浙江大學
ZHEJIANG UNIVERSITY

# Fuzzing

Yajin Zhou (http://yajin.org)

Zhejiang University

Credit: This ppt is from Chaoyuan Peng, based on
https://linuxfoundation.org/wp-content/uploads/2021-Linux-Foundation-Mentorship-Series_-Fuzzing-the-Linux-Kernel.pdf

# Testing

*Testing is the process of executing a program to find errors.*

An error is a deviation between observed behavior and specified behavior, i.e., a violation of the underlying specification:

- Functional requirements (features)
- Operational requirements (performance, usability)
- Security requirements

ref: Chauhan, R.K. and Singh, I., 2014. Latest research and development on software testing techniques and tools.

# Testing

- Manual testing
  - Unit testing (individual modules)
  - Integration testing (interaction between modules)
  - System testing (full application testing)
- Fuzzing testing
- Symbolic and concolic testing

# Fuzzing

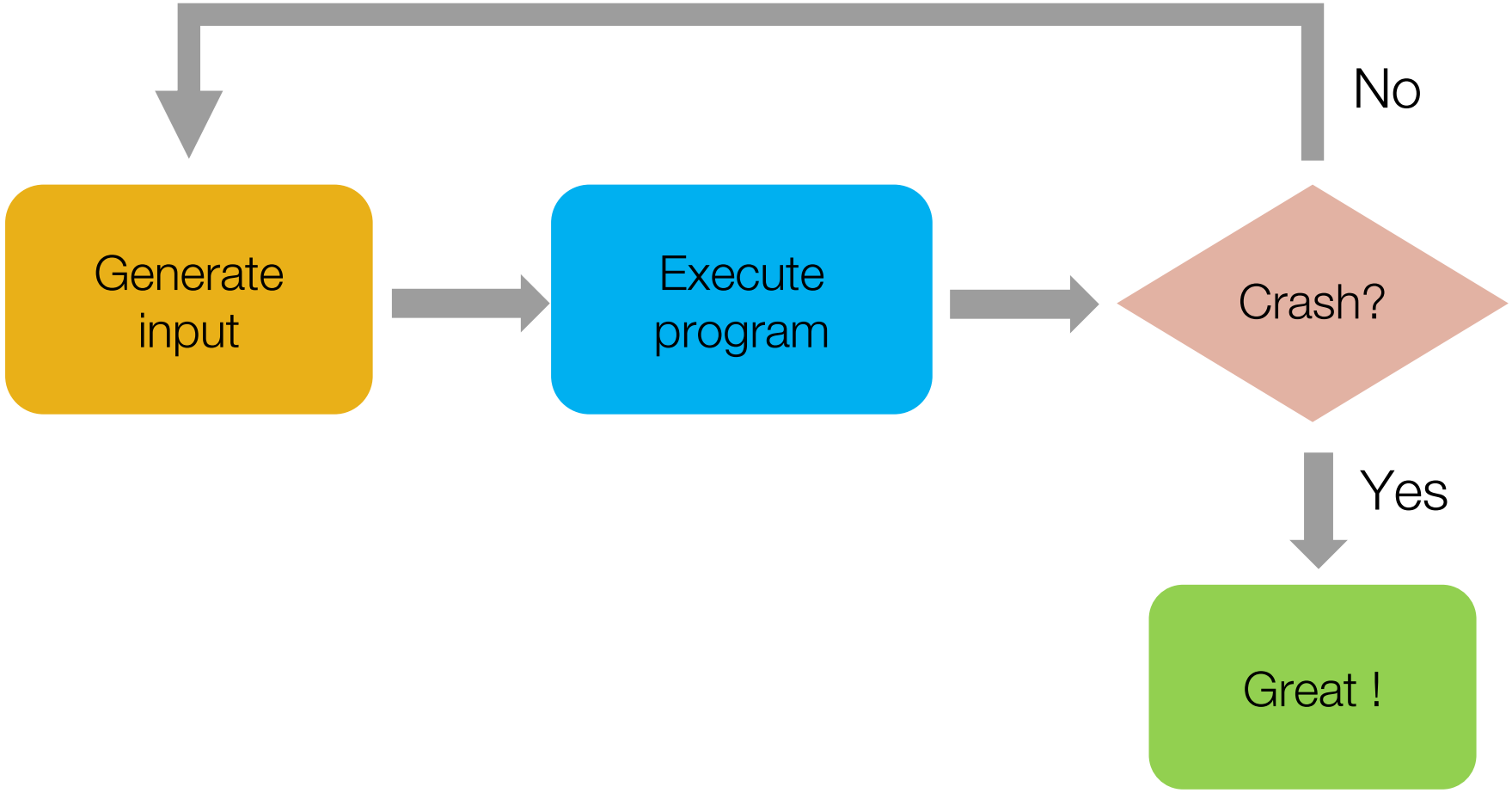
- As Its Core, Fuzzing is Random Testing
  - it starts a long time ago
  - automated software testing technique

**1981** Random testing is a cost-effective alternative to systematic testing techniques (Duran & Natos)
**1983** "The Monkey" (Capps)
**1988** Birth of the term "fuzzing" (Miller)

Monkey testing

# Fuzzing

- Feeding in random inputs until the program crashes



# Fuzzing

- Feeding in random inputs until the program crashes

— How do we inject inputs?

— How do we generate inputs?

— How do we automate the process?

— How do we execute the program?

— How do we detect bugs?

# Fuzzing

- Feeding in random inputs until the XML parser crashes

— How do we inject inputs?

— How do we generate inputs?

— How do we detect bugs?

— How do we automate the process?

# Fuzzing

- Feeding in random inputs until the XML parser crashes

— How do we inject inputs? Execute the parser with a xml file

— How do we generate inputs?

— How do we detect bugs?

— How do we automate the process?

# Fuzzing

- Feeding in random inputs until the XML parser crashes

— How do we inject inputs? execute the parser with a xml file

— How do we generate inputs?

— How do we detect bugs?

— How do we automate the process?

# Generating inputs for programs


- In case of an XML parser

Idea **#1**: just generate random binary data

```
cat /dev/urandom | xml_parser
```

# Random inputs

```
if (input[0] == '<')

   if (input[1] == 'x')

      if (input[2] == 'm')

         if (input[3] == 'l')

            // start process file
```

- Parser expects the file to start with <xml header
- We need ~2^{8}^4 guesses to get past the header check
- works poorly & incomplete

# Generating better inputs for programs


Idea #2: Model what the application should process

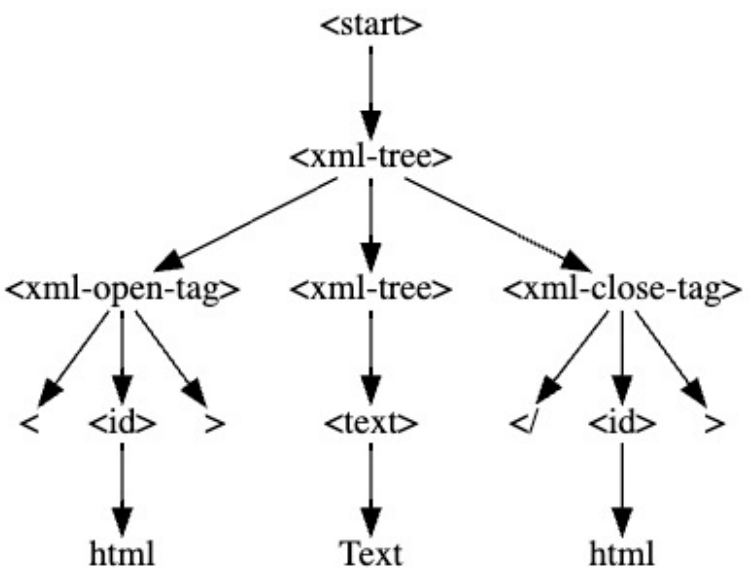
Structured inputs (a.k.a. structure-aware fuzzing)

# Structured inputs

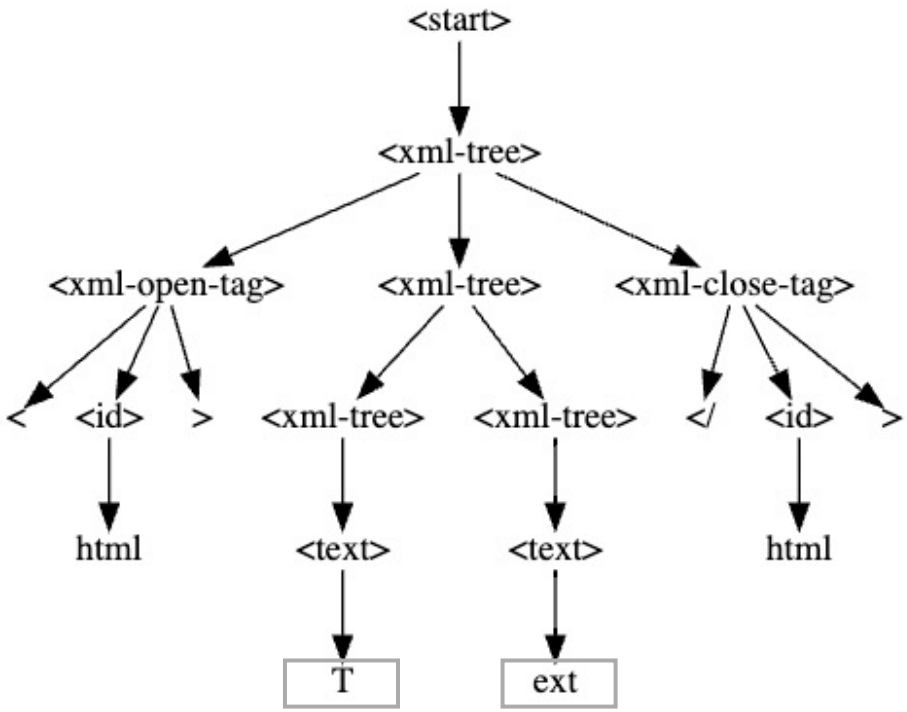
```
XML_GRAMMAR: Grammar = {
    "<start>": ["<xml-tree>"],
    "<xml-tree>": ["<text>",
                   "<xml-open-tag><xml-tree><xml-close-tag>",
                   "<xml-openclose-tag>",
                   "<xml-tree><xml-tree>"],
    "<xml-open-tag>":      ["<<id>>", "<<id> <xml-attribute>>"],
    "<xml-openclose-tag>": ["<<id>/>", "<<id> <xml-attribute>/>"],
    "<xml-close-tag>":     ["</<id>>"],
    "<xml-attribute>":     ["<id>=<id>", "<xml-attribute> <xml-attribute>"],
    "<id>":                ["<letter>", "<id><letter>"],
    "<text>":              ["<text><letter_space>", "<letter_space>"],
    "<letter>":            srange(string.ascii_letters + string.digits +
                                  "\"" + "'" + "."),
    "<letter_space>":      srange(string.ascii_letters + string.digits +
                                  "\"" + "'" + " " + "\t"),
}
```

ref: https://www.fuzzingbook.org/html/GreyboxGrammarFuzzer.html#Parsing-and-Recombining-HTML

# Structured inputs



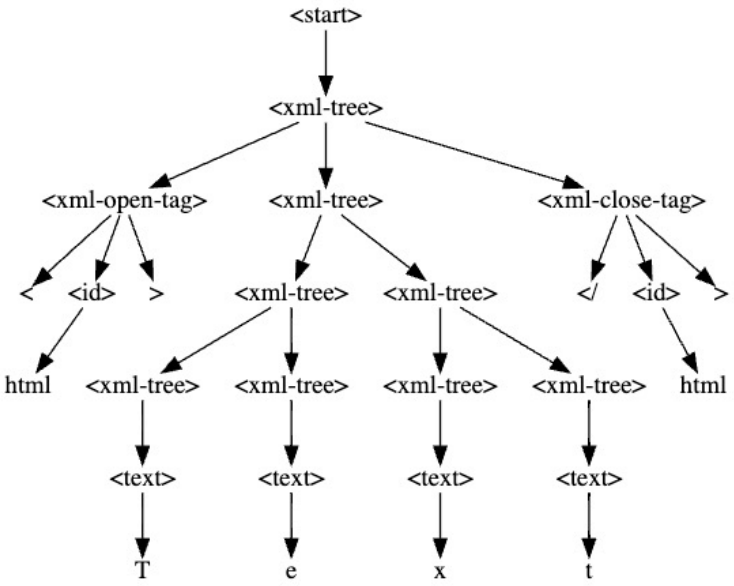

Initial structured tree




structured tree after adding nodes

# Structured inputs

After several mutation




new structured tree

```
<start>
|-<html><head><title>Hello</title></head><body>World<br/></body></html>
<xml-tree>
|-<html><head><title>Hello</title></head><body>World<br/></body></html>
|-<head><title>Hello</title></head><body>World<br/></body>
|-<head><title>Hello</title></head>
|-<title>Hello</title>
|-Hello
|-<body>World<br/></body>
|-World<br/>
|-World
|-<br/>
<xml-open-tag>
|-<html>
|-<head>
|-<title>
|-<body>
<xml-openclose-tag>
|-<br/>
<xml-close-tag>
|-</title>
|-</head>
|-</body>
|-</html>
<xml-attribute>
<id>
<text>
<letter>
<letter_space>
```

generated testcase

# Generating better inputs for programs

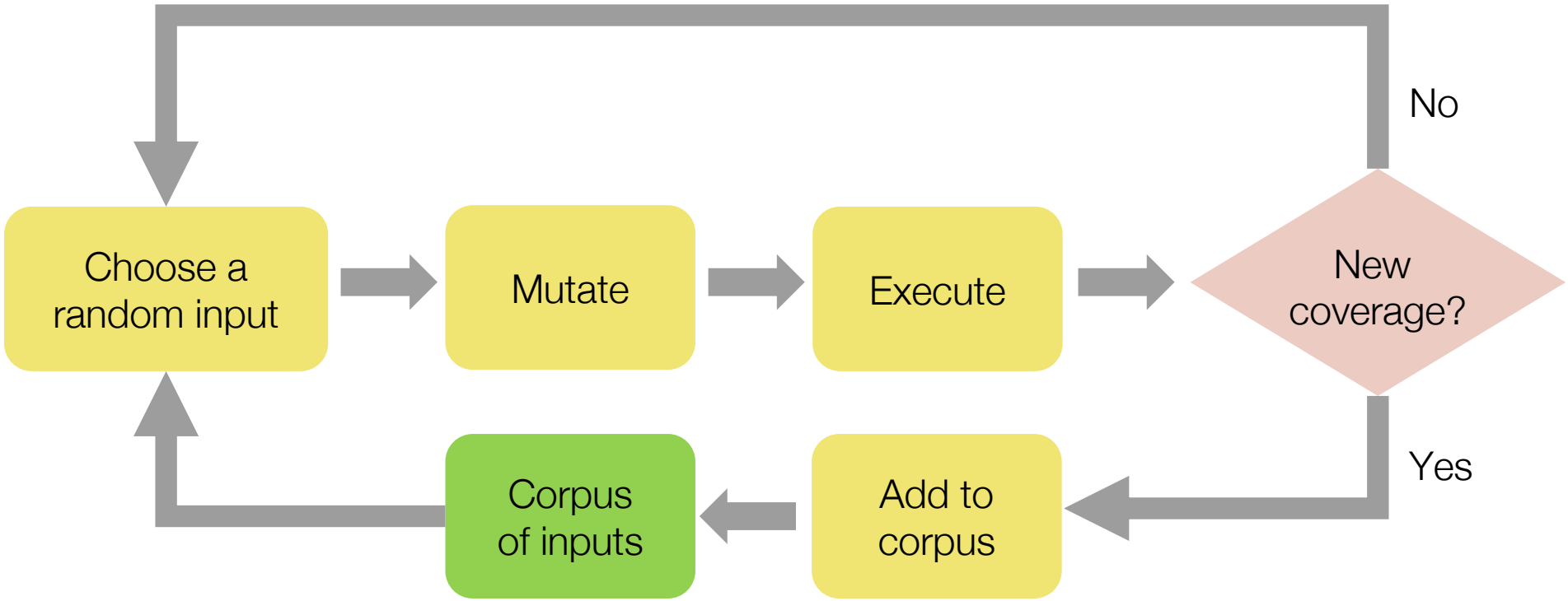

Idea #3: Coverage as completeness metric

Intuition:

- A software flaw is only detected if the flawed statement is executed.
- Effectiveness of test suite therefore depends on how many statements are executed.

Coverage-guided generation (a.k.a. coverage-guided fuzzing)

# Coverage-guided generation



# Branch Coverage

```
int arr[6];

int func(int a, int b) {
    int idx = 6;
    if (a < 6) idx -= 6; else idx -= 1;
    if (b < 6) idx -= 1; else idx += 1;
    return arr[idx]; // idx = 4
}
```

Test input:  a=10, b=1

# Branch Coverage

```
int arr[6];

int func(int a, int b) {
    int idx = 6;
    if (a < 6) idx -= 6; else idx -= 1;
    if (b < 6) idx -= 1; else idx += 1;
    return arr[idx]; // idx = 1
}
```

Test input:  a=1, b=10

# Branch Coverage

```
int arr[6];

int func(int a, int b) {
    int idx = 6;
    if (a < 6) idx -= 6; else idx -= 1;
    if (b < 6) idx -= 1; else idx += 1;
    return arr[idx];
}
```

All test inputs:  a=10, b=1 and a=1, b=10

Full  branch coverage

# Is branch coverage enough?

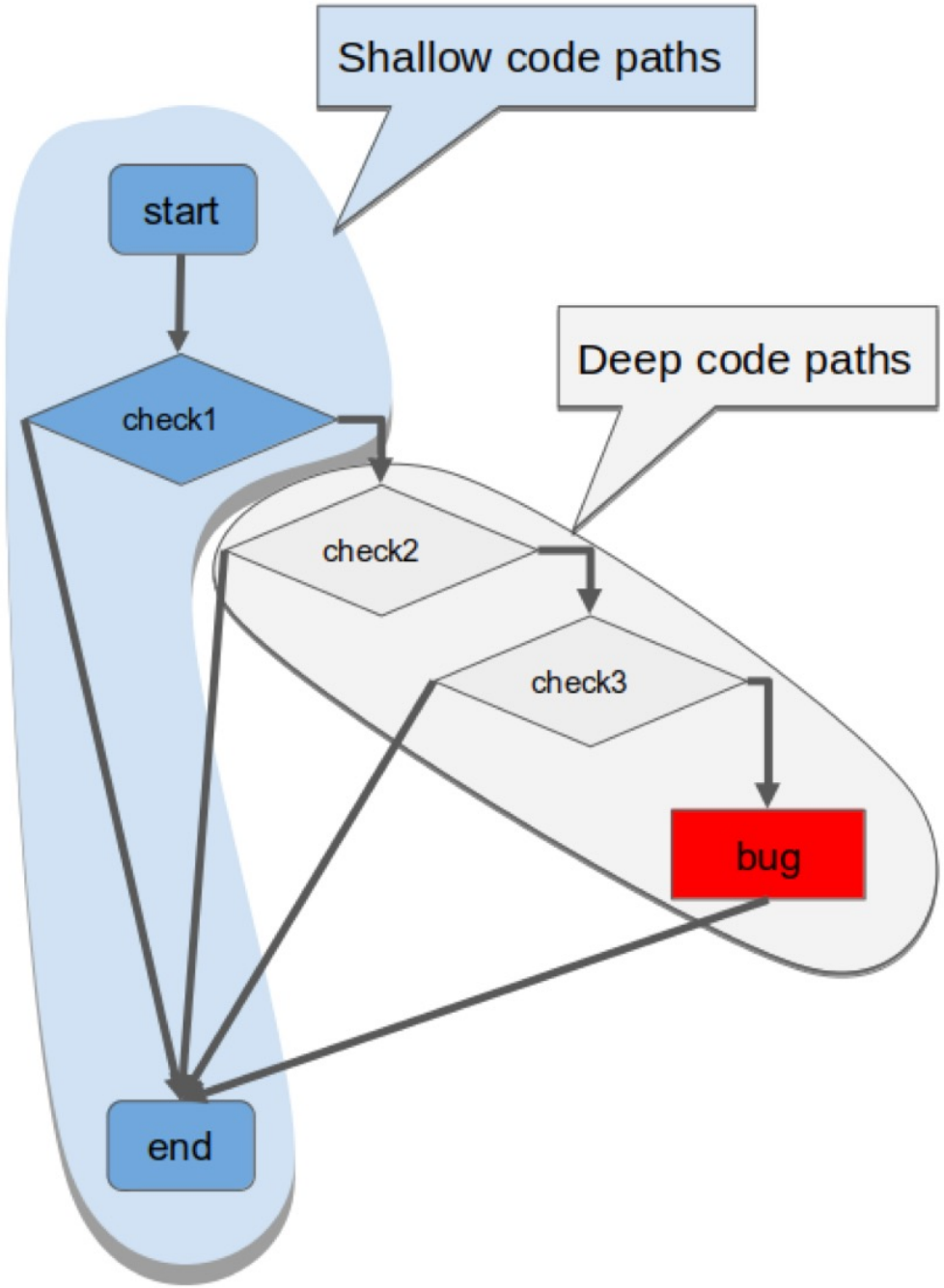
```
int arr[6];

int func(int a, int b) {
    int idx = 6;
    if (a < 6) idx -= 6; else idx -= 1;
    if (b < 6) idx -= 1; else idx += 1;
    return arr[idx];
}
```

- Not all paths are executed: Fail to find overflow bug (a = 1, b = 1)
- Full path coverage evaluates all possible paths
  - expensive (path explosion due to each branch)
  - impossible for loops
- Probabilistically covers state space

# Coverage wall

- Hard to satisfy checks (e.g., checksum)
- Chains of checks
- Fuzzer no longer makes progress after certain iterations.




ref: CS412 Software Security

# How to measure code coverage

Existing tools:

- Gcov:
  - https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
- SanitizerCoverage:
  - https://clang.llvm.org/docs/ SourceBasedCodeCoverage.html

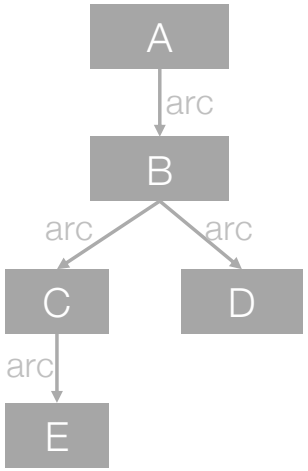
# Example

- Compile program with coverage instrumentation
  - gcc **-fprofile-arcs -ftest-coverage** gcov_example.c -o example
- Generate coverage information on the fly
  - ./example
- Convert coverage information to html report
  - gcovr --html-detailed coverage.html

```c
#include <stdio.h>
int arr[5] = { 0, 1, 2, 3, 4 };

int func(int a) {
    int idx = 5;
    if (a < 5)
        idx -= 5;
    else
        idx -= 1;
    return arr[idx];
}

int main(int argc, char *argv[]){
    if (argc >= 2){
        printf("argc>=2\n");
        return 0;
    }
    printf("gcov testing.\n");
    int result = func(argc);
    return result;
}
```




arc: possible branches taken from one basic block to another

# Example

- Generated report:
  - **Green** line means executed
  - **Red** line means never executed

**GCC Code Coverage Report**

Directory: .
File: gcov_example.c
Date: 2022-04-14 17:13:20

| | Exec | Total | Coverage |
|---|---|---|---|
| Lines: | 10 | 13 | 76.9% |
| Functions: | 2 | 2 | 100.0% |
| Branches: | 2 | 4 | 50.0% |

▸ List of functions

| Line | Branch | Exec | Source |
|---|---|---|---|
| 1 | | | `#include <stdio.h>` |
| 2 | | | `int arr[5] = { 0, 1, 2, 3, 4 };` |
| 3 | | | |
| 4 | | 1 | `int func(int a) {` |
| 5 | | 1 | `    int idx = 5;` |
| 6 | ▸ 1/2 | 1 | `    if (a < 5)` |
| 7 | | 1 | `        idx -= 5;` |
| 8 | | | `    else` |
| 9 | | ✗ | `        idx -= 1;` |
| 10 | | 1 | `    return arr[idx];` |
| 11 | | | `}` |
| 12 | | | |
| 13 | | 1 | `int main(int argc, char *argv[]){` |
| 14 | ▸ 1/2 | 1 | `    if (argc >= 2){` |
| 15 | | ✗ | `        printf("argc>=2\n");` |
| 16 | | ✗ | `        return 0;` |
| 17 | | | `    }` |
| 18 | | 1 | `    printf("gcov testing.\n");` |
| 19 | | 1 | `    int result = func(argc);` |
| 20 | | 1 | `    return result;` |
| 21 | | | `}` |
| 22 | | | |

# Fuzzing

- Feeding in random inputs until the XML parser crashes

— How do we inject inputs? execute the parser with a xml file

— How do we generate inputs? coverage-guided, mutation

— How do we detect bugs?

— How do we automate the process?

# Detecting bugs


- Program crash is not a good indicator
  - Some bugs do not cause crash immediately (e.g., memory corruptions)
  - Other bugs are not crashes (e.g., uninitialized value usage)

- Use dynamic bug detectors
  - **Sanitizers**

# Sanitizers

- Tools based on compiler instrumentation.
- Discover bugs like **integer overflow**, **heap buffer overflow**, **use after free**, etc.
- Different type of Sanitizers:
  - ASAN
  - UBSAN
  - MSAN
  - TSAN

# Address Sanitizer (ASAN)


- Fast memory error detector
- Compiler directive: **-fsanitize=address**
- Detects various issues:
  - Out-of-bounds access to heap, stack and globals
  - Use After Free
  - Use after scope
- Typical slowdown: ~2x

# Address Sanitizer (ASAN)

```
1   int main(int argc, char **argv) {
2     int *array = new int[100];
3     delete [] array;
4     return array[argc];  // use after free here
5   }
```

# compile, link, run

clang++ -g -fsanitize=address example_uaf.cc && ./a.out

```
=================================================================
==38960==ERROR: AddressSanitizer: heap-use-after-free on address 0x00010623a844 at pc 0x000104463f4c bp 0x00016b99f0e0 sp 0x00016b99f0d8
READ of size 4 at 0x00010623a844 thread T0
    #0 0x104463f48 in main example_uaf.cc:4
    #1 0x1048210f0 in start+0x204 (dyld:arm64+0x50f0)
    #2 0x4d7dfffffffffffc  (<unknown module>)

0x00010623a844 is located 4 bytes inside of 400-byte region [0x00010623a840,0x00010623a9d0)
freed by thread T0 here:
    #0 0x104913c70 in wrap__ZdaPv+0x6c (libclang_rt.asan_osx_dynamic.dylib:arm64+0x4bc70)
    #1 0x104463efc in main example_uaf.cc:3
    #2 0x1048210f0 in start+0x204 (dyld:arm64+0x50f0)
    #3 0x4d7dfffffffffffc  (<unknown module>)

previously allocated by thread T0 here:
    #0 0x10491387c in wrap__Znam+0x6c (libclang_rt.asan_osx_dynamic.dylib:arm64+0x4b87c)
    #1 0x104463ee4 in main example_uaf.cc:2
    #2 0x1048210f0 in start+0x204 (dyld:arm64+0x50f0)
    #3 0x4d7dfffffffffffc  (<unknown module>)

SUMMARY: AddressSanitizer: heap-use-after-free example_uaf.cc:4 in main
```

read freed buffer in `return array[argc];`

free buffer in `delete [] array;`

`int *array = new int[100];`

# Undefined Behavior Sanitizer (UBSAN)

- Compiler directive: -fsanitize=undefined
- Detects undefined behavior:
  - Divide by zero

```
int b = 0;
int c = a / b;
```

  - Signed integer overflow

```
int k = 0x7fffffff;
k += 1;
// 0x7fffffff + 1 cannot be represented in type 'int'
```

  - Dereferencing misaligned/null pointer

```
char * ptr = (char*)alloc_mem(LARGE_SIZE); // possible NULL pointer
*ptr = 1;
```

# Memory Sanitizer (MSAN)

- Compiler directive: **-fsanitize=memory**
- Detects uninitialized reads

```c
1   int main(int argc, char** argv) {
2     int array[10]; // uninitialized stack array
3     return array[5];
4   }
```

clang -fsanitize=memory msan_example.c

```
==108578==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x4983bd in main /home/happy/workspace/msan_example.c:3:3
    #1 0x7f93e4ca00b2 in __libc_start_main /build/glibc-sMfBJT/glibc-2.31/csu/../csu/libc-start.c:308:16
    #2 0x41c22d in _start (/home/happy/workspace/a.out+0x41c22d)

SUMMARY: MemorySanitizer: use-of-uninitialized-value /home/happy/workspace/msan_example.c:3:3 in main
```

- Typical slowdown: ~3x

# Thread Sanitizer (TSAN)

- Compiler directive: **-fsanitize=thread**
- Detects data races.

clang –fsanitize=thread tsan_example.c

```
1  #include <pthread.h>
2  int g_value;
3  void *thread1(void *x)
4  {
5      g_value = 42;
6      return x;
7  }
8  int main()
9  {
10     int cnt = 100;
11     while (--cnt > 0)
12     {
13         pthread_t t;
14         pthread_create(&t, NULL, thread1, NULL);
15         g_value = 43;
16         pthread_join(t, NULL);
17     }
18     return g_value;
19 }
```

write in thread1

write in main thread

```
==================
WARNING: ThreadSanitizer: data race (pid=107809)
  Write of size 4 at 0x000000f18418 by main thread:
    #0 main /home/happy/workspace/tsan_example.c:15:17 (a.out+0x4b5f19)

  Previous write of size 4 at 0x000000f18418 by thread T2:
    #0 thread1 /home/happy/workspace/tsan_example.c:5:13 (a.out+0x4b5e9b)

  Location is global 'g_value' of size 4 at 0x000000f18418 (a.out+0x000000f18418)

  Thread T2 (tid=107812, finished) created by main thread at:
    #0 pthread_create <null> (a.out+0x424b2b)
    #1 main /home/happy/workspace/tsan_example.c:14:9 (a.out+0x4b5f0a)

SUMMARY: ThreadSanitizer: data race /home/happy/workspace/tsan_example.c:15:17 in main
==================
ThreadSanitizer: reported 1 warnings
```

- Typical slowdown: 5x ~ 15x

# Fuzzing

- Feeding in random inputs until the XML parser crashes

— How do we inject inputs? Execute the parser with a xml file

— How do we generate inputs? coverage-guided, mutation

— How do we detect bugs? ASAN, UBSAN, MSAN, etc.

— How do we automate the process?

# Automation

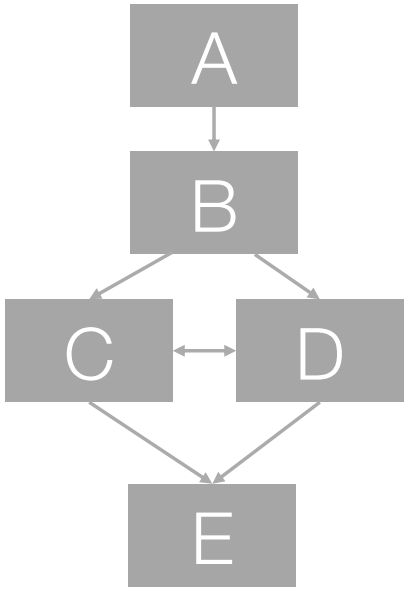

- Run program with a generated testcase
- Monitor program for coverage and crashes
- Deduplicate crashes
- Minimize testcase
- Generate reproducers
- Report crashes

# AFL


- The most well-known fuzzer currently
- Uses compiler-time instrumentation to track branch coverage
- Mutate fuzzing testcase based on previous branch coverage

# Core of AFL

- Pseudo code of coverage instrumentation:
  - cur_location = <COMPILE_TIME_RANDOM>;
  - shared_mem[cur_location ^ prev_location] ++;
  - prev_location = cur_location >> 1;
- Different coverage for the following call-chains:
  - A -> B -> C -> D -> E
  - A -> B -> D -> C -> E



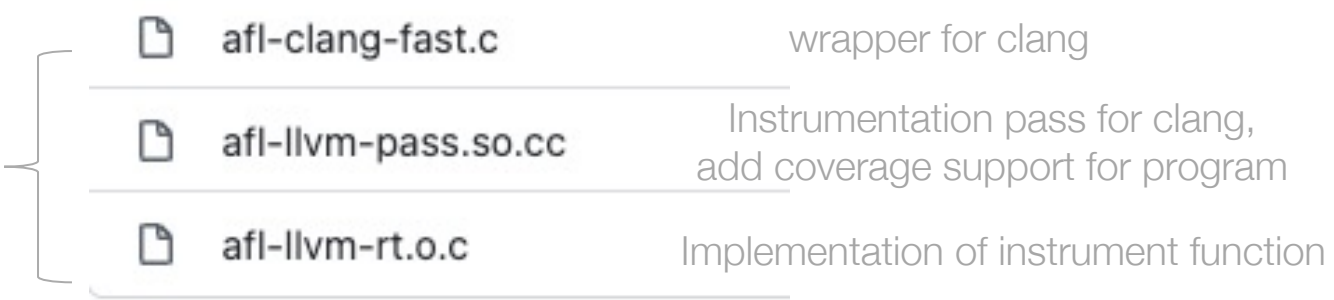
# Core of AFL

Corpus Mutation Strategies

- **Bit Flip**: flips a bit. i.e., **1->0, 0->1**
  - L/S (length of toggled bits/stepover bits): 1/1, 2/1, 4/1, 8/8…
- **Byte Flip**: flips a byte
- **Arithmetic**: subtract/add small integer to 8/16/32 bit values.
- **Havoc**: random things with bit/byte/arithmetic, etc.
- **Interest**: replace content with **known interesting value** (e.g., 65535)
- **Dictionary**: user provided dictionary or auto discovered tokens.
- **Splice**: split & combine two or more files to get a new file

# Hands On: Building AFL

- git clone https://github.com/google/AFL.git
- make
- cd llvm_mode
- make




  - **afl-clang-fast**: compiler which instruments program while building
  - **afl-fuzz**: overall fuzzer

# Hands On: Fuzzing example

```c
#include <stdio.h>
#include <unistd.h>

#define SIZE 20

int main(int argc, char *argv[])
{
    char input[SIZE] = {0};
    size_t length;
    length = read(STDIN_FILENO, input, SIZE);
    char calc_value = 0;

    if (input[0] == 'a'){
        if (input[1] == 'b'){
            if (input[2] == 'c'){
                int idx = input[3] + input[4];
                calc_value = input[idx];
            }
        }
    }

    return calc_value;
}
```

buffer overflow bug

- Example PoC:

  input buffer: “abc\x40\x40"

# Hands On: Fuzzing example

- Build program with **ASan** & **Coverage** instrumentation
  - afl-clang-fast -fsanitize=address example.c

```
21  if ( !_asan_option_detect_stack_use_after_return )
22  {
23    v4 = 0LL;
24    goto LABEL_3;
25  }
26  _asan_stack_malloc_1();
27  v4 = v3;
28  v5 = (unsigned __int64)v3;
29  if ( !v3 )
30LABEL_3:
31    v5 = (unsigned __int64)&v19[-12];
32  v19[3] = v5;
33  *(_QWORD *)v5 = 1102416563LL;
34  *(_QWORD *)(v5 + 8) = "1 32 20 7 input:8";
35  *(_QWORD *)(v5 + 16) = main;
36  v6 = v5 >> 3;
37  *(_QWORD *)(v6 + 2147450880) = 0xF3F8F8F8F1F1F1F1LL;
38  *(_DWORD *)(v6 + 2147450888) = -202116109;
39  v7 = (_BYTE *)(v5 + 32);
40  _afl_area_ptr[_start___sancov_guards] += __CFADD__(_afl_area_ptr[_start___sancov_guards], 1) + 1;
41  *(_WORD *)(v6 + 2147450884) = 0;
42  *(_BYTE *)(v6 + 2147450886) = 4;
43  _asan_memset();
44  v8 = (_QWORD *)(v5 + 32);
45  _interceptor_read();
46  v9 = *(_BYTE *)(((v5 + 32) >> 3) + 0x7FFF8000);
47  if ( v9 && ((unsigned __int8)v7 & 7) >= v9 )
48  {
49    v10 = v5 + 32;
50    v11 = _asan_report_load1(v5 + 32, v8);
51  }
52  else
53  {
54    if ( *v7 != 97 )
55    {
```

Instead of alloc array on stack,
ASan wrap the alloc on FakeStack

Increase coverage,
add counter to shared memory

# Hands On: Fuzzing example

- Build program with **ASan** & **Coverage** instrumentation
  - afl-clang-fast -fsanitize=address example.c

```
21  if ( !_asan_option_detect_stack_use_after_return )
22  {
23    v4 = 0LL;
24    goto LABEL_3;
25  }
26  _asan_stack_malloc_1();
27  v4 = v3;
28  v5 = (unsigned __int64)v3;
29  if ( !v3 )
30LABEL_3:
31    v5 = (unsigned __int64)&v19[-12];
32  v19[3] = v5;
33  *(_QWORD *)v5 = 1102416563LL;
34  *(_QWORD *)(v5 + 8) = "1 32 20 7 input:8";
35  *(_QWORD *)(v5 + 16) = main;
36  v6 = v5 >> 3;
37  *(_QWORD *)(v6 + 2147450880) = 0xF3F8F8F8F1F1F1F1LL;
38  *(_DWORD *)(v6 + 2147450888) = -202116109;
39  v7 = (_BYTE *)(v5 + 32);
40  _afl_area_ptr[_start___sancov_guards] += __CFADD__(_afl_area_ptr[_start___sancov_guards], 1) + 1;
41  *(_WORD *)(v6 + 2147450884) = 0;
42  *(_BYTE *)(v6 + 2147450886) = 4;
43  _asan_memset();
44  v8 = (_QWORD *)(v5 + 32);
45  _interceptor_read();
46  v9 = *(_BYTE *)(((v5 + 32) >> 3) + 0x7FFF8000);
47  if ( v9 && ((unsigned __int8)v7 & 7) >= v9 )
48  {
49    v10 = v5 + 32;
50    v11 = _asan_report_load1(v5 + 32, v8);
51  }
52  else
53  {
54    if ( *v7 != 97 )
55    {
```

Instead of alloc array on stack,
ASan wrap the alloc on FakeStack

Increase coverage,
add counter to shared memory

Check buffer sanity (i.e., input[0]) before read

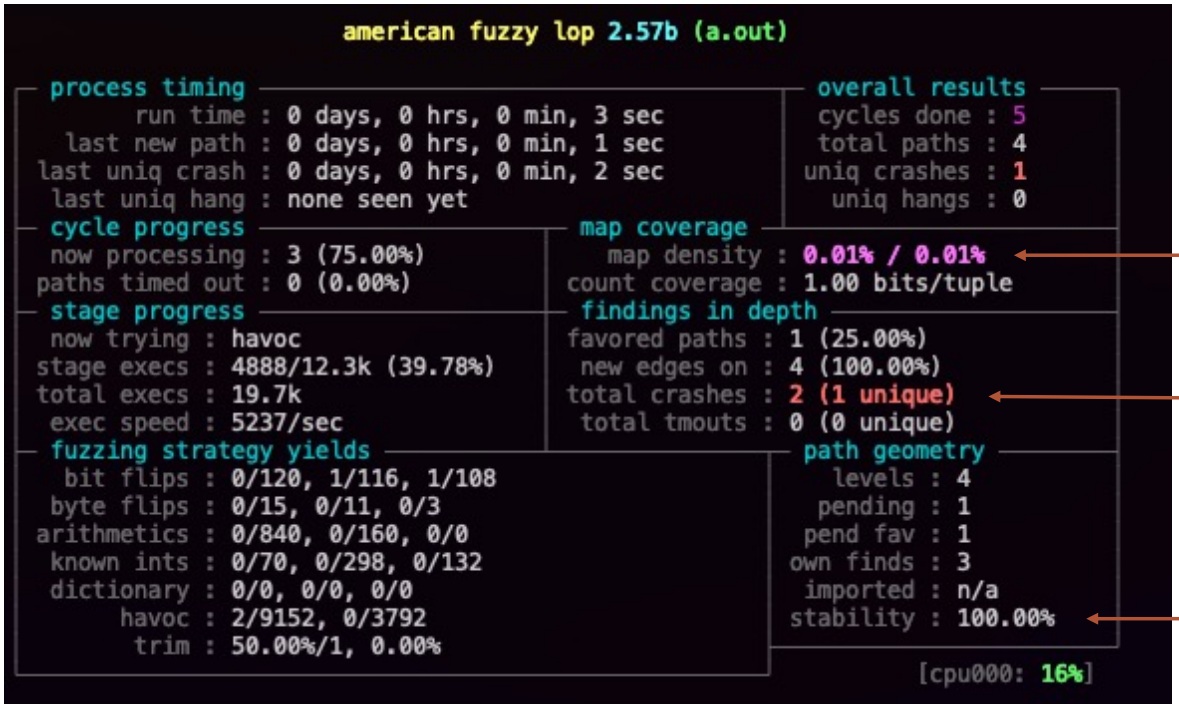
# Hands On: Fuzzing example


- Build program with **ASan** & **Coverage** instrumentation
  - afl-clang-fast -fsanitize=address example.c
- Provide initial corpus/testcase seed
  - echo 123 > corpus/seeds0
- Fuzz it
  - afl-fuzz **-D -i** input **-o** output -- ./a.out
    - -D: enable deterministic fuzzing (more mutation strategies)
    - -i dir: input directory with initial testcases.
    - -o dir: output directory for fuzzer findings

# Hands On: Fuzzing example

One cycle:

- go over all the interesting test cases discovered so far



# Hands On: Fuzzing example

## Output (fuzzer status & findings):

```
output
├── crashes
│   ├── id:000000,sig:06,src:000002,op:flip4,pos:2
│   └── README.txt
├── fuzz_bitmap
├── fuzzer_stats
├── hangs
├── plot_data
└── queue
    ├── id:000000,orig:seeds0
    ├── id:000001,src:000000,op:havoc,rep:64,+cov
    ├── id:000002,src:000001,op:flip2,pos:1,+cov
    └── id:000003,src:000002,op:havoc,rep:4,+cov

3 directories, 9 files
```

testcase which crashes the program

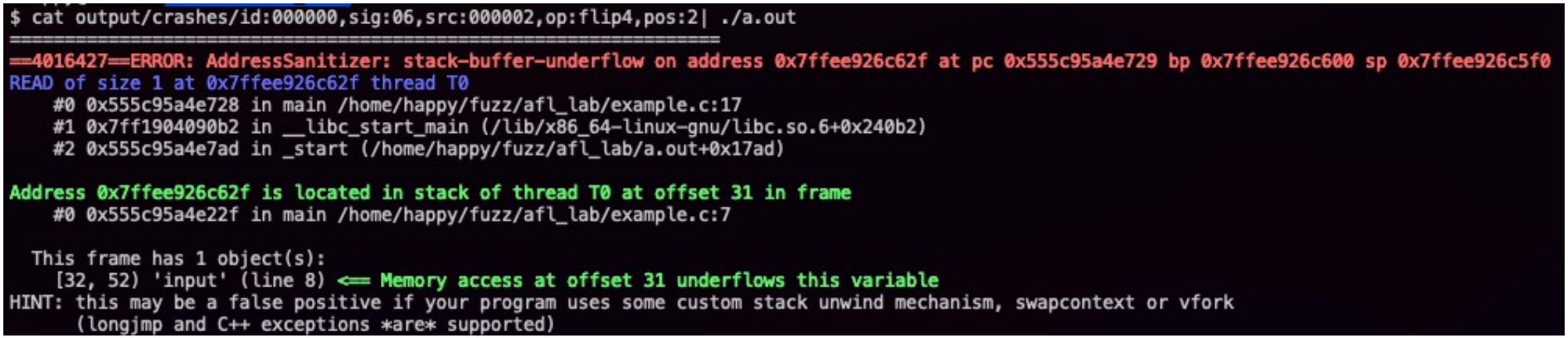
fuzzing status. For clustered fuzzers, use **afl-whatsup** to get status of fuzzers

interesting testcases discovered

# Hands On: Fuzzing example

## Reproduce crashes:

```
% hexdump -C output/crashes/id:000000,sig:06,src:000002,op:flip4,pos:2
00000000  61 62 63 ff                                       |abc.|
```



"abc\xff"

```
$ cat output/crashes/id:000000,sig:06,src:000002,op:flip4,pos:2| ./a.out
=================================================================
==4016427==ERROR: AddressSanitizer: stack-buffer-underflow on address 0x7ffee926c62f at pc 0x555c95a4e729 bp 0x7ffee926c600 sp 0x7ffee926c5f0
READ of size 1 at 0x7ffee926c62f thread T0
    #0 0x555c95a4e728 in main /home/happy/fuzz/afl_lab/example.c:17
    #1 0x7ff1904090b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x240b2)
    #2 0x555c95a4e7ad in _start (/home/happy/fuzz/afl_lab/a.out+0x17ad)

Address 0x7ffee926c62f is located in stack of thread T0 at offset 31 in frame
    #0 0x555c95a4e22f in main /home/happy/fuzz/afl_lab/example.c:7

  This frame has 1 object(s):
    [32, 52) 'input' (line 8) <== Memory access at offset 31 underflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
```

# Other fuzzers


- libFuzzer
  - In-process, coverage-guided fuzzing engine.
- LibAFL
  - Advanced fuzzing library written in Rust
  - Scales across cores and machines: Windows, Android, MacOS, Linux, no_std, etc.
- AFL++
  - superior fork to Google's AFL
  - more speed, more and better mutations, instrumentation, custom module support, etc.
- Nautilus
  - A grammar based feedback Fuzzer

# Summary

- Feeding in random inputs until the program crashes

— How do we inject inputs? Execute the parser with a xml file

— How do we generate inputs? coverage-guided, mutation

— How do we detect bugs? ASAN, UBSAN, MSAN, etc.

— How do we automate the process? AFL, libFuzzer, etc.

# Conclusion

## Three Shades of Fuzzing

### Black box

- Unaware of the program structure
- Scale but dumb

### White box

- Leverage semantic program analysis
- Smart but don’t scale too much

### Grey box

- Leverages program instrumentation
- Smart & scale

# Conclusion

Principle of Greybox Fuzzing

1. Preprocess
2. Scheduling: Choose "good" inputs
3. Input Generation: Mutations
4. Input Evaluation: Observe coverage/score
5. Configuration Updating
6. Continue

ref: The art, science, and engineering of fuzzing: A survey (Manès et al. 2019)