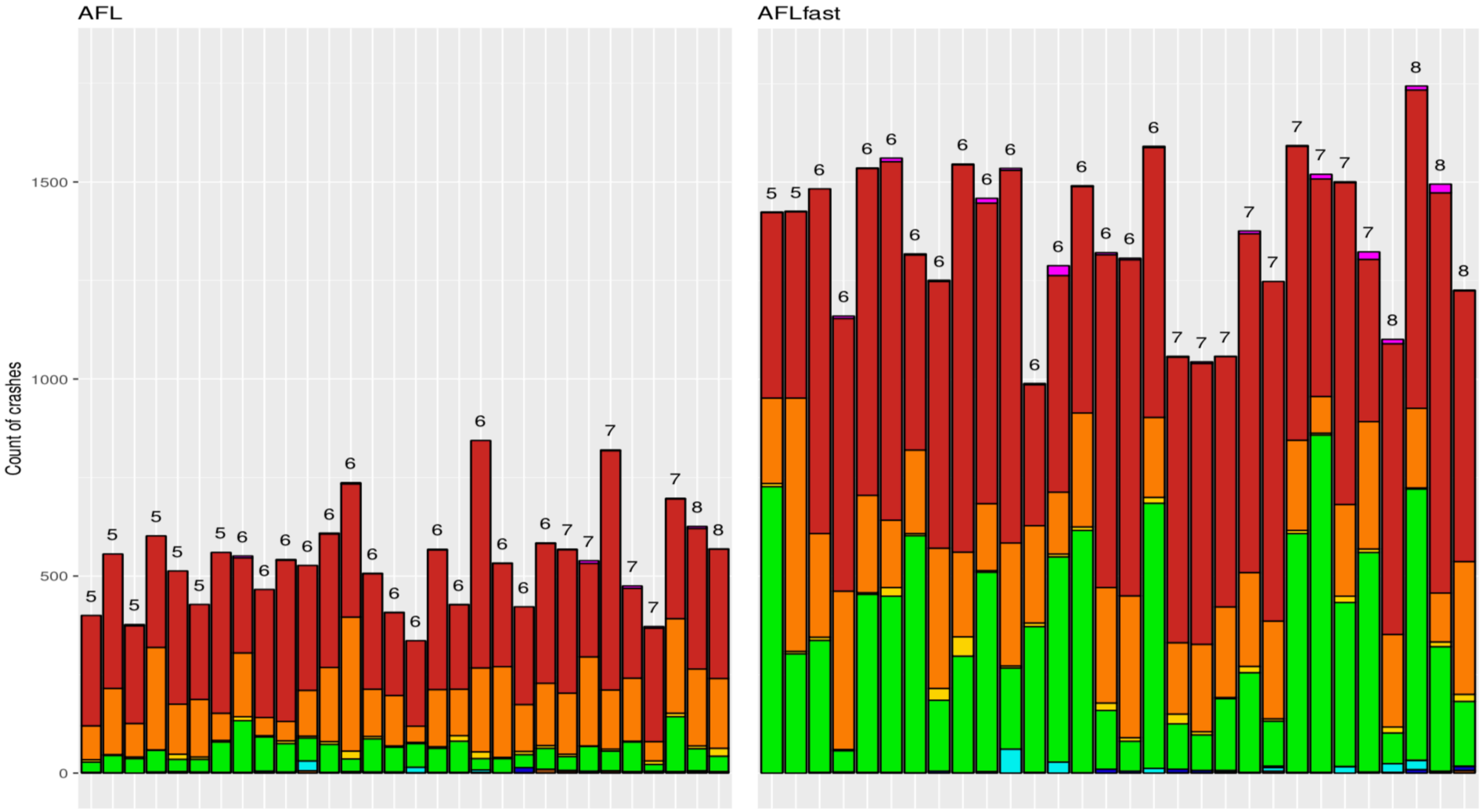
- Used commit history to find **patches since fuzzed version**
  - E.g., commit on left fixes integer overflow
- Applied patches iteratively, and re-ran against all **57,000+ crashing inputs** (post-CMIN, all 30 runs)
  - Those that no longer crash are due to this patch
  - Broke apart patches that fix multiple bugs
- Re-run must account for non-determinism
  - Used ASAN/UBSAN: “non crash” only if it found no issue

Stack Hashes (N=3)

Bug	# Hashes	Matches	False Matches	Input count
A	9	2	7	228
B	362	343	19	31,103
C	24	21	3	106
D	159	119	40	12,672
E	15	4	11	12,118
F	15	1	14	232
G	2	0	2	2
H	1	1	0	568
I	4	4	0	10
unfixed	28	12	16	98
unknown	4	0	4	4

- 57,040 inputs handled by bugfix
  - 98 inputs never fixed
  - 4 inputs “fixed” but due to some source of nondeterminism
- In general: Far less over counting
  - At most 596 hashes for **9 bugs**
  - vs. 57,040 inputs for 9 bugs
- Hashes have false negatives
  - Bug B has 343 hashes that apply just to this bug, but 19 that apply to others too

cxxfilt: AFL CMIN vs. Bugs



- No trial found more than 8 bugs
  - Out of 9 total
- 3 bugs account for most crashing inputs
  - many bugs have few inputs
  - so counting inputs misleading
- Number of crashing inputs correlates with number of bugs, but only loosely
- Mann Whitney p-value is .066 for AFLFast *bugs* > AFL bugs
  - vs. 10^{-10} for “unique” crashes

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

How are things in late 2020? Better

Paper	Where	When	Benchmarks	Baseline	Trials	Variance	Crash	Coverage	Seed	Timeout
DIE (JS)	S&P	2020	R(3)	Superion, CA	5	C	G*, C*	E (path)	R	24 H
Ijon	S&P	2020	C*, R*	A	3		G	E (path)	M	24 H
Pangolin	S&P	2020	R(9), L	A, AF, Q, D, Angora, T-Fuzz	10	M-W	G, C		M*	24 H
Retrowrite	S&P	2020	L	AFL in various modes	5	M-W	G		V	24 H
SAVIOR	S&P	2020	L, R(8)	A, AG, TFuzz, Angora, Driller, Q	5	M-W	G, S-UBSAN(1)*	L		24x3 H
EcoFuzz	Sec	2020	R(14), G	A, AF, FairFuzz, …	5	Yes	G*, C	E (path)	?	24 H
EcoFuzz 2	Sec	2020	L	Angora, VUzzer	5	?	G			5 H
FiFUZZ	Sec	2020	R(9) R(5-binutils)	A, AF, AS, FairFuzz	3	?	G, O	E	?	24 H
GreyOne	Sec	2020	L, R(19)	A, V, Angora, CollAFL, Honggfuzz, Q	5		G, C*	E (path)	R (10)	60 H
Montage (JS)	Sec	2020	R(1)	CA, JSF, JFuzzer	5	M-W	G*, S		R, G, V	72x88 H
ParmeSan	Sec	2020	G	Angorra	30	M-W	G	E	V	48 H
Superion	ICSE	2019	R(4)	A, JSF			G*, C	L, M		"100 cycles" (3 months)
Zest	ISSTA	2019	R(5)	A, QC-junit	20	Yes	G	E	V(1)	3 H
UnTracer	S&P	2019	R(8)	AFL in various modes	8	M-W, A12	-	-	?	24 H
EnFuzz	Sec	2019	L, G, R(15)	A, AF, FairFuzz, Q, libFuzzer, R	10	"within 5%"	G, S-ASAN(1)	E (path)	V	24*4 H
TIFF	ACSAC	2018	L, R(9)	AF, VUzzer	3	"marginal statistical variations"	G(*), S	L	R(4)	12 H

- 14/15 papers' results based on ground truth
  - At least for part of their benchmarks
- 10/15 also used "unique crashes"
  - Varying levels of extra effort to avoid over/undercounts
    - ASAN or UBSAN instrumentation
- 11/15 also measured code coverage

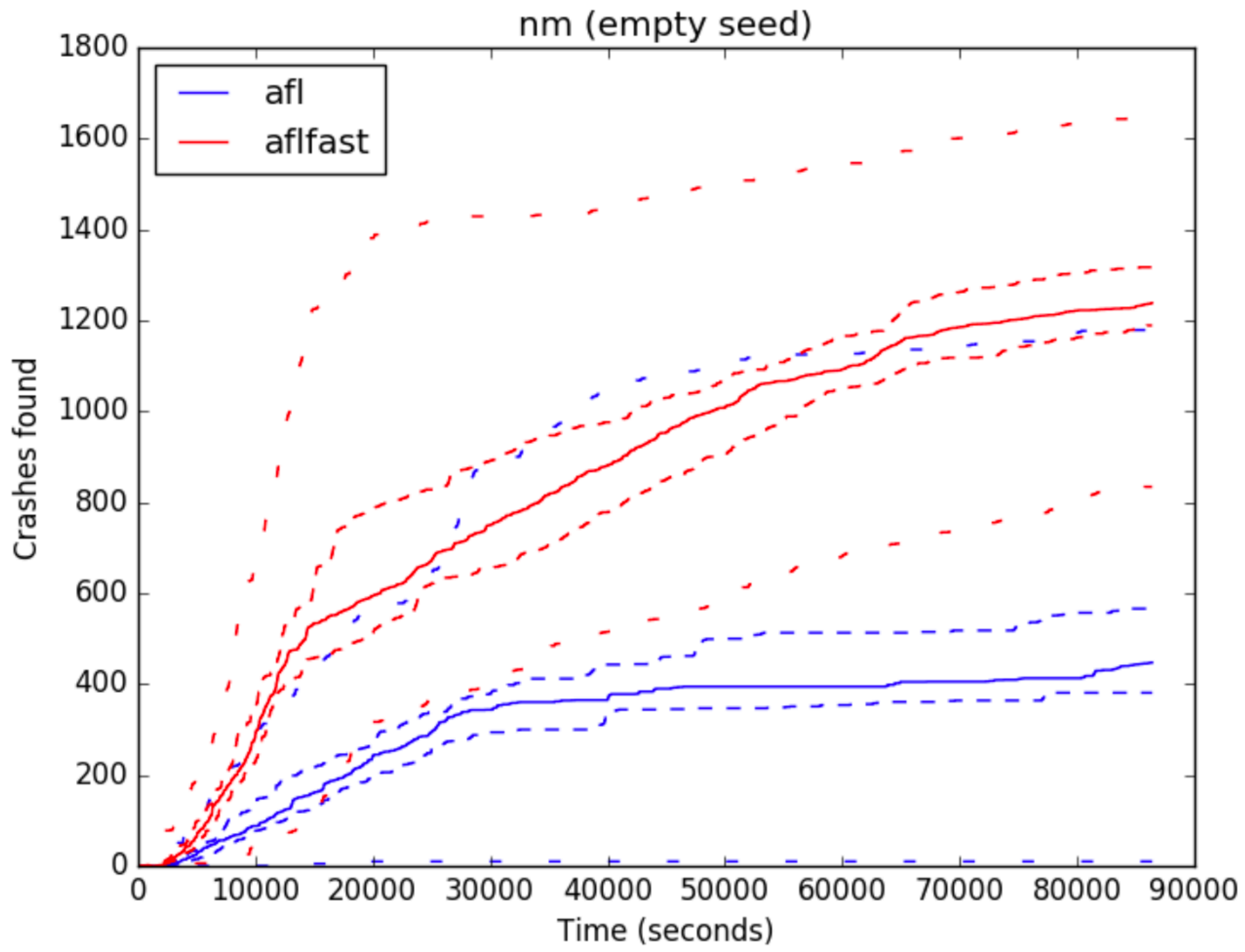
Target Programs

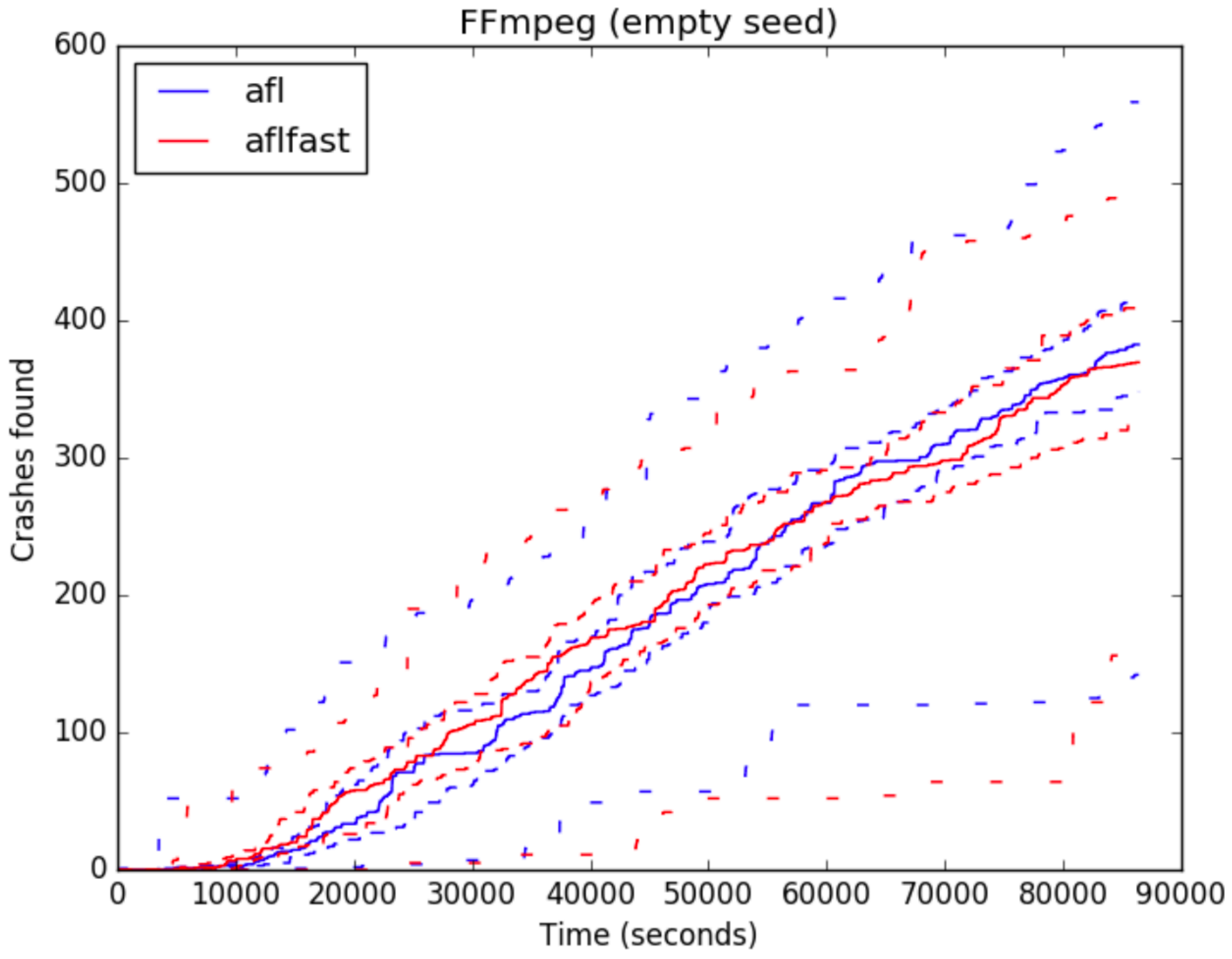
paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[8]	R(29)				G	?	N	-
FuzzSim[55]	R(101)	B	100	C	S		R/M	10D
Dowser[22]	R(7)	O	?		O		N	8H
COVERSET[45]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[9]	R(8)	A, B, Z			S		M	1H
MutaGen[29]	R(8)	R, Z			S	L	V	24H
SDF[35]	R(1)	Z, O			O		V	5D
Driller[50]	C(126)	A			G	L, E	N	24H
QuickFuzz-1[20]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[54]	R(5)	O			M	O	G, R	2H
[57]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[44]	C(63), L, R(10)	A			G, S, O		N	6H, 24H
SlowFuzz[41]	R(10)	O	100		-		N	
Steelix[33]	C(17), L, R(5)	A, V, O			C, G	L, E, M	N	5H
Skyfire[53]	R(4)	O			?	L, M	R, G	LONG
kAFL[47]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[13]	R(7)	O			G*		G	5H
Orthrus[49]	G(4), R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[27]	R(1)	O			G*		G	-
VDF[25]	R(18)				C	E	V	30D
QuickFuzz-2[21]	R(?)	O	10		G*		G, M	
IMF[23]	R(1)	O			G*	O	G	24H
[59]	S(?)	O	5		G		G	24H
NEZHA[40]	R(6)	A, L, O	100		O		R	
[56]	G(10)	A, L					V	5M
S2F[58]	L, R(8)	A, O			G	O	N	5H, 24H
FairFuzz[32]	R(9)	A	20	C		E	V/M	24H
Angora[10]	L, R(8)	A, V, O	5		G, C	L, E	N	5H
T-Fuzz[39]	C(296), L, R(4)	A, O	3		C, G*		N	24H
MEDS[24]	S(2), R(12)	O	10		C		N	6H

Evaluations

- 30/32 used real programs
  - Median of 7, as many as 100
  - 2/32 use Google Fuzz Suite
  - Fair/sufficient sample?
- 9/32 purposely-vulnerable programs (or injected bugs)
  - 5 use LAVA-M
  - 4 use CGC
  - Ecological validity?

Binutils vs. Image proc.





$p < 10^{-13}$

$p = 0.379$

From AFLFast paper

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 confirmed** (e.g., mean size is 1800 LOC)
  - 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

How are things in late 2020? Better

Paper	Where	When	Benchmarks	Baseline	Trials	Variance	Crash	Coverage	Seed	Timeout
DIE (JS)	S&P	2020	R(3)	Superion, CA	5	C	G*, C*	E (path)	R	24 H
Ijon	S&P	2020	C*, R*	A	3		G	E (path)	M	24 H
Pangolin	S&P	2020	R(9), L	A, AF, Q, D, Angora, T-Fuzz	10	M-W	G, C		M*	24 H
Retrowrite	S&P	2020	L	AFL in various modes	5	M-W	G		V	24 H
SAVIOR	S&P	2020	L, R(8)	A, AG, TFuzz, Angora, Driller, Q	5	M-W	G, S-UBSAN(1)*	L		24x3 H
EcoFuzz	Sec	2020	R(14), G	A, AF, FairFuzz, …	5	Yes	G*, C	E (path)	?	24 H
EcoFuzz 2	Sec	2020	L	Angora, VUzzer	5	?	G			5 H
FiFUZZ	Sec	2020	R(9) R(5-binutils)	A, AF, AS, FairFuzz	3	?	G, O	E	?	24 H
GreyOne	Sec	2020	L, R(19)	A, V, Angora, CollAFL, Honggfuzz, Q	5		G, C*	E (path)	R (10)	60 H
Montage (JS)	Sec	2020	R(1)	CA, JSF, JFuzzer	5	M-W	G*, S		R, G, V	72x88 H
ParmeSan	Sec	2020	G	Angorra	30	M-W	G	E	V	48 H
Superion	ICSE	2019	R(4)	A, JSF			G*, C	L, M		"100 cycles" (3 months)
Zest	ISSTA	2019	R(5)	A, QC-junit	20	Yes	G	E	V(1)	3 H
UnTracer	S&P	2019	R(8)	AFL in various modes	8	M-W, A12	-	-	?	24 H
EnFuzz	Sec	2019	L, G, R(15)	A, AF, FairFuzz, Q, libFuzzer, R	10	"within 5%"	G, S-ASAN(1)	E (path)	V	24*4 H
TIFF	ACSAC	2018	L, R(9)	AF, VUzzer	3	"marginal statistical variations"	G(*), S	L	R(4)	12 H

- Standard benchmarks in greater use
  - 7/15 use LAVA-M
  - 3/15 use GoogleTS
  - 1/15 use CGC
  - All provide ground truth
- Real-world programs often diverse, used before
  - Some impressive choices: 19 programs in one case!

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
  - Regular competition, like SAT competition?
- Google Fuzz, CGC, LAVA-M, current papers may be good starting points

New! FuzzBench

- https://github.com/google/fuzzbench

Our paper

Why do we need a fuzzer benchmarking platform?

Evaluating fuzz testing tools properly and rigorously is difficult, and typically needs time and resources that most researchers do not have access to. A study on Evaluating Fuzz Testing analyzed 32 fuzzing research papers and has found that "*no paper adheres to a sufficiently high standard of evidence to justify general claims of effectiveness*". This is a problem because it can lead to unreproducible results.

We created FuzzBench, so that all researchers and developers can evaluate their tools according to the best practices and guidelines, with minimal effort and for free.

SIGPLAN Guidelines

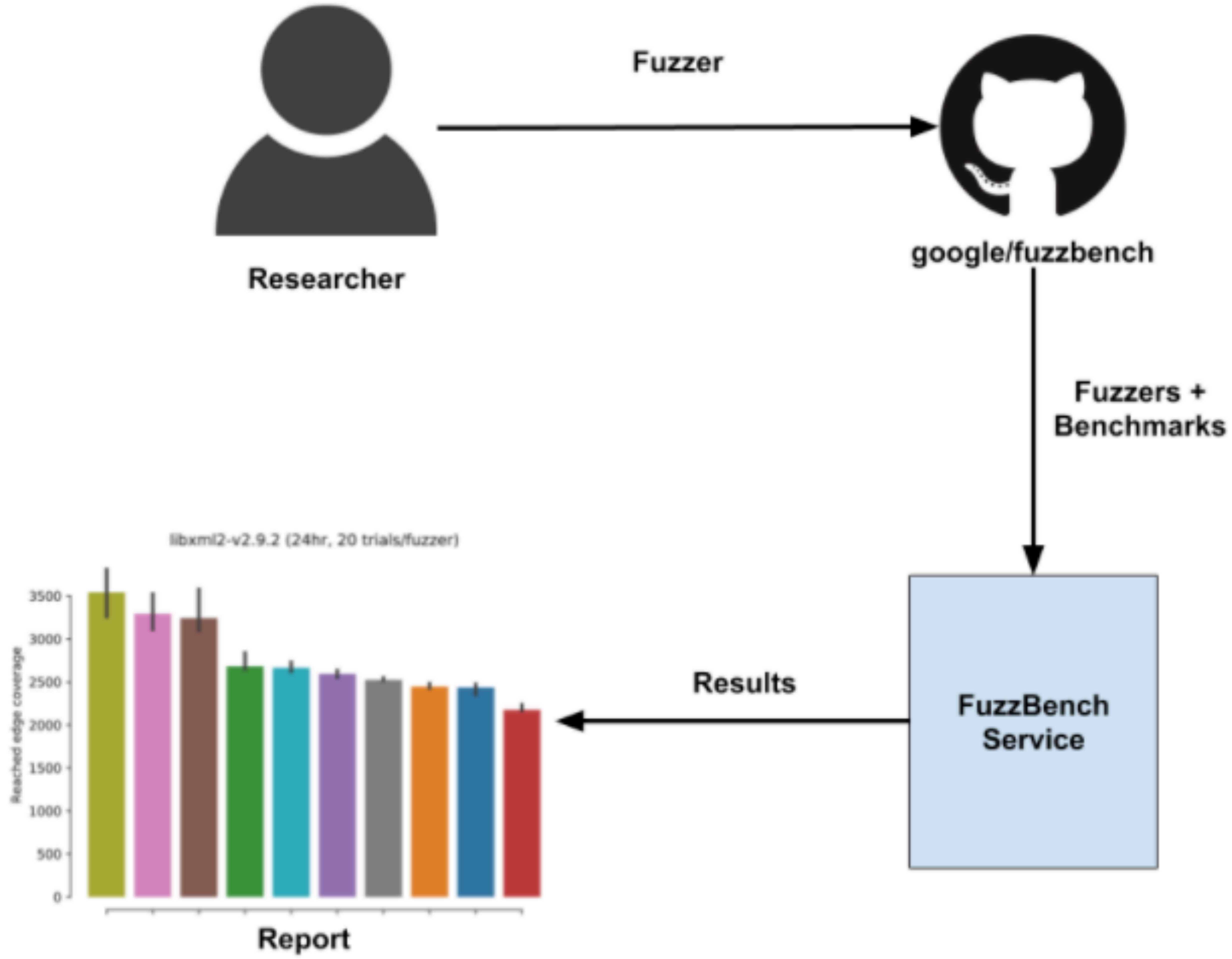
New! FuzzBench

- https://github.com/google/fuzzbench

Why do we need a fuzzer benchmarking platform?

Evaluating fuzz testing tools properly and rigorously is difficult, and typically needs time and resources that most researchers do not have access to. A study on Evaluating Fuzz Testing analyzed 32 fuzzing research papers and has found that *"no paper adheres to a sufficiently high standard of evidence to justify general claims of effectiveness"*. This is a problem because it can lead to unreproducible results.

We created FuzzBench, so that all researchers and developers can evaluate their tools according to the best practices and guidelines, with minimal effort and for free.



New! FuzzBench

- https://github.com/google/fuzzbench
- **21 programs** and libraries with **known bugs**
  - OpenSSL, SQLite3, WolfSSL, Zlib, Libpng, LibPCAP, ..
  - Can use any OSS-Fuzz project as a benchmark
- Connects to *many* fuzzers
- Measures (via **20 trials**, **24 hours**)
  - Median **total edge coverage**, and over time, per program. Graphs median.
  - **Missing**: measurement based on **ground-truth bugs**
    - Stated plans to add it

Magma: A Ground-Truth Fuzzing Benchmark

Ahmad Hazimeh
EPFL

Adrian Herrera
Australian National University & Defence Science and Technology Group

Mathias Payer
EPFL

ABSTRACT

High scalability and low running costs have made fuzz testing the de facto standard for discovering software bugs. Fuzzing techniques are constantly being improved in a race to build the ultimate bug-finding tool. However, while fuzzing excels at finding bugs in the wild, evaluating and comparing fuzzer performance is challenging due to the lack of metrics and benchmarks. For example, crash count—perhaps the most commonly-used performance metric—is inaccurate due to imperfections in deduplication techniques. Additionally, the lack of a unified set of targets results in ad hoc evaluations that hinder fair comparison.

We tackle these problems by developing *Magma*, a ground-truth fuzzing benchmark that enables uniform fuzzer evaluation and comparison. By introducing *real* bugs into *real* software, Magma allows for the realistic evaluation of fuzzers against a broad set of targets. By instrumenting these bugs, Magma also enables the collection of bug-centric performance metrics independent of the fuzzer. Magma is an open benchmark consisting of seven targets that perform a variety of input manipulations and complex computations, presenting a challenge to state-of-the-art fuzzers.

We evaluate six widely-used mutation-based greybox fuzzers (AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, and honggfuzz) against Magma over 200 000 CPU-hours. Based on the number of bugs, reached, triggered, and detected, we draw conclusions about the fuzzers' exploration and detection capabilities. This provides insight into fuzzer performance evaluation, highlighting the importance of ground truth in performing more accurate and meaningful evaluations.

1 INTRODUCTION

While these metrics provide some insight into a fuzzer's performance, we argue that they are insufficient for use in fuzzer comparisons. Furthermore, the set of target programs that these metrics are evaluated on can vary wildly across papers, making cross-paper comparisons impossible. The deficiencies of these three metrics are discussed in turn.

Crash counts. The simplest method for evaluating a fuzzer is to count the number of crashes triggered by that fuzzer, and compare this crash count with that achieved by another fuzzer on the same target program. Unfortunately, crash counts often inflate the number of actual bugs in the target program [29]. Moreover, deduplication techniques (e.g., coverage profiles, stack hashes) fail to accurately identify the root cause of these crashes [9, 29].

Bug counts. Identifying a crash's *root cause* is preferable to simply reporting raw crashes, as it avoids the inflation problem inherent in crash counts. Unfortunately, obtaining an accurate *ground-truth* bug count typically requires extensive manual triage, which in turn requires someone with extensive domain expertise and experience [1].

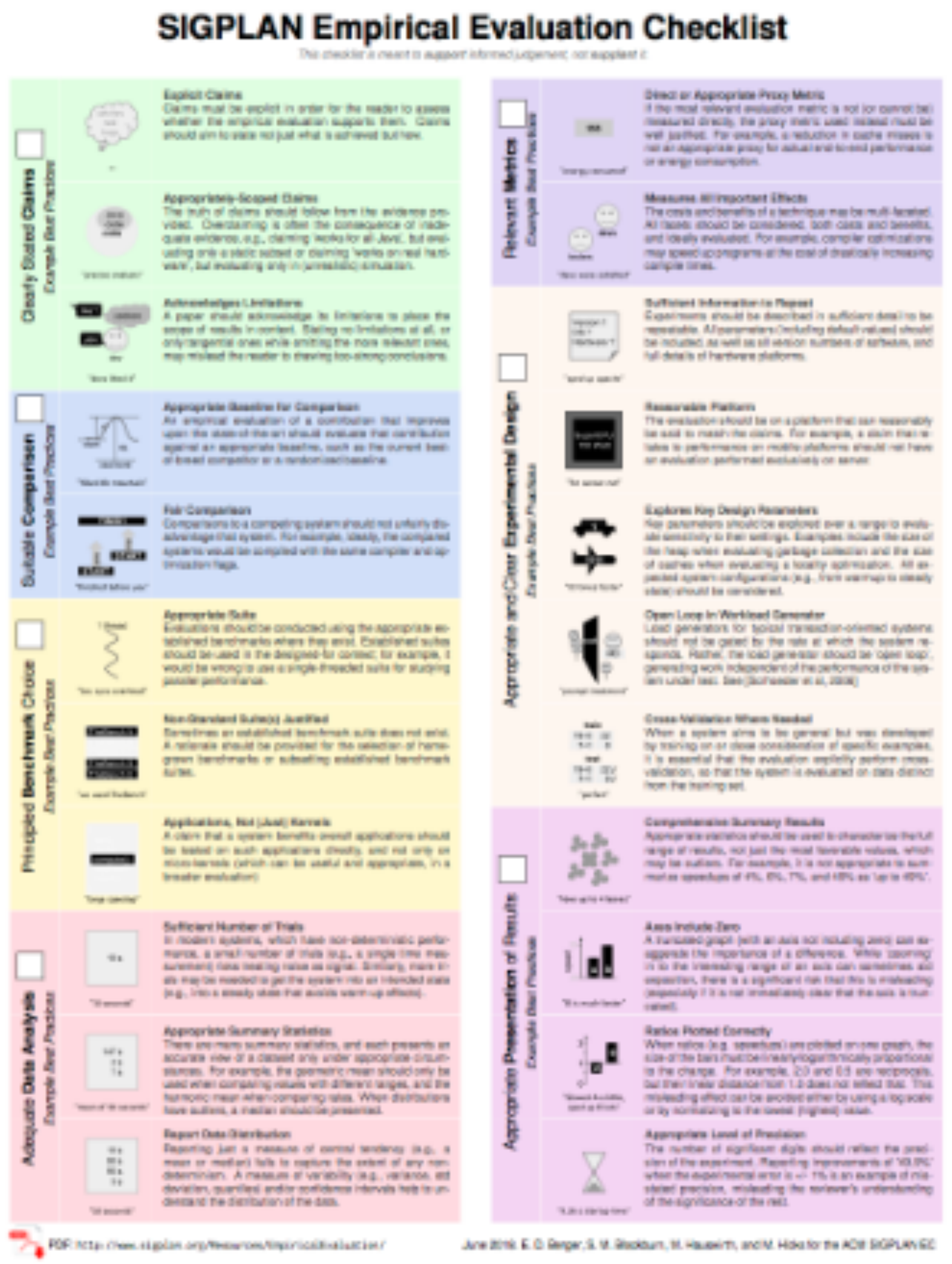
Code-coverage profiles. Due to the difficulty in obtaining ground-truth bug counts, code-coverage profiles are another performance metric commonly used to evaluate and compare fuzzing techniques. Intuitively, covering more code correlates with finding more bugs. However, previous work [29] has shown that there is a weak correlation between coverage-deduplicated crashes and ground-truth bugs, implying that higher coverage does not necessarily indicate better fuzzer effectiveness.

The deficiencies of existing performance metrics calls for a rethink of fuzzer evaluation practices. In particular, the performance metrics used in these evaluations must accurately measure a fuzzer's

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 several, including the empty seed
- Do use **longer timeout** and measure performance over time
- Use a **good benchmark suite** (to be developed!)

General advice: SIGPLAN guidelines!



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