Fuzzing (Lecture 3)

Prof. Mathy Vanhoef

DistriNet – KU Leuven – Belgium

Slides partly inspired by: https://cmu-program-analysis.github.io/2022/lecture-slides/20-fuzzing.pdf

Security testing of programs

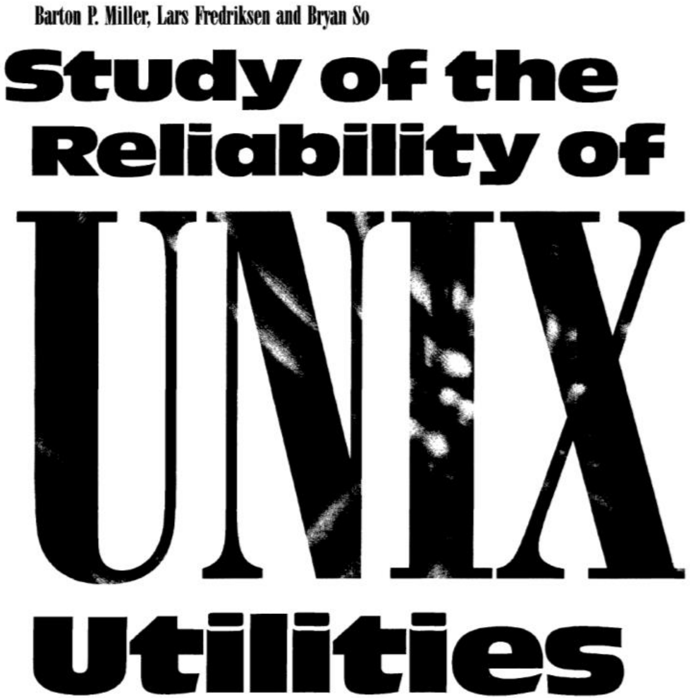
A common technique for security testing is **fuzzing**:

- Give unexpected or random input to the program
- Then monitor for crashes, failed assertions, memory leaks,…

Tested program is called the **SUT (System Under Test)**

History of fuzzing

The paper started the field of fuzz testing (1990):

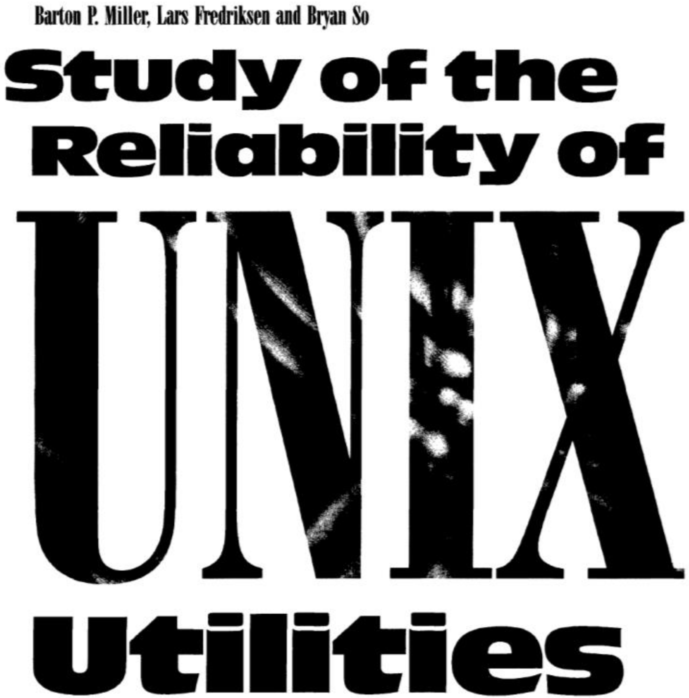

Barton P. Miller, Lars Fredriksen and Bryan So

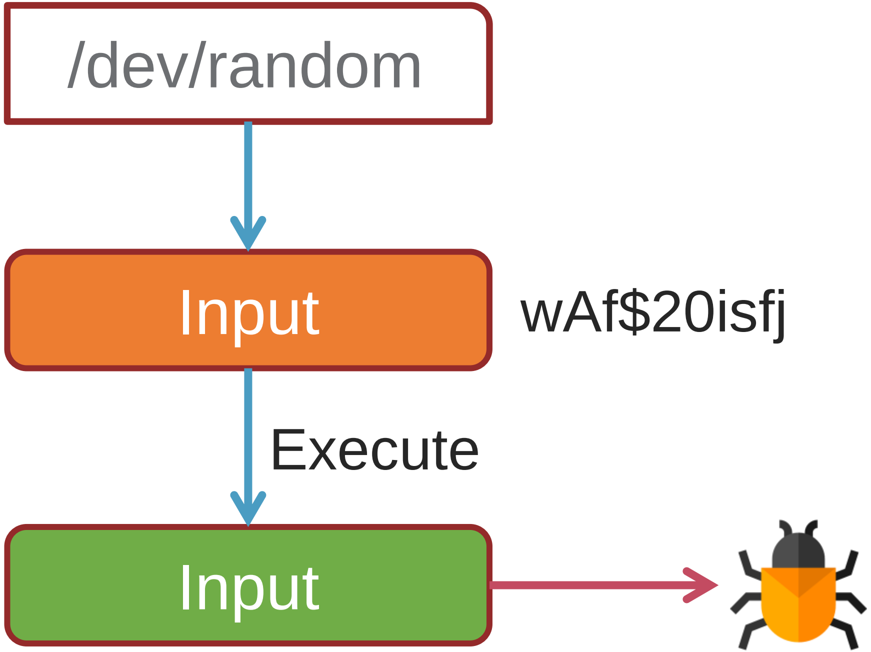
Study of the Reliability of UNIX Utilities

> "On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash."

History of fuzzing

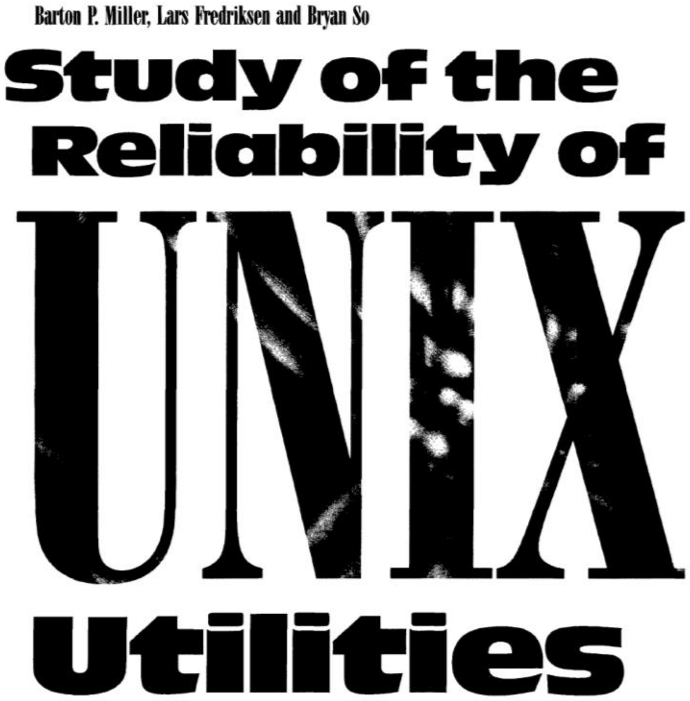
The paper started the field of fuzz testing (1990):





History of fuzzing

The paper started the field of fuzz testing (1990):



- Tested ~90 Unix tools by piping random characters into them.
- 24% of them failed.
- "… our simple testing technique has discovered a wealth of errors and is **likely to be more commonly used** (at least in the near term) **than more formal** procedures."

Why fuzzing?

Why is fuzzing useful in practice?

- Programs **often only undergo functional testing**, i.e., they are tested to handle expected inputs
- Want to test how programs will react to **unexpected inputs**, since incorrectly handled input can cause vulnerabilities!

Fuzzing is frequently used in practice:

- AFL++, LibFuzzer, libAFL, Honggfuzz, Boofuzz, and so on…

What bugs can fuzzing detect?

- Typically, bugs resulting in a **crash**: buffer-overflows, memory leaks, division-by-zero, use-after-free, assert violations, etc.
- **Root causes**: incorrect argument validation, incorrect type casting, executing untrusted code, etc.
- Possible **impact**: security, reliability, performance, correctness

Fuzzing recently had many successes

- Google discovered more than **25,000+ bugs in Chrome**...
  …and 36,000+ bugs in more than 550 open-source projects

- Using SAGE saved Microsoft millions of dollars while creating Windows 7

- The 2016 DARPA Cyber Grand Challenge winner, Mayhem, heavily relied on white-box fuzzing to find vulnerabilities

Sources:

- https://google.github.io/clusterfuzz/#trophies
- “Automated whitebox fuzz testing” by . P. Godefroid, M. Y. Levin, and D. Molnar
- http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html
- “Unleashing Mayhem on binary code” by S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley

Well-known example: American Fuzzy Lop (AFL)

Partly a "dumb" fuzzer:

- No model of input. Uses set of **seed inputs**.
- Given a test input, it **mutates** random bits.

Partly a "smart" fuzzer:

- Tracks which code in the SUT was executed.
- Adds input covering new code to the set of interesting inputs

= **coverage-guided fuzzing** = combination of dumb & smart

Example: fuzzing jpeg

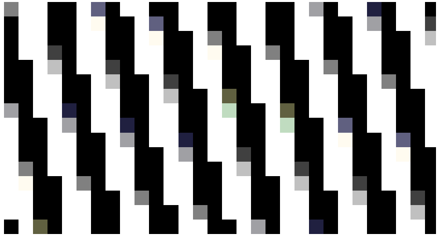
- Start with a single seed input: “hello”
- Using 7 cores for ~28 hours (i7 2.6 GHz)
- Interesting inputs are discovered, but not yet a valid jpeg file

```
$ ./djpeg id:002,op:havoc,rep:32,+cov
Premature end of JPEG file
Not a JPEG file: starts with 0xff 0xff

$ ./djpeg id:003,+cov
Premature end of JPEG file
JPEG datastream contains no image
```

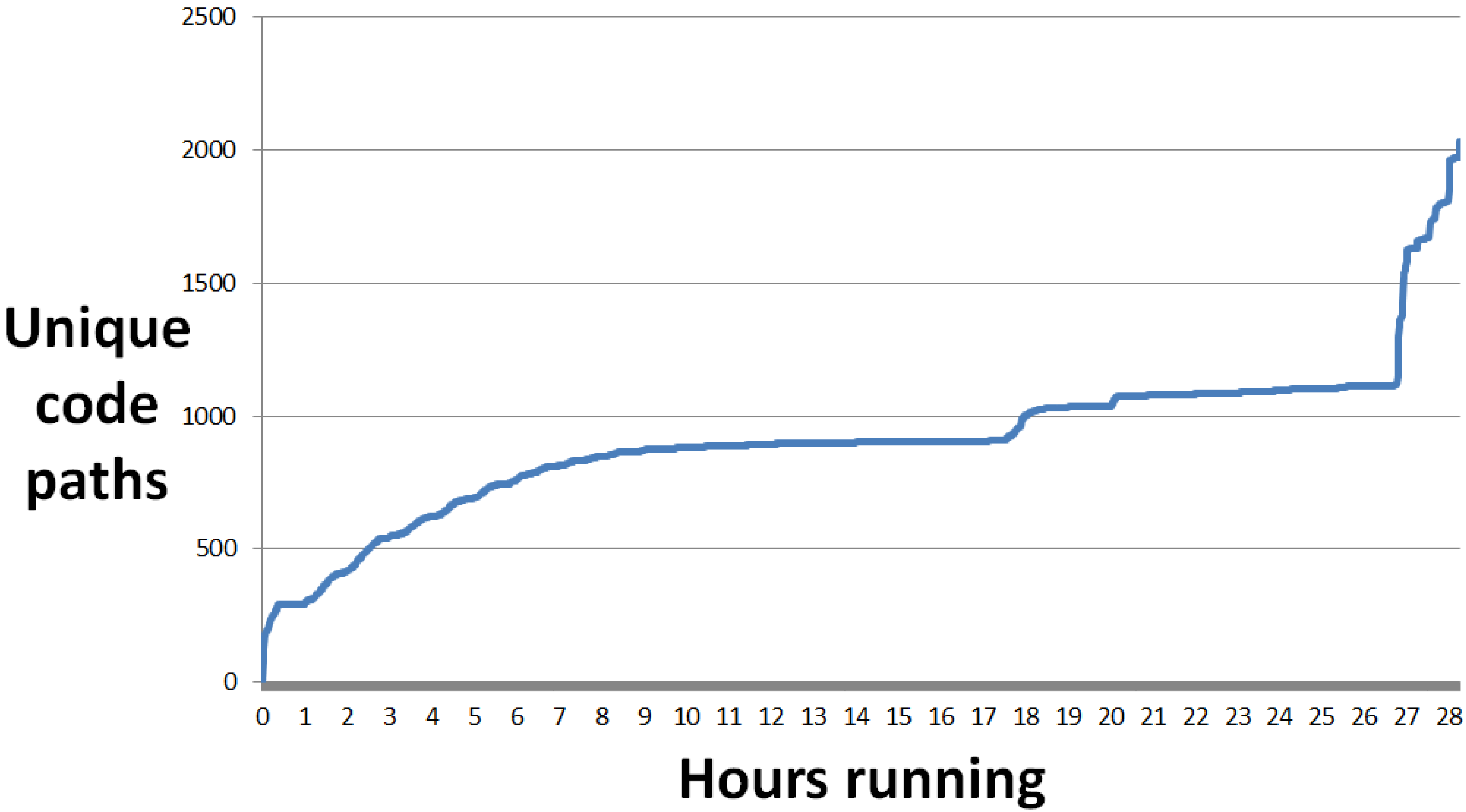
Example: fuzzing jpeg

```
$ ./djpeg id:000840,sync:fuzzer04
Corrupt JPEG data: 50 extraneous bytes before marker 0xc4
Bogus Huffman table definition

$ ./djpeg id:001032,sync:fuzzer06
Corrupt JPEG data: 2 extraneous bytes before marker 0xc9
Quantization table 0x31 was not defined
```

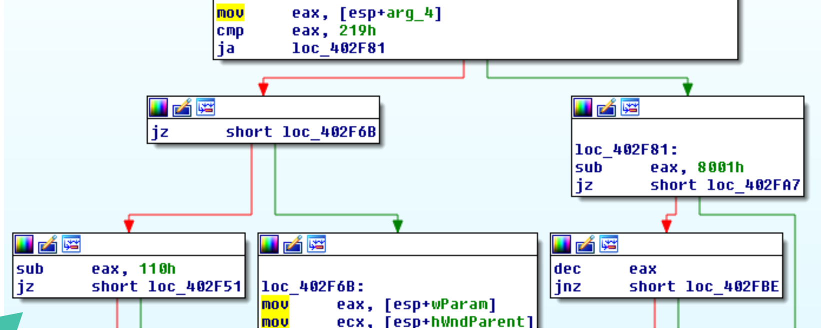
Suddenly valid jpeg’s are generated!



Downside: this can take a long time



How does AFL achieve this?



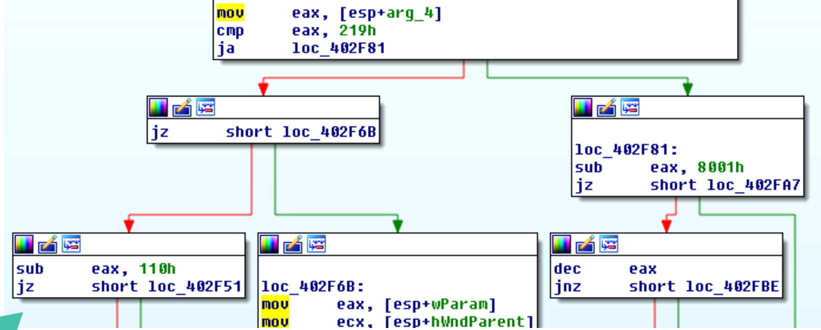
It monitors which code is executed

- Doesn't track the actual code path
- Tracks how many times a branch was taken

Example 1:

Path:	**Branches:**
A → B → C → D → E	AB, BC, CD, DE
A → B → D → C → E	AB, BD, DC, CE

How does AFL achieve this?



It monitors which code is executed

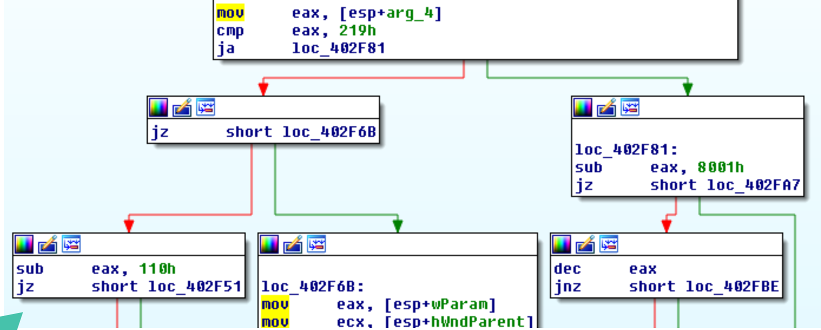
- Doesn't track the actual code path
- Tracks how many times a branch was taken

Example 2:

More hits = different path

Path:	**Branches:**
A → B → A → C	AB, BA, AC
A → B → A → B → A → C	AB, BA, AC

How does AFL achieve this?



It monitors which code is executed

- Doesn't track the actual code path
- Tracks how many times a branch was taken

Queue of "interesting" inputs:

- Take an input from this queue and mutate it
- If new path is taken, add mutation to the queue

→ AFL slowly "explores" functionality of program

Taking a step back: types of fuzzers

- **Black-box** fuzzing: assumes no access to the source code or compiled code. We can only observe input/outputs.
- **White-box** fuzzing: analyzing the internals of the SUT and the information gathered when executing the SUT.
- **Gray-box** fuzzing: obtains *some* info internal to the SUT but does not reason about the full semantics of the SUT.

Manes et al.: “Although there usually is some consensus among security experts, the distinction among black-, grey- and white-box fuzzing is not always clear.”

Source of the quote: https://arxiv.org/abs/1812.00140

Random vs mutated input

Why not generate purely random input?

› Purely random data is not very interesting input!

UPnP

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover "
MX: 1
ST: urn:dial-multiscreen-org:service:dial:1
```

Random input is likely to be (quickly) ignored

```
Asdf;kj4389bnkl;ssa0953t ][asdf  q2rjhn
0q23408jfk 902-g9  aDVQ$#Sfgv  q543tsdfg
ds dfsgdsfg addfsga
Adfgadfgad  2546ueu6sda asfa[]’]’sdf
```

Random vs mutated input

Why not generate purely random input?

› Purely random data is not very interesting input!

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover "
MX: 1
ST: urn:dial-multiscreen-org:service:dial:1
```

Mutated input more likely to explore new code paths

```
M-SEARCH * HTTP/1.1
HOAT: 239.255.255.250:190019001900
MAN: "ssdp:@@@@@discover "
MXsiEf: 1
ST: urn:dial-multiscreen-org:service:dial:1
```

How to mutate binary input?

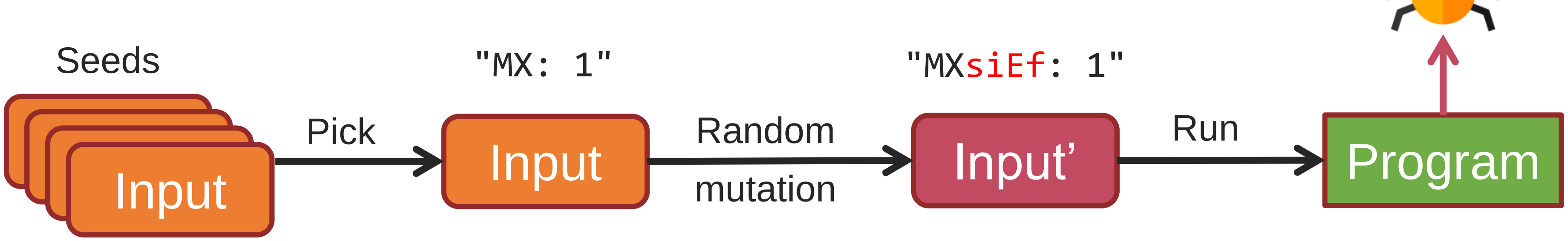
- Bit flips, byte flips
- Change random bytes
- Insert random byte chunks
- Delete random byte chunks
- Set randomly chosen byte chunks to interesting values
  - Examples: INT_MAX, INT_MIN, 0, 1, -1,…

How to mutate text input?

- Repeat words or add long sequences of characters
- Insert random symbols or keywords from a dictionary
- Set randomly chosen words to interesting values
  - Format strings: `"%s%s%s"`, `"%n%n%n"`,…
  - Command injection: `"|touch /tmp/fuzz"`, `";touch /tmp/fuzz"`,…
  - Directory traversal: `"/.../.../.../.../..."`,…
  - Binary strings: `"\xde\xad\xbe\xef"`, `"\x00\x00\x00\x00“`,…
- What are your ideas?

More examples: https://github.com/jtpereyda/boofuzz/blob/v0.4.1/boofuzz/primitives/string.py#L43

Mutation-based fuzzing



UPnP

Valid seed input

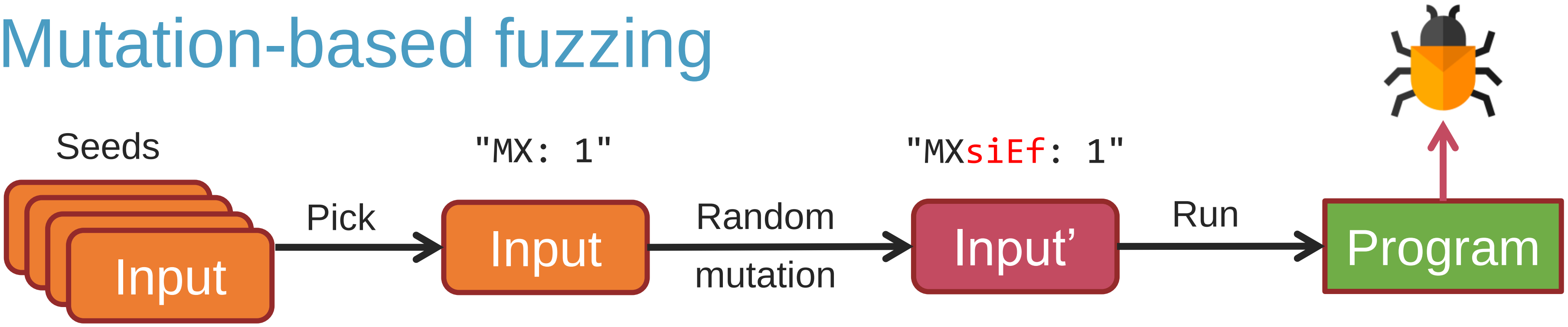
```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.0:1900
MAN: "ssdp:discover "
MX: 1
```

Mutated seed input

```
M-SEARCH * HTTP/1.1
HOAT: 239.255.255.0:19001900
MAN: "ssdp:@@@@@discover "
MXsiEf: 1
```

→ Fuzzers using this approach: radamsa, zzuf, etc.

Mutation-based fuzzing



- Advantage: a **black-box** fuzzing technique which can be used against many products. Fairly easy to set up.
- Disadvantage is **no code coverage** feedback:
  - How do you know that new code is being explored? How to know that your mutations are meaningful?
  - When to stop fuzzing?

What kind of coverage to collect?

- Function coverage
  - `foo(F, F, F);`
- Statement coverage
  - `foo(T, T, T);`
- Branch/Decision coverage
  - `foo(T, T, T);`
  - `foo(T, T, F);`
- Condition coverage
  - `foo(F, F, T);`
  - `foo(T, T, F);`

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```


Source: https://handouts.secappdev.org/handouts/2017/Jan%20Tobias%20Muehlberg/201705-secappdev-muehlberg.pdf

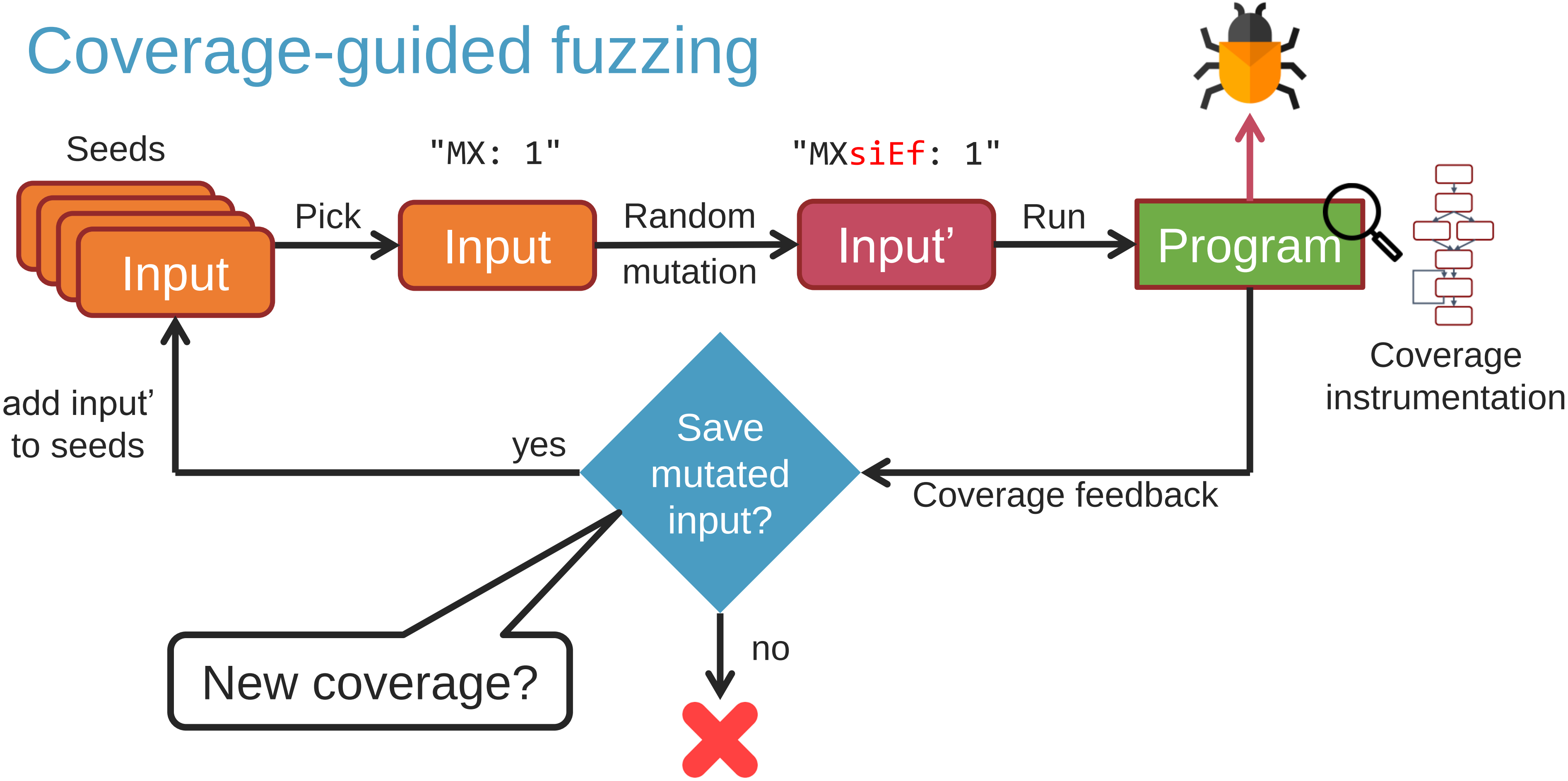
What kind of coverage to collect?

- Modified condition/decision coverage (MC/DC): every condition must be True/False once *and* affect the outcome
  - `foo(F, T, F);`
  - `foo(F, T, T);`
  - `foo(F, F, T);`
  - `foo(T, F, T);`
- Multiple condition coverage
- Parameter value coverage
- …

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Source: https://handouts.secappdev.org/handouts/2017/Jan%20Tobias%20Muehlberg/201705-secappdev-muehlberg.pdf

Coverage-guided fuzzing



Code coverage details in AFL

- Branch coverage with coarse branch-taken hit counts
- Code injected at every branch is equivalent to:

```
cur_location = <COMPILE_TIME_RANDOM>; // identify code block
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

- Every basic code block is assigned a random identifier
  - Simplifies assigning IDs when linking complex projects
  - Keeps the XOR output uniformly distributed

Code coverage details in AFL

- Branch coverage with coarse branch-taken hit counts
- Code injected at every branch is equivalent to:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

- shared_mem is added by the instrumentation (typically 64 kB)
- Bytes in this map represent (branch_src, branch_dst) tuples
  - Size of the map is chosen so that collisions are sporadic…
  - …and so the map is small enough to be rapidly analyzed/compared

Code coverage details in AFL

- Branch coverage with coarse branch-taken hit counts
- Code injected at every branch is equivalent to:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

- Shift operation in the last line tracks the directionality of tuples
  - Without this, A ^ B would be indistinguishable from B ^ A
- Shift operation also retains the identity of tight loops
  - Otherwise, A ^ A would be equal to B ^ B

Code coverage details in AFL

- Branch coverage with coarse branch-taken hit counts
- Code injected at every branch is equivalent to:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

- The tuple hit counts are divided into several buckets:
  - 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
  - Changes within a bucket are ignored. Transition from one bucket to another is flagged as an interesting new code coverage.

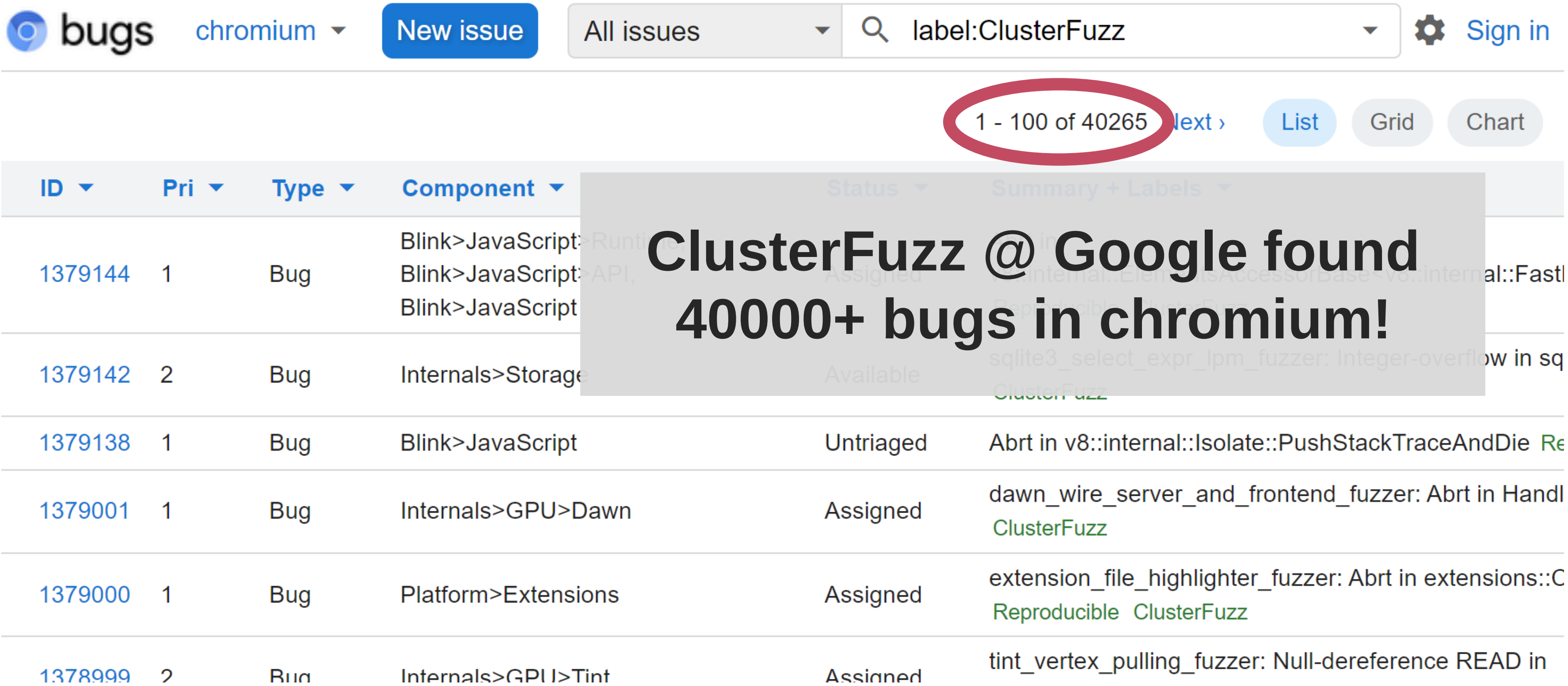
Does it work in practice?

The bug-o-rama trophy case

Yeah, it finds bugs. I am focusing chiefly on development and have not been running the fuzzer at a scale, but here are some of the notable vulnerabilities and other uniquely interesting bugs that are attributable to AFL (in large part thanks to the work done by other users):

IJG jpeg [1]	libjpeg-turbo [1] [2]	libpng [1]
libtiff [1] [2] [3] [4] [5]	mozjpeg [1]	PHP [1] [2] [3] [4] [5] [6] [7] [8]
Mozilla Firefox [1] [2] [3] [4]	Internet Explorer [1] [2] [3] [4]	Apple Safari [1]
Adobe Flash / PCRE [1] [2] [3] [4] [5] [6] [7]	sqlite [1] [2] [3] [4]...	OpenSSL [1] [2] [3] [4] [5] [6] [7]
LibreOffice [1] [2] [3] [4]	poppler [1] [2]...	freetype [1] [2]

What about fuzzing in general?



Source: https://bugs.chromium.org/p/chromium/issues/list?q=label%3AClusterFuzz&can=1

Detecting bugs

By default, a fuzzer will only detect crashes

- State-of-the-art fuzzers can also detected certain logical implementation flaws (mainly academic work)

Not all programs will immediately crash…

- Buffer overflow may overwrite unused data → no crash
- Reading memory outside of buffer → will likely succeed
- Undefined behavior may only lead to a crash on specific platforms

Sanitizers

- **Address** Sanitizer (ASAN)
- **Leak**Sanitizer (comes with ASAN)
- Threat Sanitizer (TSAN): detects **data races and deadlocks** for C++ and Go
- **Undefined-behavior** Sanitizer (UBSan)
- MemorySanitizer: detects use of **uninitialized memory**
- **Hardware-assisted** AddressSanitizer (HWASAN): variant of AddressSanitizer that consumes much less memory

Source: https://github.com/google/sanitizers

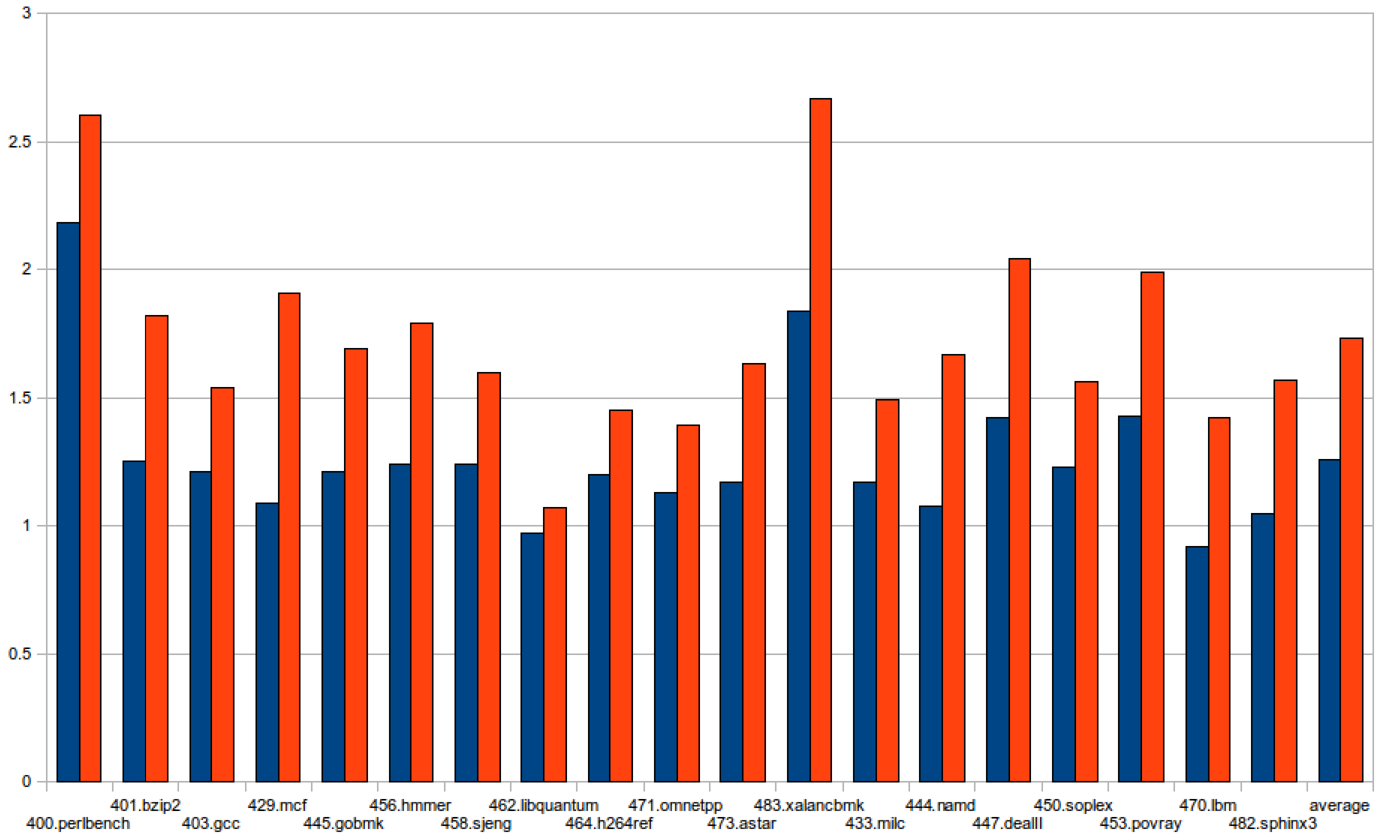
AddressSanitizer

Compile with `` `clang -fsanitize=address` ``

Asan is a memory error detector for C/C++. It finds:

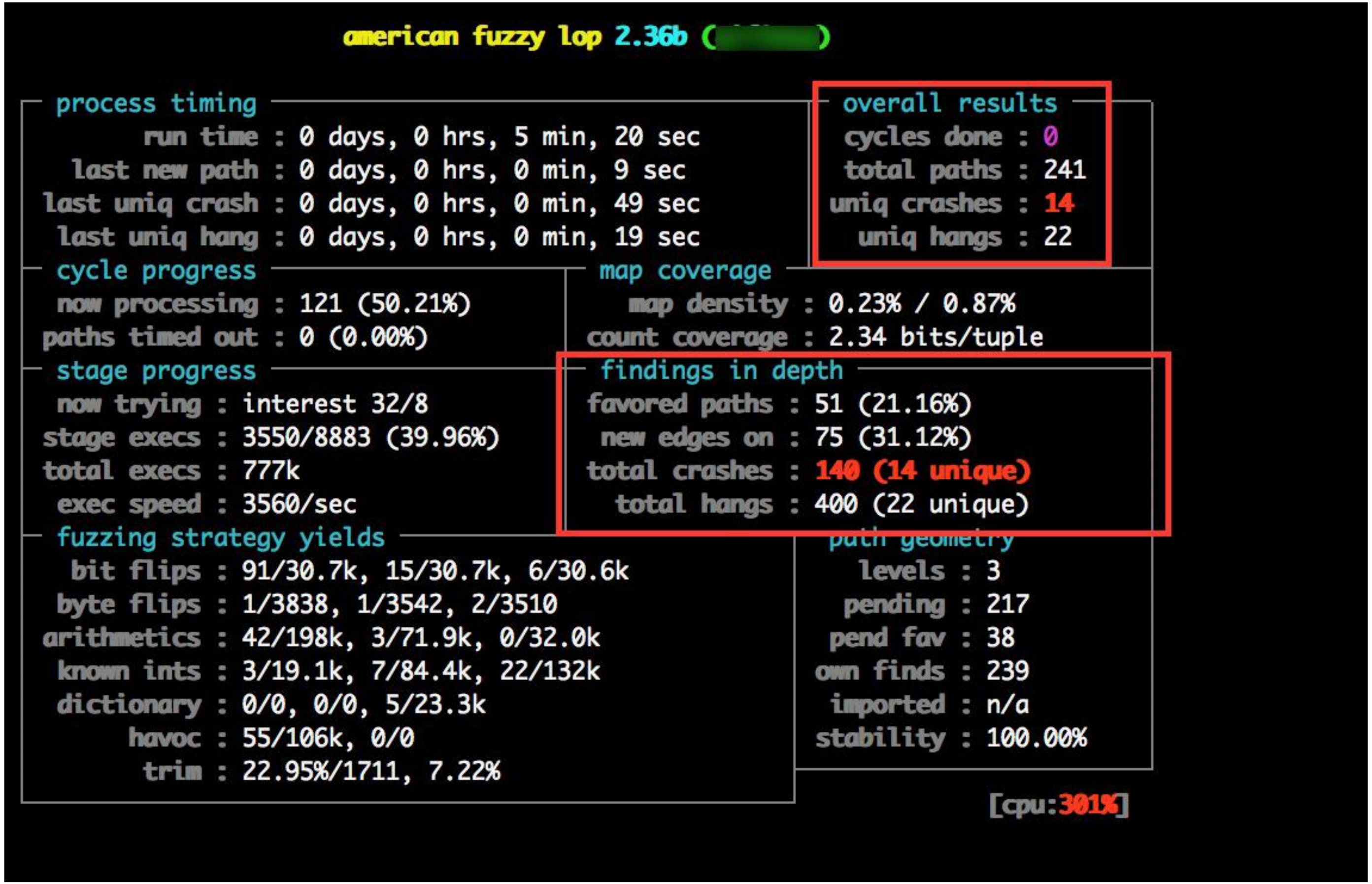
- User after free (dangling pointer dereference)
- Heap/stack/global buffer overflow
- Use after return/scope
- Initialization order bugs
- Memory leaks

AddressSanitizer: Overhead



Source https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers

Crash Triaging



Crash Triaging

Given crashing inputs x1 and x2, do they trigger the same bug?

- **Very** difficult to answer in practice
- Heuristics to check if bug(x1) == bug(x2):
  - exitcode(x1) == exitcode(x2) // or exception or error msg
  - coverage(x1) == coverage(x2)
  - stacktrace(x1) == stacktrace(x2)
  - newcoverage(x1, old) == newcoverage(x2, old)
  - fix(x1) == fix(x2)

Crash Triaging in AFL

A crash is considered unique if any of two conditions are met:

- Crash trace includes a **tuple not seen in previous** crashes
- Crash trace is **missing a tuple that was always present** in earlier crashes.

Note: may result in unique crash count inflation early on, but it exhibits a very strong self-limiting effect after early stage.

Fuzzers in the build pipeline?

Example is ClusterFuzzLite. Has a “code-change” mode:

- Will fuzz the code base for every pull request or commit
- Defaults to fuzzing for 10 minutes
- Quits after finding a single crash (will not run other fuzzers)
- Only reports new crashes (likely introduced due to commit)

Fuzzers in parallel to the build pipeline?

ClusterFuzz also has a “batch” fuzzing mode to fuzz for longer durations (called “continuous fuzzing”):

- Report all crashes, not just new ones.
- Useful to build an input corpus that can be used during “code-change” fuzzing.
- Continuous fuzzing:
  - Reuse input corpus of fuzzing old version when fuzzing new version
  - Run the fuzzers for a long time! Inputs may keep being discovered.

Many challenges when fuzzing in practice

- Fuzzing heuristics:
  - Which input to change? How many times? Which mutations?
  - Type of code coverage to use?
- Detecting bugs: what is a bug? Infinite loops? Race conditions? How to know when a bug was found?
- Debugging crashes:
  - Assuring crashes are reproducible
  - Identifying unique crashes/vulnerabilities
  - Input minimization

Many challenges when fuzzing in practice

- Fuzzing roadblocks:
  - Handling magic bytes & checksums. May require writing a test harness.
  - Dependencies in binary inputs (length of chunks, index into table)
  - Inputs with complex syntax and semantics (e.g. XML, JSON, C++)
  - Stateful applications like network protocols (see next lecture)
- There’s ongoing research to handle these obstacles

→ Fuzzing is partly a science and partly an art!

Required reading & references

Required reading:

- Technical "whitepaper" for afl-fuzz. Can be found in the afl repository in the file technical_details.txt

Optional reading:

- “The Art, Science, and Engineering of Fuzzing: A Survey” by Manes et al., IEEE Transactions on Software Engineering.
- “Registered Report: Dissecting American Fuzzy Lop” by Fioraldi et al., International Fuzzing Workshop (2022).