Fuzzing

Saurabh Jha

Disclaimer

- Many slides borrowed and in some-cases replicated from
  - Abhik Roychoudhury's lecture in ISSISP Summer School 2018
  - AFL tutorials
  - My own slides presented elsewhere

Outline

- Basics of Fuzzing
- Coverage-based Greybox Fuzzing as Markov Chain
- Fuzzing for Autonomous (AI-driven) Systems

Basics of Fuzzing

Def. Fuzzing

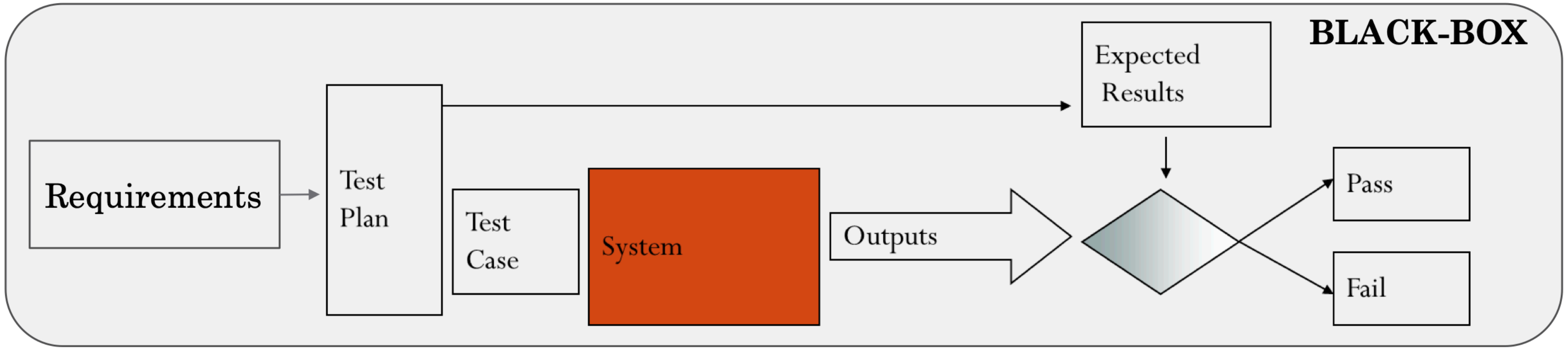
- [Input] random, no model enforced of program behavior, system, etc.
- [Reliability] application crashes or hangs
- [Automation] input generation, result checker, methodology independent of program, compiler, OS

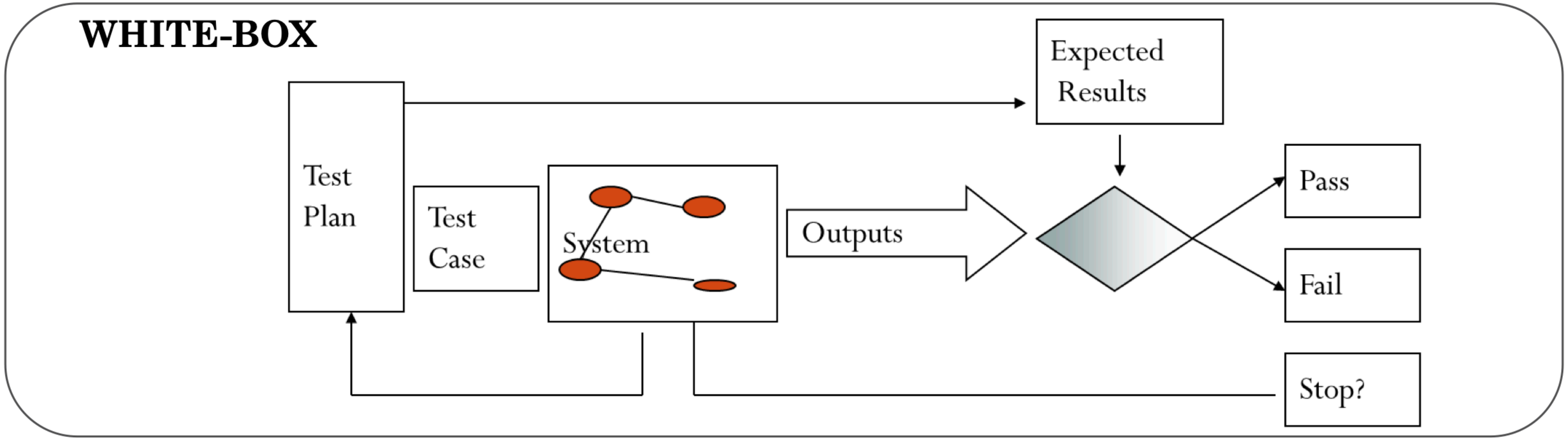
[Source] B. Miller, http://pages.cs.wisc.edu/~bart/fuzz/

Why is it important?

- Identifies bugs in application design and/or implementation
- Trustworthy applications
  - Reliability of the application
    - Users may experience hang or crash (think about hangs of your favorite app)
  - Security of the application
    - Hackers can exploit the bug to steal information (e.g., Heartbleed) or (physically) harm users (e.g., causing accidents for autonomous vehicles)
- Exciting future: New application domains for fuzzing, Automatic identification and repairs

Testing: Black, White, and Gray





First Fuzzer: Study of Reliability of Unix Utilities, Miller et al.

“While our testing strategy sounds naïve, its ability to discover fatal program bugs is impressive”

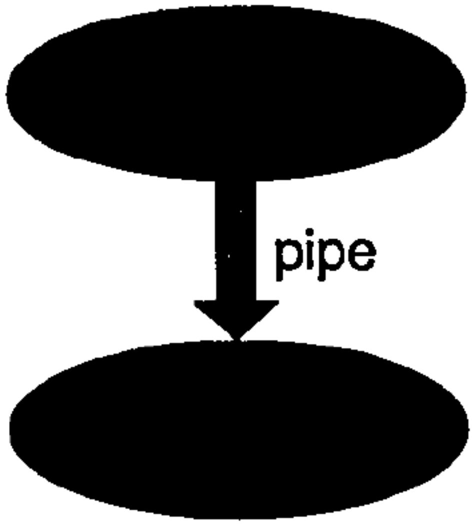


FIGURE 1. Output of Fuzz Piped to a Utility.

TABLE II. *List of Utilities Tested and the Systems on which They Were Tested (part 1)*

*●=utility crashed, ○=utility hung, *=crashed on SunOS 3.2 but not on SunOS 4.0, ⊕ = crashed only on SunOS 4.0, not 3.2. —=utility unavailable on that system. !=utility caused the operating system to crash.*

Utility	VAX (v)	Sun (s)	HP (h)	i386 (x)	AIX 1.1 (a)	Sequent (d)
adb	● ○	●	●	○	—	—
as	●			●	●	●
awk						
bc				● ○		
bib			—	—	—	—
calendar				—		
cat						
cb	●		●	●	○	●
cc						
/lib/ccom				—	—	●
checkeq				—		
checknr				—	—	
col	● ○	●	●	● ○	●	●

Industry standard for testing

Microsoft

Springfield Project - Fuzzing as a service

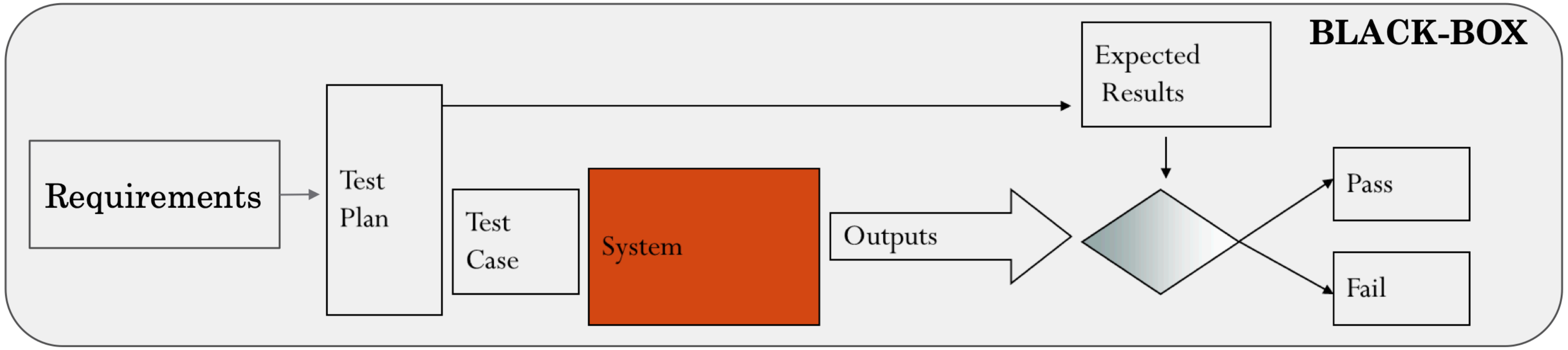
Google

OSS-Fuzz - Continuous fuzzing for open-source projects

Random Input Generation

- Mutation-based

- Generation-based



Mutation

- Inputs
  - Program P
  - Seed input x0
  - Mutation ratio 0<m ≤1
- Next step
  - Obtain an input x1 by randomly flipping m*|x0| bits
  - Run x1 and check if P crashes or terminates properly
  - In either case document the outcome, and generate next input
- End of fuzz campaign
  - When time bound is reached, or N inputs are explored for some N
  - Always make sure that bit flipping does not run same input twice.

Why depend on mutations?

- Many programs take in structured inputs
  - PDF Reader, library for manipulating TIFF, PNG images
  - Compilers which take in programs as input
  - Web-browsers, ...

- Generating a completely random input will likely crash the application with little insight gained about the underlying vulnerability

- Instead take a legal well-formed PDF file and mutate it!

Why depend on mutations?

- Principle of mutation fuzzing
    - Take a well-formed input which does not crash.
    - Minimally modify or mutate it to generate a “slightly abnormal” input
    - See if the “slightly abnormal” input crashes.

- Salient features
    - Does not depend on program at all [nature of BB fuzzing]
    - Does not even depend on input structure.
    - Yet can leverage complex input structure by starting with a well-formed seed and minimally modifying it.

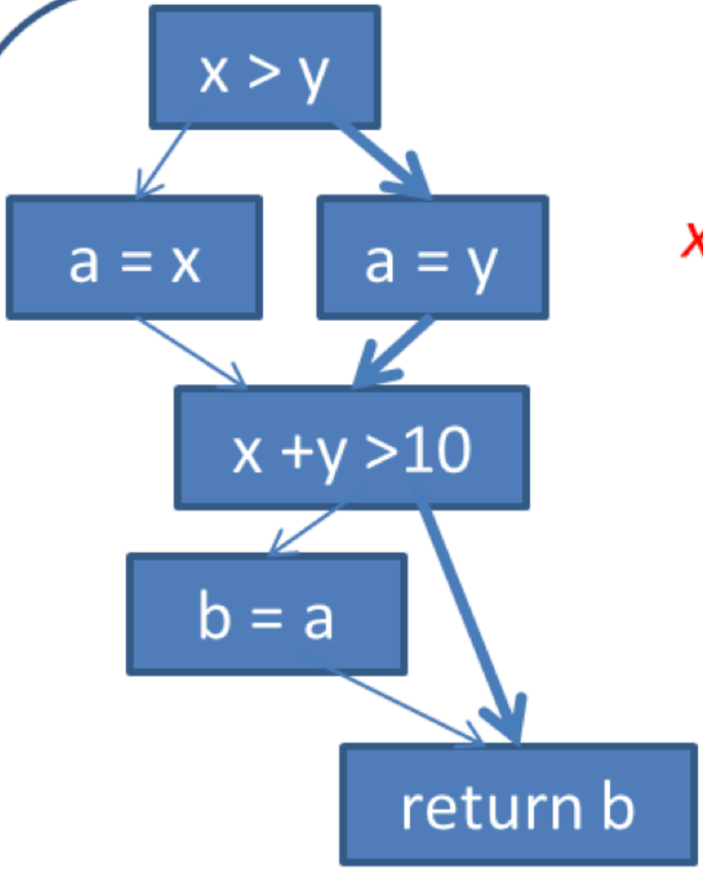
Generation Based Fuzzing

- Test cases are generated from some description of the format: RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than
- random fuzzing
- Can take significant time to set up
- E.g., SPIKE,Sulley,Mu-4000, Codenomicon, Peach Fuzzer

Mutation vs Generation

Mutation-based	Super easy to setup and automate +	Little to no protocol knowledge required +	Limited by initial corpus −	May fail for protocols with checksums, or other complexity −
Generation-based	Writing generator is labor intensive for complex protocols −	have to have spec of protocol (frequently not a problem for common ones http, snmp, etc…) −	Completeness +	Can deal with complex checksums and dependencies +

White-box Fuzzing



Cover more paths

$x \leq y \wedge x + y \leq 10$

$x \leq y \wedge \neg x + y \leq 10$

$\neg x \leq y$

Directed	D
Automated	A
Random	R
Testing	T

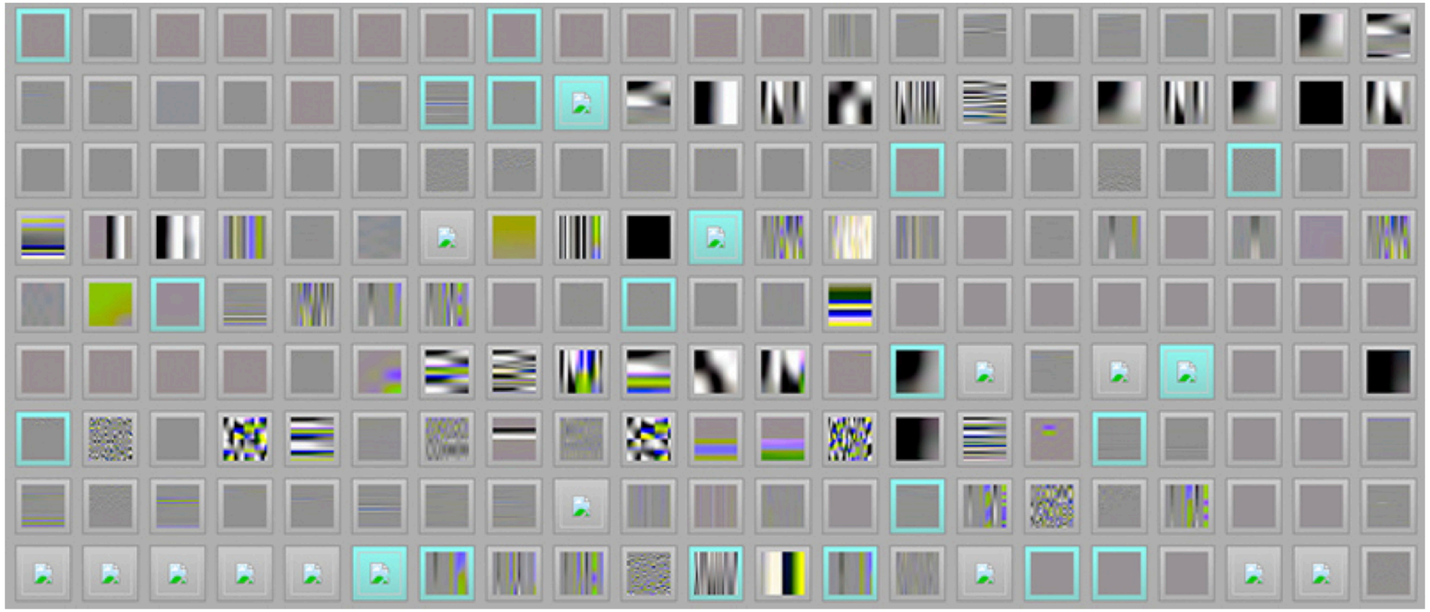
Code Coverage

- Some of the answers to our problems are found in code coverage
- To determine how well your code was tested, code coverage can give you a metric.
- But it’s not perfect (is anything?)
- Code coverage types:
  • Statement coverage – which statements have been executed • Branch coverage – which branches have been taken
  • Path coverage – which paths were taken.

Coverage-based Gray box Fuzzing as Markov Chain

Intro to American Fuzzy Lop (AFL)

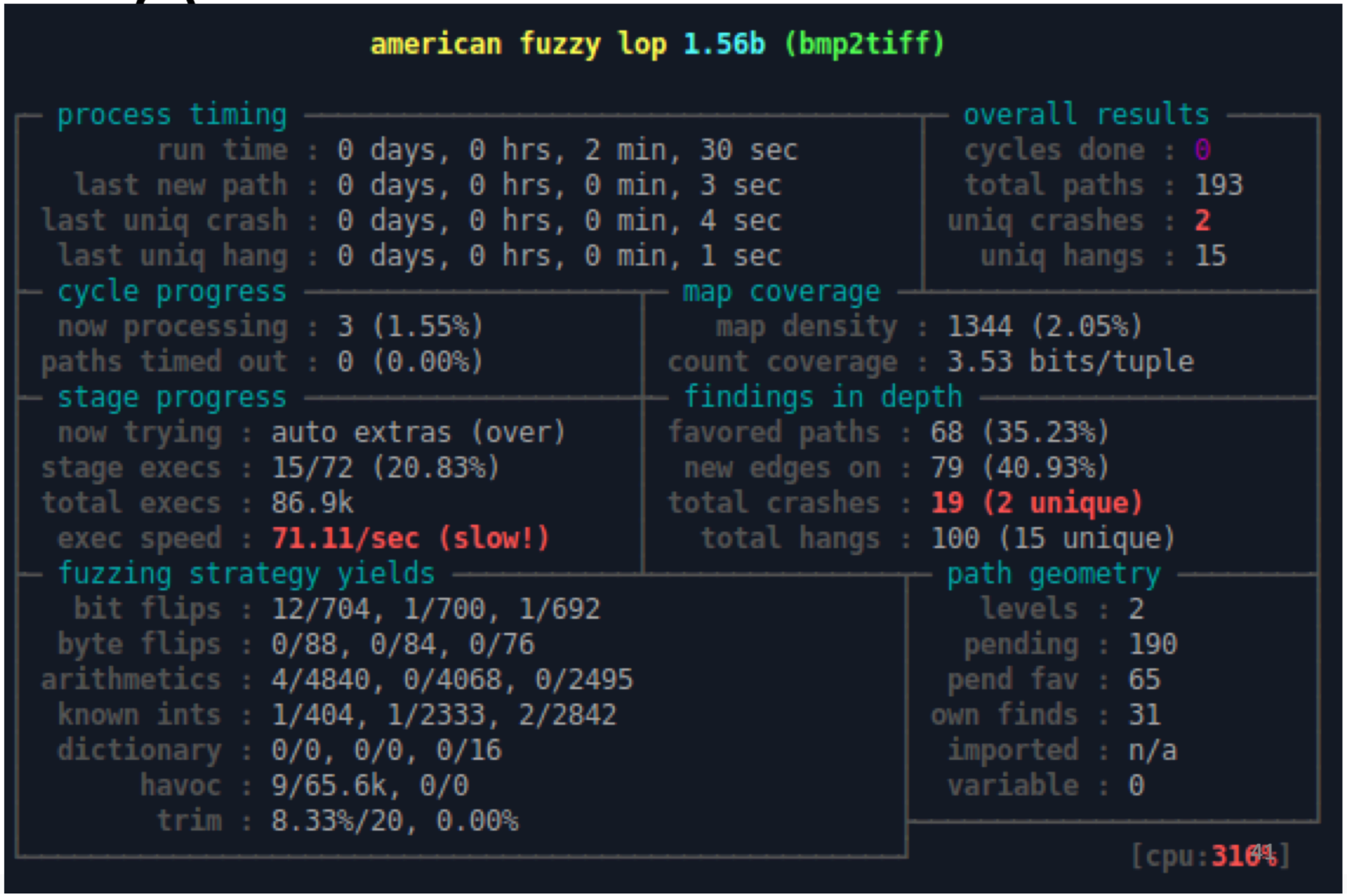
- AFL (http://lcamtuf.coredump.cx/afl/) by Michal Zalewski
- afl-fuzz -i test-cases -o findings -m none -- ./indent @@



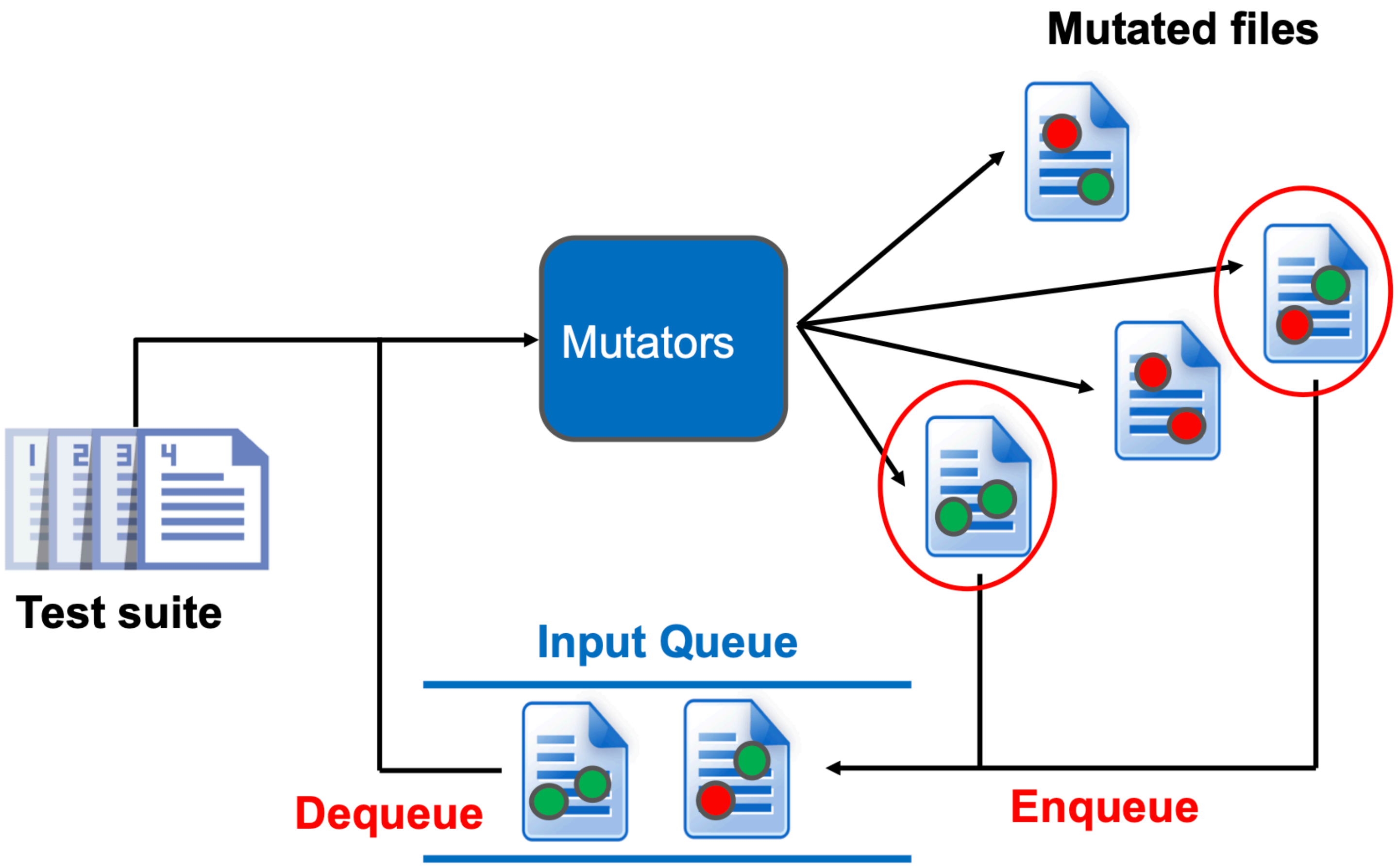
It finds bugs

IJG jpeg 1 libjpeg-turbo 1 2 libpng 1 libtiff 1 2 3 4 5 mozjpeg 1 libbpg (1) Mozilla Firefox 1 2 3 4 5 Google Chrome 1 Internet Explorer 1 2 (3) (4) LibreOffice 1 2 3 4 poppler 1 freetype 1 2 GnuTLS 1 GnuPG 1 2 (3) OpenSSH 1 2 3 bash (post-Shellshock) 1 2 tcpdump 1 2 3 4 5 6 7 Adobe Flash / PCRE 1 2 JavaScriptCore 1 2 3 4 pdfium 1 ffmpeg 1 2 3 4 libmatroska 1 libarchive 1 2 3 4 5 6 … wireshark 1 ImageMagick 1 2 3 4 5 6 7 8 … lcms (1) PHP 1 2 lame 1 FLAC audio library 1 2 libsndfile 1 2 3 less / lesspipe 1 2 3 strings (+ related tools) 1 2 3 4 5 6 7 file 1 2 dpkg 1 rcs 1 systemd-resolved 1 2 sqlite 1 2 3 libyaml 1 Info-Zip unzip 1 2 OpenBSD pfctl 1 NetBSD bpf 1 man & mandoc 1 2 3 4 5 … IDA Pro clamav 1 2 libxml2 1 glibc 1 clang / llvm 1 2 3 4 5 6 nasm 1 2 ctags 1 mutt 1 procmail 1 fontconfig 1 pdksh 1 2 Qt 1 wavpack 1 redis / lua-cmsgpack 1 taglib 1 2 3 privoxy 1 perl 1 2 3 4 5 6 libxmp radare2 1 2 fwknop metacam 1 exifprobe 1 capnproto 1

Intro to American Fuzzy Lop (AFL)



Grey-box Fuzzing, as in AFL



Space of Techniques

Search

- Random
- Biased-random
- Genetic (AFL Fuzzer)
- …

- *Low set-up overhead*
- *Fast, less accurate*
- **Use objective function to steer**

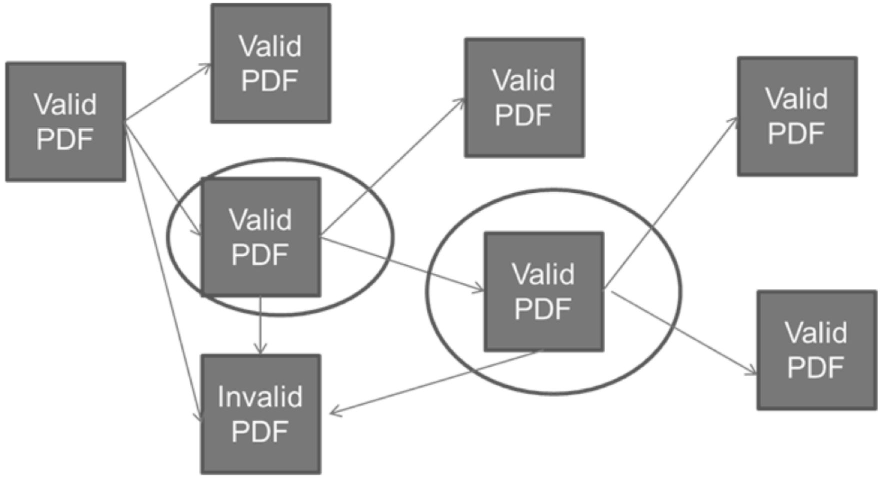
Symbolic Execution

- Dynamic Symbolic execution
- Concolic Execution
- Cluster paths based on symbolic expressions of variables
- ….

- *High set-up overhead*
- *Slow, more accurate*
- **Use logical formula to steer**

AFL Overview

- Input: Seed Inputs S
- 1: $T_{\times} = \varnothing$
- 2: T = S
- 3: if $T = \varnothing$ 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: t0 = **mutate_input**(t)
- 11: if t0 crashes then
- 12: add t0 to $T_{\times}$
- 13: else if **isInteresting**(t0) then
- 14: add t0 to T
- 15: end if
- 16: end for
- 17: until timeout reached or abort-signal
- Output: Crashing Inputs $T_{\times}$



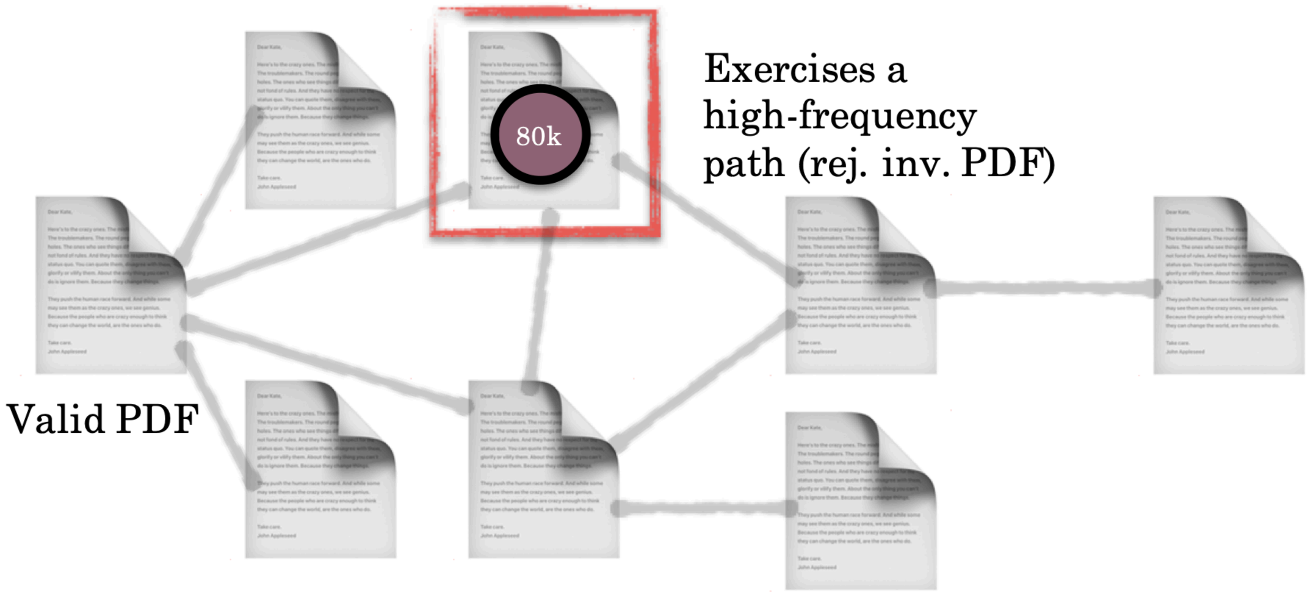
Exercises common path that rejects invalid PDF

Short paths rejecting syntactically invalid inputs are exercised by fuzzer too frequently

Core intuition

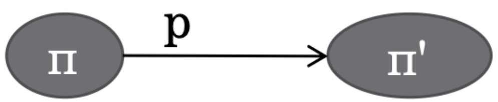
- AFL's power schedule is constant in the number of times s(i) the seed has been chosen for fuzzing

- AFL's power schedule always assigns *high* energy



Prioritize low probability paths

- ✓ Use grey-box fuzzer which keeps track of path id for a test.
- ✓ Find probabilities that fuzzing a test t which exercises π leads to an input which exercises π'



- ✓ Higher weightage to low probability paths discovered, to gravitate to those -> discover new paths with minimal effort.

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

Power Schedules

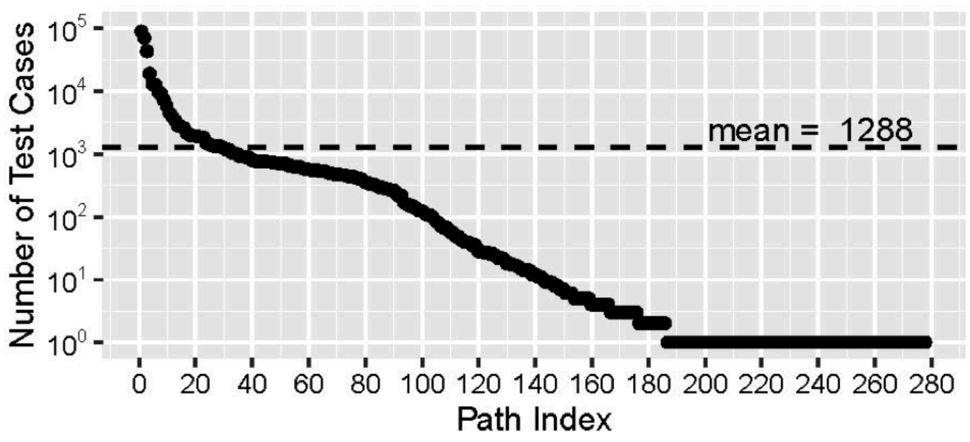
- Constant: $p(i) = \alpha(i)$
  - AFL uses this schedule (fuzzing ~1 minute)
  - $\alpha(i)$ .. how AFL judges fuzzing time for the test exercising path i

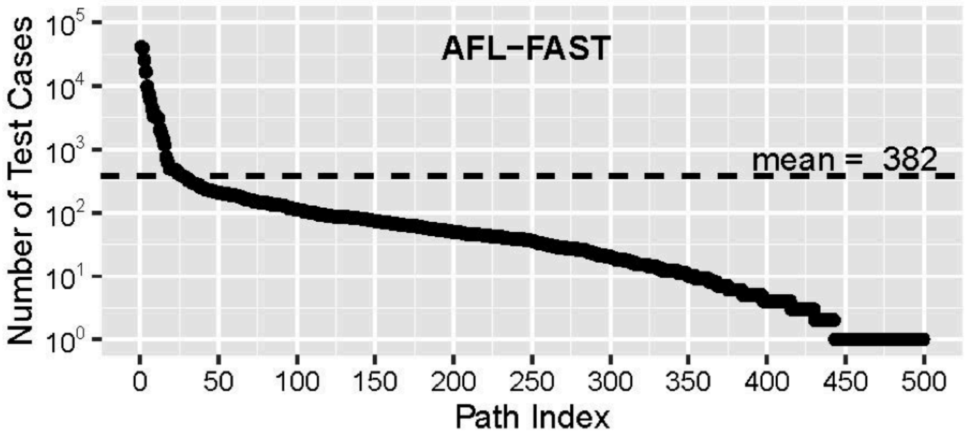
- Cut-off Exponential:

$$p(i) = \begin{cases} 0, & \text{if } f(i) > \mu \\ \min\left((\alpha(i)/\beta) * 2^{s(i)}, M \right) & \text{otherwise} \end{cases}$$

β is a constant
$s(i)$ #times the input exercising path i has been chosen for fuzzing
$f(i)$ #fuzz exercising path i (path-frequency)
μ mean #fuzz exercising a discovered path (avg. path-frequency)
M maximum energy expendable on a state

Results





Independent evaluation found crashes 19x faster on DARPA Cyber Grand Challenge (CGC) binaries

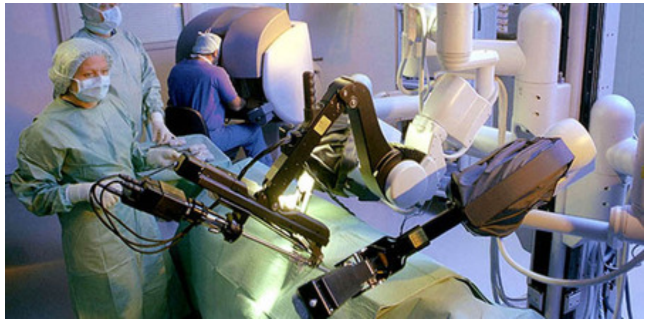
Integrated into main-line of AFL fuzzer within a year of publication (CCS16), which is used on a daily basis by corporations for finding vulnerabilities

Impact

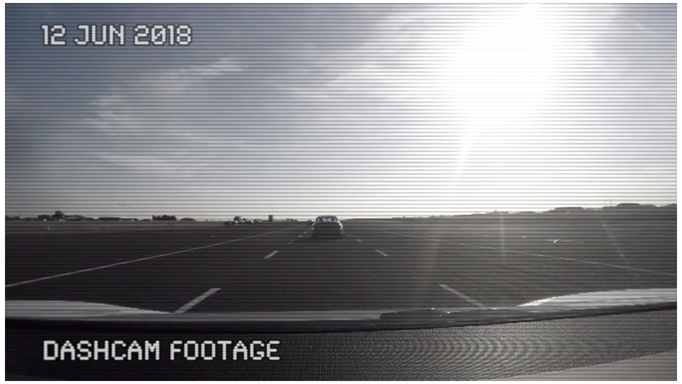
- Implemented inside AFL (version 2.33b, FidgetyAFL), and distributed approximately within one year of publication

Autonomous (AI-driven) Systems

Suite of AI-driven Systems



Resilience of Autonomous Vehicles



https://youtu.be/2W
M

https://youtu.be/jYkO7L
QC2jE

Search Bloomberg

Hyperdrive

Tesla Driver Died Using Autopilot, With Hands Off Steering Wheel

The New York Times

Self-Driving Uber Car Kills Pedestrian in Arizona, Where Robots Roam

WIRED Waymo's Self-Driving Car Crash in Arizona Revives Tough Quest

AARIAN MARSHALL AND ALEX DAVIES TRANSPORTATION 05.04.18 06:46 PM

SHARE

SHARE 396

TWEET

WAYMO'S SELF-DRIVING CAR CRASH IN ARIZONA REVIVES TOUGH QUESTIONS

Research Gap: Methods to assess end-to-end resilience, security & safety of AVs not available

Challenges and Opportunities

- Many of the functions/modules are ML algorithms consisting of back-to-back matrix multiplication
    - Coverage metric such as branch, statement, etc. do not make sense or have limited use

- Beyond hangs and crashes, the safety property includes collision, traffic rules etc.

- [Spatial resiliency] ML algorithms are inherently tolerant towards noise, and not all (random) inputs are useful

- [Temporal resilience] Physical state of such systems change over horizon of time, and ML algorithms can correct (compensate for) bad inputs/actions at time **T** in the next time-step **T+1**

Field Failure Analysis: Examining the Current State of AVs *[DSN 2018]*

Data driven analysis of failures in the field during testing of AVs

California Department of Motor Vehicles AV Testing Reports (2014 – 2016)

1,116,605 miles – **144** AVs – **12** Vendors
5328 Disengagements – **42** Accidents

❶ Disengagements

Human Initiated

AV Initiated

Failure Modes

Disengagement: A transfer of control from the autonomous system to the human driver in the case of a failure.

Accident: An collision with other vehicles, pedestrians, or property.

Quantified in terms of *disengagements per mile (DPM)* and *accident per mile (APM)*.

❷ Accidents

Field Failure Analysis: Examining the Current State of AVs *[DSN 2018]*

Results

Current AV tech in burn-in phase

- **ML/Design issues** account for 65% of failures
- 48% of disengagements are **human initiated**
- Volkswagen reported ~20% disengagements due to software hang/crashes

Comparing to Humans

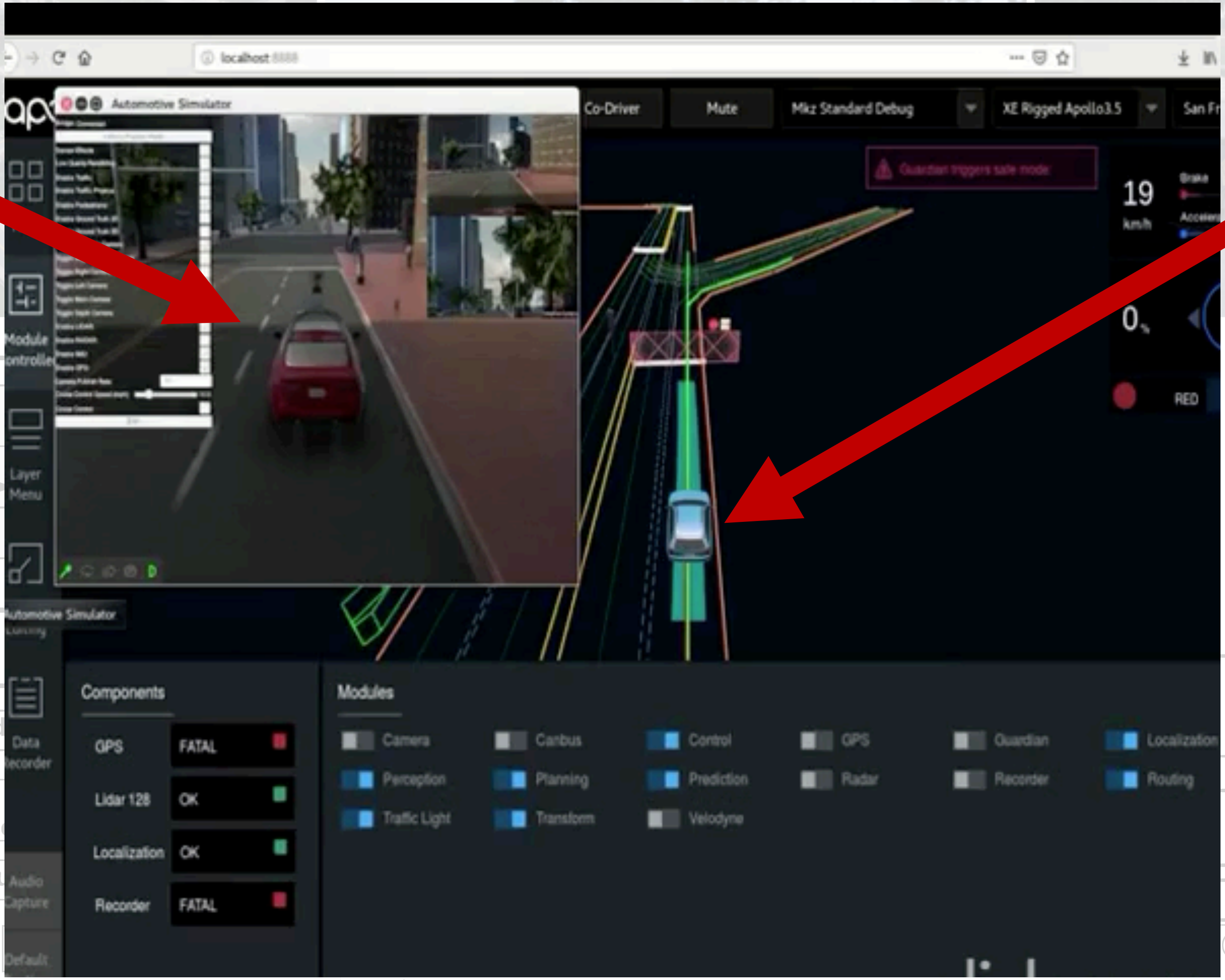
- Non-AVs are **15 – 4000×** less likely to have an accident
- All accidents reported at intersection of urban streets

Compared to other systems

- AVs are merely 4.22x worse than airplanes,
- 2.5x better than surgical robots

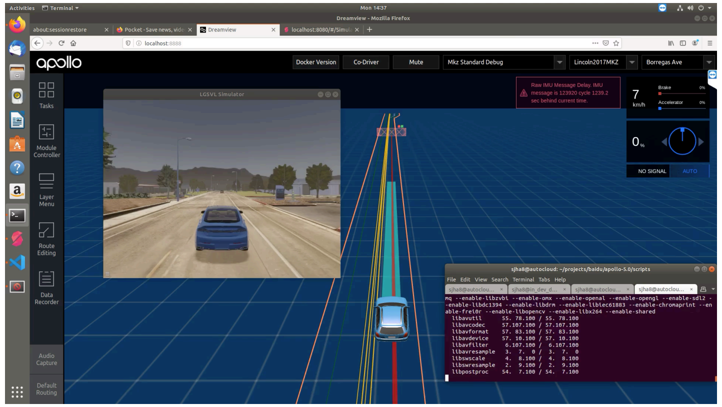
End-to-end Resilience and Safety Evaluation

AV Simulator View
- Unreal Engine or Unity-based
- Provides sensor data to AI-agent



AI-agent View
- Apollo (AI-agent) actions
- Provides actuation commands

Example Accidents



Faulty Input (bit-flip model)