Vulnerability Detection - Fuzzing

Holistic Software Security

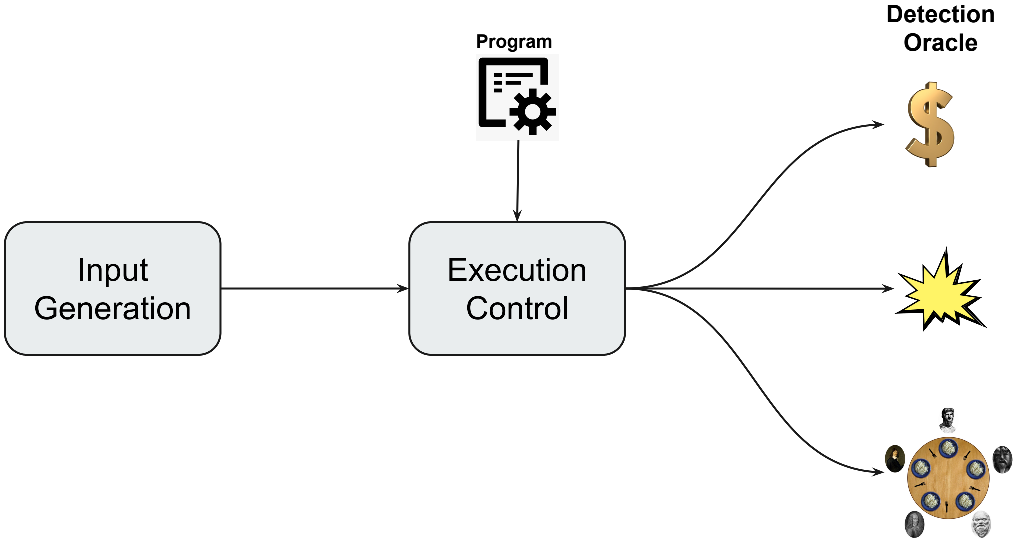
Aravind Machiry

Credits: Prof. Mayur Naik, UPenn

Fuzzing

- Automated test generation using random data.
    - Generate effective test cases, primarily using random data.

Fuzzing: High Level Idea



Input Generation

- Generate inputs (mostly randomized) to be fed into the program:
  - Random source.
  - Mutating existing inputs.
  - Based on a given input grammar.

Execution Control

- Execute the program with a given input:
    - Regular command line programs: execve and stdin.
    - OS: System calls.
    - Network programs: Send over network.
    - Input file: Save the data into a file and provide file.

Detection Oracle

- Detection of interesting program behavior:
  - Program crash.
  - Race condition.
  - High execution time.

Fuzzing Success

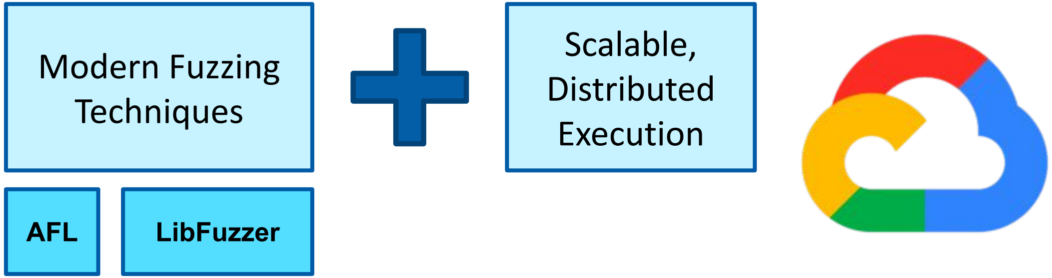
GCC Bug List Found by Random Testing (Total 79)									
date open	bug_id*	bug type	priority	rev reported	platform	component	status	date fixed(rev)(by)	File modified (lines)
3/30/2008	35764	wrong	P3	4.3.0	x86-32	target	confirmed	n/a	
5/15/2008	36238	crash	P2	4.4.0	x86-32	target	fixed	08/10 138924(Pinski)	reload1.c(1)
6/17/2008	36548	wrong	P3	136854	x86-32	middle-end	fixed	08/22 139450(Guenther)	fold-const.c(12)
6/24/2008	36613	wrong	P1	137045	x86-32	target	fixed	08/11 138955(Matz)	reload1.c(8)
6/25/2008	36635	crash	P1	137122	x86-32	target	fixed	10/08 140966(Jelinek)	cse.c(11)
7/1/2008	36691	wrong	P1	137327	x86-32	middle-end	fixed	08/04 138645(Guenther)	tree-ssa-loop-niter.c(2)
8/13/2008	37102	wrong	P1	139046	x86-32	tree-opt	fixed	10/17 141195(Macleod)	tree-outof-ssa.c(95)
8/13/2008	37103	wrong	P3	139046	x86-32	middle-end	fixed	08/14 139094(Jelinek)	fold-const.c (1)

open (837):

Title	Repro	Cause bisect	Fix bisect	Count	Last	Reported	Last activity
BUG: scheduling while atomic: syz-executor/ADDR	C	done		1	4d01h	1h08m	1h08m
BUG: sleeping function called from invalid context in __fput				1	4d01h	1h38m	1h38m
UBSAN: shift-out-of-bounds in init_sb				1	2d04h	3h19m	3h19m
BUG: sleeping function called from invalid context in __fdget_pos				1	6d00h	2d00h	7h41m
unexpected kernel reboot (6)				1	2d03h	2d02h	2d02h
INFO: task can't die in p9_client_rpc (3)				4	1d01h	2d13h	2d13h
memory leak in j1939_sk_sendmsg	C			1	6d15h	2d15h	2d05h
KASAN: use-after-free Read in v4l2_ioctl (2)	C	error		1	9d14h	5d14h	4d11h
KASAN: out-of-bounds Read in do_exit				1	10d	6d14h	4d14h
memory leak in xfrm_user_rcv_msg	C			1	12d	8d00h	12h54m
BUG: corrupted list in kobject_add_internal (3)	C	inconclusive		1	12d	8d01h	6d06h
memory leak in j1939_xtp_rx_rts	syz			1	12d	8d02h	5d09h
INFO: task hung in port100_probe	C	error		3	12d	8d04h	8d03h
general protection fault in detach_extent_buffer_page				1	14d	9d14h	9d10h

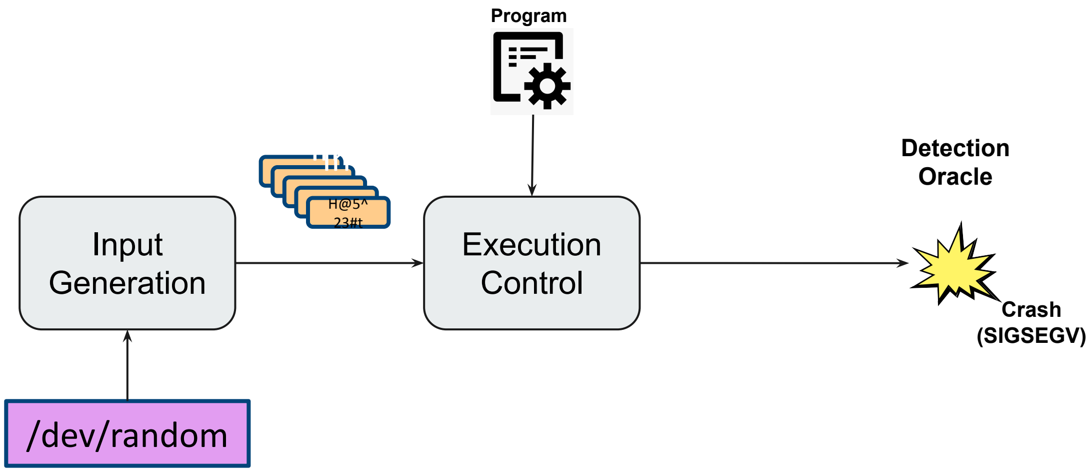
Fuzzing Success: OSS-Fuzz

- Continuous fuzzing infrastructure hosted on the Google Cloud Platform



- OSS-Fuzz has discovered over 17,400 bugs from 2016 to 2019 in many large projects (e.g. openssl, llvm, postgresql, git, firefox)

Fuzzing: Gen 1 (Random data)



Fuzzing: Gen 1

- Conducted by Barton Miller @ Univ of Wisconsin.

- 1990: Command-line fuzzer, testing reliability of UNIX programs.
  - Bombards utilities with random data

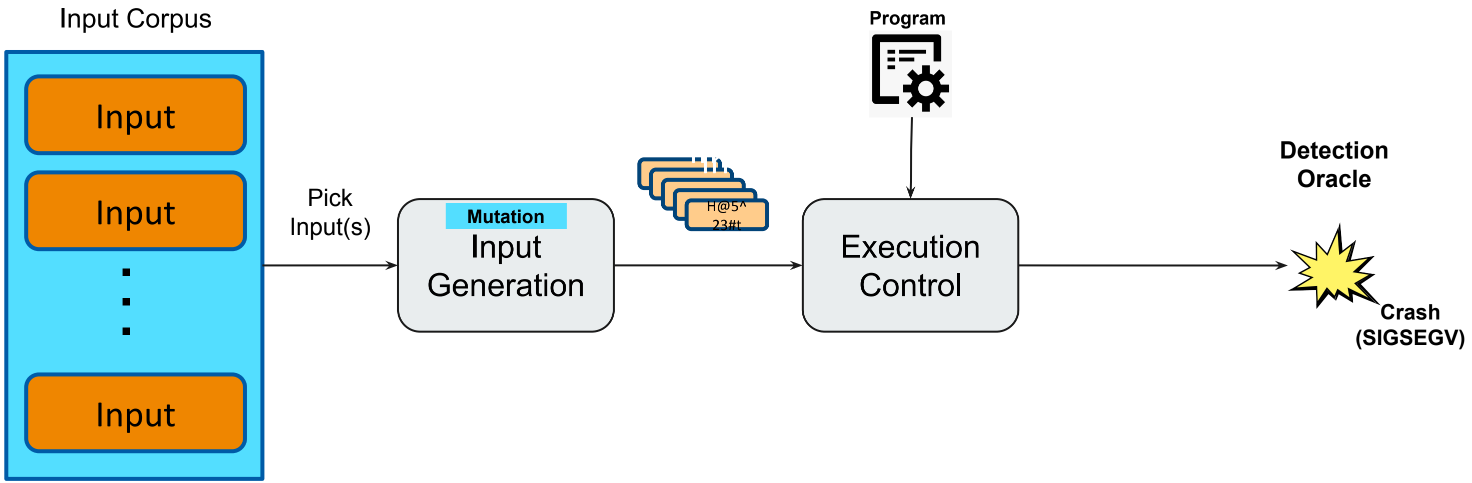
- 1995: Expanded to GUI-based programs (X Windows), network protocols, and system library APIs.

- Later: Command-line and GUI-based Windows and OS X apps.

Caused 25-33% of UNIX utility programs to crash (dump state) or hang (loop indefinitely).

- Hard to generate well formed data:
  - E.g., PNG files.

Fuzzing: Gen 2.a (Mutation based)

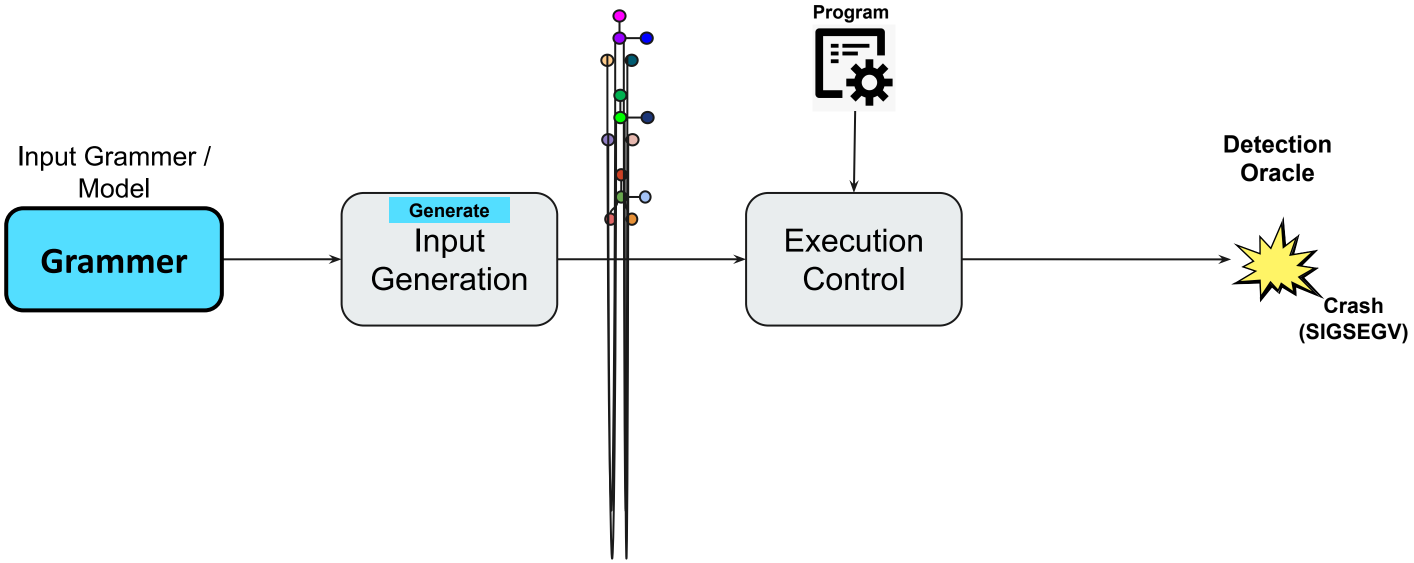


Fuzzing: Gen 2.a

- Very effective at generating semi-structured inputs.

- Still not so effective at generating highly structured inputs:
    - E.g., C files.

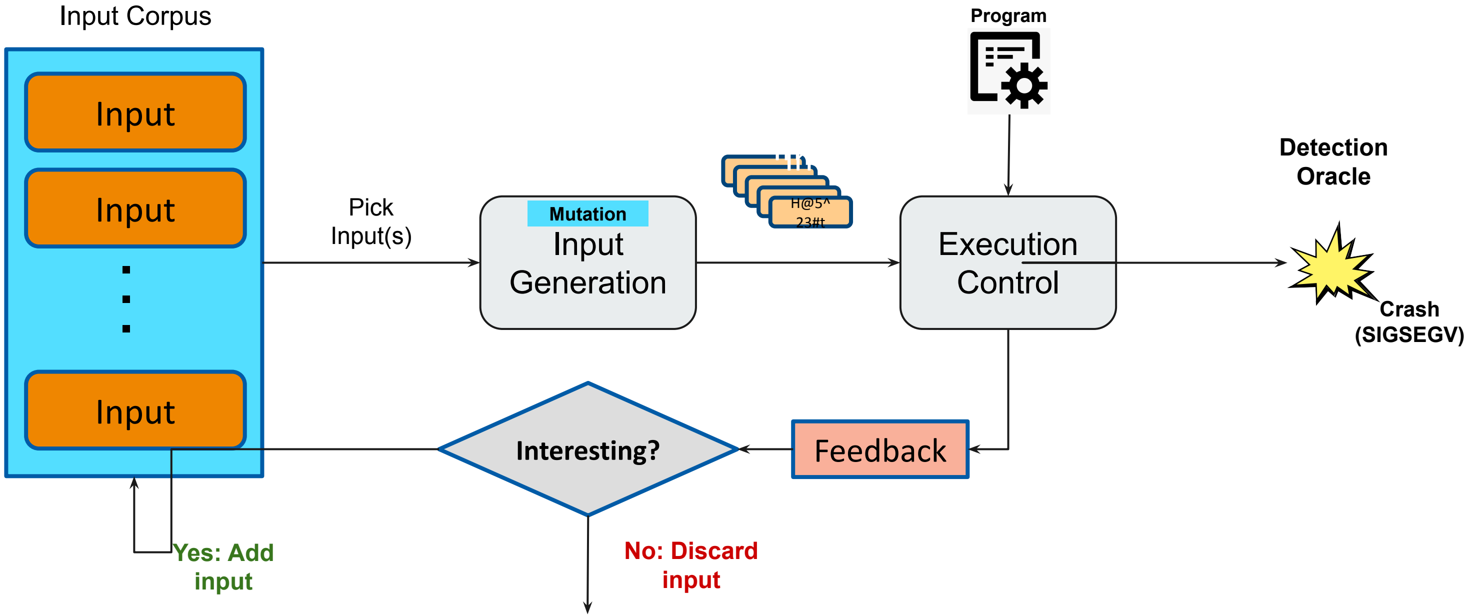
Fuzzing: Gen 2.b (Generation based)



Fuzzing: Gen 2.b

- Very effective at generating complex inputs:
  - Csmith: Generate syntactically valid but random C programs.
- Commercial tools:
  - Peach.
- Need to manually write these input grammars:
  - Domain Specific Language.
  - Large: ~200 lines

Fuzzing: Gen 3 (Feedback guided Mutation based)



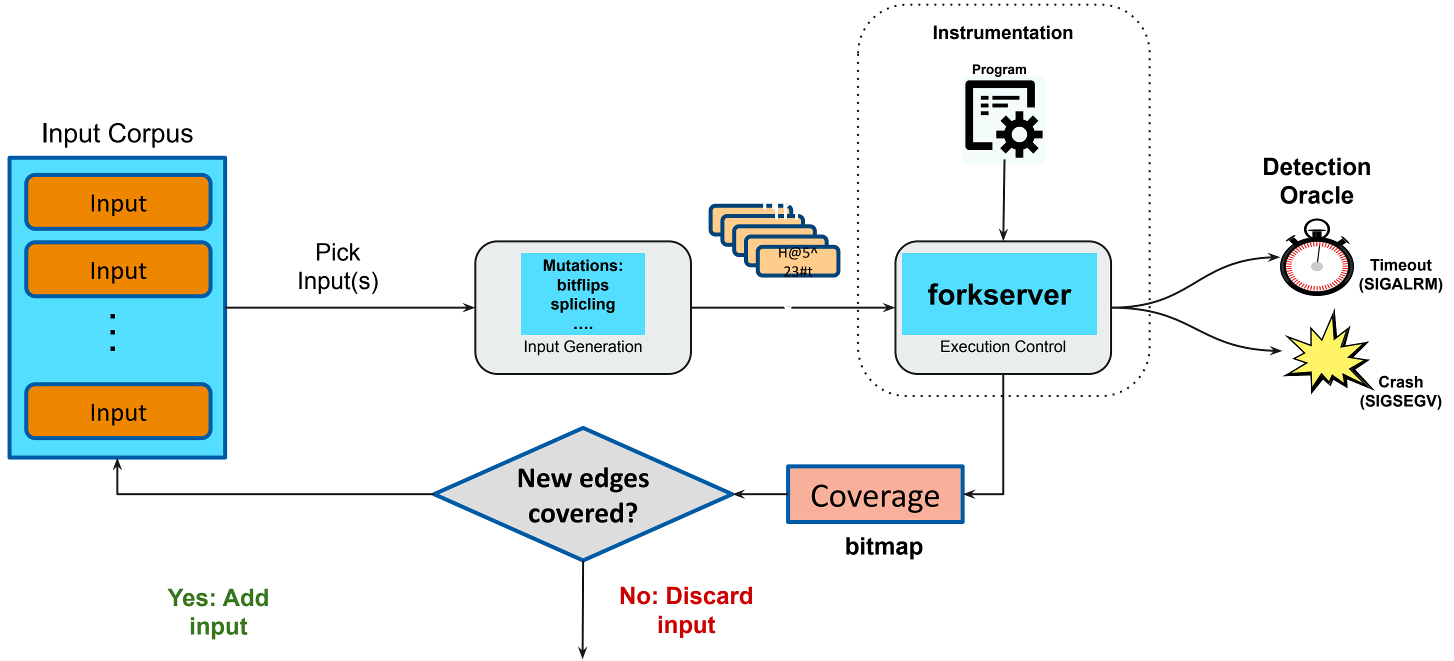
Fuzzing: Gen 3

- Extremely effective at quickly generating well-formed inputs.
- Highly successful commercial grade tool:
  - AFL (AFLPlusPlus)
- Need a way to capture feedback: Impacts performance.

Fuzzer Deep Dive : AFLPlusPlus

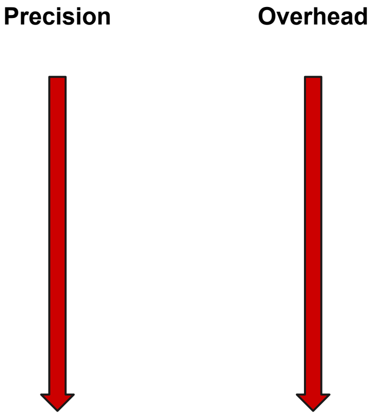
- Based on American Fuzzy Lop (AFL) developed by Michał Zalewski
- Coverage Feedback based Mutational Fuzzing.
- Highly customizable, efficient and very well maintained.
- Revolutionized fuzzing research:
    - ~30 papers since 2015.
- Found various (~200) bugs in well-maintained programs.

AFLPlusPlus (A++): Coverage guided

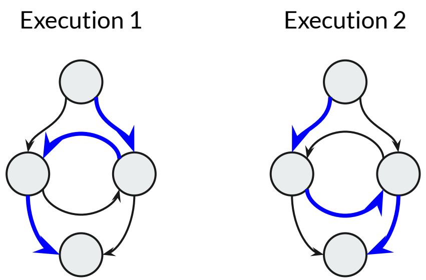


A++: Coverage Map

- Coverage choices:
  - Line or Basic block coverage:
    - Basic blocks executed.
  - **Edge coverage (used by A++):**
    - **Edges (Basic block tuple) executed.**
  - Path coverage:
    - Sequence of basic blocks executed.



Coverage choices



- Does Execution 1 and 2 have:
    - Same basic block coverage?
    - Same edge coverage?
    - Same path coverage?

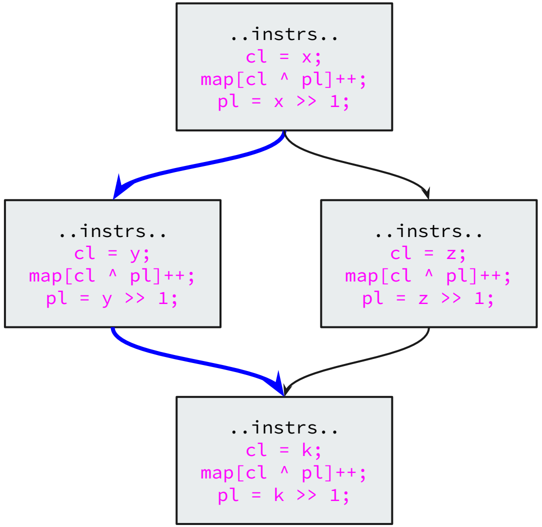
A++: Coverage Instrumentation

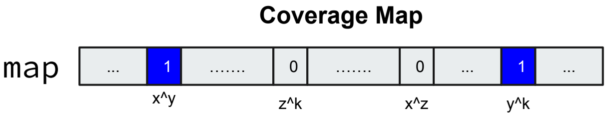
- Coverage map: Memory area in the program that stores coverage.
- Every basic block in all functions will be instrumented to update coverage map.

(Hopefully) Unique key for each edge

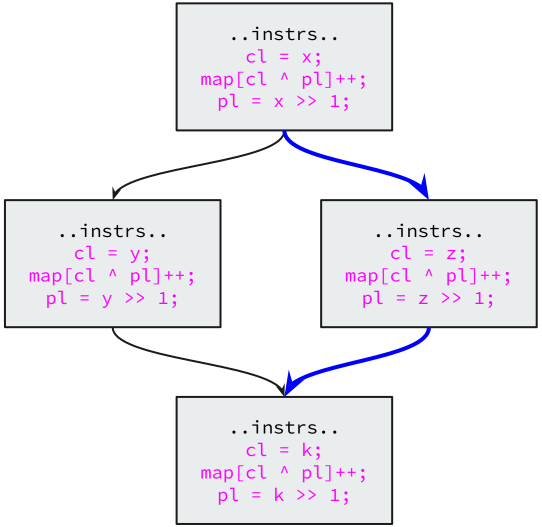
```
cur_location = <COMPILE_TIME_RANDOM>;
coverage_map[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

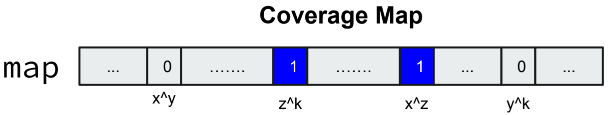
A++: Coverage Map



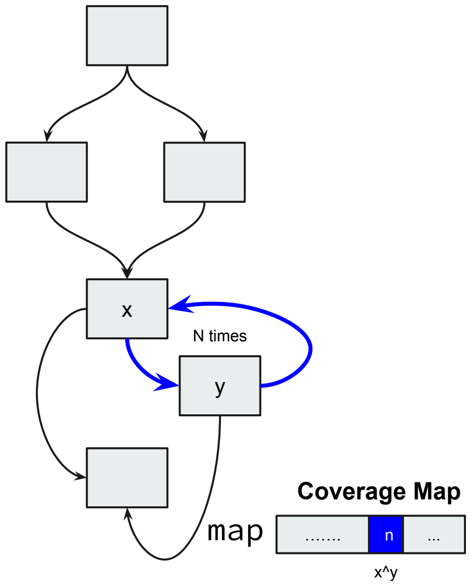


A++: Coverage Map

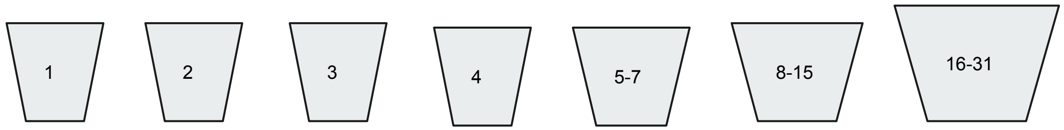




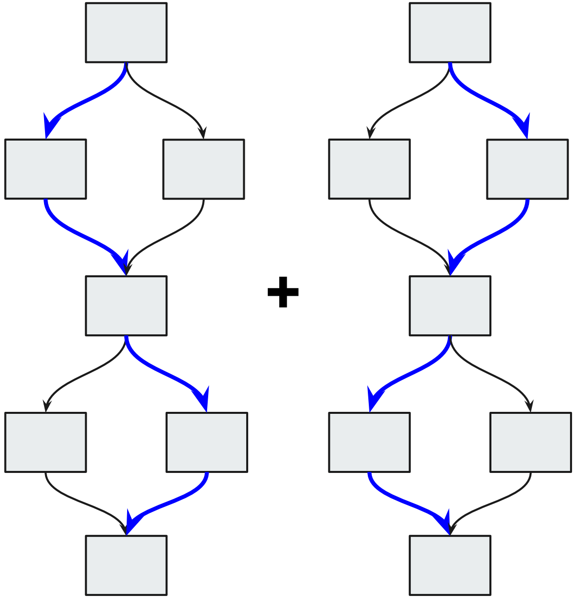
A++: Coverage Map: Bucketized edge counts



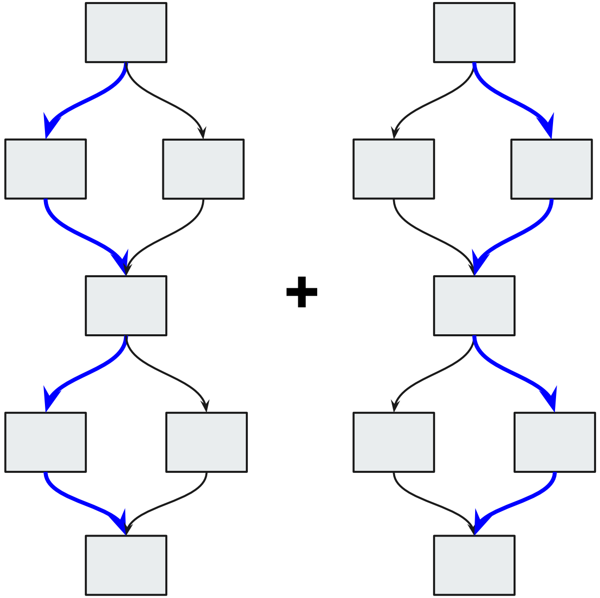
- Edge counts are bucketized:
  - E.g., Coverage map of executions with loop counts that belong to the same bucket will be considered the same.



A++: Coverage Map



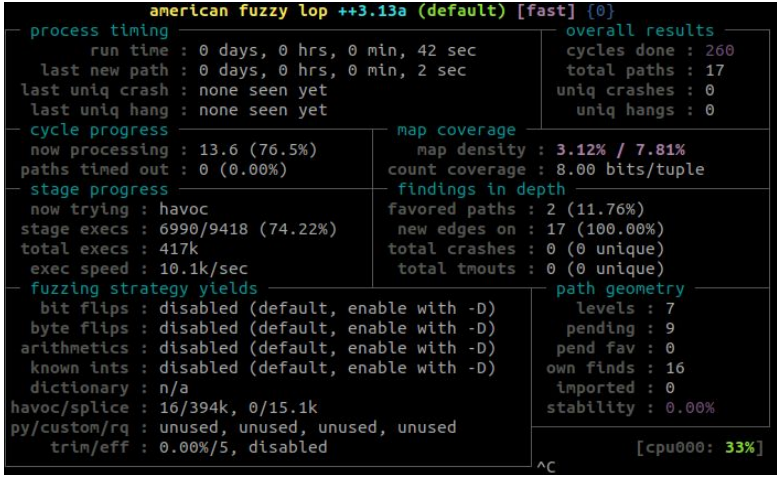
Has same coverage map as



Using A++

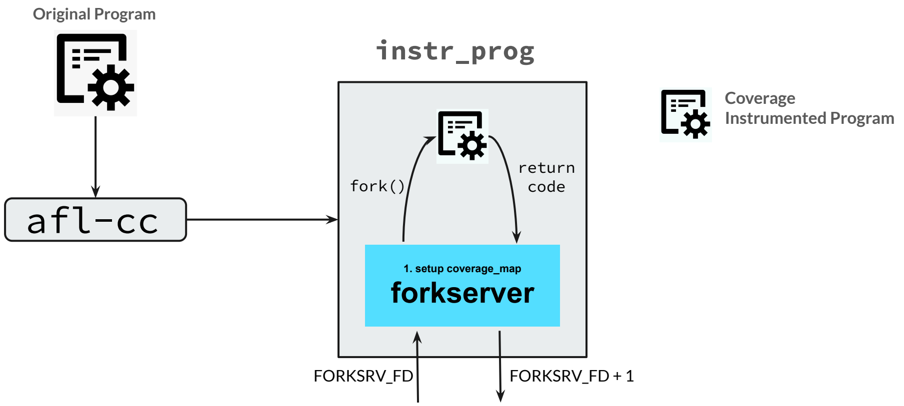
- Instrumentation: Compile the target program using afl compiler, i.e., `afl-cc`:
  - `afl-cc <.c> -o instr_prog`
  - Does:
    - Instrumentation to compute coverage.
    - Add forkserver.

- Fuzzing: Start fuzzing `instr_prog`:
  - `afl-fuzz -i <inputs_folder> -o <output_folder> -- instr_prog`



A++: Instrumentation

`afl-cc <.c> -o instr_prog`



- forkserver (`constructor`) `afl-compiler-rt.o.c`:

  Setup coverage map (shared memory map).

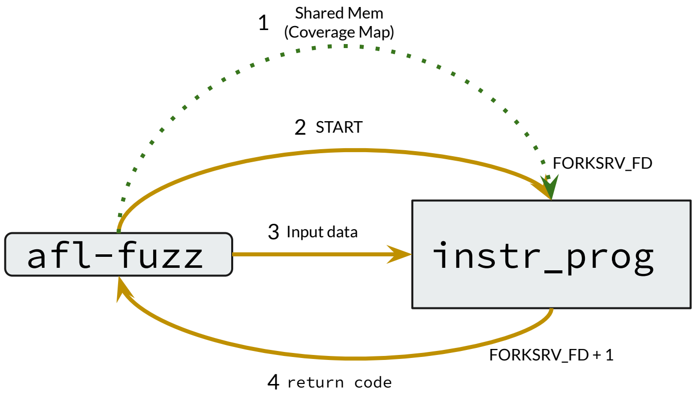
  while (1) {

  1. Wait for command at FORKSRV_FD.
  2. Once received, `fork` and start executing main of the original program:
     - Program would be writing to coverage map (shared memory).
  1. Sends the return code through FORKSRV_FD + 1

  }

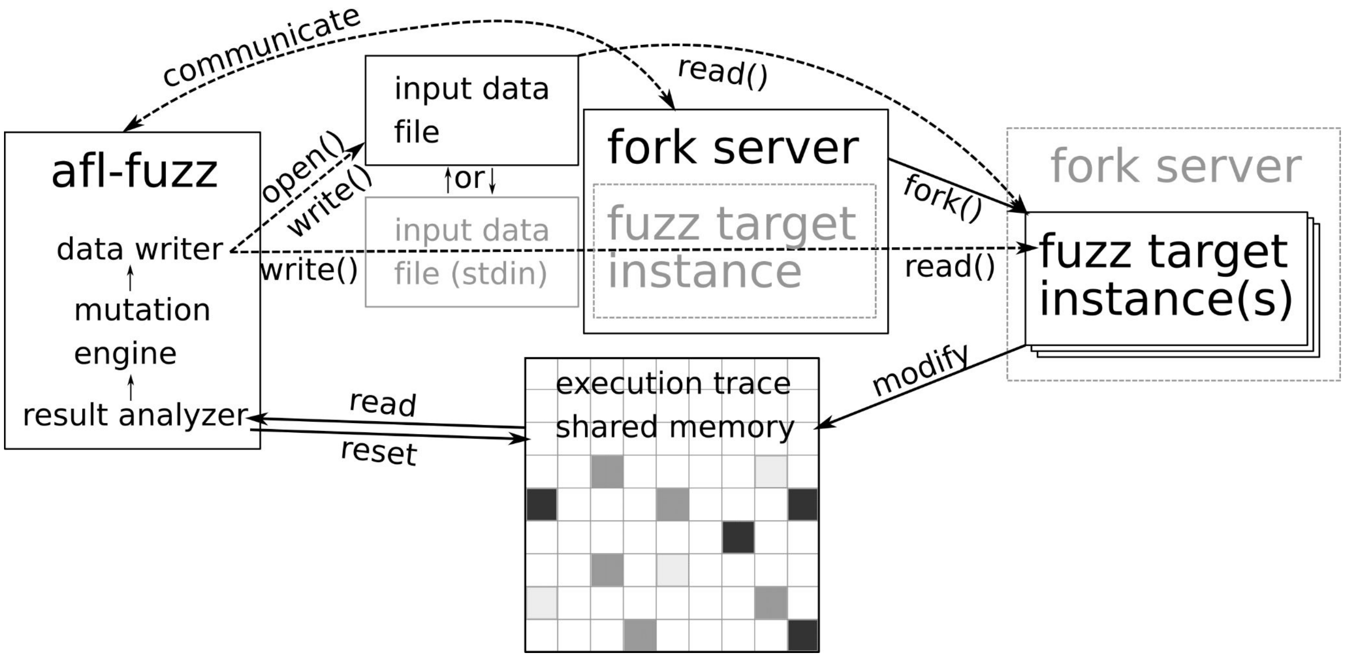
A++: Fuzzing

```
afl-fuzz -i <inputs_folder> -o <output_folder> -- instr_prog
```



- afl-fuzz (`src/afl-fuzz.c`):
  - Setup (1):
    - Send shared memory id.
  - Fuzzing Loop (2-3-4):
    - 2. START
    - 3. Input data (stdin or file)
    - 4. return code (crashes or timeout)

A++: Fuzzing


communicate
read()
input data
file
afl-fuzz
open()
write()
↑or↓
fork server
fork()
fork server
data writer
↑
mutation
engine
↑
result analyzer
input data
file (stdin)
write()
fuzz target
instance
read()
fuzz target
instance(s)
modify
read
reset
execution trace
shared memory

A++: Few drawbacks

- Effectiveness highly depends on the quality of initial test cases.
- Does not readily accepts grammar for inputs.
- Does not readily accepts other coverage metrics:
    - We may want different coverage metric for functions:
        - E.g., BB coverage for foo, edge for bar, path for baz.

Fuzzing Challenges: Input Generation

- Constrained Input:
  - Driller: Augmenting Fuzzing through Symbolic Execution [NDSS 2016]
  - Angora: Efficient Fuzzing by Principled Search [S&P 2018]
  - REDQUEEN: Fuzzing with Input-to-State Correspondence [NDSS 2019]
- Structured Input:
  - DIFUZE: Interface Aware Fuzzing for Kernel Drivers [CCS 2017]
  - WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats [ISSTA 20]

```
if (i == 345890)
{
    …
}
```

```
typedef struct {
    ISP_RT_BUF_CTRL_ENUM ctrl;
    _isp_dma_enum_ buf_id;
    ISP_RT_BUF_INFO_STRUCT *data_ptr;
    ISP_RT_BUF_INFO_STRUCT *ex_data_ptr;
    unsigned char *pExtend;
} ISP_BUFFER_CTRL_STRUCT;
```

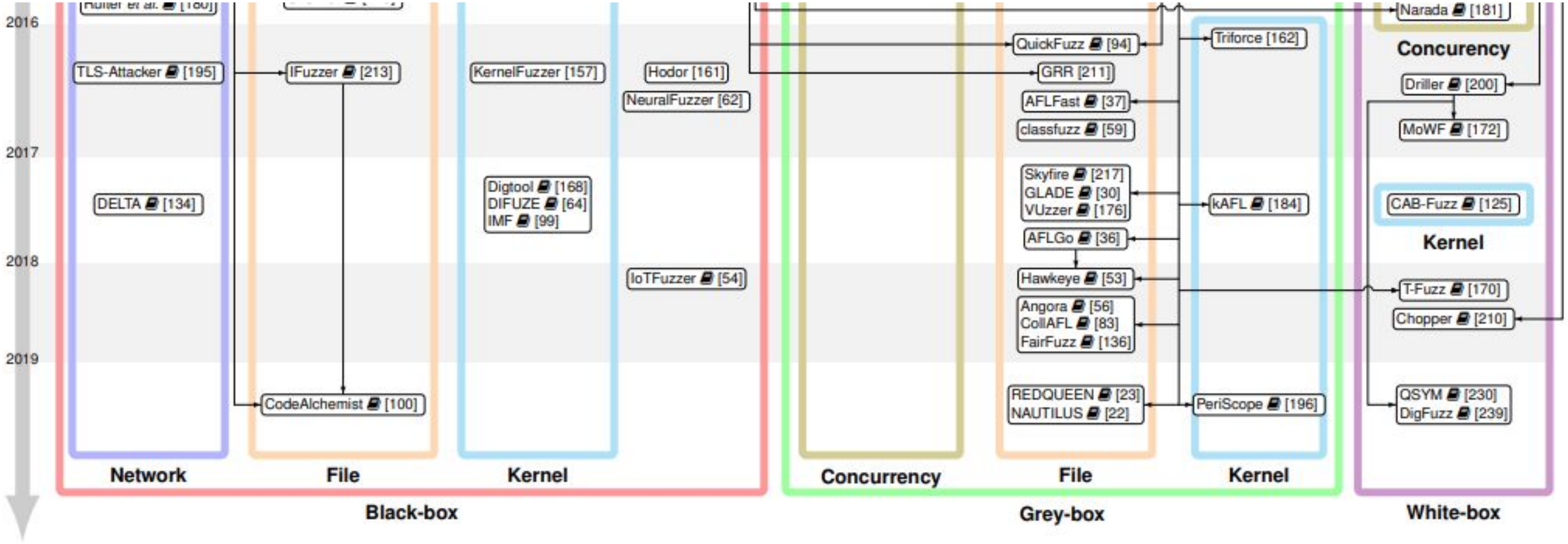
Fuzzing Challenges: Coverage metrics

- Is Path Coverage always good?
    - "Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing" [RAID 2019]
    - "CollAFL: Path-Sensitive Fuzzing" [S&P 18]

Fuzzing Challenges: Input prioritization

- Some inputs are good than other inputs?
    - “Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization” [NDSS 2020]
    - “ParmeSan: Sanitizer-guided Greybox Fuzzing” [USENIX 2020]

Fuzzing Trends



The Art, Science, and Engineering of Fuzzing: A Survey

Fuzzing Trends

- New directions:
  - ML to detect which bytes to mutate.
  - Transform the program and make it easy to fuzz (t-fuzz).
  - Combine different fuzzers: CollabFuzz: A Framework for Collaborative Fuzzing [EuroSec 2021]
- Improvements:
  - Use fancy techniques to improve different aspects of fuzzing.
- Fuzzing different applications:
  - File systems, Kernel drivers, IoT devices, etc.

Fuzzing as a generic exploratory technique

- Fuzzing allows us to find inputs that has high probability to satisfy certain goal.
    - Goal: Find more bugs:
        - Feedback: Coverage.
    - Goal: Find more temporal bugs (e.g., use-after-free, double-free):
        - Feedback: Likelihood of input triggering malloc/free.
    - Goal: Find concurrency bugs.
        - Feedback: Number of threads invoked.
    - Goal: Find denial-of-service bugs.
        - Feedback: Time taken by the input (the more time the better).
    - Goal: Type inference: Infer types for variables.
        - Feedback: Number of type-checker error (the less the better).