Software Security

Fuzzing – continued

whitebox fuzzing with SAGE

greybox fuzzing with afl

Erik Poll

Radboud Universiteit Nijmegen

Last & this week

1. Basic fuzzing with random/long inputs
2. 'Dumb' mutational fuzzing

   example: OCPP
3. Generational fuzzing aka grammar-based fuzzing

   example: GSM

4. Code-coverage guided evolutionary fuzzing with `afl`

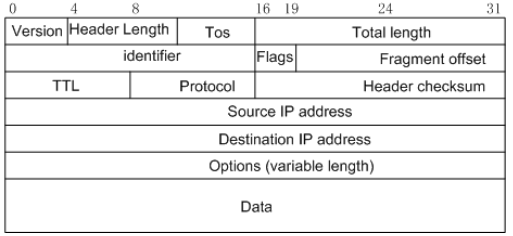
   aka grey box fuzzing or 'smart' mutational fuzzing
5. Whitebox fuzzing with SAGE

   using symbolic execution

Last week

1. **Totally dumb fuzzing** - generate random (long) inputs
2. **Mutation-based** - apply random mutations to valid inputs
   - Eg OCPP
   - Tools: Radamsa, zzuf, ...
3. **Generation-based** aka **grammar-based**
   - Eg GSM
   - Pro: can reach 'deeper' bugs than 1 & 2 ☺
   - Con: but lots of work to construct fuzzer or grammar ☹
   - Tools: SNOOZE, SPIKE, Peach, Sulley, antiparser, Netzob, ...

Less shallow



Today: more advanced strategies for testcase generation

Game changers in test-case generation:

4. **Whitebox approach of SAGE**

5. **Coverage-guided evolutionary fuzzing with afl**

   observe execution to try to learn which mutations are interesting

   - **aka greybox approach**

Whitebox fuzzing with SAGE

Whitebox fuzzing using symbolic execution

- The central problem with fuzzing: how can we generate inputs that trigger interesting code executions?

  Eg fuzzing the procedure below is unlikely to hit the error case

```
int foo(int x) {
    y = x+3;
    if (y==13) abort(); // error
}
```

- The idea behind whitebox fuzzing: if we know the code, then by analysing the code we can find interesting input values to try.
- SAGE from Microsoft Research that uses symbolic execution of x86 binaries to generate test cases.

```
m(int x,y){
    x = x + y;
    y = y - x;
    if (2*y > 8) { ...
                 }
    else if (3*x < 10){ ...
                       }
    }
```

Can you provide values for `x` and `y` that will trigger execution of the two if-branches?

Symbolic execution

```
m(int x,y){
    x = x + y;
    y = y – x;
    if (2*y > 8) { ...
                 }
    else if (3*x < 10){ ...
    }                  }
```

Suppose `x` = *N and* `y` = *M.*

`x` *becomes N+M*

`y` *becomes M - (N+M) = -N*

*if-branch taken if 2 * -N > 8, i.e. N < -4*

Aka the path condition

2nd if-branch taken if
*N ≥ -4 AND 3 * (M+N) < 10*

Given a set of constraints, an SMT solver (Yikes, Z3, …) produces values that satisfy it, or proves that it are not satisfiable.

This generates test data (i) *automatically* and (ii) *with good coverage*

- **SMT solvers can also be used for static analyses as in PREfast, or more generally, for program verification**

Symbolic execution for test generation

- Symbolic execution can be used to automatically generate test cases with good coverage
- Basic idea instead of giving variables concrete values (say 42), variables are given symbolic values (say α or N), and program is executed with these symbolic values to see when certain program points are reached
- *Downsides of symbolic execution?*
  - Very expensive (in time & space)
  - Things explode if there are loops or recursion, or if you make heavy use of the heap
  - You cannot pass symbolic values as input to some APIs, system calls, I/O peripherals, …

  SAGE mitigates these by using a *single concrete execution* to obtain *symbolic constraints* to generate *many* test inputs for *many* execution paths

SAGE example

Example program

```
void top(char input[4]) {
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 3) crash();
}
```

What would be interesting test cases?

Do you think a fuzzer could find them?

How could you find them?

SAGE example

Example program

```
void top(char input[4]) {
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 3) crash();
}
```

path contraints:

$i_0 \neq$ 'b'

$i_1 \neq$ 'a'

$i_2 \neq$ 'd'

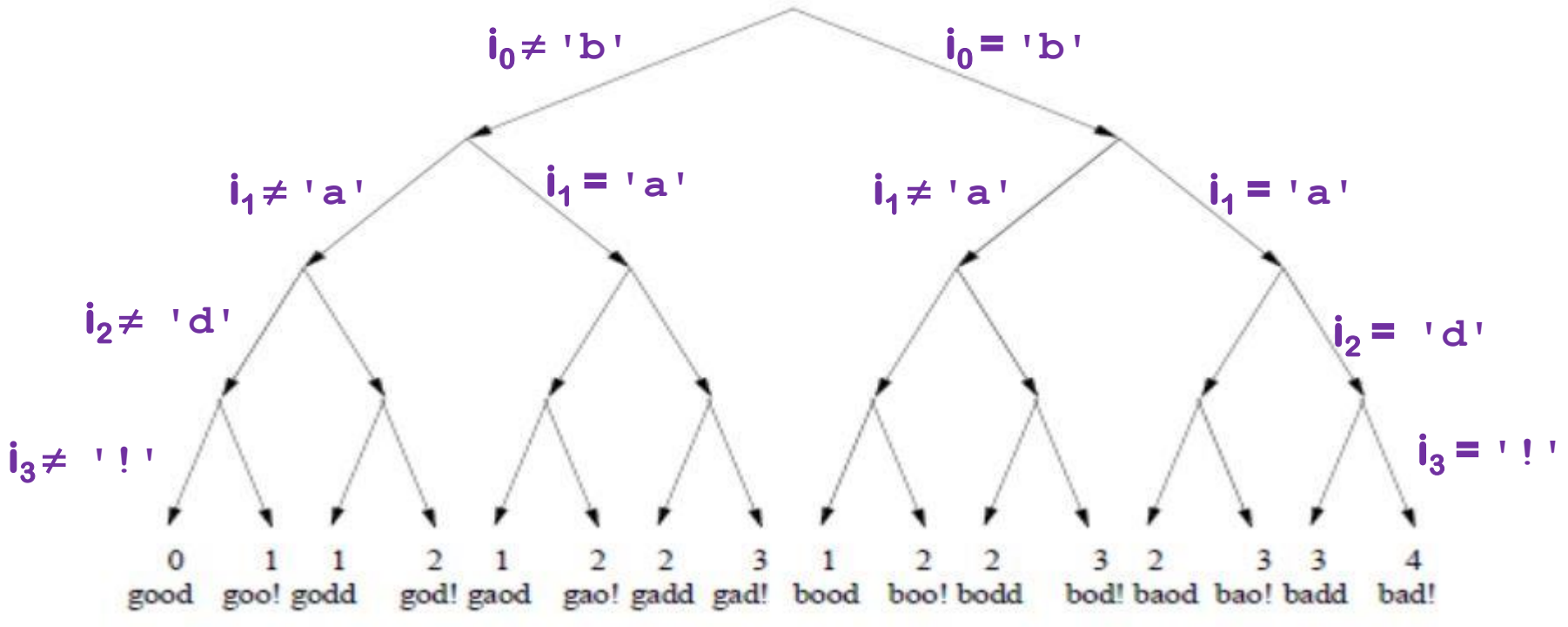
$i_3 \neq$ '!'

SAGE executes the code for some concrete `input`, say `'good'`

It then collects *path constraints* for an arbitrary symbolic input of the form $i_0 i_1 i_2 i_3$

Search space for interesting inputs

Based on this *one* execution, combining the 4 constraints found & their negations, yields $2^4 = 16$ test cases



Note: the initial execution with the input ‘good’ was not very interesting, but some of these others are

SAGE success

SAGE was very successful at uncovering security bugs, eg

> Microsoft Security Bulletin MS07-017 aka CVE-2007-0038: Critical
>
> Vulnerabilities in GDI Could Allow Remote Code Execution
>
> Stack-based buffer overflow in the animated cursor code in Windows … allows remote attackers to execute arbitrary code … via a large length value in the second (or later) `anih` block of a RIFF .ANI, cur, or .ico file, which results in memory corruption when processing cursors, animated cursors, and icons

Root cause: vulnerablity in PARSING of RIFF .ANI, cur, and ico-formats.

NB SAGE automatically generates inputs triggering this bug *without* knowing these formats

[Godefroid et al., *SAGE: Whitebox Fuzzing for Security Testing*, ACM Queue 2012]

[Patrice Godefroid, *Fuzzing: Hack, Art, and Science*, Communications of the ACM, 2020]

Coverage-guided evolutionary fuzzing with `afl`

(American Fuzzy Lop)

Evolutionary Fuzzing

Use evolution:

try random input mutations, and

observe the effect on some form of coverage, and

let only the interesting mutations evolve further

where “interesting” = resulting in ‘new’ execution paths

Aka coverage-guided evolutionary greybox fuzzing,

but terminology is a bit messy/non-standard

alf: observing jumps to find interesting inputs/input changes

input

code

line	instruction
1	**JMP 6**
2	..
3	..
4	..
5	**JZ (Jump If Zero) 7**
6	..
7	*arraycopy (dst, input[i..j]);*
8	
9	..
10	**JCXZ 2**
11	..
12	..
13	*println (part of input);*
14	..
15	**JNE 103131**
16	..

afl bitmap `shared_mem`

	1	2	3	4	5	6	7	8	9	10	11	12
1						☑[1]						
2												
3												
4												
5							☑[3]					
6												
7												
8												
9												
10		☑[2]										
11												
12												
13												
14												
15												

afl [http://lcamtuf.coredump.cx/afl]

- Code instrumented to observe execution paths:
  - if source code is available, by using modified compiler
  - if source code is not available, by running code in an emulator
- Code coverage represented as a 64KB bitmap:
  each control flow jumps is mapped to a change in this bitmap
  - different executions could result in same bitmap, but chance is small
- Mutation strategies include: bit flips, incrementing/decrementing integers, using pre-defined interesting values (eg. 0, -1, MAX_INT,....) or user-supplied dictionary, deleting/combining/zeroing input blocks, ...
- The fuzzer forks the SUT to speed up the fuzzing
- Big win: no need to specify the input format, but still good coverage

afl's instrumentation of compiled code

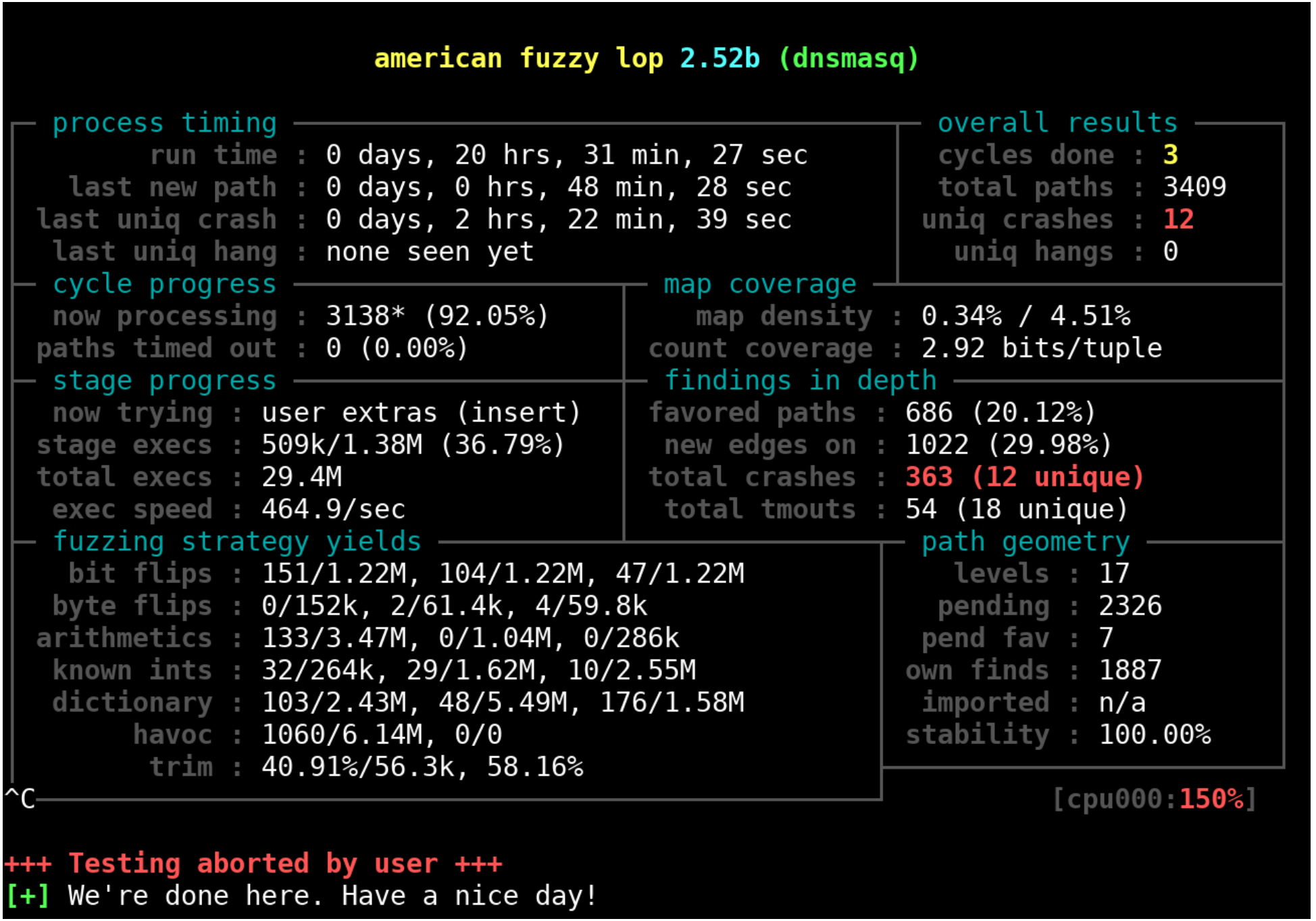
Code is injected at every branch point in the code

```
cur_location = <SOME_RANDOM_NUMBER_FOR_THIS_CODE_BLOCK>;

shared_mem[cur_location ^ prev_location]++;

prev_location = cur_location >> 1;
```

where `shared_mem` is a 64 KB memory region

Intuition: for every jump from L_1 to L_2 a different byte in `shared_mem` is changed (increased).

Which byte is determined by random values chosen at compile time inserted at source and destination of every jump


american fuzzy lop 2.52b (dnsmasq)
process timing
run time : 0 days, 20 hrs, 31 min, 27 sec
last new path : 0 days, 0 hrs, 48 min, 28 sec
last uniq crash : 0 days, 2 hrs, 22 min, 39 sec
last uniq hang : none seen yet
overall results
cycles done : 3
total paths : 3409
uniq crashes : 12
uniq hangs : 0
cycle progress
now processing : 3138* (92.05%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.34% / 4.51%
count coverage : 2.92 bits/tuple
stage progress
now trying : user extras (insert)
stage execs : 509k/1.38M (36.79%)
total execs : 29.4M
exec speed : 464.9/sec
findings in depth
favored paths : 686 (20.12%)
new edges on : 1022 (29.98%)
total crashes : 363 (12 unique)
total tmouts : 54 (18 unique)
fuzzing strategy yields
bit flips : 151/1.22M, 104/1.22M, 47/1.22M
byte flips : 0/152k, 2/61.4k, 4/59.8k
arithmetics : 133/3.47M, 0/1.04M, 0/286k
known ints : 32/264k, 29/1.62M, 10/2.55M
dictionary : 103/2.43M, 48/5.49M, 176/1.58M
havoc : 1060/6.14M, 0/0
trim : 40.91%/56.3k, 58.16%
path geometry
levels : 17
pending : 2326
pend fav : 7
own finds : 1887
imported : n/a
stability : 100.00%
^C
[cpu000:150%]
+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

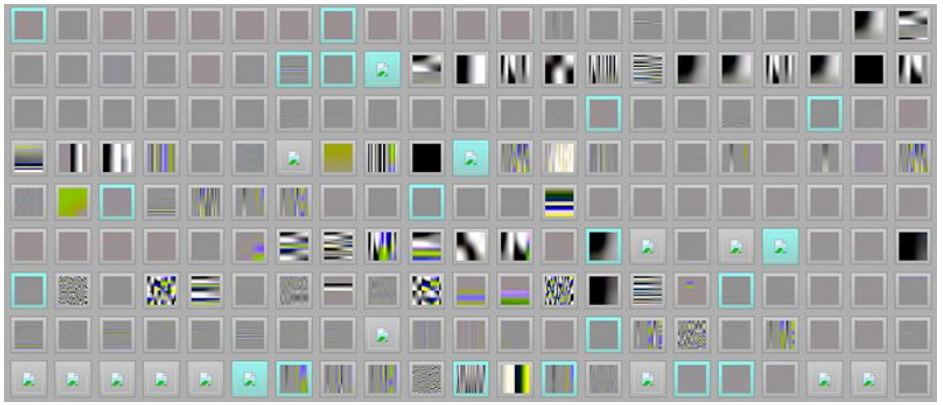
Cool example: learning the JPG file format

Fuzzing a program that expects a JPG as input, starting with 'hello world' as initial test input, afl can learn to produce legal JPG files

along the way producing/discovering error messages such as

- `Not a JPEG file: starts with 0x68 0x65`
- `Not a JPEG file: starts with 0xff 0x65`
- `Premature end of JPEG file`
- `Invalid JPEG file structure: two SOI markers`
- `Quantization table 0x0e was not defined`

and then JPGs like

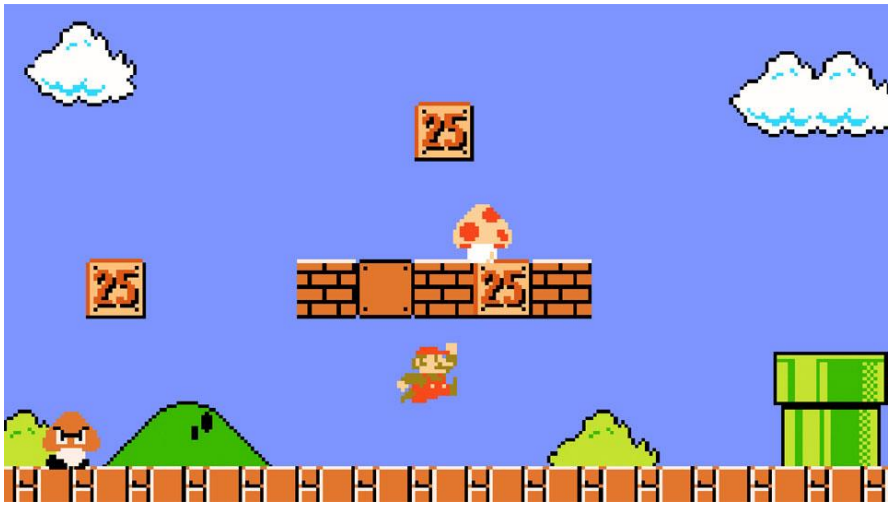


[Source http://lcamtuf.blogspot.nl/2014/11/pulling-jpegs-out-of-thin-air.html]

Other strategies in evolutionary fuzzing

Instead of maximizing path/code coverage, we can also let inputs evolve to maximize some other variable or property

- **Code may need to instrumented to let fuzzer observe that property**



Eg the x-coordinate of Super Mario

[Aschermann et al., *IJON: Exploring Deep State Spaces via Fuzzing,* IEEE S&P 2020]

https://www.youtube.com/watch?v=3PyhXlHDkNI

Conclusions

- Fuzzing is great technique to find (a certain class of) security flaws!
- If you ever write or use C(++) code, you should fuzz it.
- Challenge: getting good coverage fuzzing without too much effort

  Successful approaches include

  - White-box fuzzing based on symbolic execution with SAGE
  - Evolutionary fuzzing aka coverage guided greybox fuzzing with `afl`
- *Does fuzzing makes sense for code in other programming languages?*

  Yes, even if the kind of bugs found may have lower security impact.
- A more ambitious generation of tools not only tries to find security flaws, but also to then build exploits, eg. `angr`

To read (see links on the course page)

- Section 1 of technical white paper for afl
- Patrice Godefroid, *Fuzzing: Hack, Art, and Science* CACM 2020