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

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”
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
<table>
<thead>
<tr>
<th></th>
<th>Mutation-based</th>
<th>Generation-based</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Super easy to setup and automate</td>
<td>Writing generator is labor intensive for complex protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knowledge</td>
<td>Little to no protocol knowledge required</td>
<td>have to have spec of protocol (frequently not a problem for common ones http, snmp, etc...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complexity</td>
<td>Limited by initial corpus</td>
<td>Completeness</td>
<td></td>
<td>Can deal with complex checksums and dependencies</td>
</tr>
<tr>
<td></td>
<td>May fail for protocols with checksums, or other complexity</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
White-box Fuzzing

Cover more paths

\[ x \leq y \land x + y \leq 10 \]

\[ x \leq y \land \neg x + y \leq 10 \]

\( \neg x \leq y \)

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 @@
Intro to American Fuzzy Lop (AFL)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time</td>
<td>Cycles Done</td>
</tr>
<tr>
<td>Last New Path</td>
<td>Total Paths</td>
</tr>
<tr>
<td>Last Unique Crash</td>
<td>Uniq Crashes</td>
</tr>
<tr>
<td>Last Unique Hang</td>
<td>Uniq Hangs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now Processing</td>
<td>Map Density</td>
<td>Favored Paths</td>
</tr>
<tr>
<td>Now In Progress</td>
<td>Count Coverage</td>
<td>New Edges On</td>
</tr>
<tr>
<td>Stage ExeCs</td>
<td>Exec Speed</td>
<td>Total Crashes</td>
</tr>
<tr>
<td>Total ExeCs</td>
<td>Exec Speed</td>
<td>Total Hangs</td>
</tr>
<tr>
<td>Exec Speed</td>
<td>Exec Speed</td>
<td>Path Geometry</td>
</tr>
<tr>
<td>71.11/sec (slow!)</td>
<td>86.9k</td>
<td>Levels: 2</td>
</tr>
</tbody>
</table>

Fuzzing Strategy Yields:
- Bit Flips: 12/704, 1/700, 1/692
- Byte Flips: 0/88, 0/84, 0/76
- Arithmetics: 4/4840, 0/4068, 0/2495
- Known Ints: 1/404, 1/2333, 2/2842
- Dictionary: 0/0, 0/0, 0/16
- Havoc: 9/65.6k, 0/0
- Trim: 8.33%/20, 0.00%
Grey-box Fuzzing, as in AFL

- Mutators
- Test suite
- Input Queue
- Mutated files
- Dequeue
- Enqueue
Space of Techniques

Search
- Random
- Biased-random
- Genetic (AFL Fuzzer)
- ...

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

- Input: Seed Inputs S
- 1: $T_{x} = \varnothing$
- 2: $T = S$
- 3: if $T = \varnothing$ then
- 4: add empty file to $T$
- 5: end if
- 6: repeat
- 7: $t = \text{chooseNext}(T)$
- 8: $p = \text{assignEnergy}(t)$
- 9: for $i$ from 1 to $p$ do
- 10: $t_0 = \text{mutate_input}(t)$
- 11: if $t_0$ crashes then
- 12: add $t_0$ to $T_{x}$
- 13: else if $\text{isInteresting}(t_0)$ then
- 14: add $t_0$ to $T$
- 15: end if
- 16: end for
- 17: until timeout reached or abort-signal
- Output: Crashing Inputs $T_{x}$
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

Exercises a high-frequency path (rej. inv. PDF)

Valid PDF
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 $\pi$ leads to an input which exercises $\pi'$

\[
\pi \xrightarrow{p} \pi'
\]

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

```c
1  void crashme (char* s) {
2    if (s[0] == 'b')
3      if (s[1] == 'a')
4        if (s[2] == 'd')
5          if (s[3] == '!')
6            abort();
7  }
```
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:**

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

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

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

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).
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 – 4000x 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

Quantified in terms of disengagements per mile (DPM) and accident per mile (APM).
End-to-end Resilience and Safety Evaluation

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

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

**Campaign Manager**
- Mode Selection
  - ML Data
  - SW HW

**Injection Plan Generator**
- Event-Driven Sync. Module
- (CPUs, GPUs)

**Components**
- GPS
- LIDAR
- Camera
- Object Perception
- Path Perception
- Sensor Fusion
- Camera
- LIDAR
- GPS
- Lidar 128
- Localisation
- OK
- FATAL
- OK
- Recorder
- Control
- Perception
- Planning
- Prediction
- Radar
- Recorder
- Localization
- Routing
- Traffic Light
- Transform
- Velodyne
Example Accidents

Faulty Input (bit-flip model)