Testing - Input Generation Techniques

Advanced Software Engineering
Spring 2023

Agenda

Adminsitrivia: Reminder, project proposal

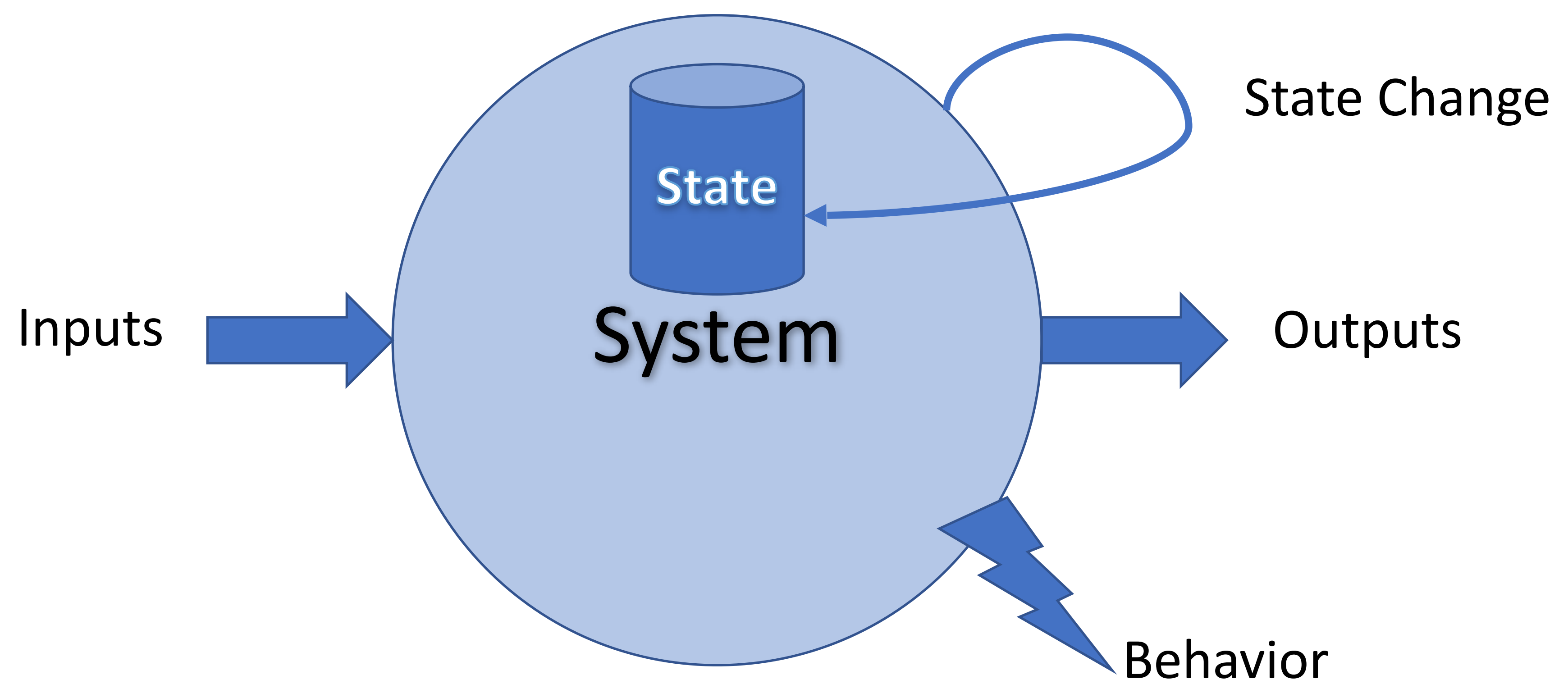
Review testing so-far

Poll

Lecture/discussion: test input generation

Review: Tests as Inputs + Oracles

- Inputs:
  - Arguments on command line
  - Files
  - Network
  - User
  - Randomness



Review: Test Oracles and Pseudo-Oracles

What is "correct" behavior?

Test oracles

- Example-based: "For a given input, some assertions should be true"
- Properties: "All inputs in some class should satisfy some property"
  - "It doesn't crash"
  - "Changing the input in some way should maintain the same output"
- Regression: "It provides the same output as it used to"
- Differential: "Two systems implementing the same spec should provide the same output"
- Human oracle: "For a given user, they should be satisfied"

Review: Mutants as a Valid Substitute for Real Faults

Mutation Analysis Tests the Tests

Idea: What if many (real) bugs could be represented by a single, one-line “mutation” to the program?

```
public contains(location: PlayerLocation): boolean {
  return (
    location.x + PLAYER_SPRITE_WIDTH / 2 > this._x &&
    location.x - PLAYER_SPRITE_WIDTH / 2 < this._x + this._width &&
    location.y + PLAYER_SPRITE_HEIGHT / 2 > this._y &&
    location.y - PLAYER_SPRITE_HEIGHT / 2 < this._y + this._height
  );
}
```

Correct code for “Contains” check in Covey.Town

```
public contains(location: PlayerLocation): boolean {
  return (
    location.x + PLAYER_SPRITE_WIDTH / 2 < this._x &&
    location.x - PLAYER_SPRITE_WIDTH / 2 < this._x + this._width &&
    location.y + PLAYER_SPRITE_HEIGHT / 2 > this._y &&
    location.y - PLAYER_SPRITE_HEIGHT / 2 < this._y + this._height
  );
}
```

Mutated (and buggy) code for “Contains” check in Covey.Town

Review: Assertions help detect bugs

Likely assertions might help developers add assertions to code

```
15.1.1:::BEGIN  100 samples
  N = size(B)                        (7 values)
  N in [7..13]                       (7 values)
  B                                  (100 values)
    All elements >= -100             (200 values)

15.1.1:::END    100 samples
  N = I = N_orig = size(B)           (7 values)
  B = B_orig                         (100 values)
  S = sum(B)                         (96 values)
  N in [7..13]                       (7 values)
  B                                  (100 values)
    All elements >= -100             (200 values)

15.1.1:::LOOP   1107 samples
  N = size(B)                        (7 values)
  S = sum(B[0..I-1])                 (96 values)
  N in [7..13]                       (7 values)
  B                                  (100 values)
    All elements in [-100..100]      (200 values)
  I in [0..13]                       (14 values)
  sum(B) in [-556..539]              (96 values)
  B[0] nonzero in [-99..96]          (79 values)
  B[-1] in [-88..99]                 (80 values)
  B[0..I-1]                          (985 values)
    All elements in [-100..100]      (200 values)
  I <= N                             (77 values)
 Negative invariants:
  N != B[-1]                         (99 values)
  B[0] != B[-1]                      (100 values)
```

Figure 2: Invariants inferred for Gries program 15.1.1 over 100 randomly generated input arrays. Invariants are shown for the beginning (precondition) and end (postcondition) of the program, as well as the loop head (the loop invariant). `B[-1]` is shorthand for `B[size(B)-1]`, the last element of array `B`, and *var*`_orig` represents *var*'s value at the start of execution. Invariants for elements of an array are listed indented under the array; in this example, no array has multiple elementwise invariants.

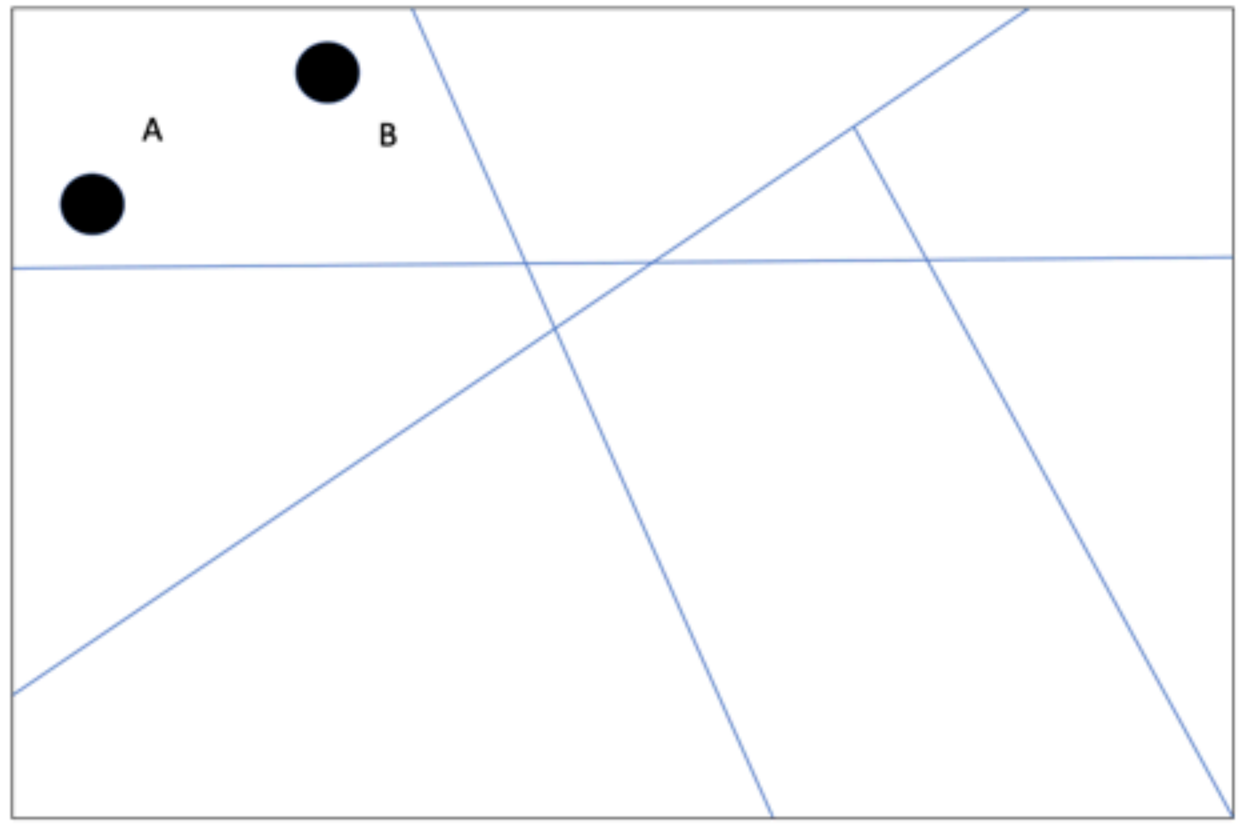
"Dynamically discovering likely program invariants to support program evolution" Ernst et al, ICSE 1999
https://doi.org/10.1145/302405.302467

Review: Use Equivalence Classes to Generate Inputs

What is a “good” test suite?

Interpretation: Coverage of Input Space

- (Manually) enumerate possible “equivalence classes” of inputs
- Ensure that each equivalence class is covered by a test
- Pay extra attention to boundary cases



If the program works for input A, it will probably work for input B

2-14 Test input generation

0 done

0 underway

Did you think it was "surprising" that random testing found bugs in unix™ utilities?

Yes

No

Total Results: 0

Do you think that the fuzzing paper was really inspired by "a dark and stormy night" with a dialup connection?

Join by Web

1. Go to **PollEv.com**
2. Enter **JBELL**
3. Respond to activity

Instructions not active. **Log in** to activate

Total Results: 0

Have you used any of these dynamic analysis tools for C before?

- Valgrind
- Address sanitizer, Memory Sanitizer
- Thread Sanitizer
- UndefinedBehaviorSanitizer
- I try to avoid programming in C, so I definitely haven't used these tools for C

Total Results: 0

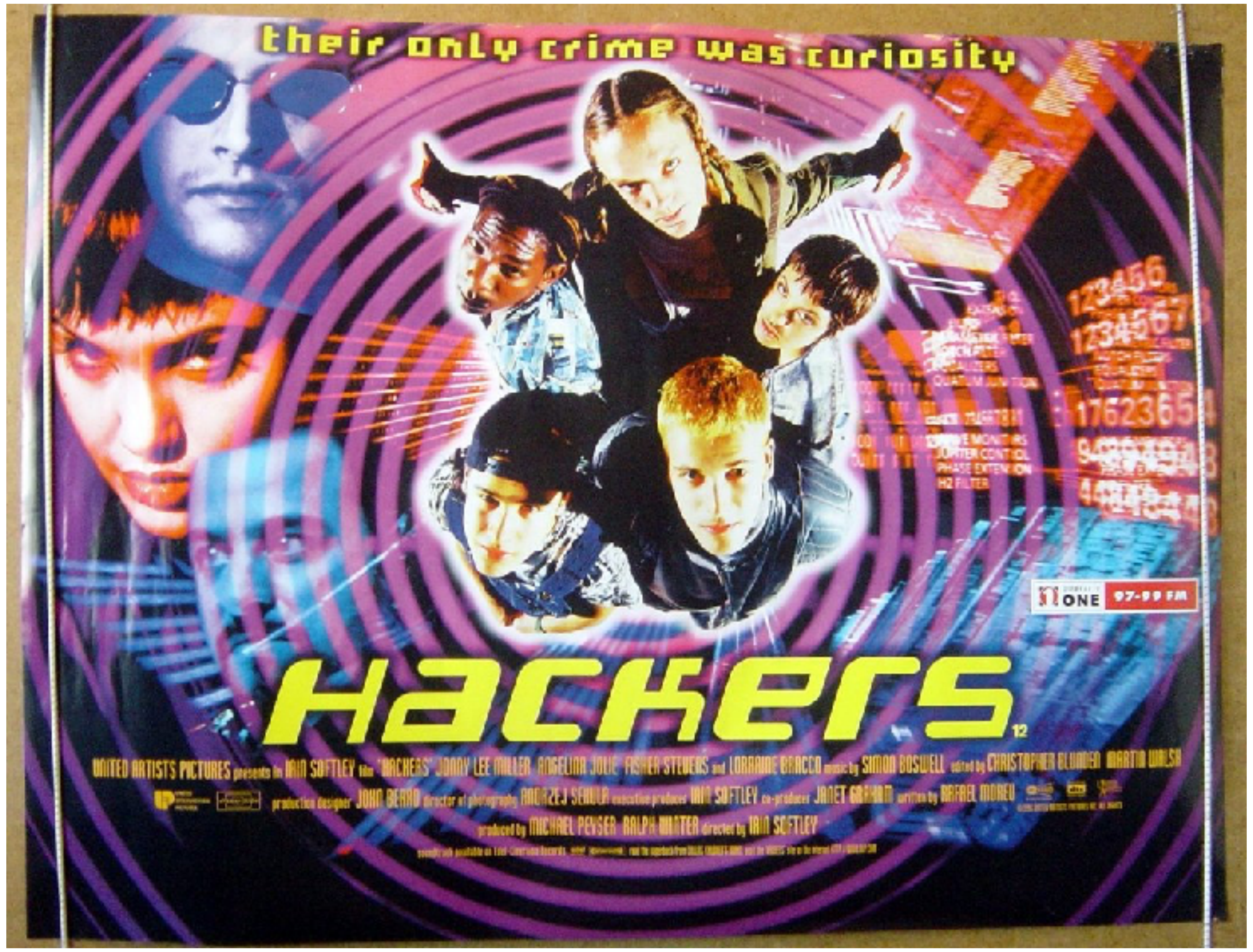
Spot the bug

```
char inputPassword[BUFSIZE];
char realPassword[17];
strncpy(realPassword, "mySecretPassword", 17);
gets(inputPassword);
```

Sidebar: 1990's Cultural Reference [2,3]

Computer Fraud and Abuse Act of 1986

Robert Morris
MIT Professor
Y Combinator Co-Founder



1995 Movie with Jonny Lee Miller and Angelina Jolie

What testing strategy will find this bug?

Assume: we have a perfect oracle for detecting buffer overflows when they occur

```c
char inputPassword[BUFSIZE];
char realPassword[17];
strncpy(realPassword, "mySecretPassword", 17);
gets(inputPassword);
```

It was a dark and stormy night

"Fuzz testing"

- Generate a continuous string of random (?) characters
  - Printable only
  - Printable + control characters
  - Also with/without null byte characters
- Options to specify: Length of input, random seed
- Inputs are files, or for interactive applications, Ptyjig
- Oracle - program crash or hang

Sidebar: Unix™

Reminder - OSS discussion

UNIX, BSD and GNU

Slide Subtitle

```
Security policy loaded: Apple Mobile File Integrity (AMFI)
calling mpo_policy_init for Sandbox
Security policy loaded: Seatbelt sandbox policy (Sandbox)
calling mpo_policy_init for Quarantine
Security policy loaded: Quarantine policy (Quarantine)
Copyright (c) 1982, 1986, 1989, 1991, 1993
        The Regents of the University of California. All rights reserved.

MAC Framework successfully initialized
using 16384 buffer headers and 10240 cluster IO buffer headers
AppleKeyStore starting (BUILT: Sep 19 2014 00:11:30)
```

BSD Copyright in OS X boot sequence

- 1978: UC Berkeley begins distributing their own derived version of Unix (BSD)
- 1983: AT&T broken up by DOJ, UNIX licensing changed: no more source releases
- Also 1983: “Starting this Thanksgiving I am going to write a complete Unix-compatible software system called GNU (Gnu’s Not Unix), and give it away free to everyone who can use it”

GNU logo (a gnu wildebeest)

Evaluating Random Testing

Generating random inputs is “surprisingly effective” at finding bugs in Unix™

- 88 utility programs, 6 operating systems, 109 crashes/hangs in total
- Why are there so many buggy programs in Unix?
- Do the “comments on the results” apply similarly today, and how?

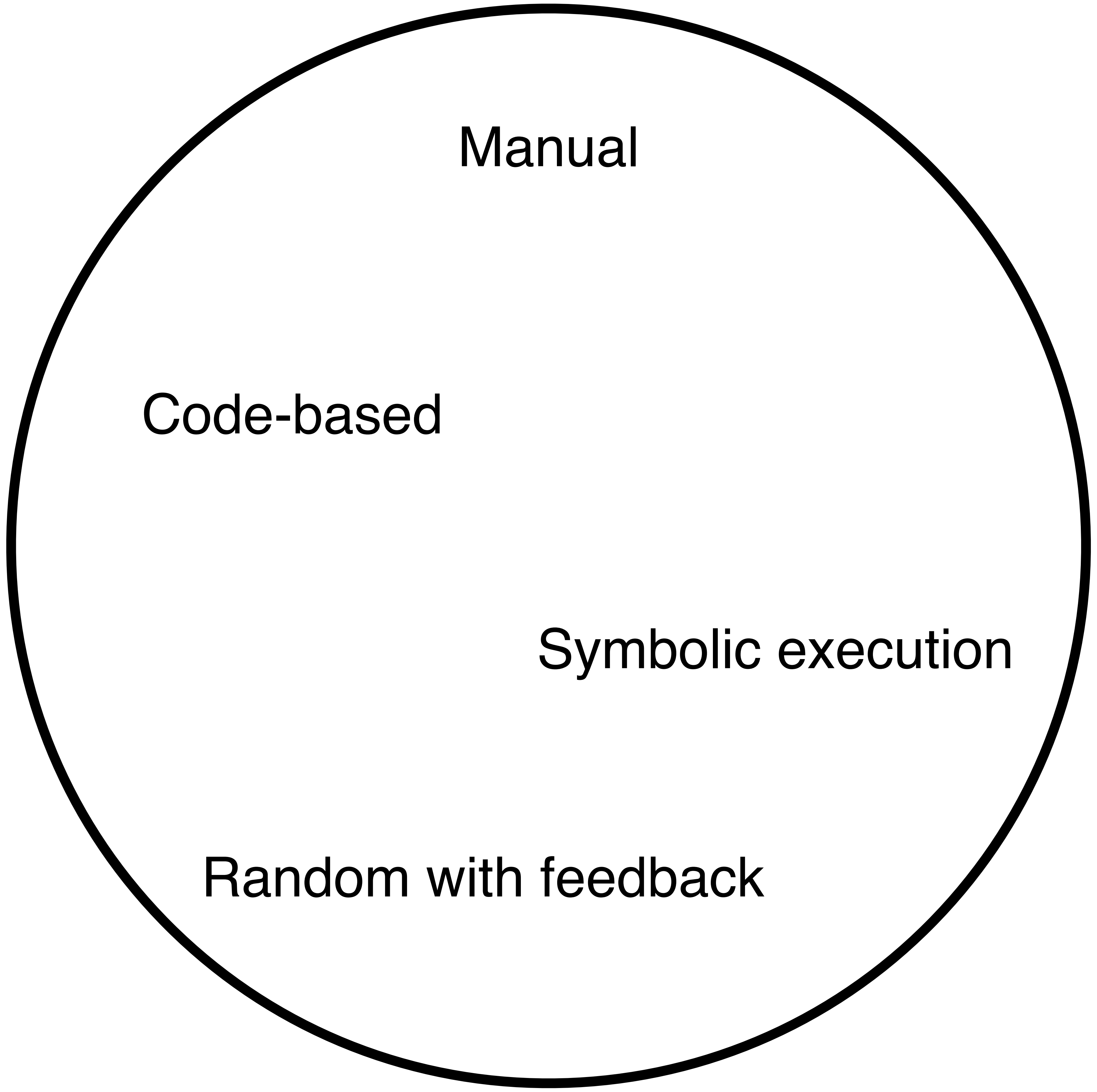
Beyond Random Testing

How do we generate an input1 that reveals the crash?

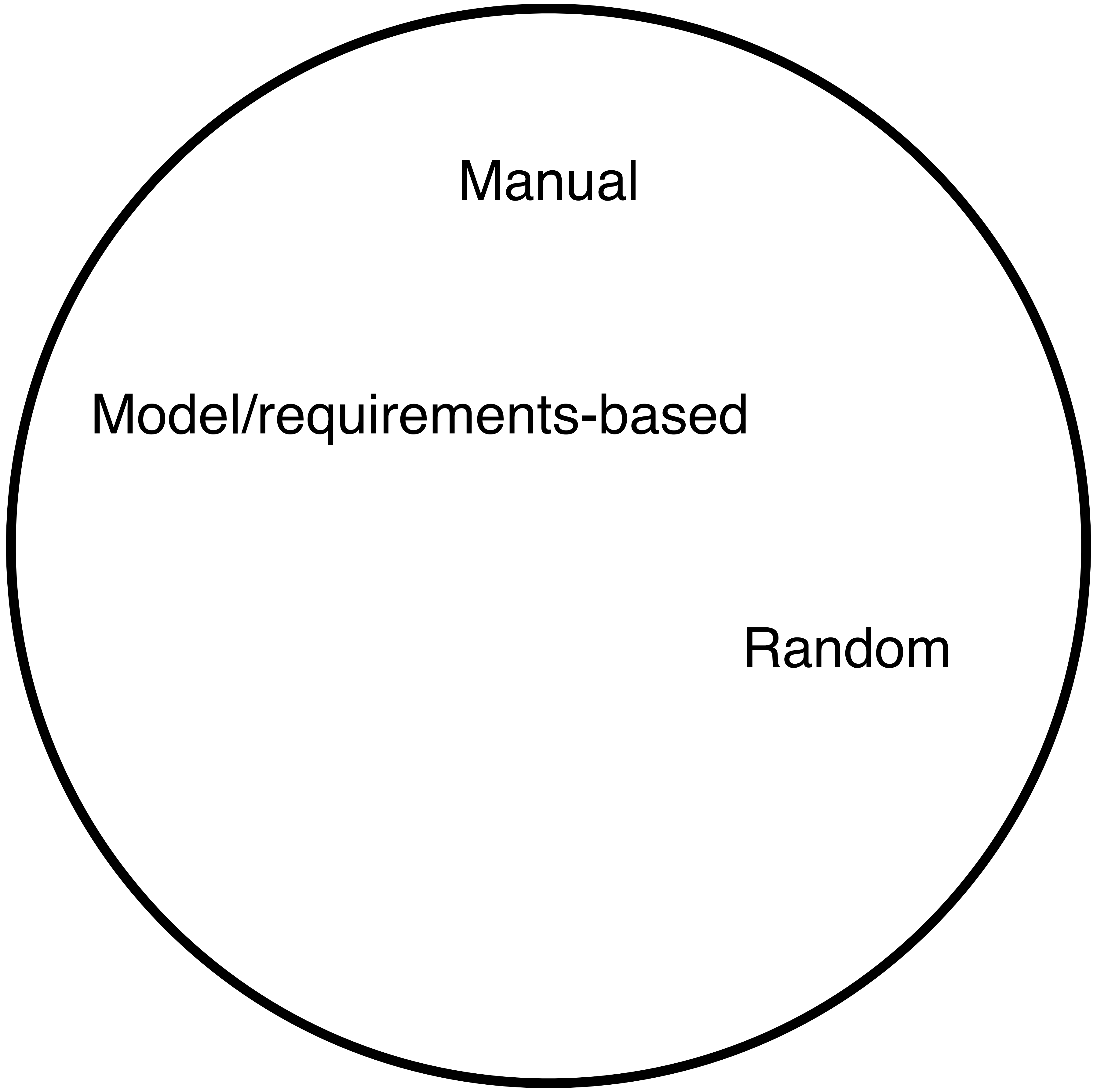
```
void magic(byte input1){
    if(input1 == 45){
        crash();
    }
}

void magic2(byte input1, byte input2){
    if(input1 == 45){
        if(input2 == 36){
            crash();
        }
    }
}
```

Test Input Generation Strategies



“White box” (We look at the code)



“Black box” (We do not look at the code)

Beyond Random Testing

- Symbolic execution: for each input, represent it as a symbolic value (instead of concrete number), then detect constraints on inputs, create and solve logical formulas to get inputs

- Random fuzzing, but with some hints: “The numbers 45 and 36 seem lucky”

- Random fuzzing, but with guidance: “Using 45 as input1 seems interesting”

```
void magic2(byte input1, byte input2){
    if(input1 == 45){
       if(input2 == 36){
           crash();
        }
    }
}
```

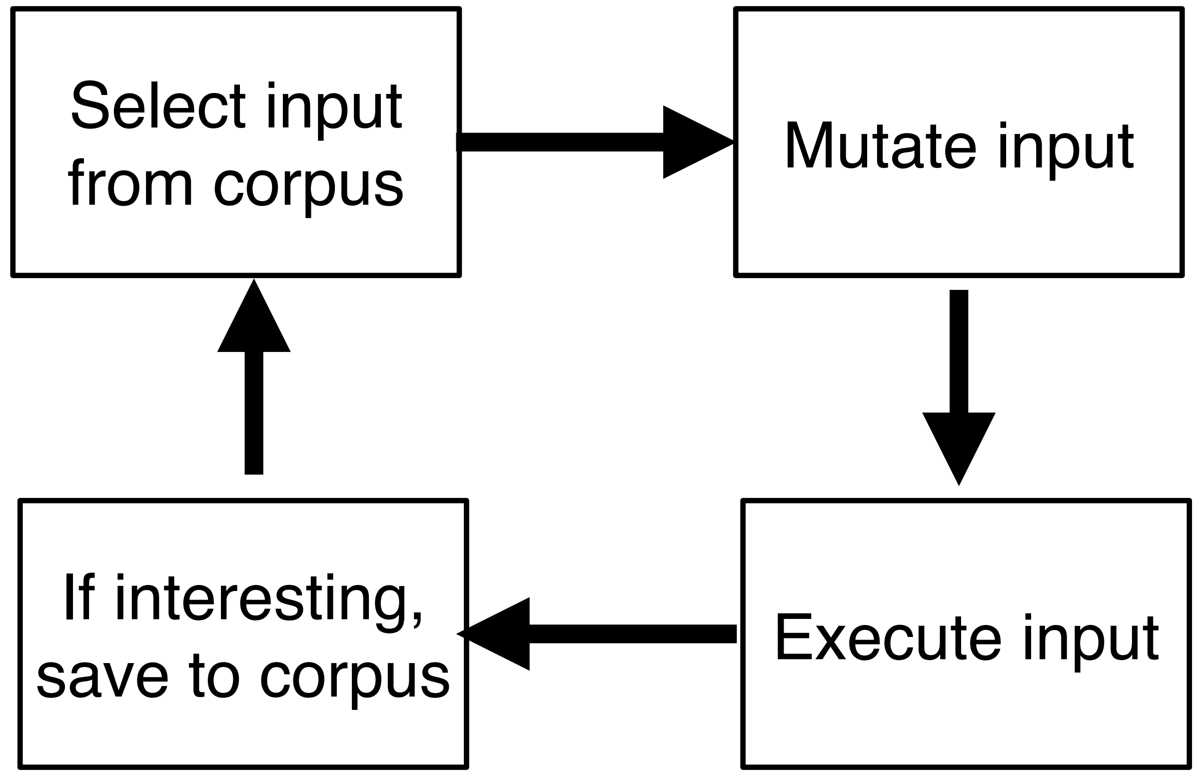
Feedback-Guided Fuzzing

```
void magic2(byte input1, byte input2){
    if(input1 == 45){    //B1
       if(input2 == 36){ //B2
            crash();
       }
    }
}
```

input1	input2	B1	B2
0	0	F	
10	68	F	
45	**0**	**T**	**F**
14	0	F	
45	100	T	F
45	**36**	**T**	**T**

Feedback-Guided Fuzzing

Overview




```
void magic2(byte input1, byte input2){
    if(input1 == 45){     //B1
       if(input2 == 36){ //B2
            crash();
        }
    }
}
```

input1	input2	B1	B2
0	0	F	
10	68	F	
45	**0**	**T**	**F**
14	0	F	
45	100	T	F
45	**36**	**T**	**T**

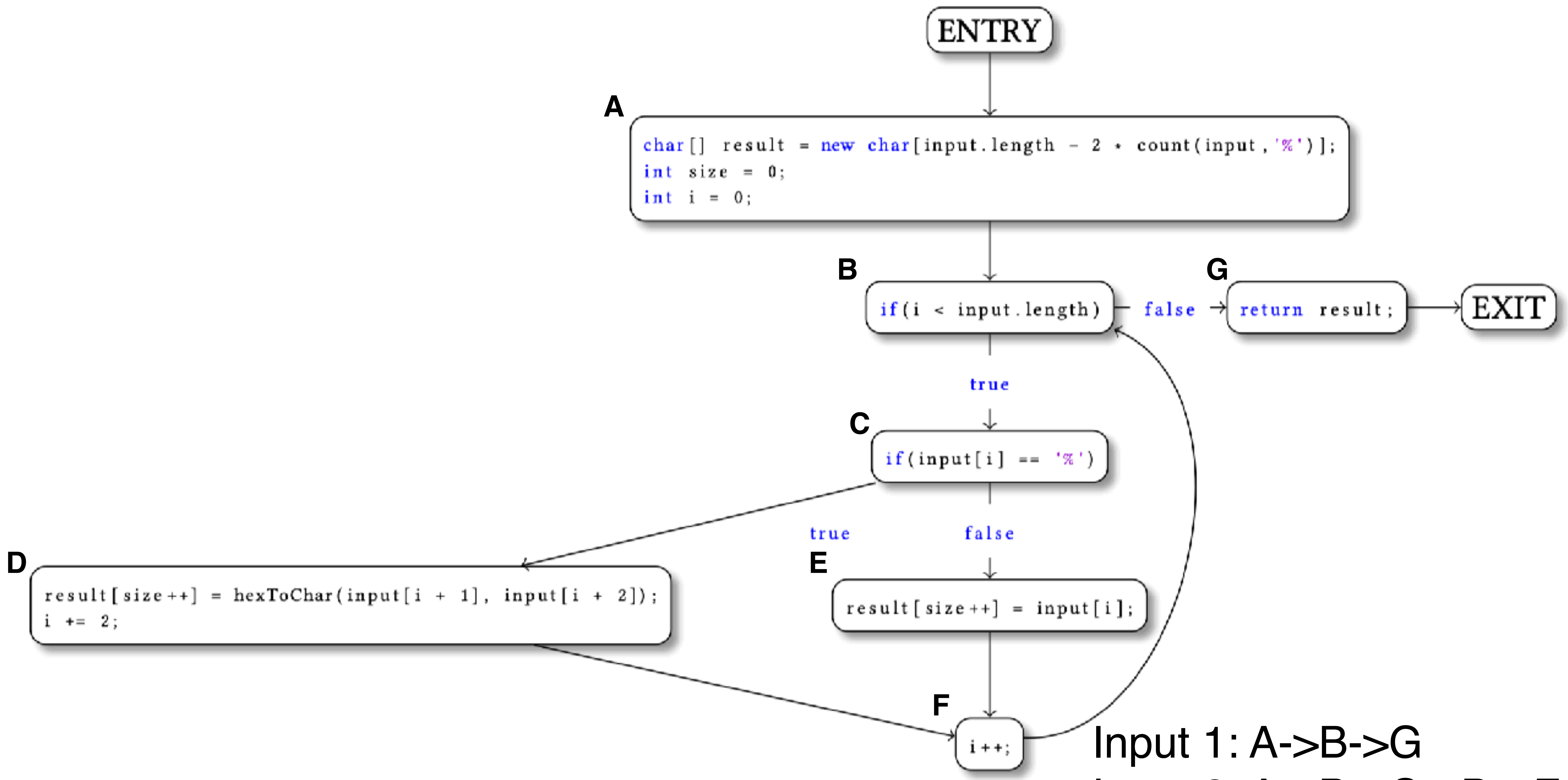
Feedback-Guided Fuzzing

Design goals: AFL

- Speed - fuzz at native speed
  - Rationale: worst-case should never be worse than brute force
- Reliability - avoid complex instrumentation
  - Rationale: Instrumentation is brittle
- Simplicity - limit number of knobs provided to users
- Chainability - make it easy to interact with fuzzed applications

AFL Tracks Edge Coverage

“Interesting” inputs reveal new edges, or new coarse hit counts



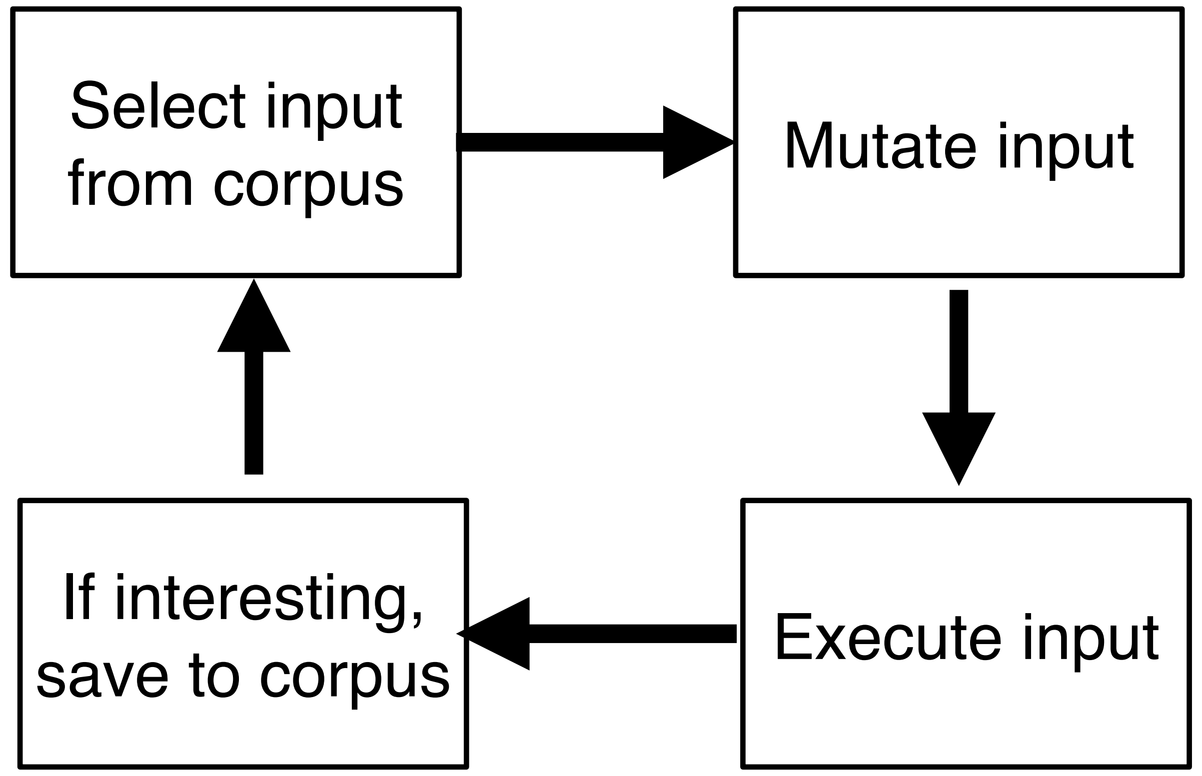
Input 1: A->B->G

Input 2: A->B->C->D->F->B->G

Input 3: A->B->C->D->F->B->C->D->F->B->G

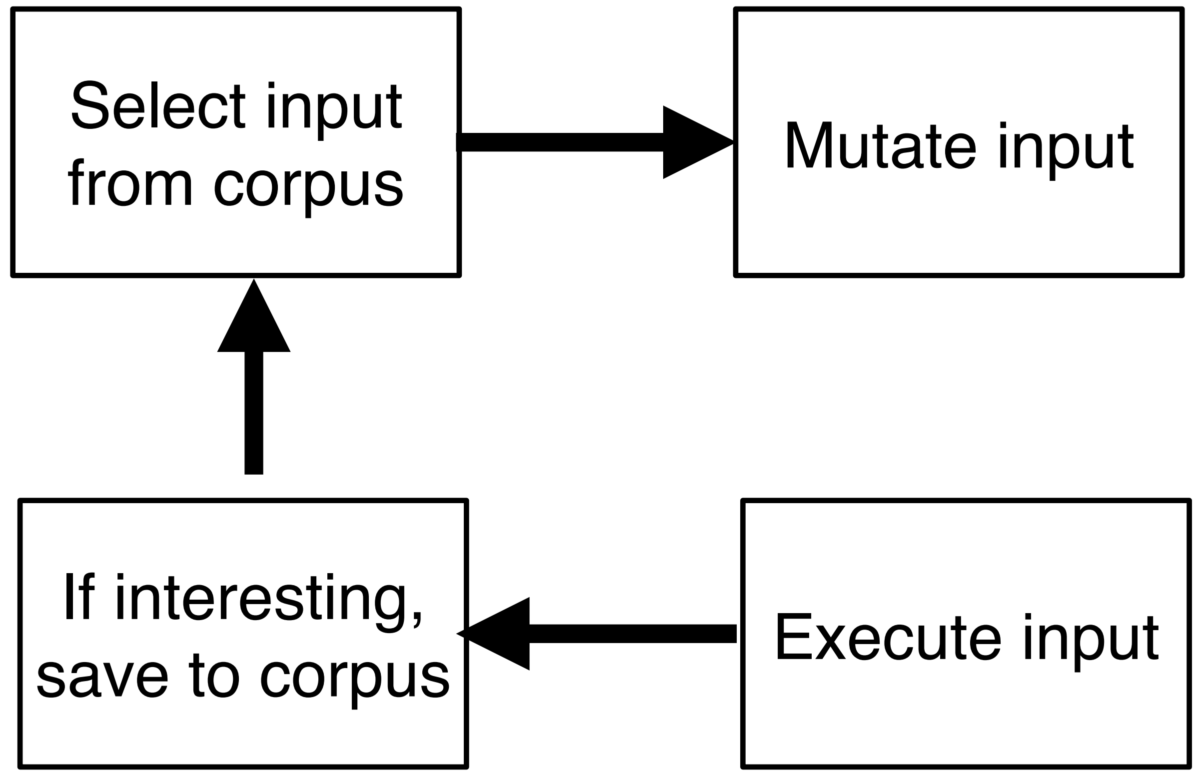
AFL Selects Inputs with Heuristics

- Some inputs might cover a superset of what others cover
- Some inputs might be longer to run, or are just otherwise larger
- AFL prefers inputs that are faster, favoring those that cover the same or a superset of branch edges in less time



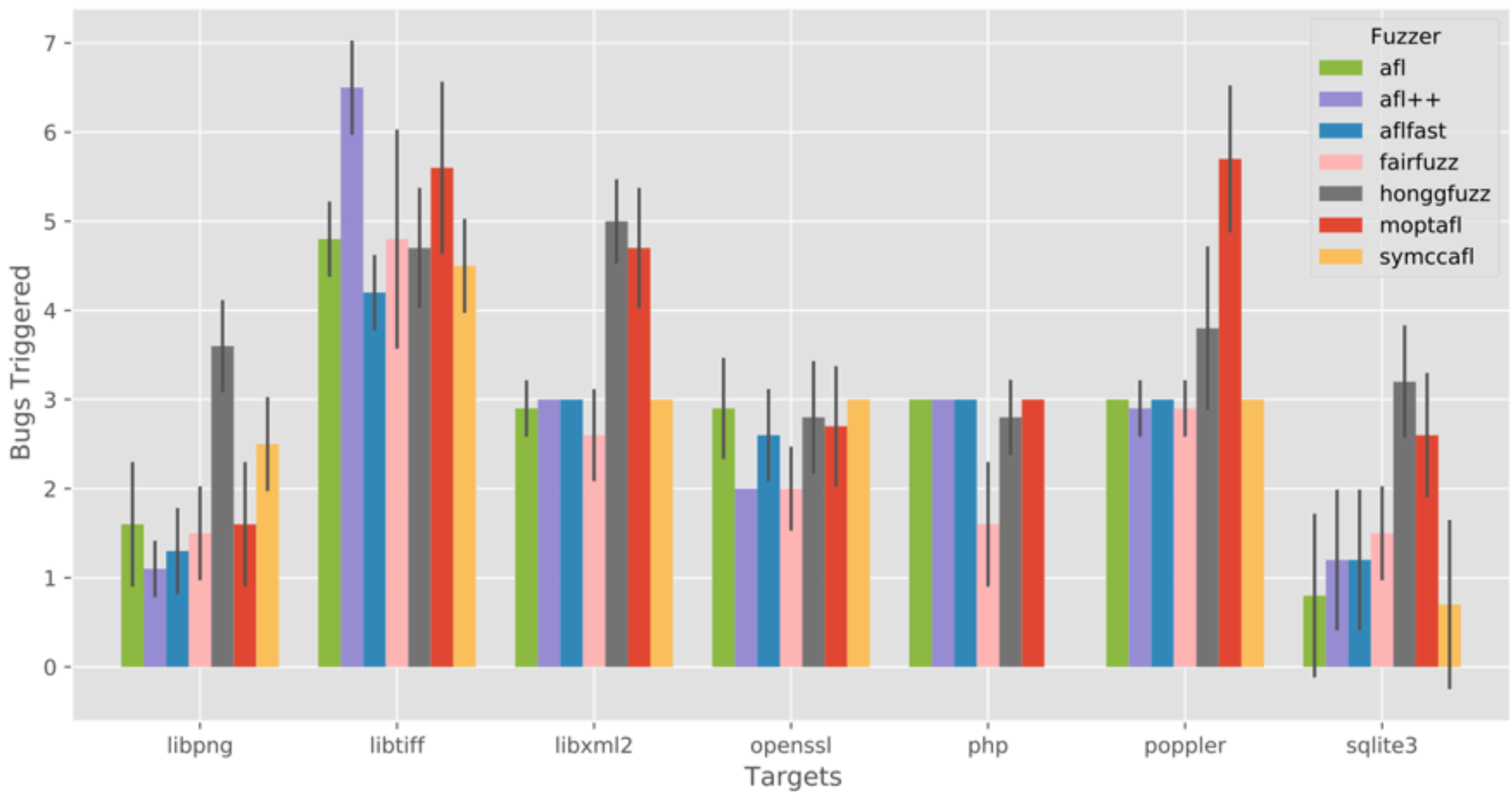
AFL has several mutation strategies

- Deterministic bit flips
- Addition and subtraction of small ints
- Swap integers for interesting values (-1, 256, etc)
- Stacked random tweaks (multiple at a time)
- Splice multiple files together



AFL Remains Popular/Effective

“AFL++” incorporates many individual improvements over past decade



“Magma: A Ground-Truth Fuzzing Benchmark” Hazimeh, Herrera and Payer. Proc of ACM on Measurement and Analysis of Computer Systems, 2021
https://doi.org/10.1145/3428334

Challenges/risks that come with fuzzing

Aside from "how to generate the inputs"

- How to de-duplicate bugs? 100's of inputs might trigger the "same" bug
- How to minimize failure-inducing inputs?
- How to know when we are done fuzzing, and how much resources to commit?

How SQLite is Tested

Core test harnesses

- “TCL”
- “TH3” (licensed)
- SQL Logic tests - differential testing
- dbsqlfuzz - proprietary fuzz tester, inputs are database file and query

SQLite tests environmental inputs

“Anomaly Testing”

- Out of memory errors
- I/O errors
- Crashes

SQLite has 100% branch and MC/DC coverage

- “Defensive” programming concerns
- Why is the test suite run three times for coverage?

```
void assert(booolean value){
    if(value){ //Should not be reachable
        crash();
    }
}
```

SQLite uses Dynamic Analysis with Tests

Additional runtime checks for invalid behavior

- Assertions
- Valgrind
- Memsys2
- Journal assertions
- Undefined behavior checks

Sanitizers Help Detect Bugs, but Aren’t Free

Address Sanitizer/Contiki-NG µIP case study

Table 3: Number of times and mean time-to-exposure (HH:MM:SS) for the seven vulnerabilities in the code base of μIP.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	**10** 00:17:20	**10** 00:35:40	**10** 00:03:00	0 ◷	**10** 00:53:29	**10** 00:23:59	**10** 00:49:58	**10** 00:01:39
uIP-ext-hdr	**10** 03:32:17	**10** 03:23:20	**10** 00:12:11	**10** 00:50:12	**10** 02:44:41	**10** 00:57:23	9 05:05:31	**10** 00:11:35
uIP-len	5 06:59:39	0 ◷	4 09:03:11	0 ◷	5 08:48:08	5 04:45:32	3 01:24:00	1 01:35:04
uIP-buf-next-hdr	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷
uIP-RPL-classic-prefix	6 06:21:52	2 18:52:46	7 03:57:22	0 ◷	6 09:55:47	**10** 05:14:50	2 07:11:56	0 ◷
uIP-RPL-classic-div	7 10:46:12	6 11:09:41	8 07:35:17	4 16:52:41	4 10:54:35	5 08:05:55	3 01:25:26	6 06:00:12
uIP-RPL-classic-sllao	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷

Table 5: Number of times and mean time-to-exposure for the μIP vulnerabilities using AddressSanitizer instrumentation.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	8 00:17:24	**10** 00:34:34	**10** 00:19:53	0 ◷	**10** 00:48:04	**10** 00:15:08	**10** 00:37:30	**10** 00:31:03
uIP-ext-hdr	**10** 05:15:10	**10** 02:30:14	**10** 01:20:44	**10** 01:11:22	**10** 02:17:21	**10** 01:53:00	**10** 03:33:16	**10** 02:38:00
uIP-len	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	2 11:57:49
uIP-RPL-classic-prefix	2 13:25:17	0 ◷	2 21:58:18	0 ◷	1 03:59:56	1 08:19:18	0 ◷	1 17:06:14
uIP-RPL-classic-div	0 ◷	0 ◷	0 ◷	2 09:50:03	1 02:41:05	0 ◷	0 ◷	0 ◷

Table 6: Impact of AddressSanitizer for the vulnerabilities in the code base of μIP. The table shows performance differences from Table 3: a positive impact is denoted with an upward arrow (▲) and negative impact with a downward arrow (▼). An integer denotes the change in the number of trials exposing the vulnerability; for similar number of trials the time difference is shown. A number of trials and a time denote vulnerabilities that a fuzzing tool exposed only on the sanitized code.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	▼ 2	▲ 00:01:06	▼ 00:16:53	—	▲ 00:05:25	▲ 00:08:51	▲ 00:12:28	▼ 00:29:24
uIP-ext-hdr	▼ 01:42:53	▲ 00:53:06	▼ 01:08:33	▼ 00:21:10	▲ 00:27:20	▼ 00:55:37	▲ 1	▼ 02:26:25
uIP-len	▼ 5	—	▼ 4	—	▼ 5	▼ 5	▼ 3	▲ 1
uIP-RPL-classic-prefix	▼ 4	▼ 2	▼ 5	—	▼ 5	▼ 9	▼ 2	▲ 1
uIP-RPL-classic-div	▼ 7	▼ 6	▼ 8	▼ 2	▼ 3	▼ 5	▼ 3	▼ 6

“So Many Fuzzers, So Little Time✱: Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack” Poncelet, Sagonas and Tsiftes, ASE 2022

https://doi.org/10.1145/3551349.3556946

Sanitizers Help Detect Bugs, but Aren’t Free

Effective Type Sanitizer/Contiki-NG µIP case study

Table 3: Number of times and mean time-to-exposure (HH:MM:SS) for the seven vulnerabilities in the code base of μIP.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	**10** 00:17:20	**10** 00:35:40	**10** 00:03:00	0 ◷	**10** 00:53:29	**10** 00:23:59	**10** 00:49:58	**10** 00:01:39
uIP-ext-hdr	**10** 03:32:17	**10** 03:23:20	**10** 00:12:11	**10** 00:50:12	**10** 02:44:41	**10** 00:57:23	9 05:05:31	**10** 00:11:35
uIP-len	5 06:59:39	0 ◷	4 09:03:11	0 ◷	5 08:48:08	5 04:45:32	3 01:24:00	1 01:35:04
uIP-buf-next-hdr	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷
uIP-RPL-classic-prefix	6 06:21:52	2 18:52:46	7 03:57:22	0 ◷	6 09:55:47	**10** 05:14:50	2 07:11:56	0 ◷
uIP-RPL-classic-div	7 10:46:12	6 11:09:41	8 07:35:17	4 16:52:41	4 10:54:35	5 08:05:55	3 01:25:26	6 06:00:12
uIP-RPL-classic-sllao	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷	0 ◷

Table 9: Number of times and mean time-to-exposure for the μIP vulnerabilities and EffectiveSan instrumentation.

Id	AFL-clang	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	**10** 00:10:49	**10** 00:09:19	**10** 00:16:19	0 ◷	**10** 00:06:07	**10** 00:14:56	**10** 00:20:15	**10** 00:05:52
uIP-ext-hdr	**10** 01:03:07	**10** 03:35:05	**10** 00:24:04	0 ◷	**10** 00:42:26	**10** 01:15:08	**10** 00:35:02	**10** 00:24:05
uIP-len	**10** 00:44:24	0 ◷	**10** 00:25:34	**10** 04:06:35	**10** 02:29:14	**10** 02:02:46	**10** 02:01:25	**10** 00:17:42
uIP-buf-next-hdr	2 12:47:46	0 ◷	3 08:36:57	0 ◷	1 01:52:32	2 00:29:24	2 07:13:00	7 06:41:59
uIP-RPL-classic-prefix	0 ◷	0 ◷	3 13:28:30	0 ◷	0 ◷	2 04:51:57	2 13:23:10	5 03:22:02
uIP-RPL-classic-div	3 22:04:40	0 ◷	3 19:27:27	0 ◷	2 04:29:34	1 02:31:07	2 18:44:25	6 08:53:54

Table 10: Impact of EffectiveSan for the vulnerabilities in the code base of μIP (differences from Table 3).

Id	AFL-gcc/-clang	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	▲ 00:06:31	▲ 00:26:21	▼ 00:13:19	—	▲ 00:47:22	▲ 00:09:03	▲ 00:29:43	▼ 00:04:13
uIP-ext-hdr	▲ 02:29:10	▼ 00:11:45	▼ 00:11:53	▼ 10	▲ 02:02:15	▼ 00:17:45	▲ 1	▼ 00:12:30
uIP-len	▲ 5	—	▲ 6	▲ 10	▲ 5	▲ 5	▲ 7	▲ 9
uIP-buf-next-hdr	▲ 2	—	▲ 3	—	▲ 1	▲ 2	▲ 2	▲ 7
uIP-RPL-classic-prefix	▼ 6	▼ 2	▼ 4	—	▼ 6	▼ 8	▼ 06:11:14	▲ 5
uIP-RPL-classic-div	▼ 4	▼ 6	▼ 5	▼ 2	▼ 2	▼ 4	▼ 1	▼ 02:53:42

“So Many Fuzzers, So Little Time✱: Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack” Poncelet, Sagonas and Tsiftes, ASE 2022
https://doi.org/10.1145/3551349.3556946

Fuzzing Structured Inputs

Example: Find bugs in C compilers

- Motivation: bugs in C compilers can be devastati
- The oracle is "easy": compare behavior across optimization levels and across compilers and versions
- Generating inputs to find those bugs is hard
  - Undefined behavior
  - Atypical code may be under-represented in developer test suites

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{ jxyang, chenyang, eeide, regehr }@cs.utah.edu


```c
int foo (void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y;
}
```

Figure 1. We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

1. Introduction

The theory of compilation is well developed, and there are compiler frameworks in which many optimizations have been proved correct. Nevertheless, the practical art of compiler construction involves a morass of trade-offs between compilation speed, code quality, code debuggability, compiler modularity, compiler retargetability, and other goals. It should be no surprise that optimizing compilers—like all complex software systems—contain bugs.

Miscompilations often happen because optimization safety checks are inadequate, static analyses are unsound, or transformations are flawed. These bugs are out of reach for current and future automated program-verification tools because the specifications that need to be checked were never written down in a precise way, if they were written down at all. Where verification is impractical, however, other methods for improving compiler quality can succeed. This paper reports our experience in using testing to make C compilers better.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

For the past three years, we have used Csmith to discover bugs in C compilers. Our results are perhaps surprising in their extent: to date, we have found and reported more than 325 bugs in mainstream C compilers including GCC, LLVM, and commercial tools. Figure 1 shows a representative example. Every compiler that we have tested, including several that are routinely used to compile safety-critical embedded systems, has been crashed and also shown to silently miscompile valid inputs. As measured by the responses to our bug reports, the defects discovered by Csmith are important. Most of the bugs we have reported against GCC and LLVM have been fixed. Twenty-five of our reported GCC bugs have been classified as P1, the maximum, release-blocking priority for GCC defects. Our results suggest that fixed test suites—the main way that compilers are tested—are an inadequate mechanism for quality control.

We claim that Csmith is an effective bug-finding tool in part because it generates tests that explore atypical combinations of C language features. Atypical code is *not* unimportant code, however; it is simply underrepresented in fixed compiler test suites. Developers who stray outside the well-tested paths that represent a compiler's "comfort zone"—for example by writing kernel code or embedded systems code, using esoteric compiler options, or automatically generating code—can encounter bugs quite frequently. This is a significant problem for complex systems. Wolfe [30], talking about independent software vendors (ISVs) says: "An ISV with a complex code can work around correctness, turn off the optimizer in one or two files, *and usually they have to do that for any of the compilers they use*" (emphasis ours). As another example, the front

"Finding and Understanding Bugs in C Compilers" Yang et al, PLDI 2011
https://doi.org/10.1145/1993316.1993532

Randomly Generating C Programs

Procedure

- Begin with grammar for subset of C
- Pick an allowable production from the grammar
- Generate that production and any targets needed
- If it’s a non-terminal production then recurse
- Handle dataflow transfer through each new production, keeping track of in-scope locals, globals etc
- Perform safety checks (avoid undefined behavior)

“Finding and Understanding Bugs in C Compilers” Yang et al, PLDI 2011
https://doi.org/10.1145/1993316.1993532

Randomly Generated C Programs Find Bugs

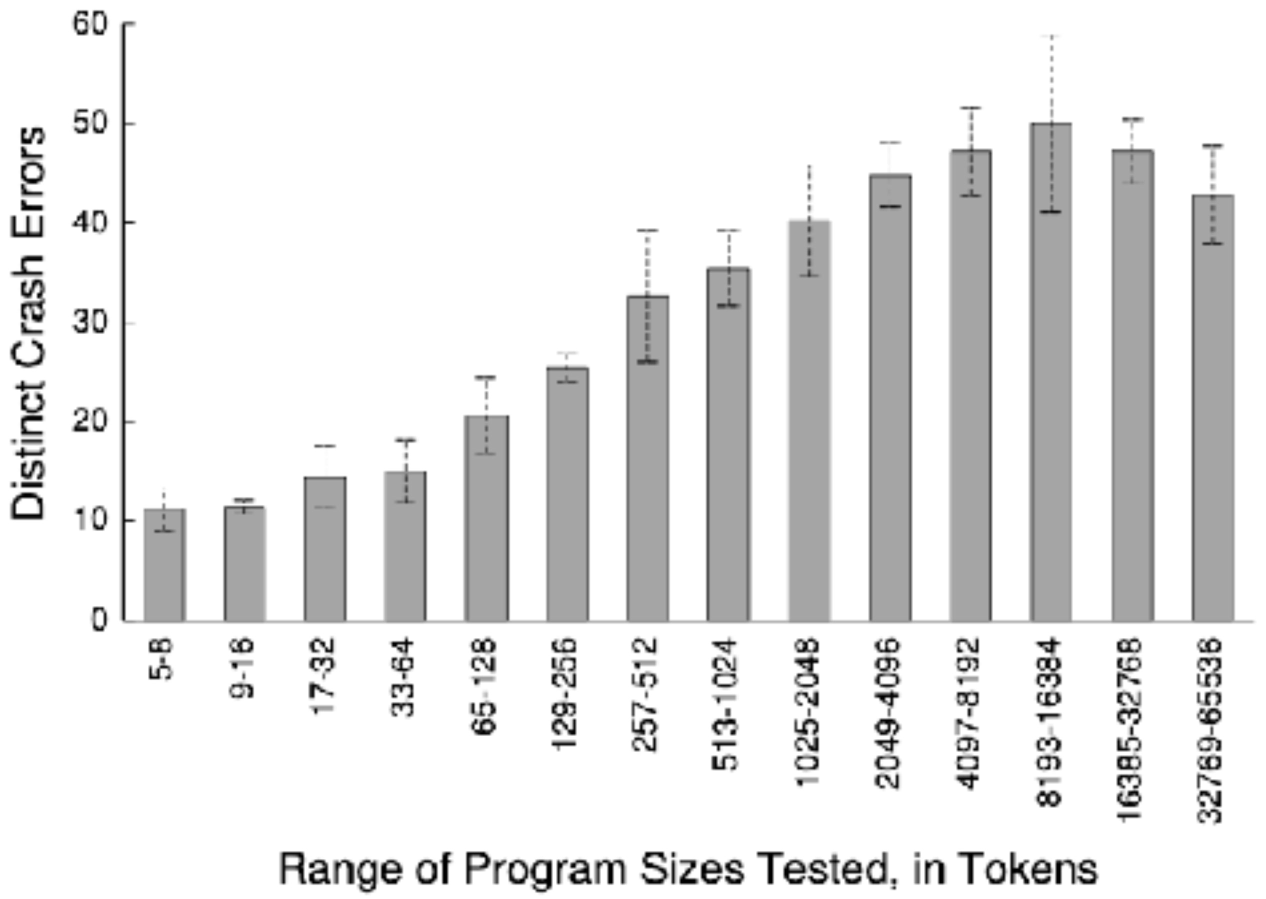


Figure 4. Number of distinct crash errors found in 24 hours of testing with Csmith-generated programs in a given size range

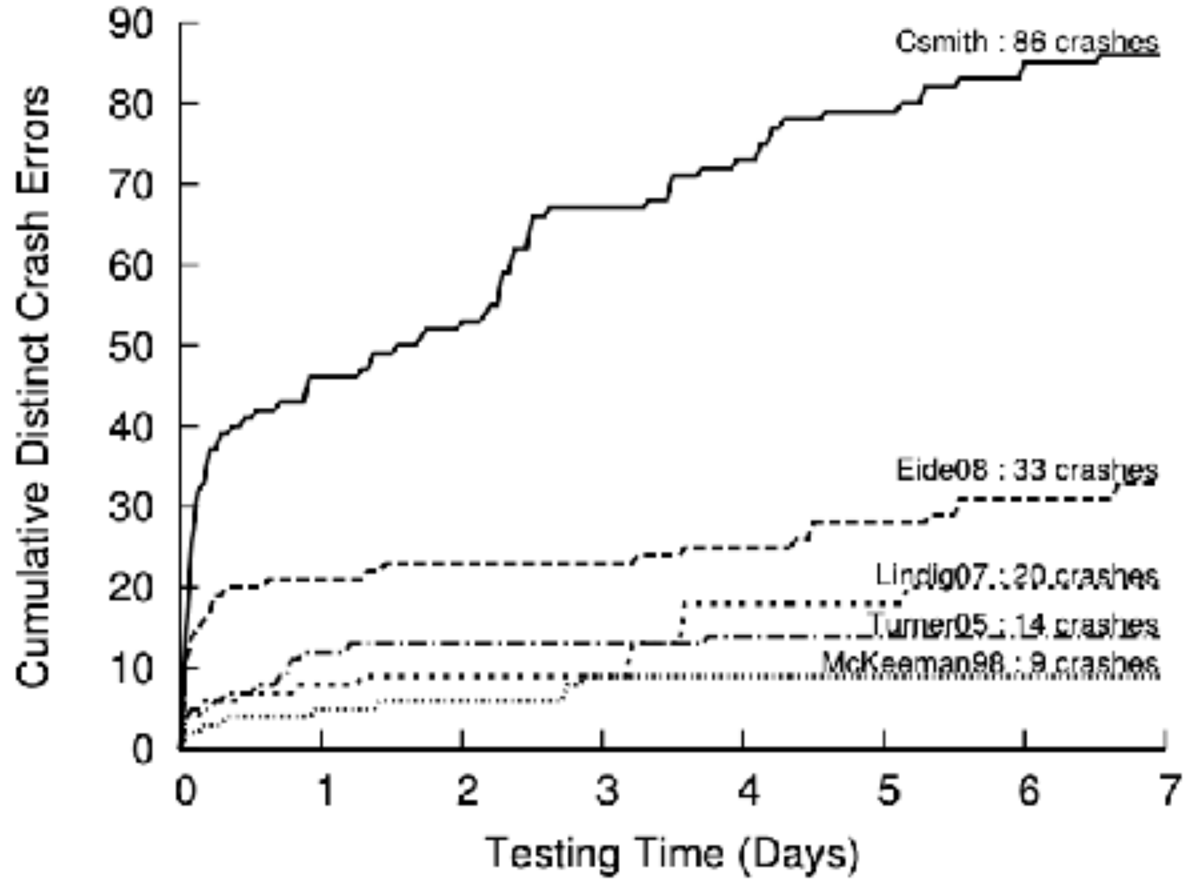


Figure 5. Comparison of the ability of five random program generators to find distinct crash errors

"Finding and Understanding Bugs in C Compilers" Yang et al, PLDI 2011
https://doi.org/10.1145/1993316.1993532