

Software Security

Fuzzing – continued

whitebox fuzzing with SAGE

greybox fuzzing with afl

Erik Poll

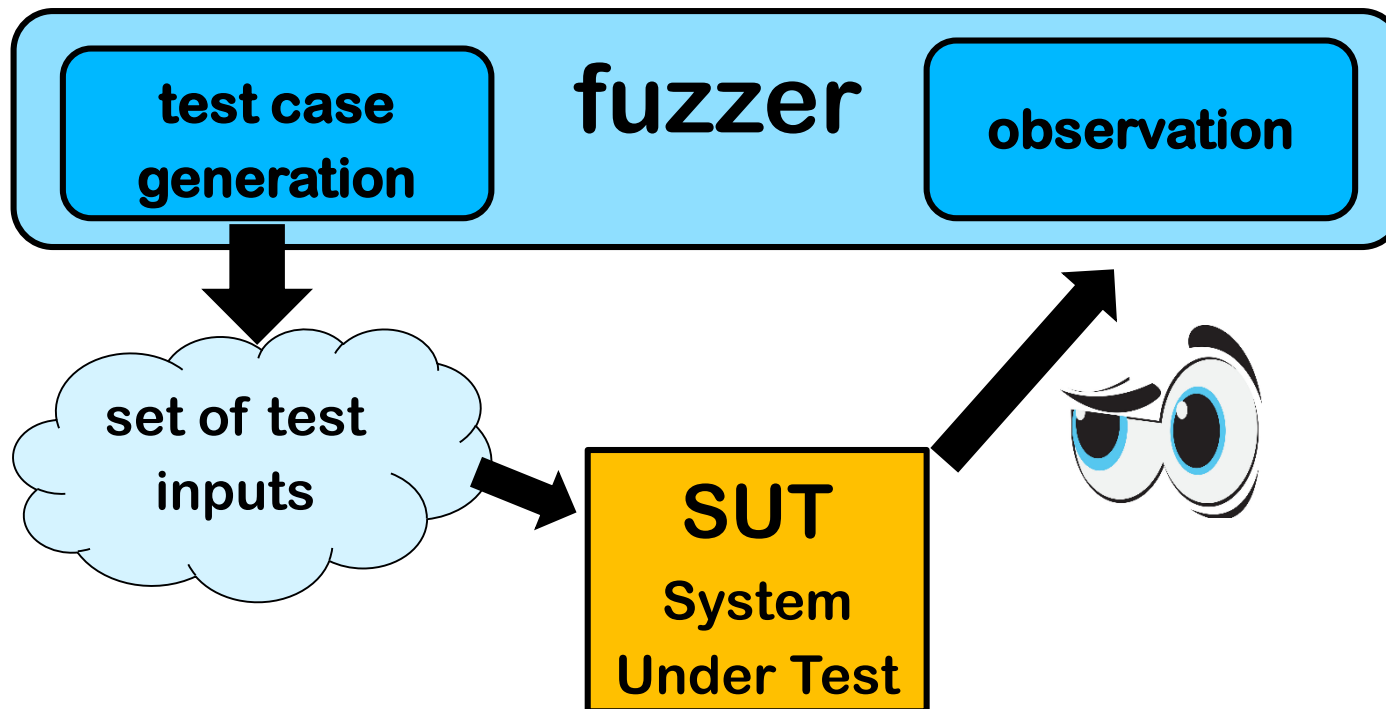
Radboud Universiteit Nijmegen



Fuzzing

(semi) *automatically* generate 'random' inputs and check if an application *crashes* or *misbehaves in observable way*

Great for certain classes of bugs, esp. memory corruption bugs



First tool for this: `fuzz` for UNIX

[Miller et al., An empirical study of the reliability of UNIX utilities, CACM 1990]

Last week

1. Totally dumb fuzzing

generate random (long) inputs

2. Mutation-based

apply random mutations to valid inputs

- Tools: **Radamsa**, **zzuf**, ...

3. Generation-based aka grammar-based

generate semi-well-formed inputs
based on knowledge of file format or protocol

- With tailor-made fuzzer for a specific input format, eg **FrankenCert**, or a generic fuzzer configured with a grammar
- 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**, ...

All fuzzers use variations & combinations of these 3 approaches

Less
shallow



| | | | | | | |
|---------------------------|---------------|----------|-------|-----------------|----|----|
| 0 | 4 | 8 | 16 | 19 | 24 | 31 |
| Version | Header Length | | Tos | Total length | | |
| identifier | | | Flags | Fragment offset | | |
| TTL | | Protocol | | Header checksum | | |
| Source IP address | | | | | | |
| Destination IP address | | | | | | |
| Options (variable length) | | | | | | |
| Data | | | | | | |

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

*$N \geq -4$ 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
 - 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 constraints:

$i_0 \neq 'b'$

$i_1 \neq 'a'$

$i_2 \neq 'd'$

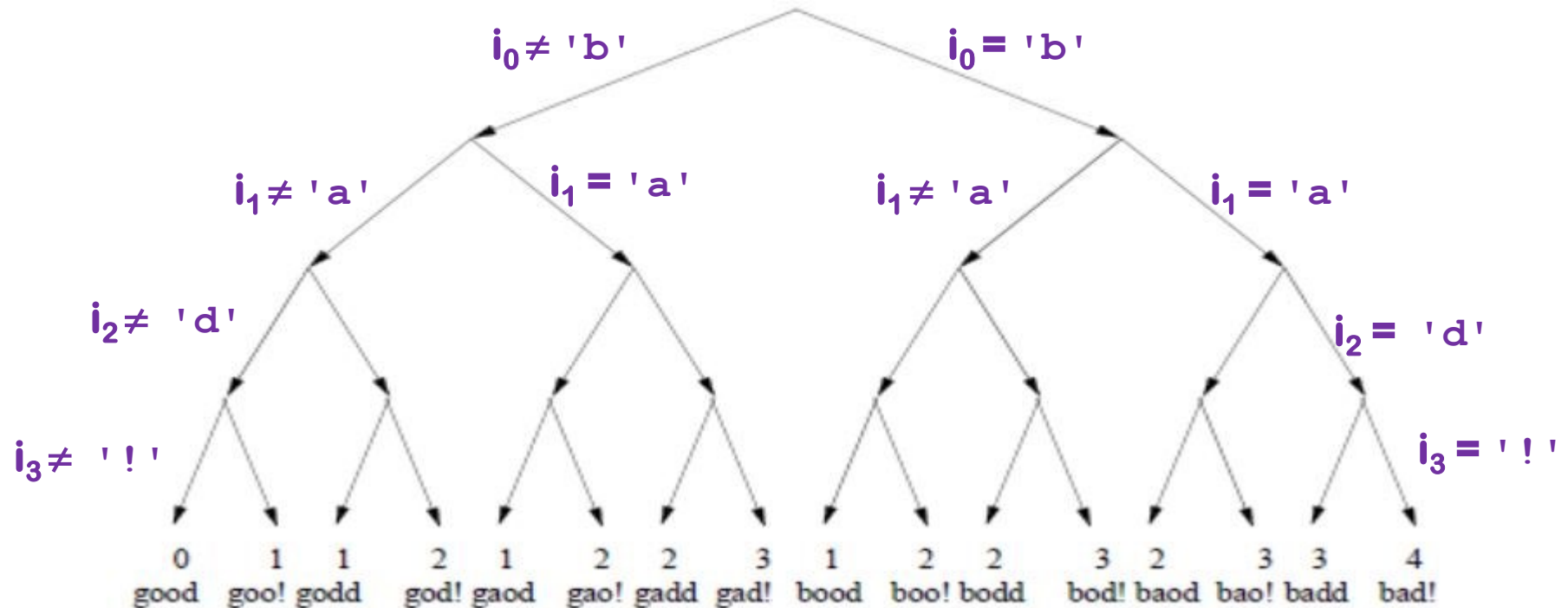
$i_3 \neq '!'$

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

It then collects *path constraints* for an arbitrary **symbolic input** of the form $i_0i_1i_2i_3$

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: vulnerability 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 greybox, but terminology is a bit messy...

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 `src` to `dest` a different byte in `shared_mem` is changed.

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

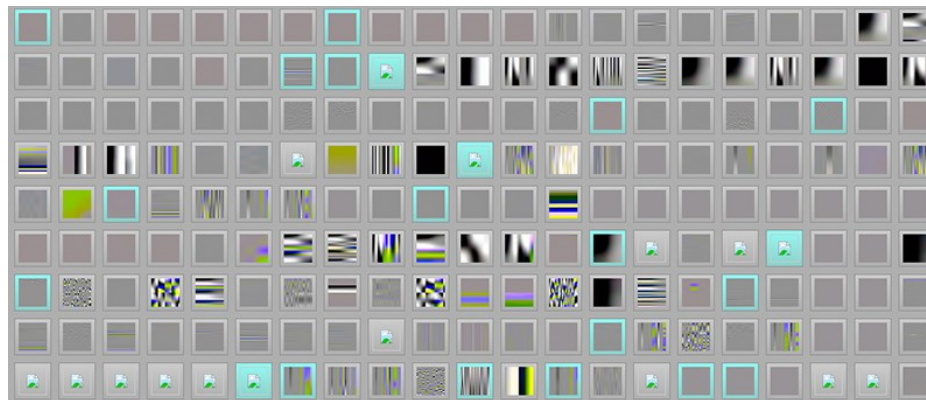
Cool example: learning the JPG file format

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>]

Vulnerabilities found with afl

IJG jpeg [1](#)
libtiff [1](#) [2](#) [3](#) [4](#) [5](#)
Mozilla Firefox [1](#) [2](#) [3](#) [4](#)
Adobe Flash / PCRE [1](#) [2](#) [3](#) [4](#)
LibreOffice [1](#) [2](#) [3](#) [4](#)
GnuTLS [1](#)
PuTTY [1](#) [2](#)
bash (post-Shellshock) [1](#) [2](#)
pdfium [1](#) [2](#)
BIND [1](#) [2](#) [3](#) ...
Oracle BerkeleyDB [1](#) [2](#)
FLAC audio library [1](#) [2](#)
strings (+ related tools) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#)
Info-Zip unzip [1](#) [2](#)
NetBSD bpf [1](#)
clang / llvm [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) ...
mutt [1](#)
pdksh [1](#) [2](#)
redis / lua-cmsgpack [1](#)
perl [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#)...
SleuthKit [1](#)
exifprobe [1](#)
Xerces-C [1](#) [2](#) [3](#)
libjpeg-turbo [1](#) [2](#)
mozjpeg [1](#)
Internet Explorer [1](#) [2](#) [3](#) [4](#)
sqlite [1](#) [2](#) [3](#) [4](#)...
poppler [1](#)
GnuPG [1](#) [2](#) [3](#) [4](#)
ntpd [1](#) [2](#)
tcpdump [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#)
ffmpeg [1](#) [2](#) [3](#) [4](#) [5](#)
QEMU [1](#) [2](#)
Android / libstagefright [1](#) [2](#)
libsndfile [1](#) [2](#) [3](#) [4](#)
file [1](#) [2](#) [3](#) [4](#)
libtasn1 [1](#) [2](#) ...
man & mandoc [1](#) [2](#) [3](#) [4](#) [5](#) ...
nasm [1](#) [2](#)
procmail [1](#)
Qt [1](#) [2](#)...
taglib [1](#) [2](#) [3](#)
libxmp
fwknop [reported by author]
jhead [?]
metacam [1](#)
libpng [1](#)
PHP [1](#) [2](#) [3](#) [4](#) [5](#)
Apple Safari [1](#)
OpenSSL [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#)
freetype [1](#) [2](#)
OpenSSH [1](#) [2](#) [3](#)
nginx [1](#) [2](#) [3](#)
JavaScriptCore [1](#) [2](#) [3](#) [4](#)
libmatroska [1](#)
lcms [1](#)
iOS / ImageIO [1](#)
less / lesspipe [1](#) [2](#) [3](#)
dpkg [1](#) [2](#)
OpenBSD pfctl [1](#)
IDA Pro [reported by authors]
ctags [1](#)
fontconfig [1](#)
wavpack [1](#)
privoxy [1](#) [2](#) [3](#)
radare2 [1](#) [2](#)
X.Org [1](#) [2](#)
capnproto [1](#)
djvulibre [1](#)

Other strategies in evolutionary fuzzing

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



Eg the **x-coordinate of Super Mario**

Code has to be instrumented to let fuzzer observe that property

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

<https://www.youtube.com/watch?v=3PyhXIHDkNI>

Conclusions

- Fuzzing is great technique to find (a certain kind of) security flaws!
- If you ever write or use some C(++) code, you should fuzz it.
- The bottleneck: how to do smart fuzzing without too much effort

Successful approaches include

- White-box fuzzing based on symbolic execution with **SAGE**
- Evolutionary mutation-based fuzzing with **afl**
- 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)

- **David Wheeler**, *The Apple goto fail vulnerability: lessons learned*
- **Patrice Godefroid**, *Fuzzing: Hack, Art, and Science* **CACM 2020**