Fuzzing

Mustakimur R. Khandaker

Fuzzing

Fuzzing is an automated software testing technique

- Generate invalid, unexpected, or random inputs to the program.
  Inputs are often file based or network based.
- The program is monitored for errors, Crashes, assertions, sanitizers…

Dumb fuzzing:

- Blindly mutate existing valid inputs.

Smart fuzzing:

- **Generation based:** generate inputs according to protocol specification.
- **Guided fuzzing:** collect feedback to guide the next round of mutation.

Mutation Based Fuzzing

Little or no knowledge of the structure of the inputs is assumed.

Anomalies are added to existing valid inputs.

Anomalies may be completely *random or follow some heuristics*.

Examples:

- **interest:** -1, 0x8000000, 0xffff, etc.
- **bitflip:** flipping 1,2,3,4,8,16,32 bits.
- **havoc:** random tweak in fixed length.
- **extra:** dictionary, remove Null, etc..

Example Tools:

- Taof, GPF, ProxyFuzz, Peach Fuzzer, etc.

Example: Fuzzing a PDF Viewer

- Google for .pdf (about 1 billion results).
- Crawl pages to build a corpus.
- Use fuzzing tool (or script to).
  - Grab a file.
  - Mutate that file.
  - Feed it to the program.
  - Record if it crashed (and input that crashed it).

Advantage & Disadvantage

```
bool is_ELF(Elf32_Ehdr eh)

{

    /* ELF magic bytes are 0x7f,'E','L','F'

     * Using  octal escape sequence to represent 0x7f

     */

    if(!strncmp((char*)eh.e_ident, "\177ELF", 4)) {

        printf("ELFMAGIC \t= ELF\n");

        /* IS a ELF file */

        return 1;

    } else {

        printf("ELFMAGIC mismatch!\n");

        /* Not ELF file */

        return 0;

    }

}
```

Strengths

- Super easy to setup and automate.
- Little to no protocol knowledge required.

Weaknesses

- Limited by initial corpus.
- Limited code coverage.
- May fail for protocols with checksums, those which depend on challenge response, etc.

Generation-based Fuzzing

Test cases are generated from specification of input format.

- e.g., RFC, documentation, etc.
- Wireshark has a whole family of protocol specification.

Anomalies are added to each possible spot in the inputs.

```
//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, based on colortype
                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");
...
```

Knowledge of protocol should give better results than random fuzzing.

Continue …

```
<?xml encoding="utf-8"  version="1.0" ?>
<test>
<xslo:variable test="self::xhtml:pre"
select="seda:Contains/seda:Appraisal"  test="$xml2rfc-ext-strip-vbare='true'" />
<x:elements-xslt>     <vra-p:versionOf>     <xsl:copy>     &#x2203;     <wot:signed>
<x:param select="@dur2" />     &#x39D;     </wot:signed>     <jabberID>
<errorname>  <vra-p:versionOf>     &#x3D2;     &#x3B5;     &#x3B2;      
</vra-p:versionOf>  <w:endnote test="$generate.component.toc != 0" >
&#x2592;     &pt235;     <xsl:value-of test="empty($jpackageDoc)" />
&le;     &olarr;     </w:endnote>  </errorname>     <xsl:copy>
<body>     &pt456;     &#x251c;     &#x0342;     </body>
<seeie test="$mode = 'fit'" >     &pt13456;     &#xF4;
<people>  <axsl:choose>  <vra-p:versionOf>     &phi;     &#x03C1;
&#x202A;
</vra-p:versionOf>    </axsl:choose>    </people>
</seeie>  </xsl:copy>           </jabberID>        </xsl:copy>
</vra-p:versionOf>        </x:elements-xslt>    </test>
```

FIGURE 1: XML Fuzzing File Generated by Skyfire.

Continue ...

Strengths

- Completeness.
- Can deal with complex dependencies e.g. checksums.

Weaknesses

- Have to have spec of protocol.
  - It is possible to automatically extract protocol spec from program.
- Writing generator can be labor intensive for complex protocols.
- The spec is not the code.

White Box vs. Black Box Fuzzing

Black box fuzzing: sending the malformed input without any verification of the code paths traversed.

White box fuzzing: sending the malformed input and verifying the code paths traversed. Modifying the inputs to attempt to cover all code paths.

Technique	Effort	Code coverage	Defects Found
black box + mutation	10 min	50%	25%
black box + generation	30 min	80%	50%
white box + mutation	2 hours	80%	50%
white box + generation	2.5 hours	99%	100%

Guided Fuzzing

Dumb fuzzing often hits the same code again and again (Black box) without real progress.
Generation based – stop eventually.

Guided fuzzing **collects feedback** to guide future test case generation.

- e.g., code-coverage guided fuzzing tries to explore new code with each new generated input.

Code coverage is a metric to determine how much code has been executed.

American Fuzzy Lop (AFL) is the most popular guided fuzzing tool.

Code Coverage Guided Fuzzing

AKA grey-box fuzzing.

A type of evolutionary fuzzing:

- Monitor the program execution to obtain the code coverage.
- Synthesize new corpora based on the code coverage improvement.

Mutate the inputs that explored the most new code first.

Evolutionary fuzzing can still stuck.

- Use static analysis to give fuzzing a “push” (white-box fuzzing).
  - e.g., symbolic execution / constraint solver (e.g., Driller).

Types of Code Coverage

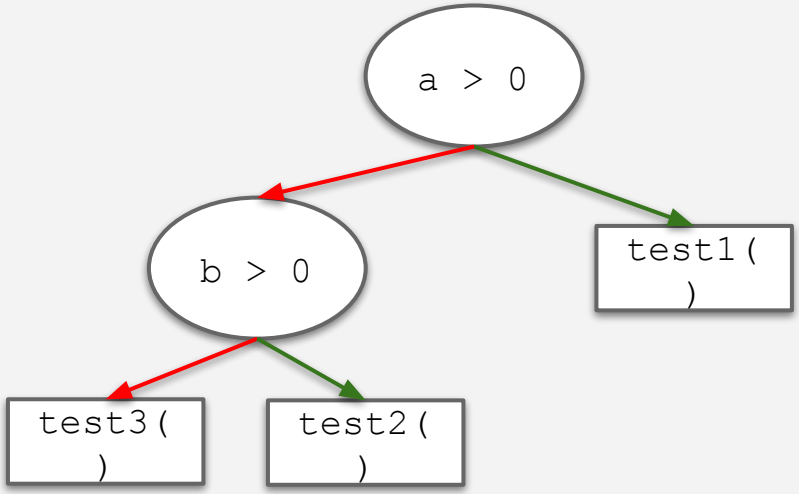
Line coverage
- Measures how many lines of source code have been executed.

Branch coverage
- Measures how many branches in code have been taken (conditional jmps).

Path coverage
- Measures how many paths have been taken.

Example

```
if (a > 0){
 test1();
} else {
 if (b > 0)
   test2();
 else
   test3();
}
```



- How many test cases for 100% line coverage?
- How many test cases for 100% branch coverage?
- How many test cases for 100% paths?

Fuzzing Resources

AFL (American fuzzy lop):

- american fuzzy lop (2.52b).

Syzkaller: unsupervised coverage-guided kernel fuzzer.

- google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer.

Driller: augmenting AFL with symbolic execution!

- shellphish/driller: Driller: augmenting AFL with symbolic execution!.
- used in Cyber Grand Challenge.

Awesome fuzzing:

- secfigo/Awesome-Fuzzing: A curated list of fuzzing resources (Books, courses - free and paid, videos, tools, tutorials and vulnerable applications to practice on) for learning Fuzzing and initial phases of Exploit Development like root cause analysis..

Fuzzing Rules of Thumb

Protocol specific knowledge is very helpful.
- Generational tends to beat random, better spec's make better fuzzers.

More fuzzers are better.
- Each implementation will vary, different fuzzers find different bugs.

The longer you run, the more bugs you find.

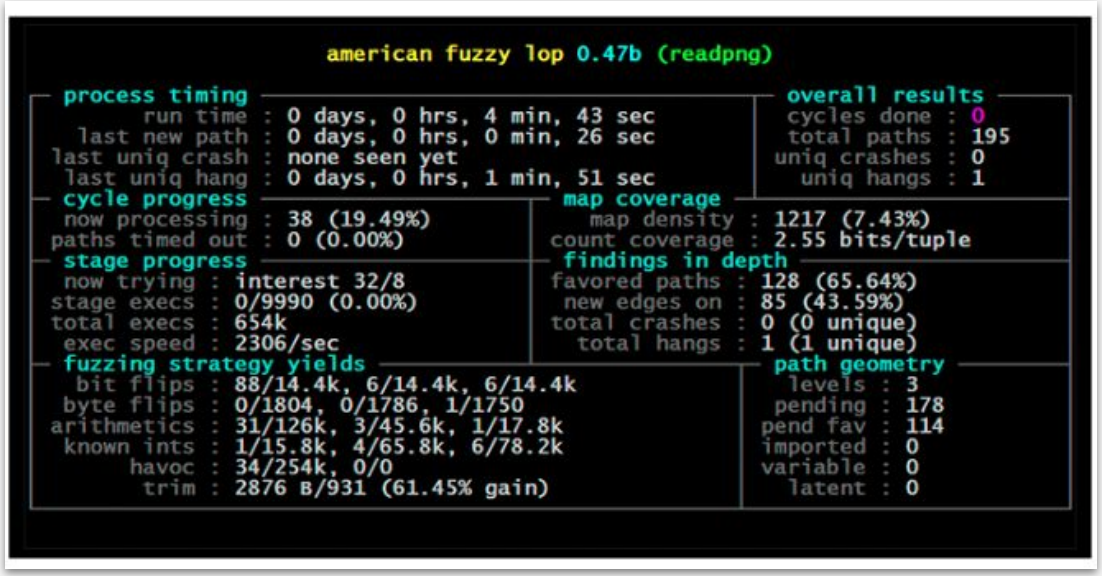
Best results come from guiding the process.
- Code coverage can be very useful for guiding the process.

AFL

american fuzzy lop (2.52b)
AFL (american fuzzy lop) — AFL 2.53b documentation

Code coverage guided genetic fuzzer.

- A set of carefully research rules to mutate the inputs.
- It can synthesizing complex file semantics.
- It has a crash explorer, test case minimizer, and fault-triggering allocator, and syntax analyzer.



Key Idea

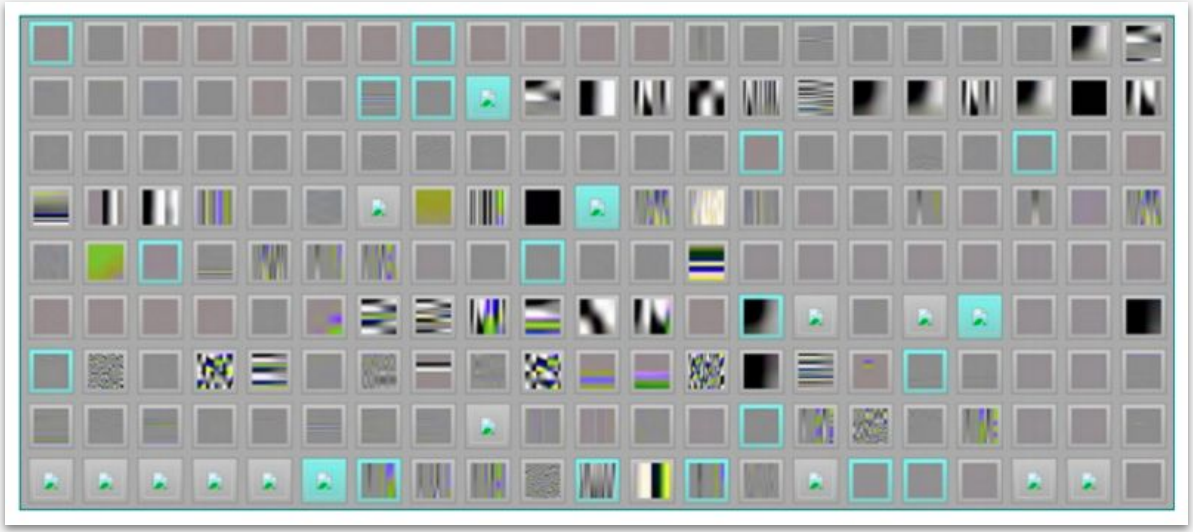
https://lcamtuf.coredump.cx/afl/technical_details.txt

Mapping input to state transitions.

- Instrumentation: Compiler (Source Code) or QEMU (Binary).

Avoiding redundant paths.

- If you see the duplicated state, throw out.
- If you see the new path, keep it for further exploration.



AFL Operations

Mutation strategies

- Highly deterministic at first – bit flips, add/sub integer values, and choose interesting integer values.
- Then, non-deterministic choices – insertions, deletions, and combinations of test cases.

afl-cmin takes a given folder of potential test cases, then runs each one and compares the feedback it receives to all rest of the testcases to find the best testcases which most efficiently express the most unique code paths.

afl-tmin works on only a specified file to avoid wasting CPU cycles fiddling with bits and bytes that are useless relative to the code paths the testcase might express.

Setup & Build

AFL User Guide — AFL 2.53b documentation

- Download and build AFL.
- Create initial test cases, the seeds.
- Build the program with afl-gcc/afl-g++ and run AFL.

```
$ cd afl-demo
$ mkdir testcases
$ cd $_
$ echo "your first test input" >01.txt
$ echo "your second test input" >02.txt
```

```
$ cd afl-demo
$ mkdir aflbuild
$ cd $_
$ CC=/path/to/afl/afl-2.51b/afl-gcc CXX=/path/to/afl/afl-2.51b/afl-g++ cmake ..
$ make
$ /path/to/afl/afl-2.51b/afl-fuzz -i ../testcases -o ../findings ./afldemo
```

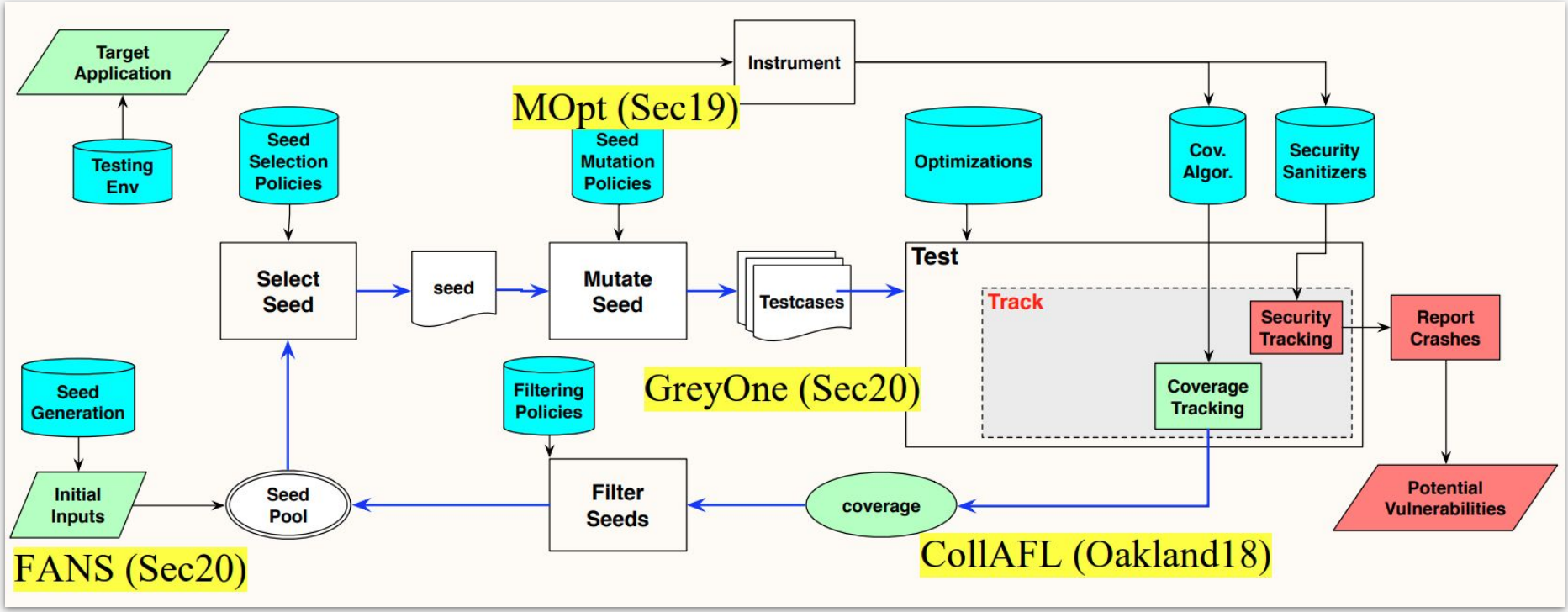
AFL Output

Shows the results of the fuzzer.

- e.g., provides inputs that will cause the crash.

- File **“fuzzer_stats”** provides summary of stats – UI.
- File **“plot_data”** shows the progress of fuzzer.
- Directory **“queue”** shows inputs that led to paths.
- Directory **“crashes”** contains input that caused crash.
- Directory **“hangs”** contains input that caused hang.

Fuzzing Research



HappY HackinG