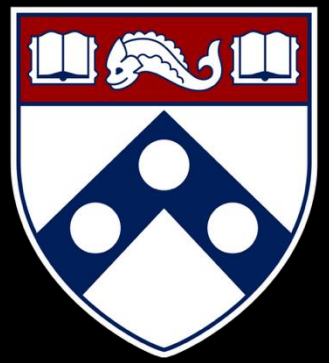
Secure Systems Engineering and Management

A Data-driven Approach

Michael Hicks



Coverage-guided Fuzzing

UPenn CIS 7000-003
Spring 2026

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
  - These can lead to security vulnerabilities
- **Complements functional testing**
  - Test features (and lack of misfeatures) directly
  - Normal tests can be starting points for fuzz tests

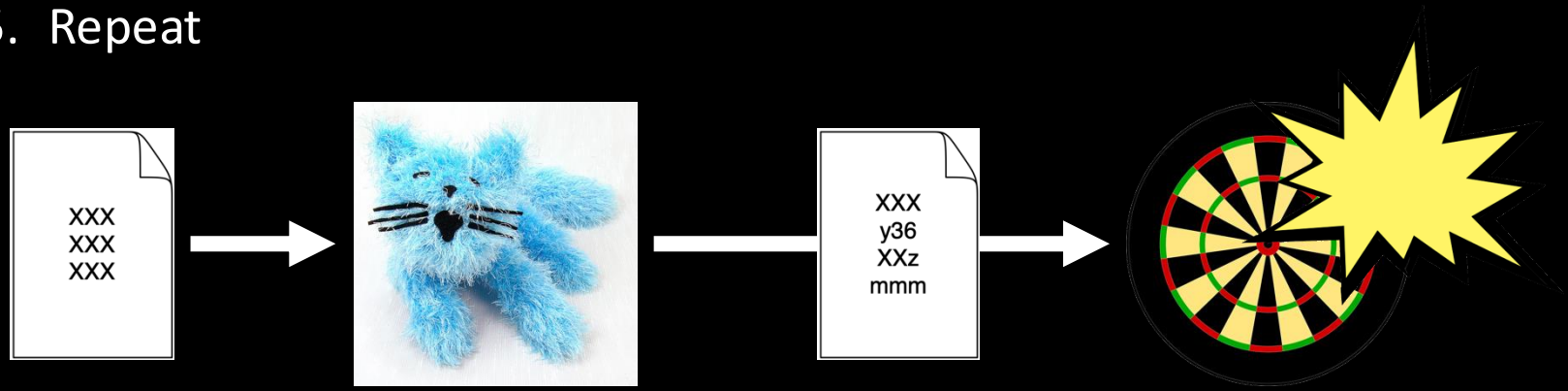
Context: Quality Gates for Secure Software

Gate	When	What
Compiler	Every build	Type errors, warnings
Tests + linting	Check-in / CI	Functional correctness, bug patterns
Fuzzing	Post check-in / nightly	Bugs that tests miss
Release fuzzing	Before shipping	Extended campaign (days/weeks)

A kind of "penetration testing"

The Fuzzing Loop

1. Start with some seed inputs
2. Pick one to mutate
3. Feed the mutated input to the program
4. If something “interesting” happened, keep it
5. Repeat



The Key Questions

- What counts as **“interesting”**?
- **Which** input do you mutate next?
- **How many** mutations do you generate?

Different answers → dramatically different results

Fuzzing Starts in 1990

Miller et al. piped random strings into UNIX utilities

Crashed 25–33% of them

An Empirical

When we use basic operating system facilities, such as the kernel and major utility programs, we expect a high degree of reliability. These parts of the system are used frequently and this frequent use implies that the programs are well-tested and working correctly. To make a systematic statement about the correctness of a program, we should probably use some form of formal verification. While the technology for program verification is advancing, it has not yet reached the point where it is easy to apply (or commonly applied) to large systems.

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing: On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash ("core dump"); on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us to believe that there might be serious bugs lurking in the systems that we regularly used.

This scenario motivated a systematic test of the utility programs running on various versions of the Unix operating system. The project proceeded in four steps: (1) programs were constructed to generate random characters, and to help test interactive utilities: (2) these programs were used to test a large number of utilities on random input strings to see if they crashed; (3) the strings (or types of strings) that crash these programs were identified; and (4) the causes of the program crashes were identified and the common mistakes that cause these crashes were categorized. As a result of testing almost 90 different utility programs on seven versions of Unix™, we were able to crash more than 24% of these programs. Our testing included versions of Unix that underwent commercial product testing. A byproduct of this project is a list of bug reports (and fixes) for the crashed programs and a set of tools available to the systems community.

There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashs or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

This type of study is important for several reasons: First, it contributes to the testing community a large list of real bugs. These bugs can provide test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, one of the bugs that we found was caused by the same programming practice that provided one of the security holes to the Internet worm (the "gets finger" bug) [2,3] We have found additional bugs that might indicate future security holes. Third, some of the crashes were caused by input that might be carelessly typed—some strange and unexpected errors were uncovered by this method of testing. Fourth, we sometimes inadvertently feed programs noisy input (e.g., trying to edit or view an object module). In these cases, we would like some meaningful and predictable response. Fifth, noisy phone lines are a reality, and major utilities (like shells and editors) should not crash because of them. Last, we were interested in the interactions between our random testing and more traditional industrial software testing.

While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache testing. When testing network protocols, a module can be inserted in the data stream. This module randomly perturbs the packets (either destroying them or modifying them) to test the protocol's error detection and recovery features. Random testing has been used in evaluating complex hardware, such as multiprocessor cache coherence protocols [4]. The state space of the device, when combined with the memory architecture, is large enough that it is difficult to generate systematic tests. In the multiprocessor example, random generation of test cases helped cover a large part of the state space and simplify the generation of cases.

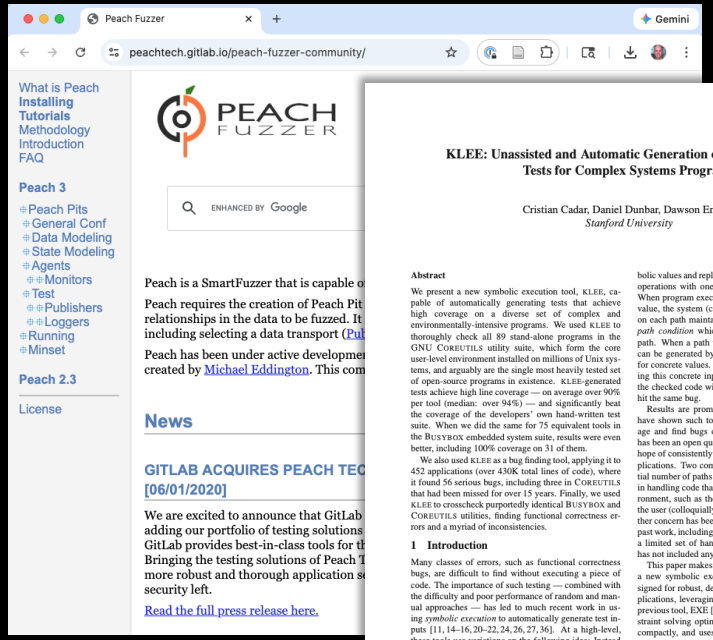
Unix is a trademark of AT&T Bell Laboratories.

2000s: Smarter Input Generation

Grammar-based: SPIKE, Peach, Sulley — Encode format knowledge to generate valid-ish inputs

Symbolic execution: SAGE, KLEE — Solve path constraints to explore systematically

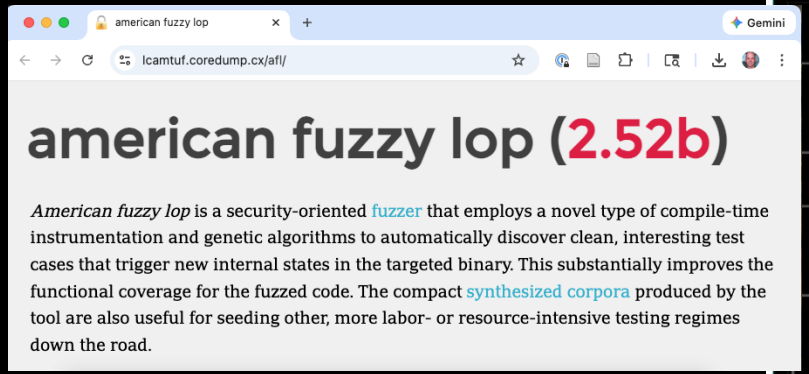
- “white box” as compared to Miller’s “black box” approach

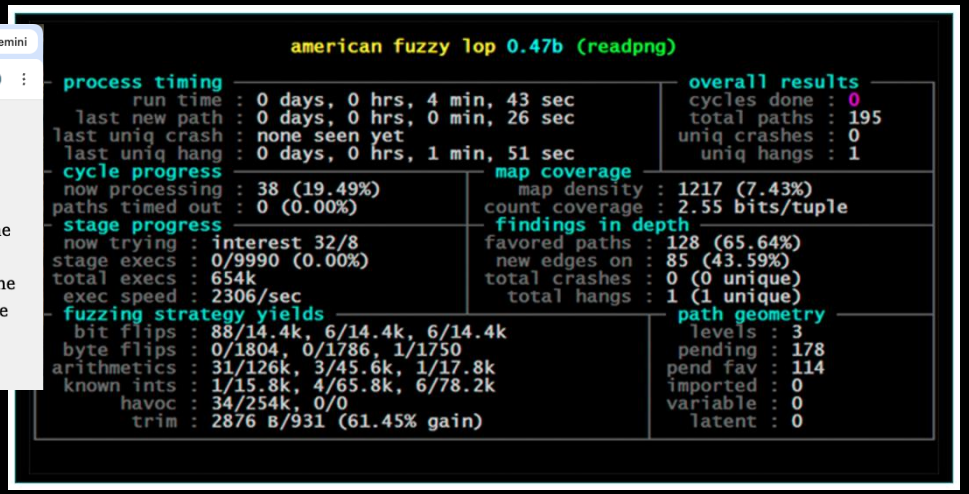


2013: AFL Changes Everything

Michal Zalewski (lcamtuf)'s insight: **lightweight instrumentation** gives coverage feedback without program analysis

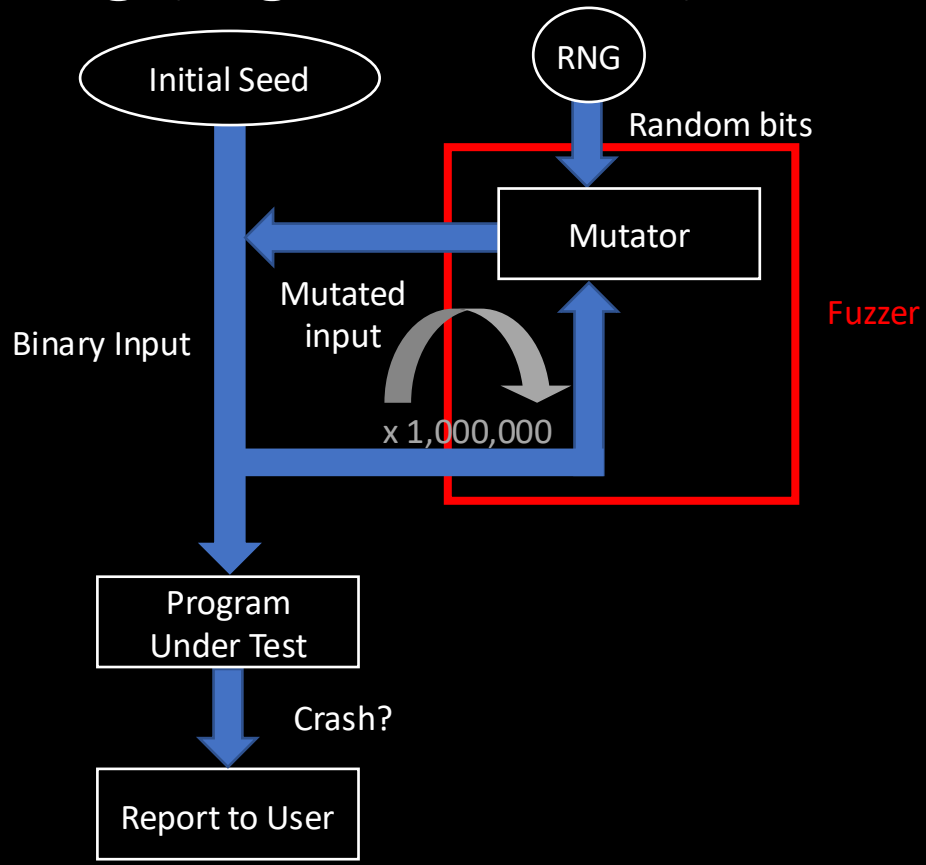
Greybox = black-box speed + white-box intelligence





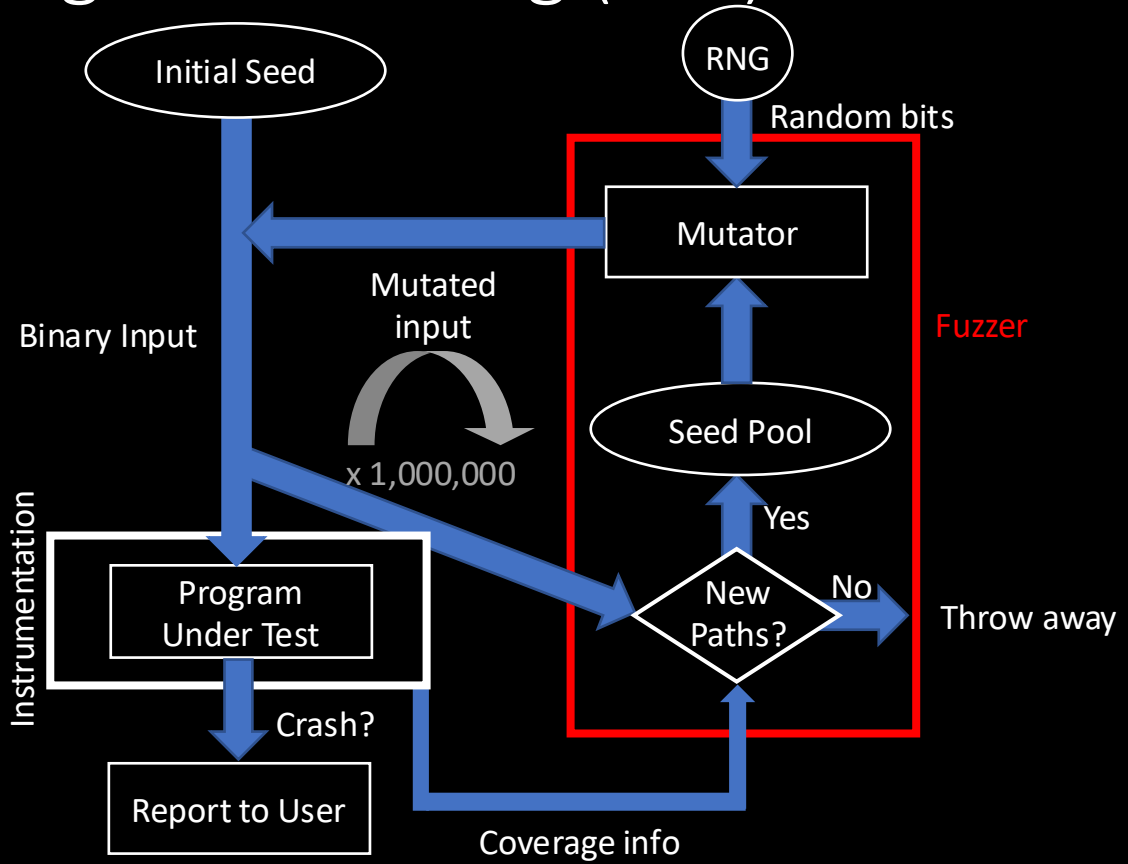
Blackbox fuzzing (e.g., Radamsa)

Repeatedly mutate an initial (binary) seed

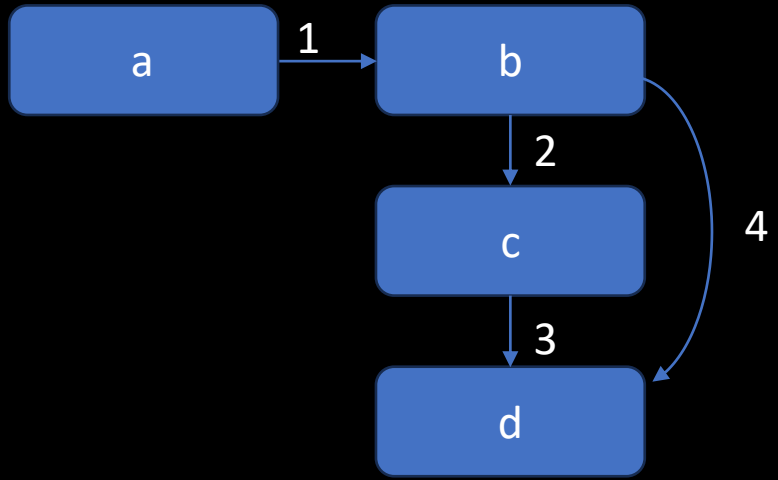


Coverage-guided fuzzing (AFL)

Only mutate seeds that induced new execution paths

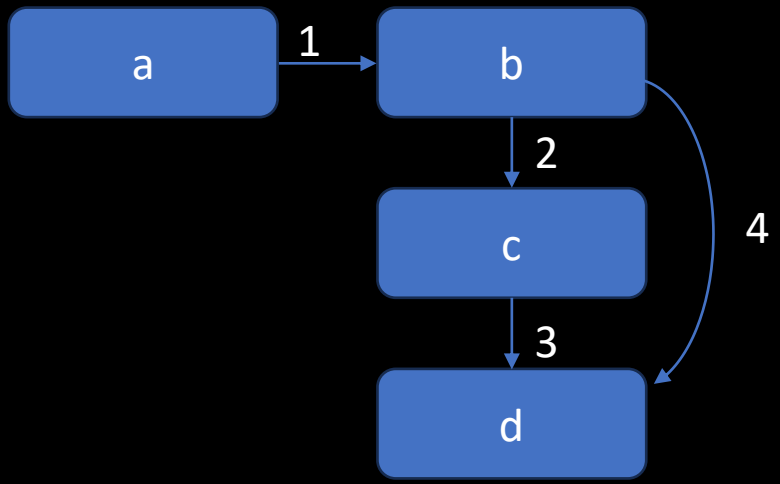


AFL's Instrumentation



AFL's Instrumentation

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

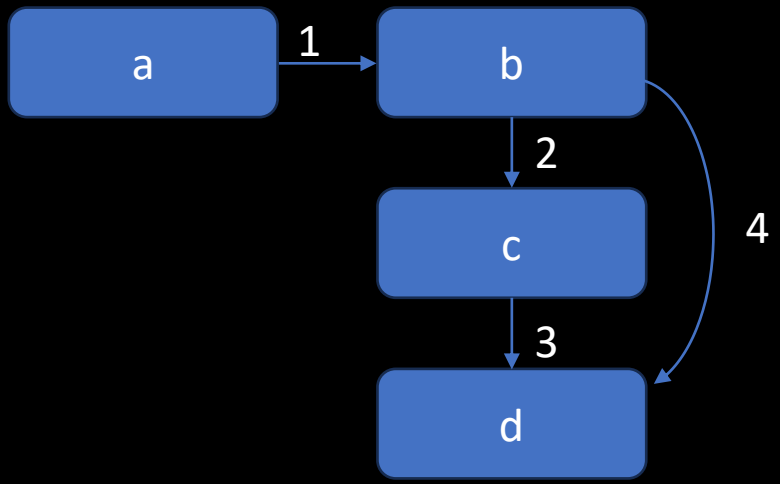


Edge	Index
1	b^(a>>1)
2	c^(b>>1)
3	d^(c>>1)
4	d^(b>>1)
…	

Edge Coverage, Not Block Coverage

Edges capture **control flow transitions**, not just “which code ran”

After each run, hit counts are **bucketed**: 1, 2, 4, 8, 16, 32, 64, 128+



Edge	Index	P1	P2
1	b^(a>>1)	1	1
2	c^(b>>1)	1	
3	d^(c>>1)	1	
4	d^(b>>1)		1
…			

The Fuzzing Loop

1. Start with some seed inputs
2. Pick one to mutate
3. Feed the mutated input to the program
4. If something “interesting” happened, keep it
5. Repeat

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S

```
1:  T_x = ∅
2:  T = S
3:  if T = ∅ then
4:      add empty file to T
5:  end if
6:  repeat
7:      t = chooseNext(T)
8:      p = assignEnergy(t)
9:      for i from 1 to p do
10:         t' = mutate_input(t)
11:         if t' crashes then
12:             add t' to T_x
13:         else if isInteresting(t') then
14:             add t' to T
15:         end if
16:     end for
17: until timeout reached or abort-signal
```

Output: Crashing Inputs T_x

Choose and Allocate

- Pick an input: CHOOSENEXT
  - Circular queue, favorites first
- How many mutations? ASSIGNENERGY
  - ~80K per seed (~1 min)

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S

```
1:  T_x = ∅
2:  T = S
3:  if T = ∅ then
4:      add empty file to T
5:  end if
6:  repeat
7:      t = chooseNext(T)
8:      p = assignEnergy(t)
9:      for i from 1 to p do
10:         t' = mutate_input(t)
11:         if t' crashes then
12:             add t' to T_x
13:         else if isInteresting(t') then
14:             add t' to T
15:         end if
16:     end for
17: until timeout reached or abort-signal
```

Output: Crashing Inputs T_x

Mutate and Evaluate

- Mutate the input: MUTATE_INPUT:
  - bit flips, arithmetic, boundary values, block splice
- Evaluate
  - Run instrumented binary, read bitmap
- Interesting? ISINTERESTING
  - new edge or new hit-count bucket → keep

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S

```
1:  T_✗ = ∅
2:  T = S
3:  if T = ∅ then
4:      add empty file to T
5:  end if
6:  repeat
7:      t = chooseNext(T)
8:      p = assignEnergy(t)
9:      for i from 1 to p do
10:         t' = mutate_input(t)
11:         if t' crashes then
12:             add t' to T_✗
13:         else if isInteresting(t') then
14:             add t' to T
15:         end if
16:     end for
17: until timeout reached or abort-signal
```

Output: Crashing Inputs $T_✗$

A Simple Target Program

```
void crashme (char* s) {
  if (s[0] == 'b')
    if (s[1] == 'a')
      if (s[2] == 'd')
        if (s[3] == '!')
          abort();
}
```

- How would a blackbox fuzzer behave on this program?
- How would AFL behave?

A Simple Target Program

```
void crashme (char* s) {
  if (s[0] == 'b')
    if (s[1] == 'a')
      if (s[2] == 'd')
        if (s[3] == '!')
          abort();
}
```

```
void crashme2 (char* s) {
  if (*(int *)s == 0x2a1646162)
    abort();
}
```

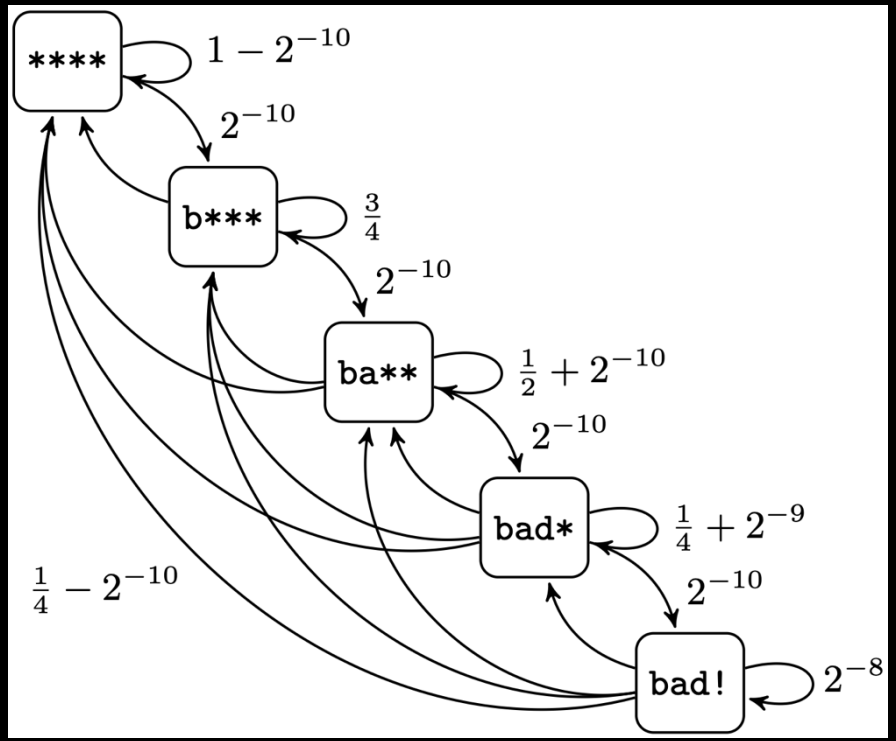
- How about this program?

Interlude: AFL++ Demo

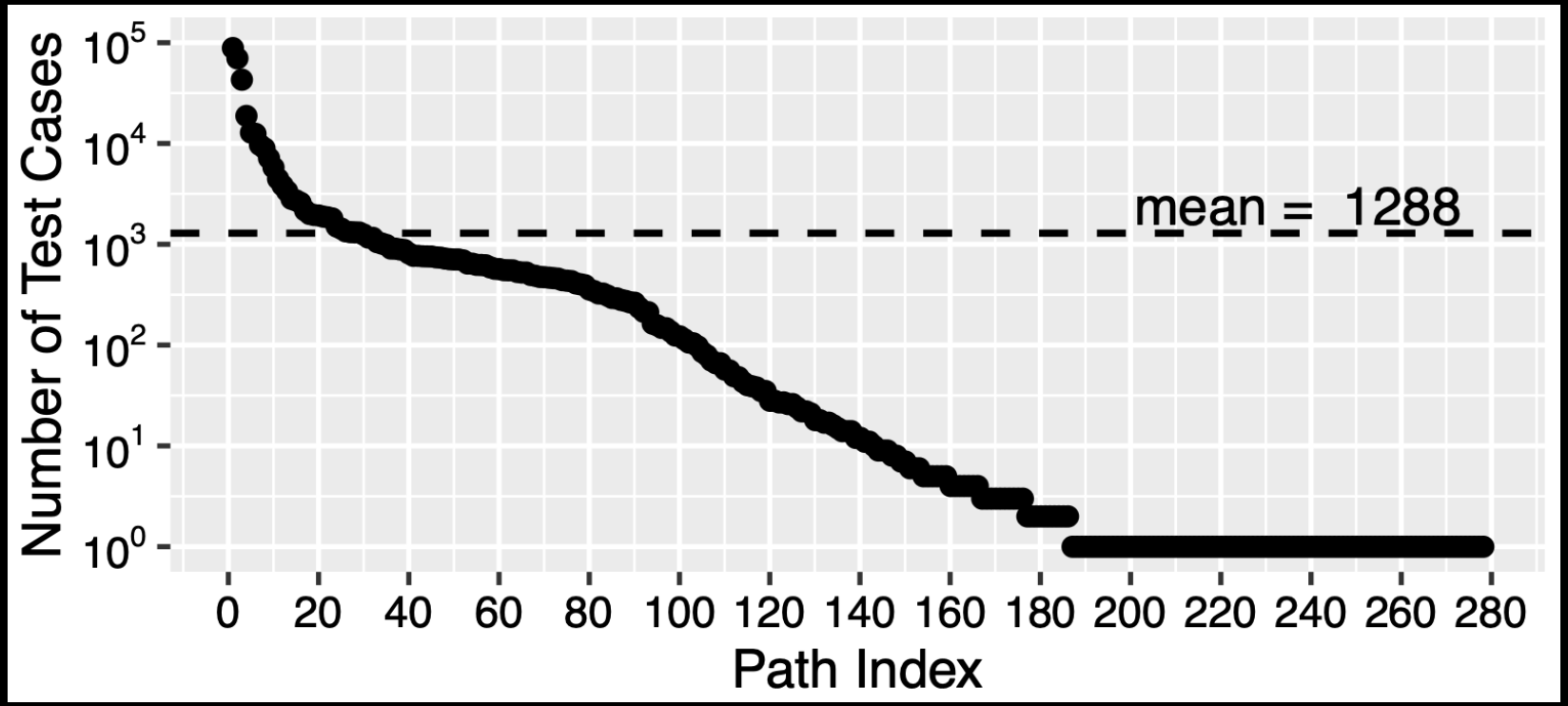
Five Paths, Wildly Different Probabilities

Path	Pattern	Probability
0	`****`	~255/256
1	`b***`	~1/256 × 255/256
2	`ba**`	~1/65K
3	`bad*`	~1/16M
4	`bad!`	~1/4B

Modeling as a Markov Chain



The Long Tail of Path Frequency



AFL's Two Scheduling Problems

1. **More energy than needed** on well-explored paths
2. **Too much energy in high-density regions** of the path space

The Waste: 256K Inputs to Find One Crash

#Total Tests	State	Explored States
1	`****`	`****`
$2^{16}+1$	`b***`	`****`, `b***`
$2 \cdot 2^{16}+1$	`ba**`	`****`, `b***`, `ba**`
$3 \cdot 2^{16}+1$	`bad*`	`****`, `b***`, `ba**`, `bad*`
$4 \cdot 2^{16}+1$	`bad!`	`****`, `b***`, `ba**`, `bad*`, `bad!`

Figure 3: The crash is found after $2^{18} = 256k$ inputs were generated by fuzzing when $p = 2^{16}$ is constant.

The Waste: 256K Inputs to Find One Crash

	State	****	b***	ba**	bad*	bad!
1	ba**	$1 \cdot 2^7$	$1 \cdot 2^7$	$2 \cdot 2^7$	0	0
2	****	$5 \cdot 2^7$	$1 \cdot 2^7$	$2 \cdot 2^7$	0	0
3	b***	$6 \cdot 2^7$	$4 \cdot 2^7$	$2 \cdot 2^7$	0	0
4	ba**	$7 \cdot 2^7$	$5 \cdot 2^7$	$4 \cdot 2^7$	1	0
5	****	$11 \cdot 2^7$	$5 \cdot 2^7$	$4 \cdot 2^7$	1	0
6	b***	$12 \cdot 2^7$	$8 \cdot 2^7$	$4 \cdot 2^7$	1	0
7	bad*	$13 \cdot 2^7$	$9 \cdot 2^7$	$5 \cdot 2^7$	$1 \cdot 2^7$	0
8	ba**	$14 \cdot 2^7$	$10 \cdot 2^7$	$7 \cdot 2^7$	$1 \cdot 2^7$	0
9	****	$18 \cdot 2^7$	$10 \cdot 2^7$	$7 \cdot 2^7$	$1 \cdot 2^7$	0
10	b***	$19 \cdot 2^7$	$13 \cdot 2^7$	$7 \cdot 2^7$	$1 \cdot 2^7$	0
11	bad*	$20 \cdot 2^7$	$14 \cdot 2^7$	$8 \cdot 2^7$	$2 \cdot 2^7$	**1**

Figure 4: Total #fuzz exercising the corresponding path when fuzzing the given state. Too much energy assigned to state ** and not enough to state bad* once it is discovered. Lines indicate new cycles.**

Power Schedules: The Key Idea

ASSIGNENERGY decides **how many mutations** to generate from seed t_i

Three variables control the decision:

- **s(i)**: times seed t_i has been chosen from the queue
- **f(i)**: total fuzz generated that exercises path i
- **α(i)**: AFL's quality score for the seed

Schedule 1: AFL's Default (EXPLOIT)

```
function ASSIGNENERGY(t_i):
    return α(i)
```

Energy is **constant** — same amount every time t_i is chosen

Schedule 2: Cut-Off Exponential (COE)

```
function ASSIGNENERGY(t_i):
    if f(i) > μ:
        return 0
    else:
        return min(α(i)/β · 2^s(i), M)
```

Path is above-average frequency; skip it entirely

Skip high-frequency paths; **exponentially increase** energy for rare ones

Schedule 3: FAST (AFLFast's Default)

```
function ASSIGNENERGY(t_i):
    if f(i) > μ:
        return 0
    else:
        return min(α(i)/β · 2^s(i) / f(i), M)
```

Like COE, but also **divides by f(i)** — less energy for well-fuzzed paths

The Payoff: 4K Inputs Instead of 256K

#Tests	State	Explored States
1	`****`	`****`
2^{10}	`b***`	`****`, `b***`
$2 \cdot 2^{10}$	`ba**`	`****`, `b***`, `ba**`
$3 \cdot 2^{10}$	`bad*`	`****`, `b***`, `ba**`, `bad*`
$4 \cdot 2^{10}$	`bad!`	`****`, `b***`, `ba**`, `bad*`, `bad!`

Figure 5: The crash is found after $2^{12} = 4k$ inputs were generated by fuzzing with a power schedule.

AFLFast Results

- **9 CVEs** in GNU binutils (previously unreported)
- **7× faster** than AFL at exposing known vulnerabilities
- **3 CVEs** not found by AFL in 24 hours at all
- Used by Team Codejitsu → **2nd place** at DARPA Cyber Grand Challenge

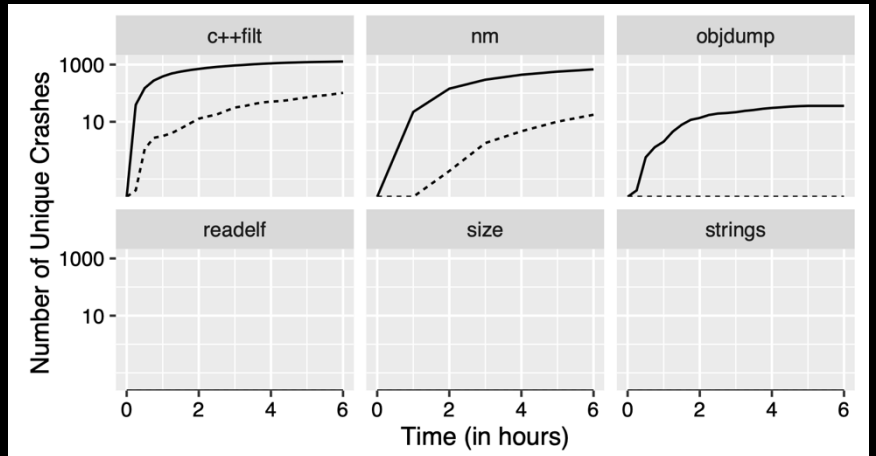
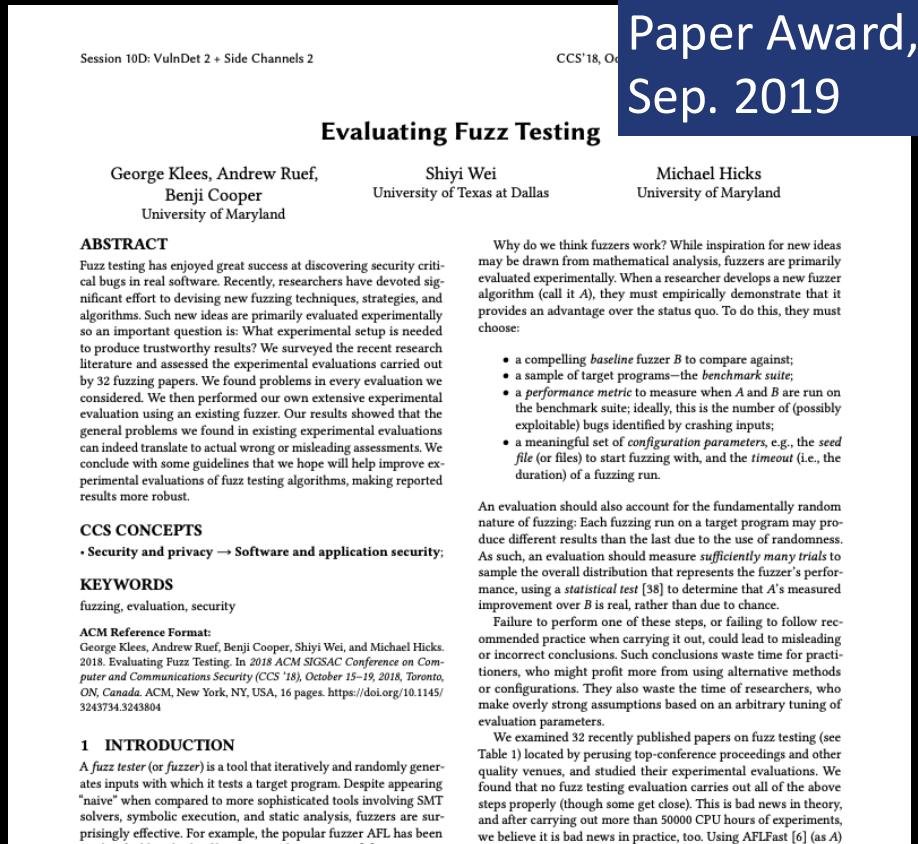


Figure 6: #Crashes over time (on a log-scale) for AFLFast (solid line) vs. AFL (dashed line)

Evaluating Fuzzing Results

- We looked at 32 published papers from 2012-2018
- We found that most papers did some things right, but none were perfect
- Carried out experiments on AFLFast that demonstrated the (real) costs of not doing so
  - Raises questions about the strength of published results

Won NSA Best Scientific Cybersecurity Paper Award, Sep. 2019



Session 10D: VulnDet 2 + Side Channels 2 CCS'18, Oc

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

1 INTRODUCTION

A *fuzz tester* (or *fuzzer*) is a tool that iteratively and randomly generates inputs with which it tests a target program. Despite appearing "naive" when compared to more sophisticated tools involving SMT solvers, symbolic execution, and static analysis, fuzzers are surprisingly effective. For example, the popular fuzzer AFL has been

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:

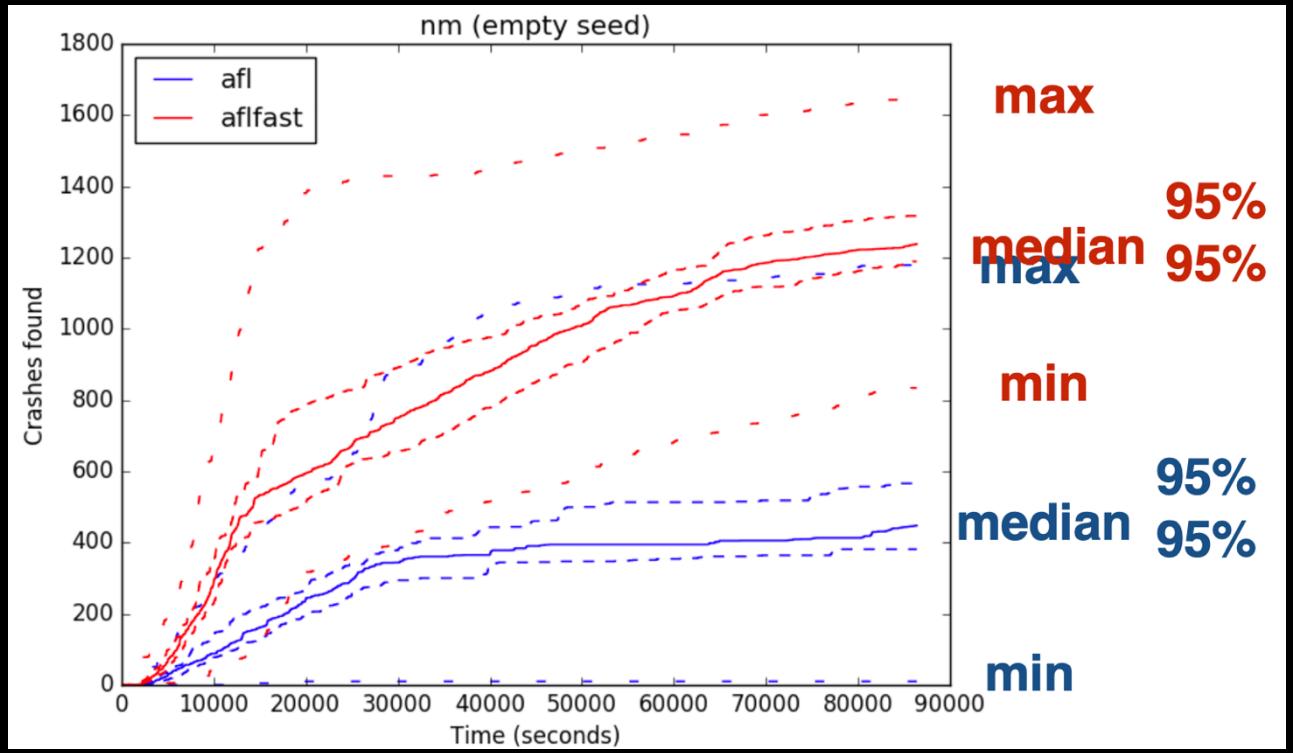
- a compelling *baseline* fuzzer *B* to compare against;
- a sample of target programs—the *benchmark suite*;
- a *performance metric* to measure when *A* and *B* are run on the benchmark suite; ideally, this is the number of (possibly exploitable) bugs identified by crashing inputs;
- a meaningful set 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 evaluation parameters.

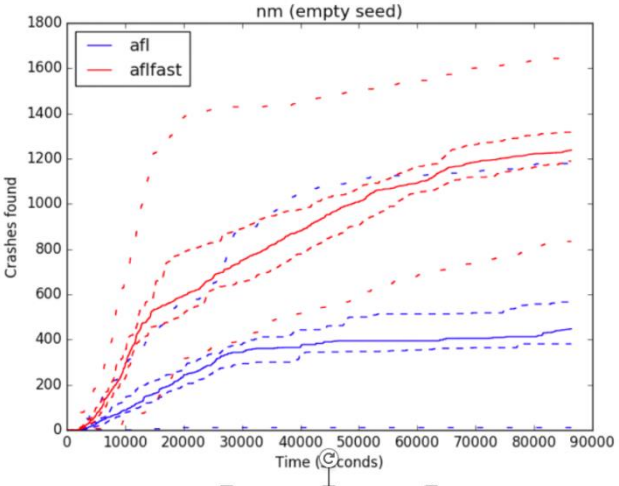
We examined 32 recently published papers on fuzz testing (see Table 1) located by perusing top-conference proceedings and other quality venues, and studied their experimental evaluations. We found that no fuzz testing evaluation carries out all of the above steps properly (though some get close). This is bad news in theory, and after carrying out more than 50000 CPU hours of experiments, we believe it is bad news in practice, too. Using AFLFast [6] (as *A*)

Fuzzing is Random: One Run Proves Nothing



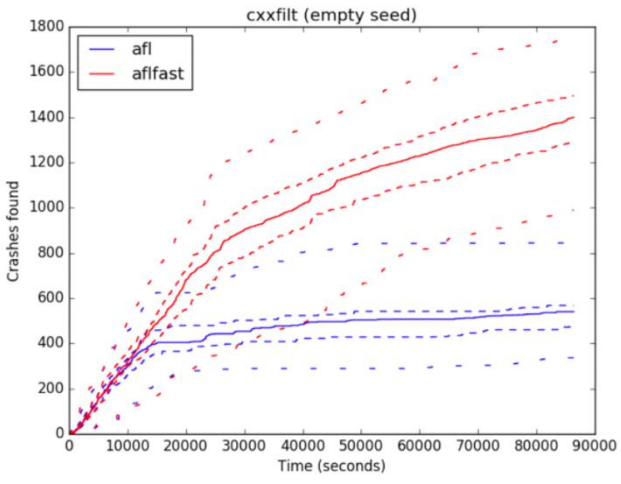
17/32 papers said nothing about multiple trials

Hypothesis testing: Need Mann Whitney U



$p < 10^{-13}$

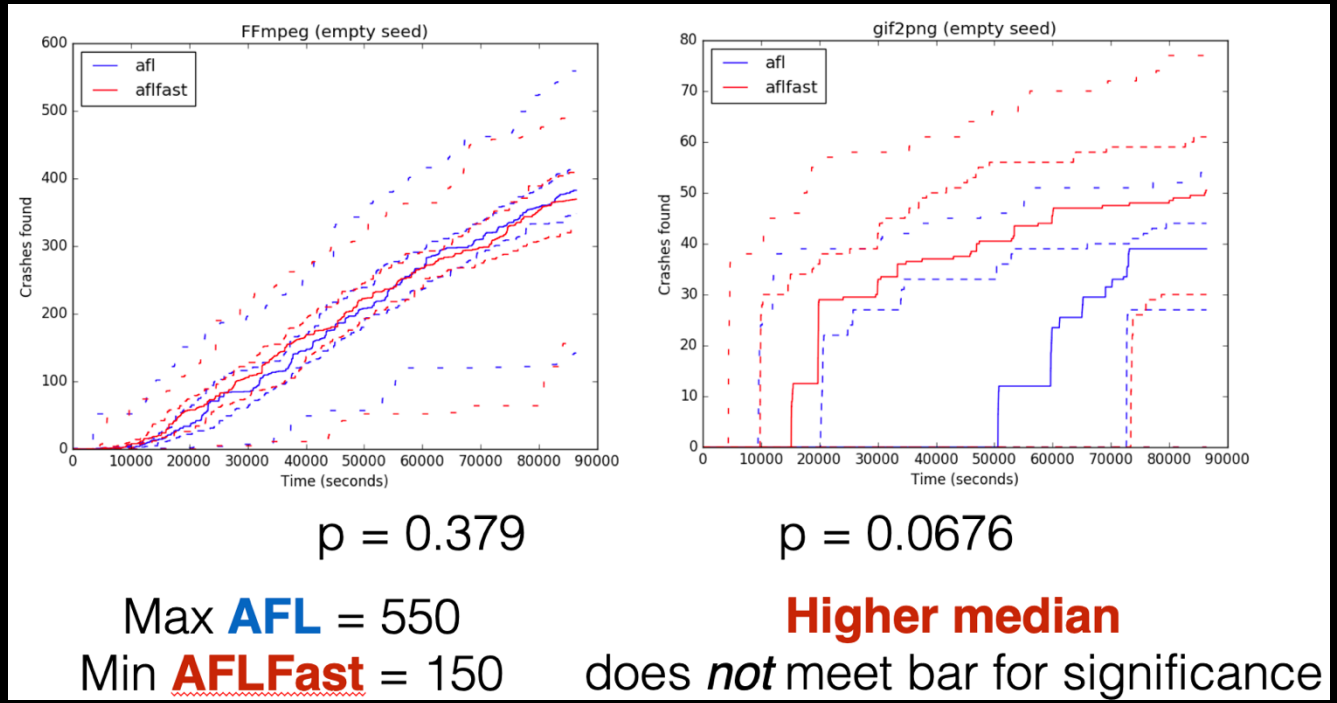
significant variance
in performance



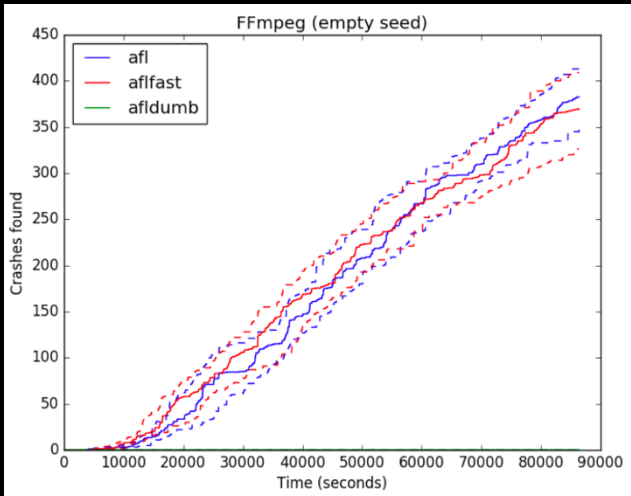
$p < 10^{-10}$

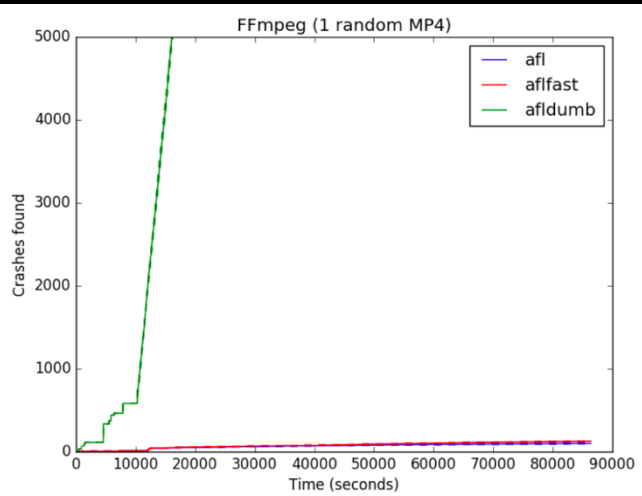
Higher median
clearly better

Hypothesis testing: Need Mann Whitney U



Seeds Have a Big Impact





<u>empty seed</u>

(AFLFast vs. AFL) $p = 0.379$

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

<u>1-made</u>

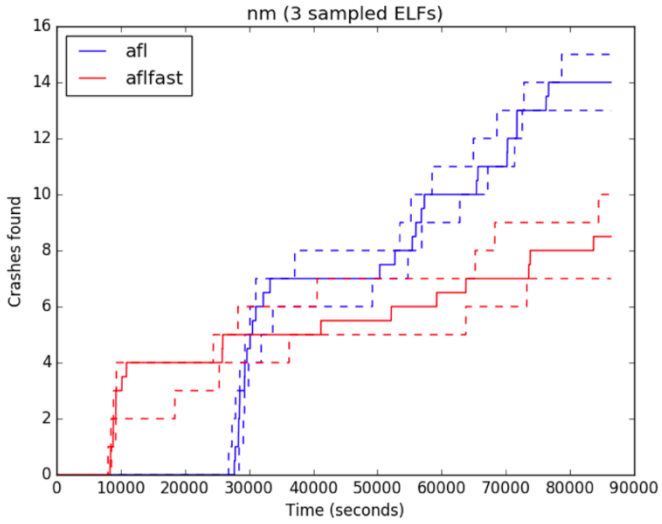
$p = 0.048$

$p < 10^{-11}$

30/32 papers used non-empty seed

- 10 say nothing else (N)
- 9 used valid seed but no details (V)

Timeouts too



3-sampled

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

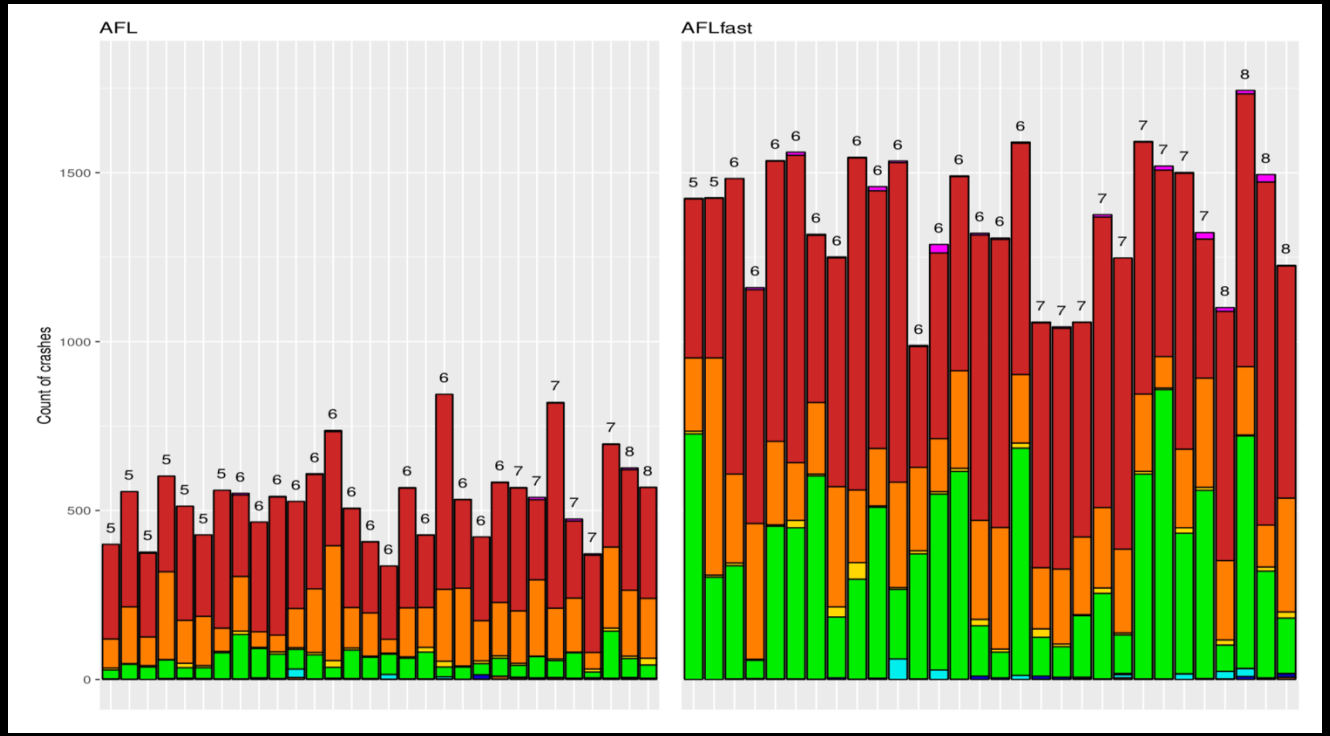
24 hours: $p = 0.000105$ **AFL** is better

- 10/32 papers ran 24 hours
- 7/32 papers ran 5 or 6 hours
- Others less, or much more

“Unique Crashes” is a Broken Metric

57,142 “unique” crashes in cxxfilt → **9 actual bugs**

That’s a **6,339× inflation**

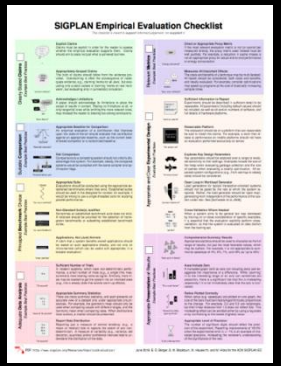


Recommendations

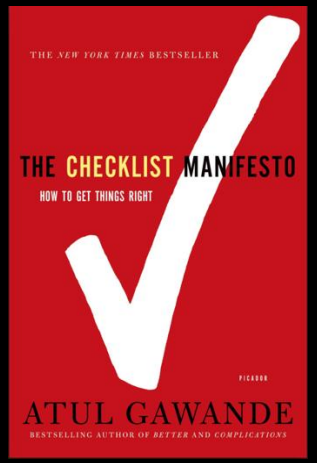
1. Multiple trials (≥30) with statistical tests (Mann-Whitney U) to account for randomness
2. Diverse seeds including the empty seed — report seed details
3. Timeouts of at least 24 hours with performance plotted over time
4. Measure actual bugs found (ground truth), not “unique crashes”
5. Test on a diverse benchmark suite, not just a few hand-picked programs

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



https://sigplan.org/Resources/EmpiricalEvaluation/



Influence: 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

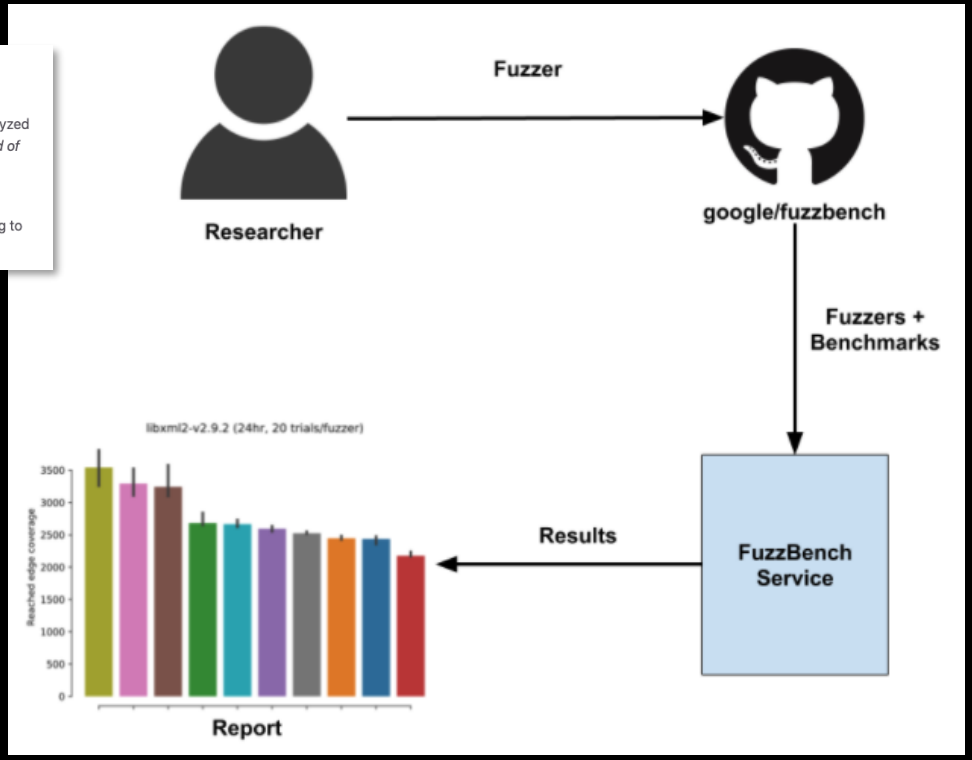
https://github.com/google/fuzzbench

Influence: 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.



https://github.com/google/fuzzbench

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

FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing

Zenong Zhang[†], *Zach Patterson*[†], *Michael Hicks*[‡], *and Shiyi Wei*[†]
[†] *University of Texas at Dallas* [‡] *University of Maryland and Amazon*[*]

Abstract

Fuzz testing is an active area of research with proposed improvements published at a rapid pace. Such proposals are assessed *empirically*: Can they be shown to perform better than the status quo? Such an assessment requires a benchmark of target programs with well-identified, realistic bugs. To ease the construction of such a benchmark, this paper presents FIXREVERTER, a tool that automatically injects realistic bugs in a program. FIXREVERTER takes as input a *bugfix pattern* which contains both code syntax and semantic conditions. Any code site that matches the specified syntax is *undone* if the semantic conditions are satisfied, as checked by static analysis, thus (re)introducing a likely bug. This paper focuses on three bugfix patterns, which we call *conditional-abort*, *conditional-execute*, and *conditional-assign*, based on a study of fixes in a corpus of Common Vulnerabilities and Exposures (CVEs). Using FIXREVERTER we have built REVBUGBENCH, which consists of 10 programs into which we have injected nearly 8,000 bugs; the programs are taken from FuzzBench and Binutils, and represent common targets of fuzzing evaluations. We have integrated REVBUGBENCH into the FuzzBench service, and used it to evaluate five fuzzers. Fuzzing performance varies by fuzzer and program, as desired/expected. Overall, 219 unique bugs were reported, 19% of which were detected by just one fuzzer.

by running it on a set of target programs, comparing its performance against that of one or more baseline fuzzers.

A key question is what performance measure to use. One popular measure, employed by Google's FuzzBench [3], is *code coverage*; if a fuzzer *A* (the improvement) is able to generate tests that execute more distinct lines/branches in a target program than the baseline *B*, then one could argue *A* will find more bugs. Multiple studies have been performed with the goal of understanding the relationship between code coverage and bug finding [4–6]. Unfortunately, a recent study finds that while there is a strong correlation between the coverage achieved and the number of bugs found by a fuzzer, there is not strong agreement on which fuzzer is superior if coverage is used to compare the fuzzers [7].

Another popular measure is to count the number of distinct, crash-inducing inputs generated, a.k.a. *unique crashes*. Since two different inputs can easily trigger the same bug, researchers often employ deduplication heuristics; two popular heuristics are AFL's "coverage profiles" and fuzzy stack hashes [8]. However, a study by Klees et al. [9] showed that both heuristics could still yield many false positives (many "deduplicated" inputs still trigger the same bug) and also some false negatives ("deduplicating" an input can actually remove evidence of a distinct bug). For one program, their study found that a result that appeared to show fuzzer *A* was superior to baseline *B* disappeared when ground truth was used, rather

Fuzzing from 2013 to today:

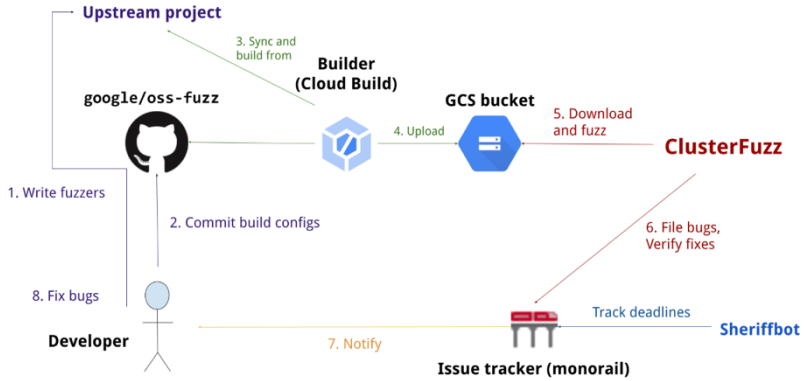
libFuzzer

```
struct Expr {
  char op;   // '+', '-', '*', '/'
  int32_t a; // left operand
  int32_t b; // right operand
};


// Evaluates the expression
int64_t EvalExpr(const Expr& e);
```

```
extern "C" int LLVMFuzzerTestOneInput(
  const uint8_t* data, size_t size) {

  // 1. build a typed Expr from raw fuzzer bytes.
  FuzzedDataProvider fdp(data, size);

  Expr e;
  e.op = fdp.PickValueInArray<char>({'+', '-', '*', '/'});
  e.a = fdp.ConsumeIntegral<int32_t>();
  e.b = fdp.ConsumeIntegral<int32_t>();

  // 2. CALL — invoke the function under test.
  int64_t result;
  try {
    result = EvalExpr(e);
  } catch (const std::invalid_argument&) {
    return 0; // unknown op — ignore gracefully
  }

  // 3. ASSERT — check always-true property
  if (e.op == '+') {
    assert(result - e.b == e.a);
  }

  return 0; // non-zero = treat input as uninteresting
}
```

OSS-Fuzz

Overview



Documentation

Read our detailed documentation to learn how to use OSS-Fuzz.

Trophies

As of May 2025, OSS-Fuzz has helped identify and fix over 13,000 vulnerabilities and 50,000 bugs across 1,000 projects.

OSS-Fuzz | Documentation

google.github.io/oss-fuzz/

OSS-Fuzz

Search OSS-Fuzz

OSS-Fuzz on GitHub

OSS-Fuzz

Fuzz testing is a well-known technique for uncovering programming errors in software. Many of these detectable errors, like buffer overflow, can have serious security implications. Google has found thousands of security vulnerabilities and stability bugs by deploying guided in-process fuzzing of Chrome components, and we now want to share that service with the open source community.

In cooperation with the Core Infrastructure Initiative and the OpenSSF, OSS-Fuzz aims to make common open source software more secure and stable by combining modern fuzzing techniques with scalable, distributed execution. Projects that do not qualify for OSS-Fuzz (e.g. closed source) can run their own instances of ClusterFuzz or ClusterFuzzLite.

We support the libFuzzer, AFL++, Honggfuzz, and Centipede fuzzing engines in combination with Sanitizers, as well as ClusterFuzz, a distributed fuzzer execution environment and reporting tool.

Currently, OSS-Fuzz supports C/C++, Rust, Go, Python and Java/JVM code. Other languages supported by LLVM may work too. OSS-Fuzz supports fuzzing x86_64 and i386 builds.

Project history

OSS-Fuzz was launched in 2016 in response to the Heartbleed vulnerability, discovered in OpenSSL, one of the most popular open source projects for encrypting web traffic. The vulnerability had the potential to affect almost every internet user, yet was caused by a relatively simple memory buffer overflow bug that could have been detected by fuzzing—that is, by running the code on randomized inputs to intentionally cause unexpected behaviors or crashes. At the time, though, fuzzing was not widely used and was cumbersome for developers, requiring extensive manual effort.

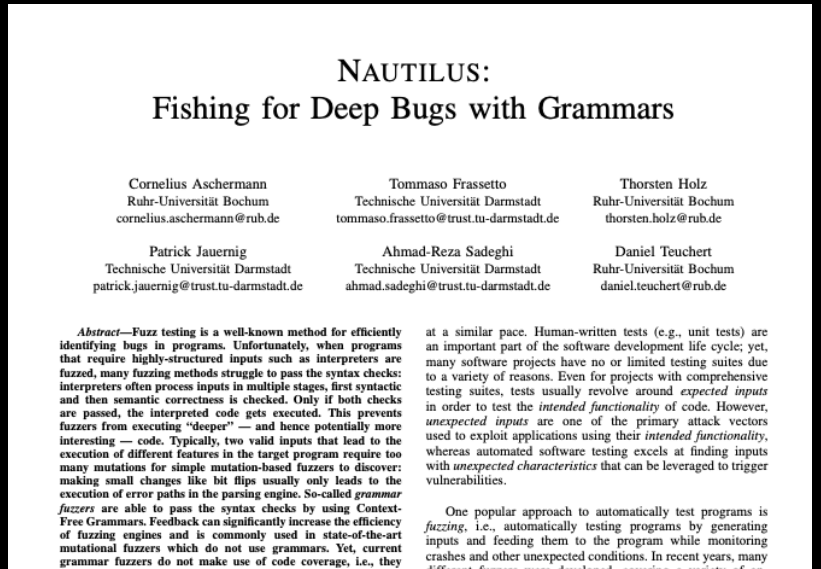
Google created OSS-Fuzz to fill this gap: it's a free service that runs fuzzers for open source

https://en.wikipedia.org/wiki/Buffer_overflow

Structure-Aware Fuzzing

Mutation-based fuzzing struggles with **highly structured inputs**

Grammar-aware mutators preserve structure while exploring content



NAUTILUS:
Fishing for Deep Bugs with Grammars

Cornelius Aschermann
Ruhr-Universität Bochum
cornelius.aschermann@rub.de

Tommaso Frassetto
Technische Universität Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de

Thorsten Holz
Ruhr-Universität Bochum
thorsten.holz@rub.de

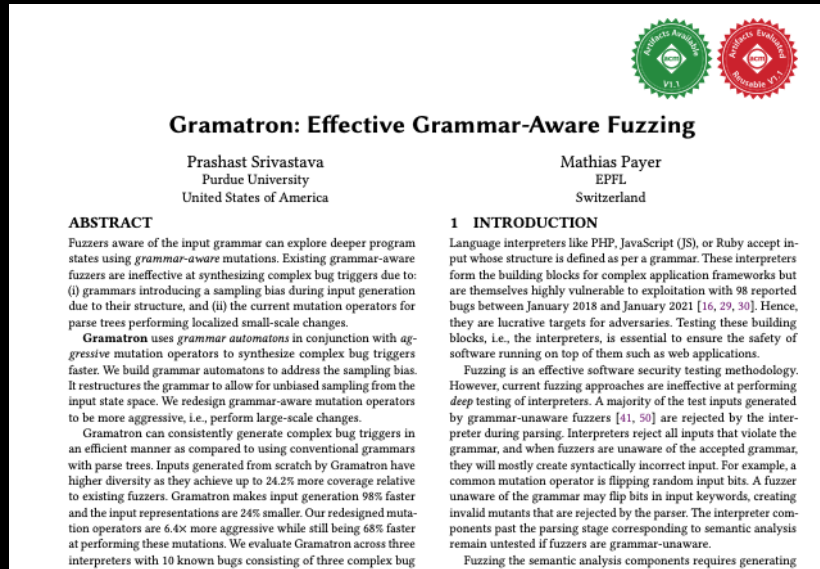
Patrick Jauernig
Technische Universität Darmstadt
patrick.jauernig@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

Daniel Teuchert
Ruhr-Universität Bochum
daniel.teuchert@rub.de

***Abstract*—Fuzz testing is a well-known method for efficiently identifying bugs in programs. Unfortunately, when programs that require highly-structured inputs such as interpreters are fuzzed, many fuzzing methods struggle to pass the syntax checks: interpreters often process inputs in multiple stages, first syntactic and then semantic correctness is checked. Only if both checks are passed, the interpreted code gets executed. This prevents fuzzers from executing "deeper" — and hence potentially more interesting — code. Typically, two valid inputs that lead to the execution of different features in the target program require too many mutations for simple mutation-based fuzzers to discover: making small changes like bit flips usually only leads to the execution of error paths in the parsing engine. So-called *grammar fuzzers* are able to pass the syntax checks by using Context-Free Grammars. Feedback can significantly increase the efficiency of fuzzing engines and is commonly used in state-of-the-art mutational fuzzers which do not use grammars. Yet, current grammar fuzzers do not make use of code coverage, i.e., they** at a similar pace. Human-written tests (e.g., unit tests) are an important part of the software development life cycle; yet, many software projects have no or limited testing suites due to a variety of reasons. Even for projects with comprehensive testing suites, tests usually revolve around *expected inputs* in order to test the *intended functionality* of code. However, *unexpected inputs* are one of the primary attack vectors used to exploit applications using their *intended functionality*, whereas automated software testing excels at finding inputs with *unexpected characteristics* that can be leveraged to trigger vulnerabilities.

One popular approach to automatically test programs is *fuzzing*, i.e., automatically testing programs by generating inputs and feeding them to the program while monitoring crashes and other unexpected conditions. In recent years, many



Gramatron: Effective Grammar-Aware Fuzzing

Prashast Srivastava
Purdue University
United States of America

Mathias Payer
EPFL
Switzerland

ABSTRACT

Fuzzers aware of the input grammar can explore deeper program states using *grammar-aware* mutations. Existing grammar-aware fuzzers are ineffective at synthesizing complex bug triggers due to: (i) grammars introducing a sampling bias during input generation due to their structure, and (ii) the current mutation operators for parse trees performing localized small-scale changes.

Gramatron uses *grammar automatons* in conjunction with *aggressive* mutation operators to synthesize complex bug triggers faster. We build grammar automatons to address the sampling bias. It restructures the grammar to allow for unbiased sampling from the input state space. We redesign grammar-aware mutation operators to be more aggressive, i.e., perform large-scale changes.

Gramatron can consistently generate complex bug triggers in an efficient manner as compared to using conventional grammars with parse trees. Inputs generated from scratch by Gramatron have higher diversity as they achieve up to 24.2% more coverage relative to existing fuzzers. Gramatron makes input generation 98% faster and the input representations are 24% smaller. Our redesigned mutation operators are 6.4× more aggressive while still being 68% faster at performing these mutations. We evaluate Gramatron across three interpreters with 10 known bugs consisting of three complex bug

1 INTRODUCTION

Language interpreters like PHP, JavaScript (JS), or Ruby accept input whose structure is defined as per a grammar. These interpreters form the building blocks for complex application frameworks but are themselves highly vulnerable to exploitation with 98 reported bugs between January 2018 and January 2021 [16, 29, 30]. Hence, they are lucrative targets for adversaries. Testing these building blocks, i.e., the interpreters, is essential to ensure the safety of software running on top of them such as web applications.

Fuzzing is an effective software security testing methodology. However, current fuzzing approaches are ineffective at performing *deep* testing of interpreters. A majority of the test inputs generated by grammar-unaware fuzzers [41, 50] are rejected by the interpreter during parsing. Interpreters reject all inputs that violate the grammar, and when fuzzers are unaware of the accepted grammar, they will mostly create syntactically incorrect input. For example, a common mutation operator is flipping random input bits. A fuzzer unaware of the grammar may flip bits in input keywords, creating invalid mutants that are rejected by the parser. The interpreter components past the parsing stage corresponding to semantic analysis remain untested if fuzzers are grammar-unaware.

Fuzzing the semantic analysis components requires generating

Hybrid Fuzzing

Fuzzing for breadth + **symbolic execution** for depth

Driller: Augmenting Fuzzing Through Selective Symbolic Execution

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang,
Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
{stephens,jmg,salls,dutcher,fish,jacopo,yans,chris,vigna}@cs.ucsb.edu

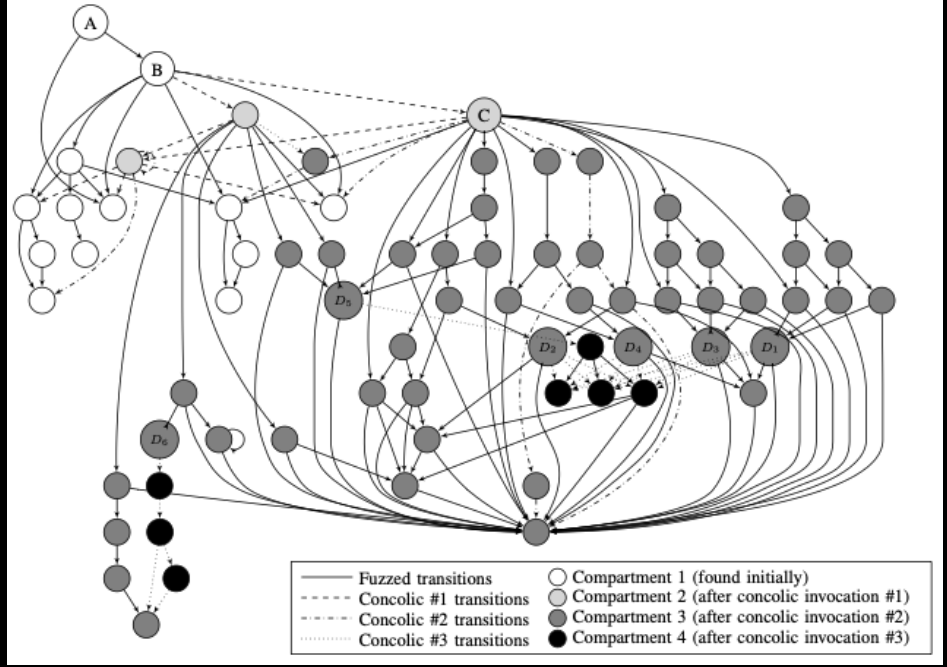
***Abstract*—Memory corruption vulnerabilities are an ever-present risk in software, which attackers can exploit to obtain unauthorized access to confidential information. As products with access to sensitive data are becoming more prevalent, the number of potentially exploitable systems is also increasing, resulting in a greater need for automated software vetting tools. DARPA recently funded a competition, with millions of dollars in prize money, to further research focusing on automated vulnerability finding and patching, showing the importance of research in this area. Current techniques for finding potential bugs include static, dynamic, and concolic analysis systems, which each having their own advantages and disadvantages. A common limitation of systems designed to create inputs which trigger vulnerabilities is that they only find shallow bugs and struggle to exercise deeper paths in executables.**

We present Driller, a hybrid vulnerability excavation tool which leverages fuzzing and selective concolic execution in a complementary manner, to find deeper bugs. Inexpensive fuzzing is used to exercise *compartments* of an application, while concolic execution is used to generate inputs which satisfy the complex checks separating the compartments. By combining the strengths of the two techniques, we mitigate their weaknesses, avoiding the path explosion inherent in concolic analysis and the incompleteness of fuzzing. Driller uses selective concolic execution to explore only the paths deemed interesting by the fuzzer and to generate inputs for conditions that the fuzzer cannot satisfy. We evaluate Driller on 126 applications released in the qualifying

Whereas such vulnerabilities used to be exploited by independent hackers who wanted to push the limits of security and expose ineffective protections, the modern world has moved to nation states and cybercriminals using such vulnerabilities for strategic advantage or profit. Furthermore, with the rise of the *Internet of Things*, the number of devices that run potentially vulnerable software has skyrocketed, and vulnerabilities are increasingly being discovered in the software running these devices [29].

While many vulnerabilities are discovered by hand, manual analysis is not a scalable method for vulnerability assessment. To keep up with the amount of software that must be vetted for vulnerabilities, an automated approach is required. In fact, DARPA has recently lent its support to this goal by sponsoring two efforts: VET, a program on developing techniques for the analysis of binary firmware, and the Cyber Grand Challenge (CGC), in which participants design and deploy automated vulnerability scanning engines that will compete against each other by exploiting binary software. DARPA has funded both VET and the Cyber Grand Challenge with millions of dollars in research funding and prize money, demonstrating the strong interest in developing a viable approach to automated binary analysis.

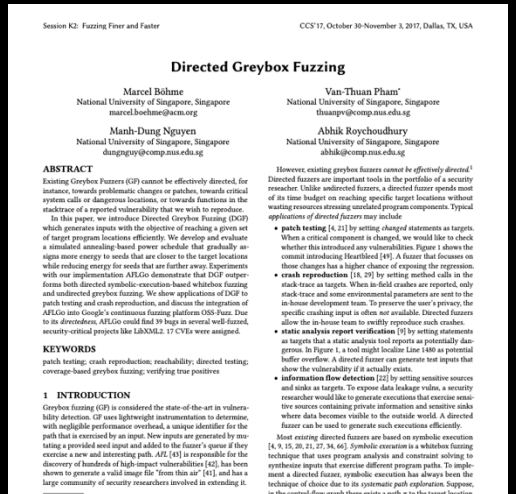
Naturally, security researchers have been actively designing



Directed & Differential Fuzzing

Directed: guide fuzzing toward **specific code locations** — AFLGo, Hawkeye — e.g., fuzz recently patched code

Differential: compare **multiple implementations** of the same spec — Find semantic bugs, not just crashes



Session K2: Fuzzing Finer and Faster CCS'17, October 30-November 3, 2017, Dallas, TX, USA

Directed Greybox Fuzzing

Marcel Böhme
National University of Singapore, Singapore
marcel.boehme@acm.org

Van-Thuan Pham*
National University of Singapore, Singapore
thuanpv@comp.nus.edu.sg

Manh-Dung Nguyen
National University of Singapore, Singapore
dungnguy@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Existing Greybox Fuzzers (GF) cannot be effectively directed, for instance, towards problematic changes or patches, towards critical system calls or dangerous locations, or towards functions in the stacktrace of a reported vulnerability that we wish to reproduce.

In this paper, we introduce Directed Greybox Fuzzing (DGF) which generates inputs with the objective of reaching a given set of target program locations efficiently. We develop and evaluate a simulated annealing-based power schedule that gradually assigns more energy to seeds that are closer to the target locations while reducing energy for seeds that are further away. Experiments with our implementation AFLGo demonstrate that DGF outperforms both directed symbolic-execution-based whitebox fuzzing and undirected greybox fuzzing. We show applications of DGF to patch testing and crash reproduction, and discuss the integration of AFLGo into Google's continuous fuzzing platform OSS-Fuzz. Due to its *directedness*, AFLGo could find 39 bugs in several well-fuzzed, security-critical projects like LibXML2. 17 CVEs were assigned.

KEYWORDS

patch testing; crash reproduction; reachability; directed testing; coverage-based greybox fuzzing; verifying true positives

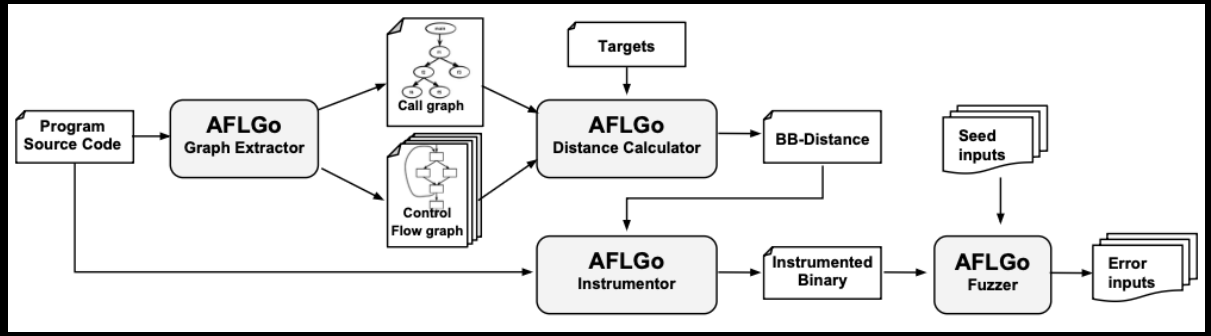
1 INTRODUCTION

Greybox fuzzing (GF) is considered the state-of-the-art in vulnerability detection. GF uses lightweight instrumentation to determine, with negligible performance overhead, a unique identifier for the path that is exercised by an input. New inputs are generated by mutating a provided seed input and added to the fuzzer's queue if they exercise a new and interesting path. AFL [43] is responsible for the discovery of hundreds of high-impact vulnerabilities [42], has been shown to generate a valid image file "from thin air" [41], and has a large community of security researchers involved in extending it.

However, existing greybox fuzzers *cannot be effectively directed*.[1] Directed fuzzers are important tools in the portfolio of a security reseacher. Unlike undirected fuzzers, a directed fuzzer spends most of its time budget on reaching specific target locations without wasting resources stressing unrelated program components. Typical *applications of directed fuzzers* may include

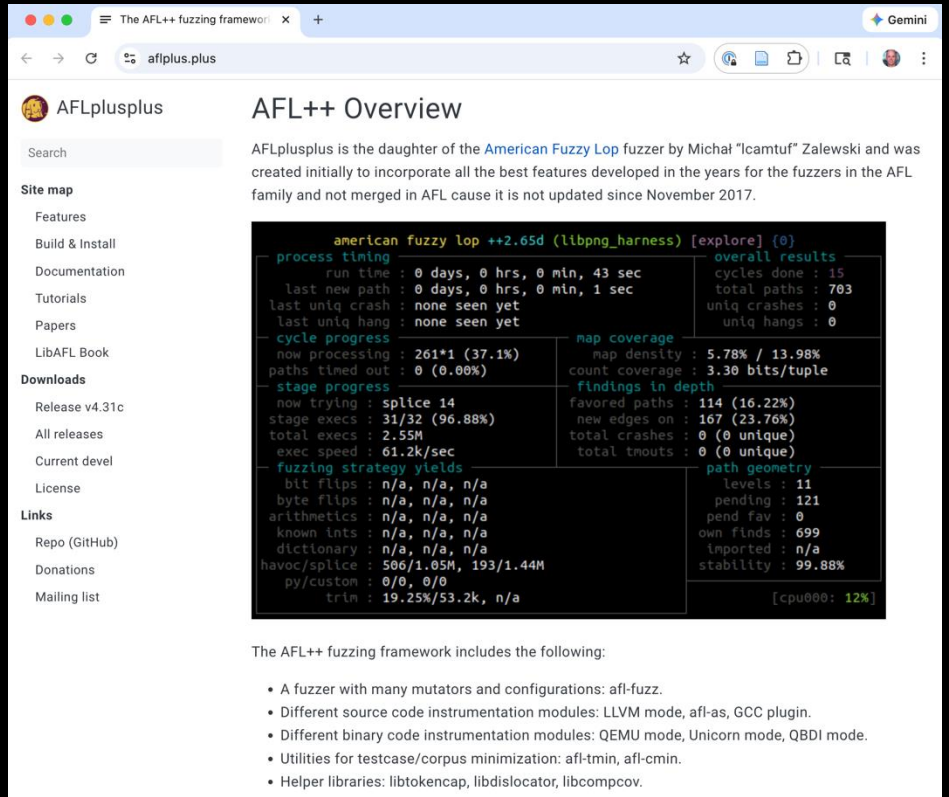
- **patch testing** [4, 21] by setting *changed* statements as targets. When a critical component is changed, we would like to check whether this introduced any vulnerabilities. Figure 1 shows the commit introducing Heartbleed [49]. A fuzzer that focusses on those changes has a higher chance of exposing the regression.
- **crash reproduction** [18, 29] by setting method calls in the stack-trace as targets. When in-field crashes are reported, only stack-trace and some environmental parameters are sent to the in-house development team. To preserve the user's privacy, the specific crashing input is often *not* available. Directed fuzzers allow the in-house team to swiftly reproduce such crashes.
- **static analysis report verification** [9] by setting statements as targets that a static analysis tool reports as potentially dangerous. In Figure 1, a tool might localize Line 1480 as potential buffer overflow. A directed fuzzer can generate test inputs that show the vulnerability if it actually exists.
- **information flow detection** [22] by setting sensitive sources and sinks as targets. To expose data leakage vulns, a security researcher would like to generate executions that exercise sensitive sources containing private information and sensitive sinks where data becomes visible to the outside world. A directed fuzzer can be used to generate such executions efficiently.

Most *existing* directed fuzzers are based on symbolic execution [4, 9, 15, 20, 21, 27, 34, 46]. *Symbolic execution* is a whitebox fuzzing technique that uses program analysis and constraint solving to synthesize inputs that exercise different program paths. To implement a directed fuzzer, symbolic execution has always been the technique of choice due to its *systematic path exploration*. Suppose,



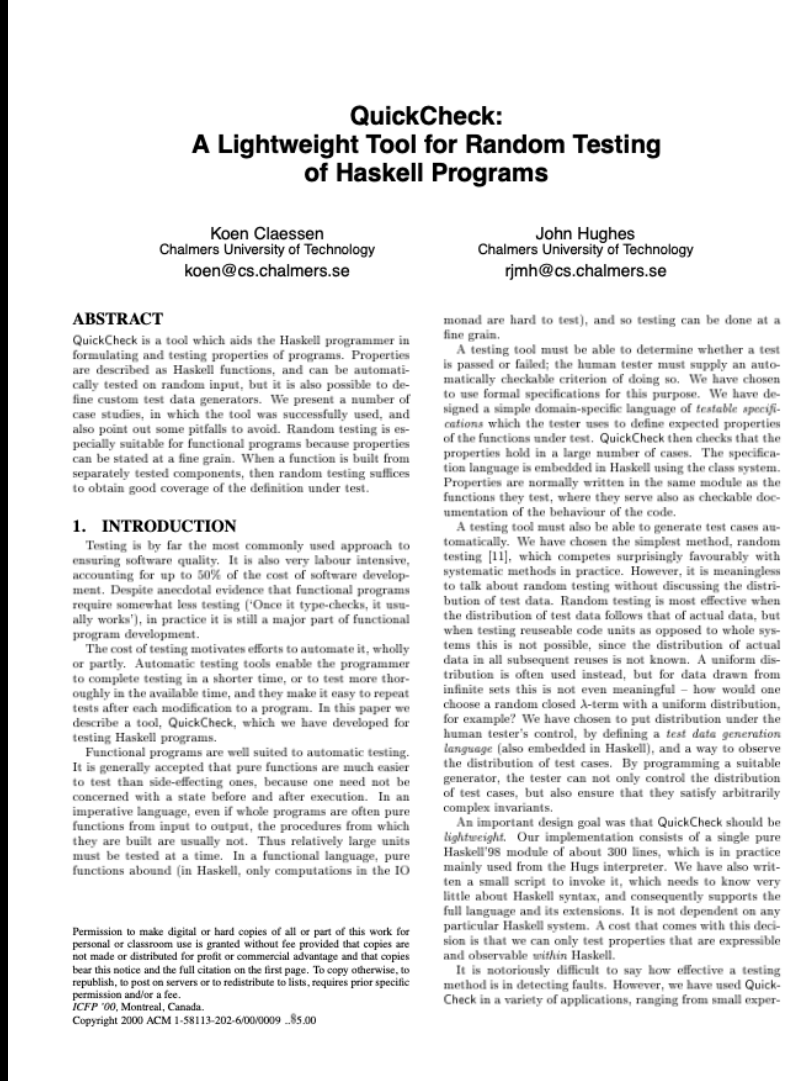
The Modern Ecosystem: AFL++

- Community fork of AFL
  - **better mutators, CMPLOG, persistent mode**
  - custom mutator API — plug in domain knowledge
- Regularly performs the best on FuzzBench baseline evaluations



Properties to Test: Beyond Crashes

- Sanitizers
  - AddressSanitizer (buffer overflows, use-after-free)
  - MemorySanitizer (uninitialized reads)
  - UndefinedBehaviorSanitizer
  - …
- Property-based testing
  - QuickCheck
  - But also: See what we were doing with libFuzzer



QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

ABSTRACT

QuickCheck is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random input, but it is also possible to define custom test data generators. We present a number of case studies, in which the tool was successfully used, and also point out some pitfalls to avoid. Random testing is especially suitable for functional programs because properties can be stated at a fine grain. When a function is built from separately tested components, then random testing suffices to obtain good coverage of the definition under test.

1. INTRODUCTION

Testing is by far the most commonly used approach to ensuring software quality. It is also very labour intensive, accounting for up to 50% of the cost of software development. Despite anecdotal evidence that functional programs require somewhat less testing ('Once it type-checks, it usually works'), in practice it is still a major part of functional program development.

The cost of testing motivates efforts to automate it, wholly or partly. Automatic testing tools enable the programmer to complete testing in a shorter time, or to test more thoroughly in the available time, and they make it easy to repeat tests after each modification to a program. In this paper we describe a tool, QuickCheck, which we have developed for testing Haskell programs.

Functional programs are well suited to automatic testing. It is generally accepted that pure functions are much easier to test than side-effecting ones, because one need not be concerned with a state before and after execution. In an imperative language, even if whole programs are often pure functions from input to output, the procedures from which they are built are usually not. Thus relatively large units must be tested at a time. In a functional language, pure functions abound (in Haskell, only computations in the IO monad are hard to test), and so testing can be done at a fine grain.

A testing tool must be able to determine whether a test is passed or failed; the human tester must supply an automatically checkable criterion of doing so. We have chosen to use formal specifications for this purpose. We have designed a simple domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test. QuickCheck then checks that the properties hold in a large number of cases. The specification language is embedded in Haskell using the class system. Properties are normally written in the same module as the functions they test, where they serve also as checkable documentation of the behaviour of the code.

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random testing [11], which competes surprisingly favourably with systematic methods in practice. However, it is meaningless to talk about random testing without discussing the distribution of test data. Random testing is most effective when the distribution of test data follows that of actual data, but when testing reuseable code units as opposed to whole systems this is not possible, since the distribution of actual data in all subsequent reuses is not known. A uniform distribution is often used instead, but for data drawn from infinite sets this is not even meaningful – how would one choose a random closed λ-term with a uniform distribution, for example? We have chosen to put distribution under the human tester's control, by defining a *test data generation language* (also embedded in Haskell), and a way to observe the distribution of test cases. By programming a suitable generator, the tester can not only control the distribution of test cases, but also ensure that they satisfy arbitrarily complex invariants.

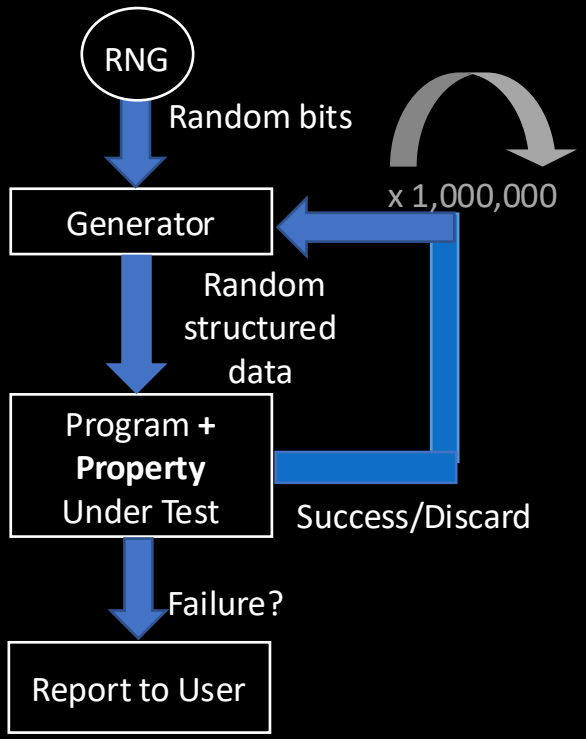
An important design goal was that QuickCheck should be *lightweight*. Our implementation consists of a single pure Haskell'98 module of about 300 lines, which is in practice mainly used from the Hugs interpreter. We have also written a small script to invoke it, which needs to know very little about Haskell syntax, and consequently supports the full language and its extensions. It is not dependent on any particular Haskell system. A cost that comes with this decision is that we can only test properties that are expressible and observable *within* Haskell.

It is notoriously difficult to say how effective a testing method is in detecting faults. However, we have used QuickCheck in a variety of applications, ranging from small exper-

ICFP '00, Montreal, Canada.

Property-based testing (PBT)

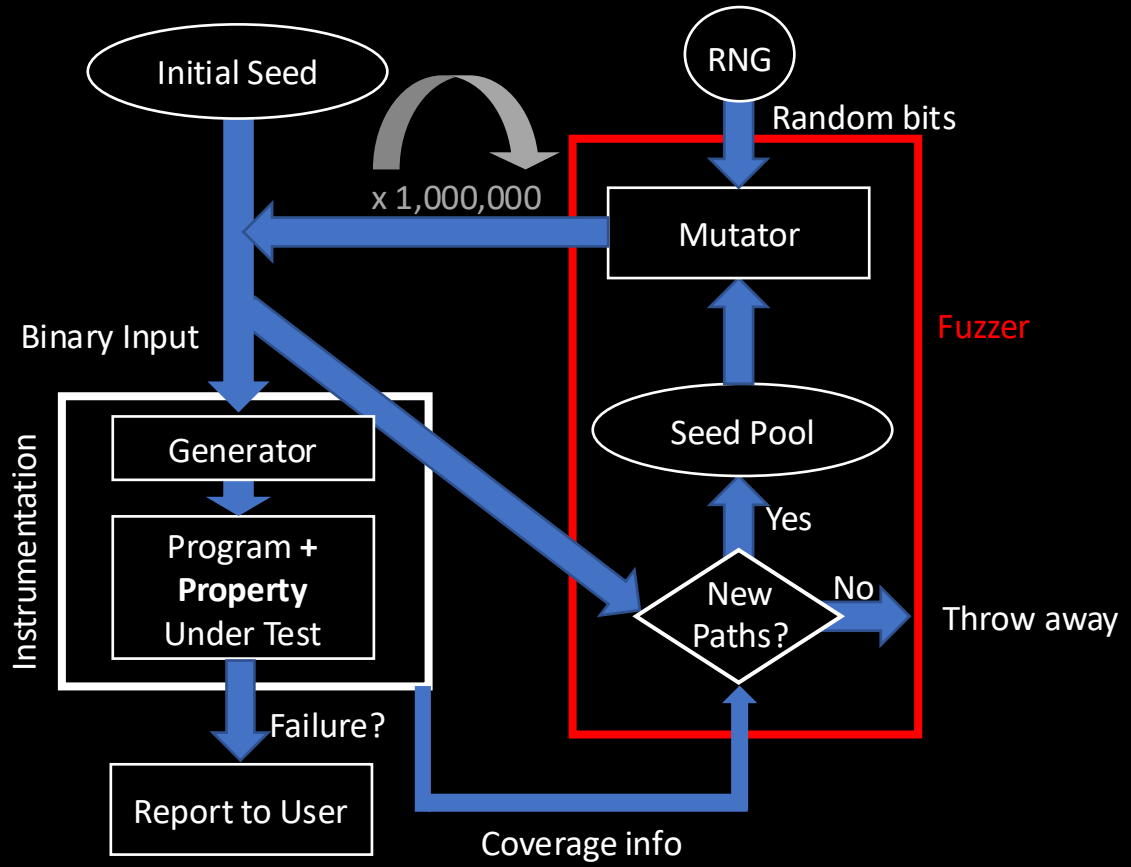
Generate random *structures*; check *general* properties



How to combine PBT with fuzzing?

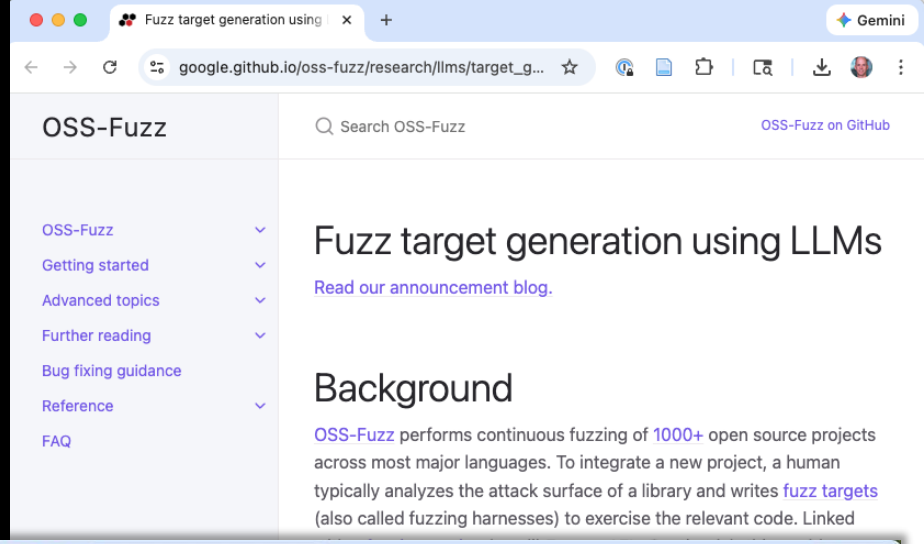
Parameterized coverage-guided fuzzing (PCGF)

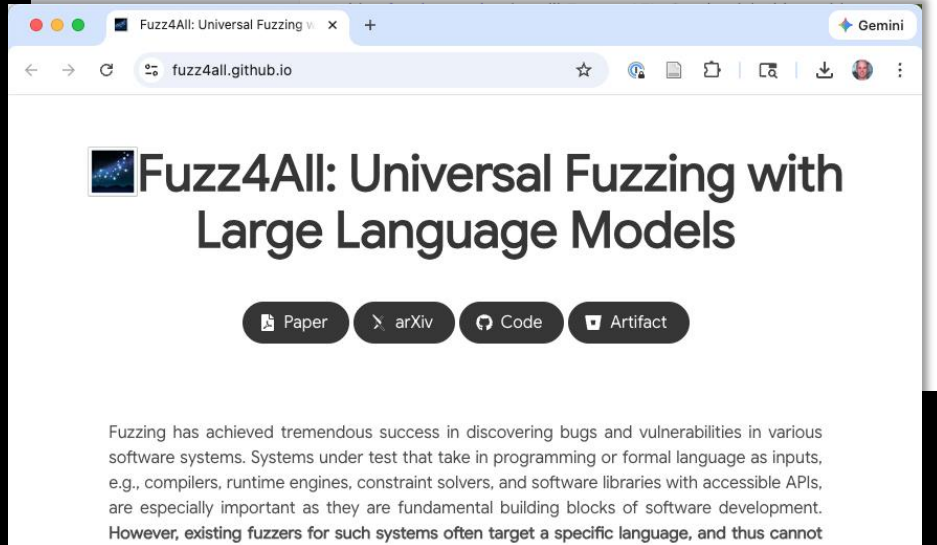
Generate structures from bits, which are mutated by the fuzzer



LLM-Assisted Fuzzing

- Write **fuzzing harnesses** automatically
- Generate **seed inputs** that understand the format
- Produce **targeted mutations** informed by code understanding





Key Takeaways

1. **Coverage feedback** is the key ingredient — transforms random testing into guided exploration
2. **Scheduling matters** — how you allocate effort across seeds can yield orders-of-magnitude improvement
3. **Evaluation is hard** — randomness, seeds, timeouts, and broken metrics can mislead
4. Fuzzing is now **infrastructure** — a baseline expectation, not a research curiosity