Fuzzing

Suman Jana

*Acknowledgements: Dawn Song, Kostya Serebryany, Peter Collingbourne

Techniques for bug finding

Automatic test case generation Static analysis Program verification

Fuzzing

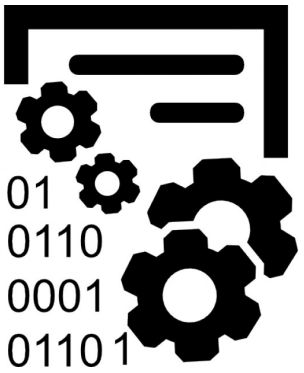
Dynamic symbolic execution

Lower coverage
Lower false positives
Higher false negatives

Higher coverage
Higher false positives
Lower false negatives

Blackbox fuzzing

Random input



Test program

Miller et al. ‘89

Blackbox fuzzing

- Given a program simply feed random inputs and see whether it exhibits incorrect behavior (e.g., crashes)
- Advantage: easy, low programmer cost
- Disadvantage: inefficient
  - Inputs often require structures, random inputs are likely to be malformed
  - Inputs that trigger an incorrect behavior is a a very small fraction, probably of getting lucky is very low

Fuzzing

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, etc.) or network based (http, SNMP, etc.)

Problem detection

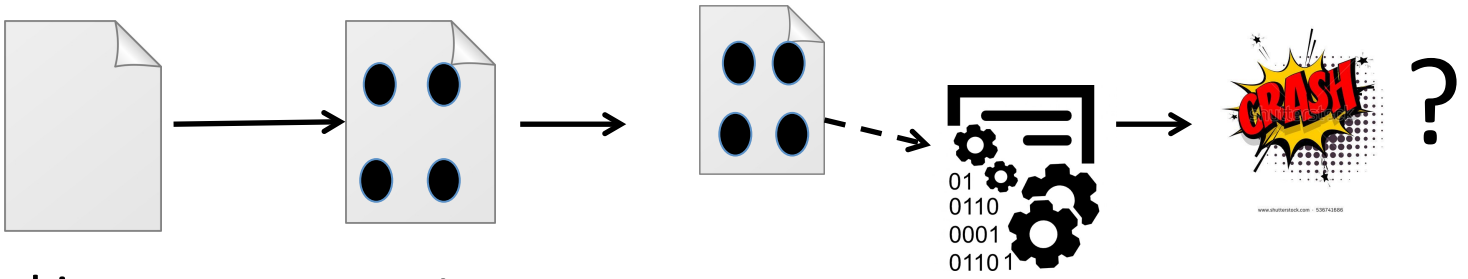
- See if program crashed
  - Type of crash can tell a lot (SEGV vs. assert fail)
- Run program under dynamic memory error detector (valgrind/purify/AddressSanitizer)
  - Catch more bugs, but more expensive per run.
- See if program locks up
- Roll your own dynamic checker e.g. valgrind skins

Regression vs. Fuzzing

	Regrssion	Fuzzing
Definition	Run program on many normal inputs, look for badness	Run program on many abnormal inputs, look for badness
Goals	Prevent normal users from encountering errors (e.g., assertion failures are bad)	Prevent attackers from encountering exploitable errors (e.g., assertion failures are often ok)

Enhancement 1: Mutation-Based fuzzing

- Take a well-formed input, randomly perturb (flipping bit, etc.)
- 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 (e.g., remove NULL, shift character forward)
- Examples: ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Example: fuzzing a PDF viewer

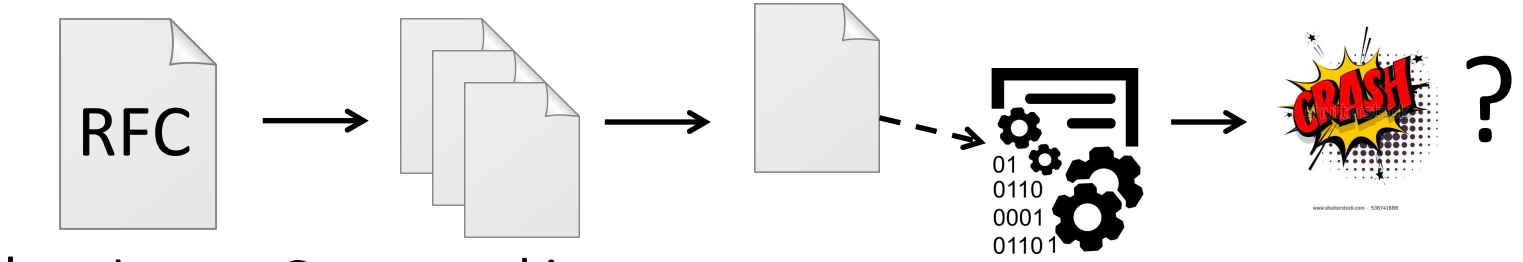
- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script)
  - Collect seed PDF files
  - Mutate that file
  - Feed it to the program
  - Record if it crashed (and input that crashed it)

Mutation-based fuzzing

- Super easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus
- May fail for protocols with checksums, those which depend on challenge

Enhancement II: Generation-Based Fuzzing

- Test cases are generated from some description of the input format: RFC, documentation, etc.
  - Using specified protocols/file format info
  - E.g., SPIKE by Immunity
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



Enhancement II: Generation-Based Fuzzing

```
//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

Mutation-based vs. Generation-based

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fall for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: Completeness, can deal with complex dependncies (e.g, checksum)
  - Cons: writing generators is hard, performance depends on the quality of the 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

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric that can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. gcov, lcov

Line coverage

- **Line/block coverage**: Measures how many lines of source code have been executed.
- For the code on the right, how many test cases (values of pair (a,b)) needed for full(100%) line coverage?

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```

Branch coverage

- Branch coverage: Measures how many branches in code have been taken (conditional jmps)
- For the code on the right, how many test cases needed for full branch coverage?

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```

Path coverage

- Path coverage: Measures how many paths have been taken
- For the code on the right, how many test cases needed for full path coverage?

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```

Benefits of Code coverage

- Can answer the following questions
  - How good is an initial file?
  - Am I getting stuck somewhere?

```
if (packet[0x10] < 7) { //hot path
} else { //cold path }
```

  - How good is fuzzerX vs. fuzzerY
  - Am I getting benefits by running multiple fuzzers?

Problems of code coverage

- For:

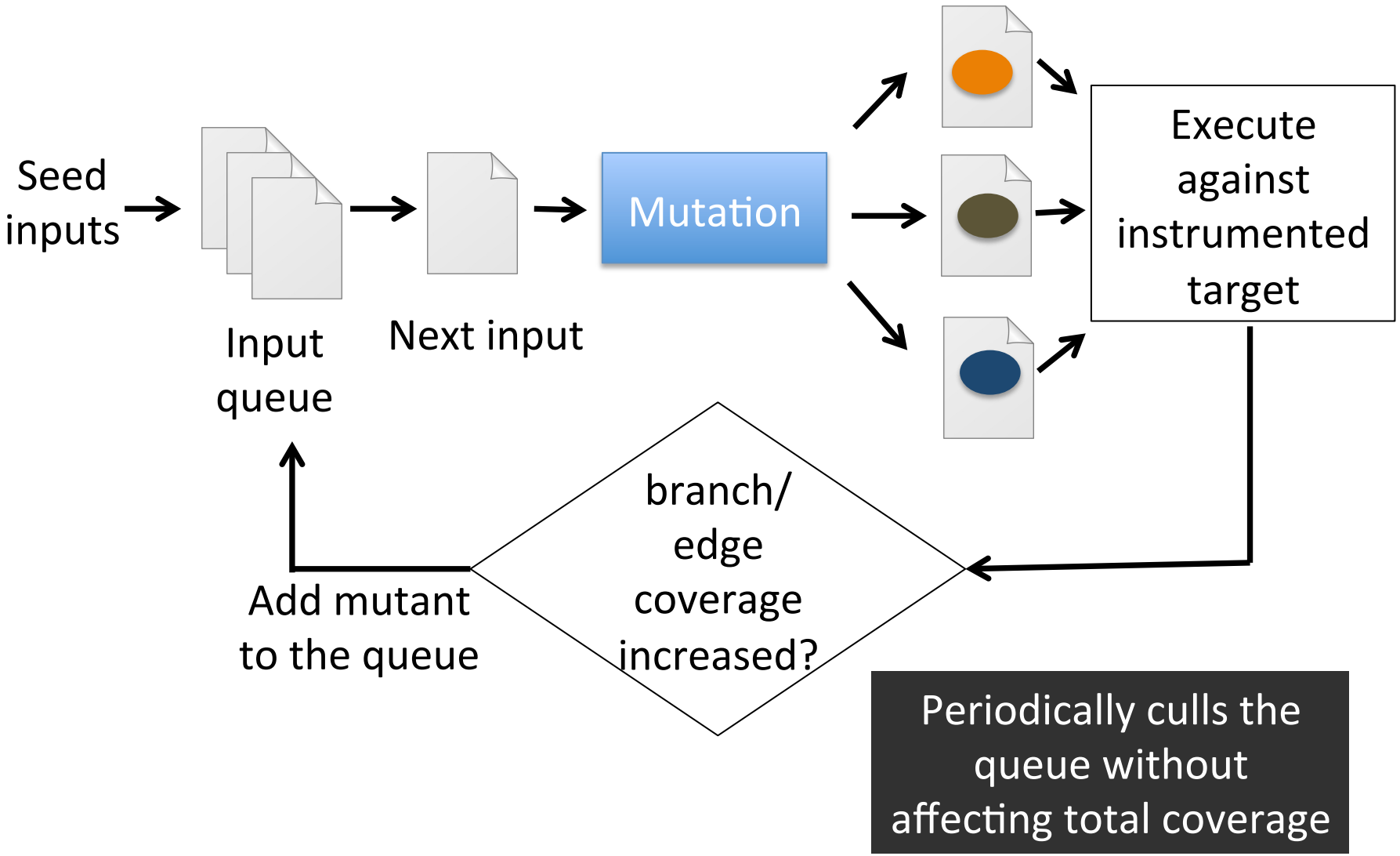
```
mySafeCopy(char *dst, char* src) {
    if(dst && src)
        strcpy(dst, src); }
```

- Does full line coverage guarantee finding the bug?
- Does full branch coverage guarantee finding the bug?

Enhancement III: Coverage-guided gray-box 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 (AFL)



AFL

- Instrument the binary at compile-time
- Regular mode: instrument assembly
- Recent addition: LLVM compiler instrumentation mode
- Provide 64K counters representing all edges in the app
- Hashtable keeps track of # of execution of edges
  - 8 bits per edge (# of executions: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+)
  - Imprecise (edges may collide) but very efficient
- AFL-fuzz is the driver process, the target app runs as separate process(es)

Data-flow-guided fuzzing

- Intercept the data flow, analyze the inputs of comparisons
  - Incurs extra overhead
- Modify the test inputs, observe the effect on comparisons
- Prototype implementations in libFuzzer and go-fuzz

Fuzzing challenges

- How to seed a fuzzer?
  - Seed inputs must cover different branches
  - Remove duplicate seeds covering the same branches
  - Small seeds are better (Why?)

- Some branches might be very hard to get past as the # of inputs statisfying the conditions are very small
  - Manually/automatically transform/remove those branches

Hard to fuzz code

```
void test (int n) {
 if (n==0x12345678)
    crash();
}
```

needs 2^32 or 4 billion attempts
In the worst case

Make it easier to fuzz

```
void test (int n) {
  int dummy = 0;
  char *p = (char *)&n;
  if (p[3]==0x12) dummy++;
  if (p[2]==0x34) dummy++;
  if (p[1]==0x56) dummy++;
  if (p[0]==0x56) dummy++;
  if (dummy==4)
    crash();
}
```

needs around 2^10 attempts

Fuzzing rules of thumb

- Input-format knowledge is very helpful
- Generational tends to beat random, better specs make better fuzzers
- Each implementation will vary, different fuzzers find different bugs
  - More fuzzing with is better
- The longer you run, the more bugs you may find
  - But it reaches a plateau and saturates after a while
- Best results come from guiding the process
- Notice where you are getting stuck, use profiling (gcov, lcov)!