CSE 523S
System Security

Fuzzing

Spring 2022

Patrick Crowley

Original slides from Lyall Cooper's MS project, updated by Hila Ben Abraham, Jon Shidal, John DeHart, and Heng Yin

Why So Many Vulnerabilities?

Why So Many Vulnerabilities?

- BUGS!!
  - Remember how we started the semester
- Assuming we can’t avoid bugs, what can we do?
  - Find them on time.
- This requires TESTING!!

Software Testing in a Box

- White box testing
  - Often "unit" testing, static analysis, code coverage
  - Tests internal parts of program individually
  - You choose what gets tested, you often test your own programs based on how you “know” they should work.
- Black box testing
  - Assume you don't know/care how program internals work
  - Tests what a program does, not how it does it
  - Provide input, ensure output is correct and no errors
- Both are important!

What is Fuzzing?

- Fuzzing is a form of vulnerability analysis.
- Process: an automated process that injects many anomalous test cases into a program.
  - Test cases may include invalid, unexpected, or random data
  - The tested application is monitored for errors (outputs, crashes, or potential memory leaks)

Example

Standard HTTP GET request

- § GET /index.html HTTP/1.1

Anomalous requests

- § AAAAAA...AAAA /index.html HTTP/1.1
- § GET ///////index.html HTTP/1.1
- § GET %n%n%n%n%n%n.html HTTP/1.1
- § GET /AAAAAAAAAAAAA.html HTTP/1.1
- § GET /index.html HTTTTTTTTTTTTTP/1.1
- § GET /index.html HTTP/1.1.1.1.1.1.1.1
- etc...

Unit Testing vs Fuzzing

- Unit testing or Regression testing:
  - Run the program on many normal inputs and edge cases.
  - **Goal:** ensure expected behavior and good user experience by finding bugs.
- Fuzzing
  - Run the program with many abnormal inputs.
  - **Goal:** prevent attackers from encountering exploitable errors, by pointing developers to areas of the program that need attention.

Pros and Cons of Fuzzing

- Pros:
  - Great at finding memory/safety/error handling bugs
  - Can be fully automated, easily run
  - Find bugs usually missed by humans
- Cons:
  - Hard to do well
  - Only finds certain classes of bugs
  - Can take a long time
    - If input is n bytes long, 256^n possible inputs!
  - Doesn't test program correctness

Types of Fuzzers

- Mutation Based - “Dumb Fuzzing”
  - Mutate existing data samples to create test data
- Generation Based - “Smart Fuzzing”
  - Define new tests based on models of the input.
- Evolutionary
  - Generate inputs based on response from program

Mutation Based Fuzzing

- Mutative takes starting input and makes changes (mutates) to it to generate many inputs.
  - Little or no knowledge of the structure of the inputs is assumed
  - Modifications can be random or follow heuristics
- Easy to set up.
- May fail for protocols with checksums, or apps that depend on challenge response, etc.
- Example, take a pdf file as an input, modify it and send to the pdf client app

Generation Based Fuzzing

- Takes specification, generates input based on it
  - Based on RFC and other types documentation.
- Anomalies are added to each possible input
  - Requires some knowledge about the tested application.
  - May take time to set up.

Generation Based Fuzzing - Example

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);     // Width
                s_push_int(0x14, 1);     // Height
                s_push_int(0x8, 3);      // Bit Depth - should be 1,2,4,8,16, base
                s_push_int(0x3, 3);      // ColorType - should be 0,2,3,4,6
                s_binary("00 00");       // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);      // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Sample PNG spec

How Much Fuzzing Is Enough?

- Mutation-based-fuzzers may generate an infinite number of test cases.
  - When has the fuzzer run long enough?
- Generation-based fuzzers may generate a finite number of test cases.
  - What happens when they’re all run and no bugs are found?

- Code coverage techniques can help answering those questions.

Code Coverage

- Code coverage is a metric that can be used to determine how much code has been executed.
- This can help us quantify the fuzzing process, determine the starting input, and to decide which or how many fuzzers we want to use.
- Different measures are
  - Line coverage: how many lines of source code have been executed.
  - Branch coverage: how many branches in code have been taken (conditional jmps)
  - Path coverage: how many paths have been taken

- Code coverage is not perfect, but is often used to guide fuzzing tools

Software Testing in a Box

https://docs.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10)?redirectedfrom=MSDN

- **Whitebox fuzzing:**
  - “Sending of malformed data with verification that all target code paths were hit”
- **Blackbox fuzzing:**
  - “Sending of malformed data without actual verification of which code paths were hit and which were not”

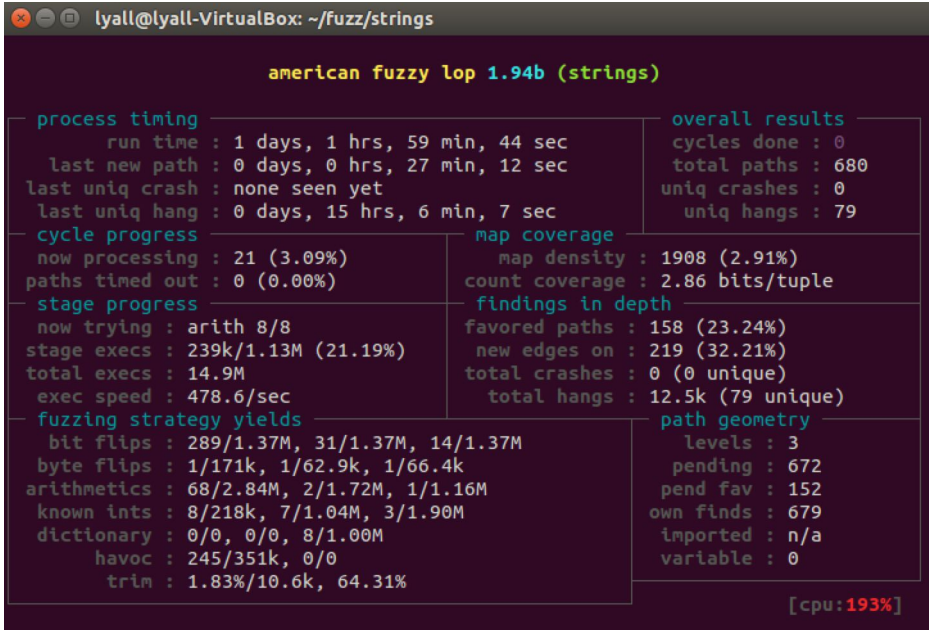
Technique	Effort	Code coverage	Defects found
Combination of black box + dumb	10 min	50%	25%
Combination of white box + dumb	30 min	80%	50%
Combination of black box + smart	2 hr	80%	50%
Combination of white box + smart	2.5 hr	99%	100%

Coverage-guided Greybox Fuzzing

- Special type of mutation-based fuzzing
  - Run mutated inputs on instrumented program and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use genetic algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases
  - Examples: AFL, libfuzzer

American fuzzy lop

- aka *afl-fuzz*
- Coverage-guided **gray-box** fuzzing
- "... is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary." - http://lcamtuf.coredump.cx/afl/
- Requires minimal setup, just compile and provide example inputs



afl-fuzz bug trophy case

- libjpeg-turbo
- libpng
- libtiff
- mozjpeg
- sqlite
- ffmpeg
- ntpd
- OpenSSL
- OpenSSH
- tcpdump
- wireshark
- Firefox
- Internet Explorer
- Safari
- Adobe Flash
- bash
- Android
- iOS
- LibreOffice
- … many more!
- see http://lcamtuf.coredump.cx/afl/#bugs

How does it work?

- Broadly, it attempts to get the program into unusual states:

1. Compiles program with gcc wrapper (afl-gcc) that adds custom instrumentation to compiled binary
2. Creates new test cases by mutating test cases currently in queue
3. Runs inputs likely to get program into new state (as based off instrumentation)
4. Minimizes and saves test cases that reached new states
5. Repeat 2-5

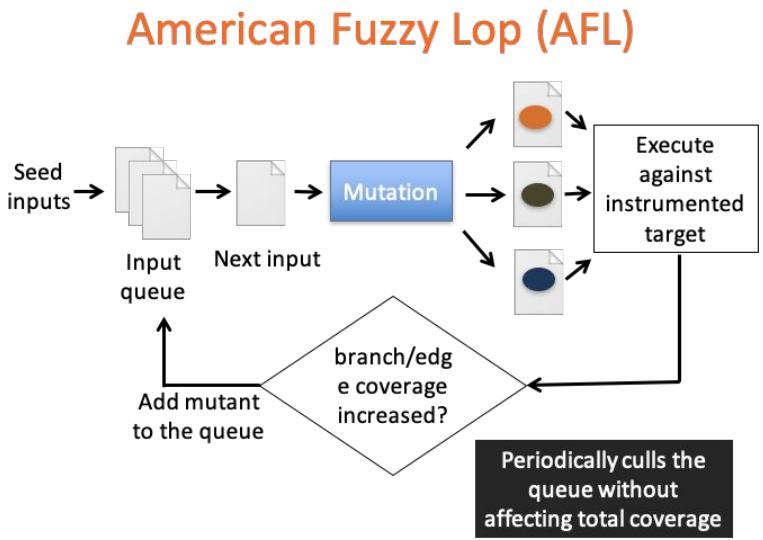


Figure by Suman Jana

Basic afl-fuzz operation steps

1. Get target program source
2. Compile with afl-gcc
3. Choose starting test case(s)
   a. Smaller is almost always better
   b. afl source comes with many good starting test cases
4. Start afl-fuzz
5. Wait!
6. Investigate results

Some of the provided test cases:

js:

```
if (1==1) eval('1');
```

rtf:

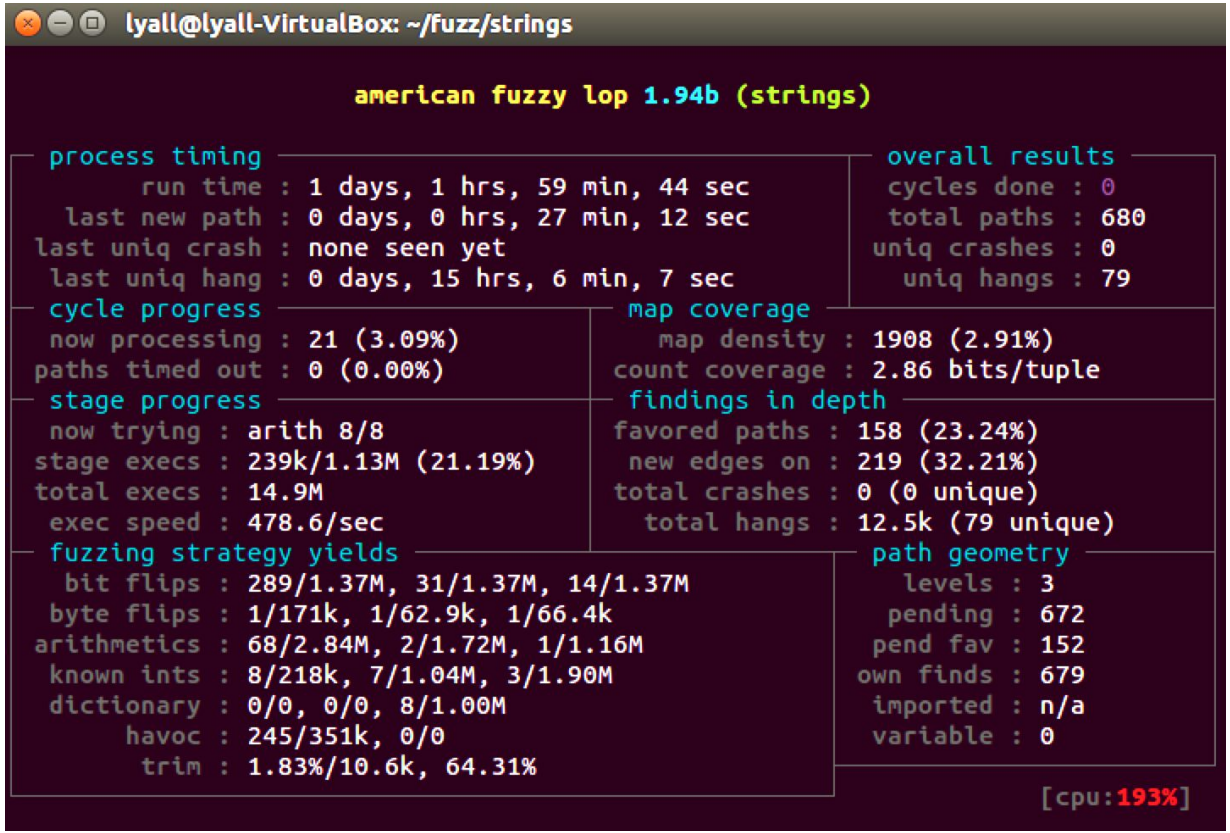
```
Test
```

xml:

```
<a b="c">d</a>
```

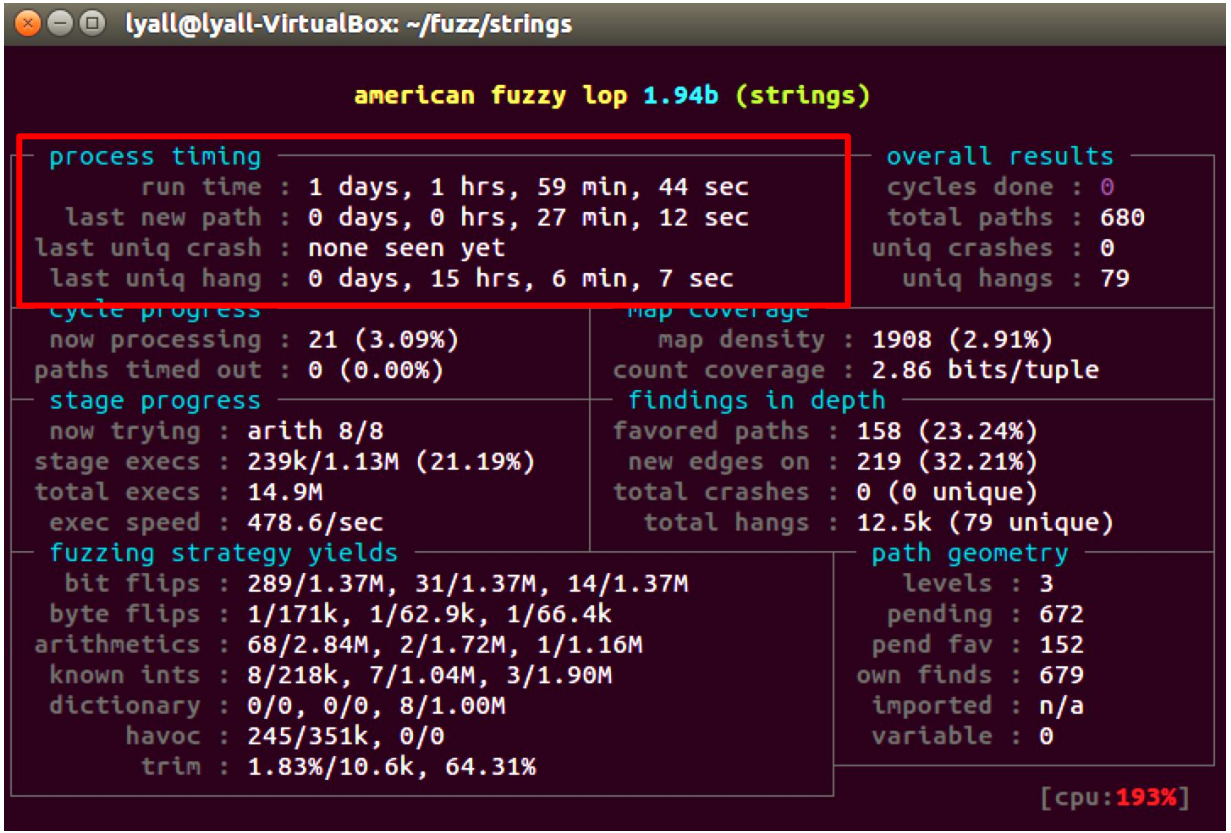
png (only 218 bytes):

Understanding the status screen



Understanding the status screen

Process Timing:
How long has it been running?
How long between discoveries?



Understanding the status screen

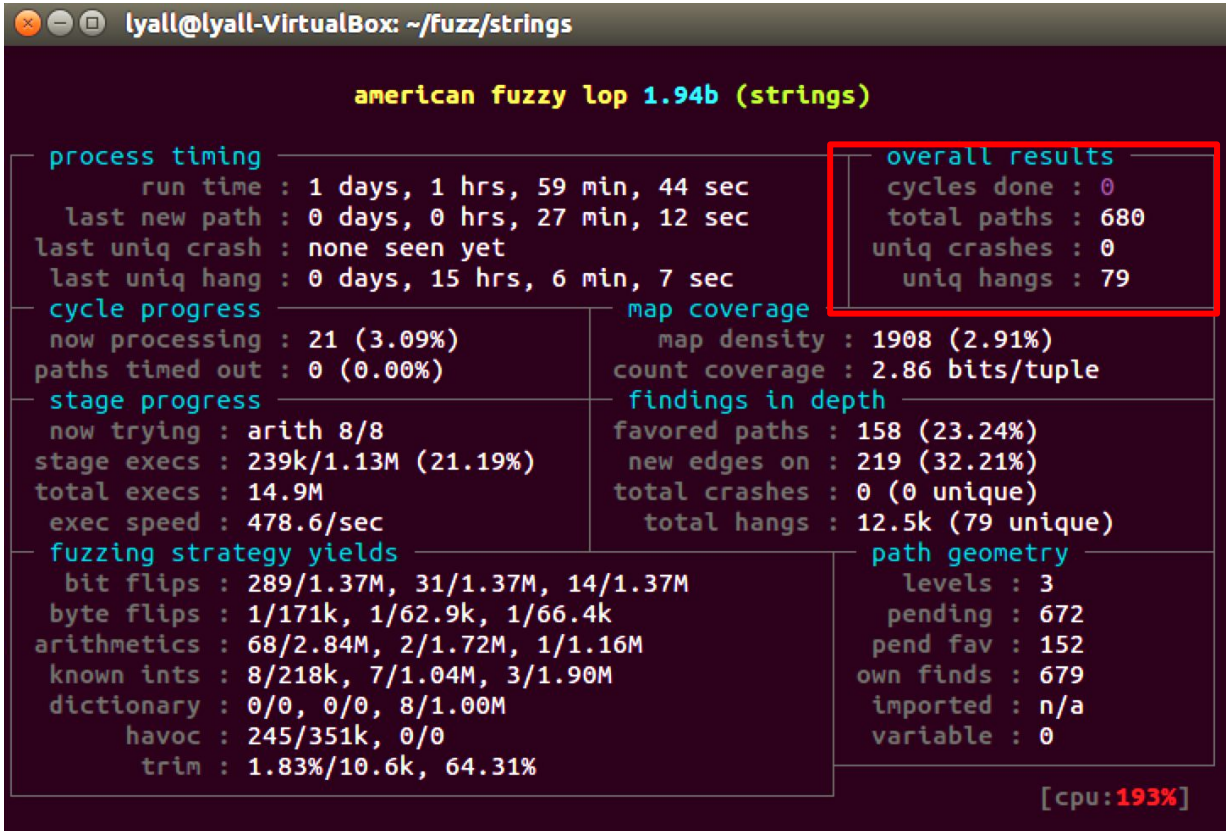
Results

Cycle: # of passes over test cases discovered.

Paths: test cases

Crashes:

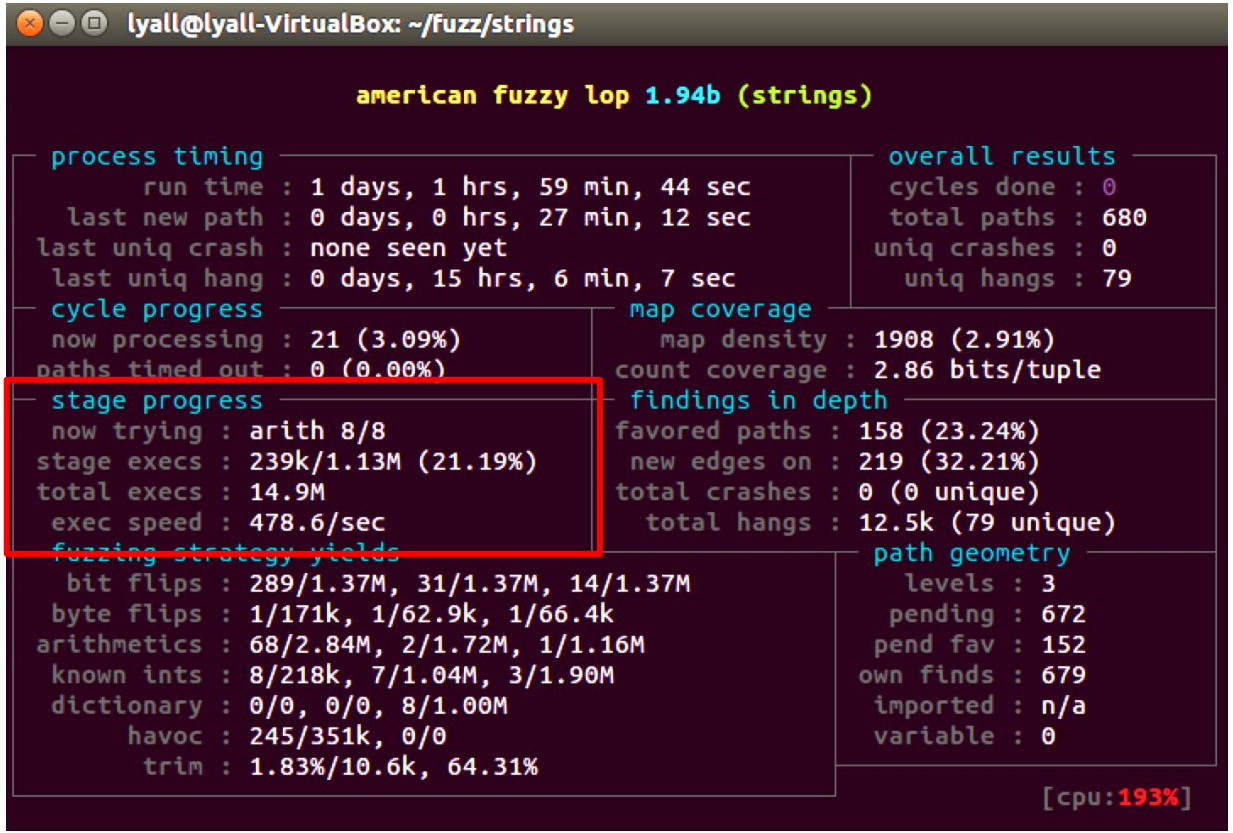
Hangs:



Understanding the status screen

Stage

What is it doing right now.



Understanding the status screen

Strategy

Fuzzing can use different strategies for building its tests.