Topics in Binary Program Analysis

CS-UY 3943 / CS-GY 9223

Prof Dolan-Gavitt

Getting Tests for Coverage

- Last time we looked at one way of generating test suites: symbolic execution
- However, we saw some problems with symbolic execution as well:
  - Path explosion: we may generate too many states to explore
  - Constraint complexity

A Pathological Example

- if (md5(input) & 0xFF) == 0) bug();
- Symbolic execution will have a very difficult time with this – inverting an MD5 hash is painful
- By contrast fuzzing will find it very quickly (1/256 random inputs will satisfy it)

The Main Fuzzing Advantage: Speed

- Symbolic execution is much, much slower than concrete execution
- This means that a fuzzer can try thousands of inputs per second (depending on the target)
- Sometimes speed beats smarts!

An Incomplete History

- 1981: Duran and Ntafos, "A report on random testing"
- 1983: Apple's "Monkey" (generated random UI events to test first Mac)
- 1988: Bart Miller, "An Empirical Study of the Reliability of UNIX Utilities"
- 1990s: crashme, X11 fuzzers
- 2000s: fuzzing frameworks: SPIKE, Sulley, PEACH
- 2005: DART – directed fuzzing
- 2008: SAGE – concolic fuzzing
- 2013-present: The "smart fuzzer" revolution (AFL, libfuzzer)

Fuzzing like it's 1988

Utility	VAX (v)	Sun (s)	HP (h)	i386 (x)	AIX 1.1 (a)	Sequent (d)
adb	●○	●	●	○	–	–
as	●			●	●	●
awk						
bc				●○		
bib			–	–	–	–
calendar				–		
cat						
cb	●		●	●	○	●
cc						
~						~
latex			–	–	–	–
lex	●	●	●	●	●	●
lint						
lisp		–		–	–	–
look	●	○	●	●	–	●

Table 2: List of Utilities Tested and the Systems on which They Were Tested (part 1)

*● = utility crashed, ○ = utility hung, * = crashed on SunOS 3.2 but not on SunOS 4.0,*
⊕ = crashed only on SunOS 4.0, not 3.2. – = utility unavailable on that system.
! = utility caused the operating system to crash.

Fuzzer Taxonomy

- Generative vs mutation-based
- "Dumb" or "smart" (w.r.t. input structure)
- White-box / grey-box / black-box

Generative vs Mutational

- The basic distinction here: whether you craft inputs from scratch or mutate existing ones
- Generational fuzzing: inputs created from scratch
- Mutational: inputs created by mutating a set of seeds
    - We can do this in stages: mutate, pick the best candidates, mutate those more, etc.

Mutation Fuzzing: Operators

- The success of a mutational fuzzer is highly dependent on its mutation operators
- AFL uses the following ones:
  - Sequential bit flips (up to 4 sequential bits)
  - Byte flips (1, 2, and 4 bytes at a time)
  - Arithmetic: add or subtract small integer values
  - Setting "well-known" integers (e.g., -1, 256, 1024, MAX_INT-1, MAX_INT)
  - Block delete / duplicate (overwrite and insert)
  - Splicing two inputs together

Dumb vs Smart

- Dumb strategy: just generate random bit strings
- *Grammar-based* fuzzers are on the "smart" side
  - Write down a complete grammar specifying your input
  - Then generate strings that match this grammar
  - Downside: building a correct grammar is a lot of work
  - Downside: May need to break the grammar to find bugs
- Note: dumb is not necessarily bad...

Example Grammar: HTTP Dates

```
HTTP-date    = rfc1123-date | rfc850-date | asctime-date
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date  = weekday "," SP date2 SP time SP "GMT"
asctime-date = wkday SP date3 SP time SP 4DIGIT
date1        = 2DIGIT SP month SP 4DIGIT
               ; day month year (e.g., 02 Jun 1982)
date2        = 2DIGIT "-" month "-" 2DIGIT
               ; day-month-year (e.g., 02-Jun-82)
date3        = month SP ( 2DIGIT | ( SP 1DIGIT ))
               ; month day (e.g., Jun  2)
time         = 2DIGIT ":" 2DIGIT ":" 2DIGIT
               ; 00:00:00 - 23:59:59
wkday        = "Mon" | "Tue" | "Wed"
             | "Thu" | "Fri" | "Sat" | "Sun"
weekday      = "Monday" | "Tuesday" | "Wednesday"
             | "Thursday" | "Friday" | "Saturday" | "Sunday"
month        = "Jan" | "Feb" | "Mar" | "Apr"
             | "May" | "Jun" | "Jul" | "Aug"
             | "Sep" | "Oct" | "Nov" | "Dec"
```

What Color Box?

- White/grey/black-box refers to how much the fuzzer knows about the program it's fuzzing
- Whitebox fuzzers get source code access, can perform arbitrary analyses
- Blackbox fuzzers don't look at the program at all
- Greybox fuzzers are in between: they get some limited amount of insight into program structure

Blackbox Fuzzing

- Here we have no access to the program, so we just run inputs on it
- Advantage: this means we can test any target (including remote services)
- Disadvantage: may not be very efficient

Whitebox Fuzzing

- With whitebox fuzzing, we can do deep analysis of the program to decide what to fuzz and how
- We can examine *dataflow* through the program: Ganesh et al., **Taint-based Directed Whitebox Fuzzing**
- We can leverage symbolic execution! This is the approach taken by **SAGE** (Godefroid et al.)

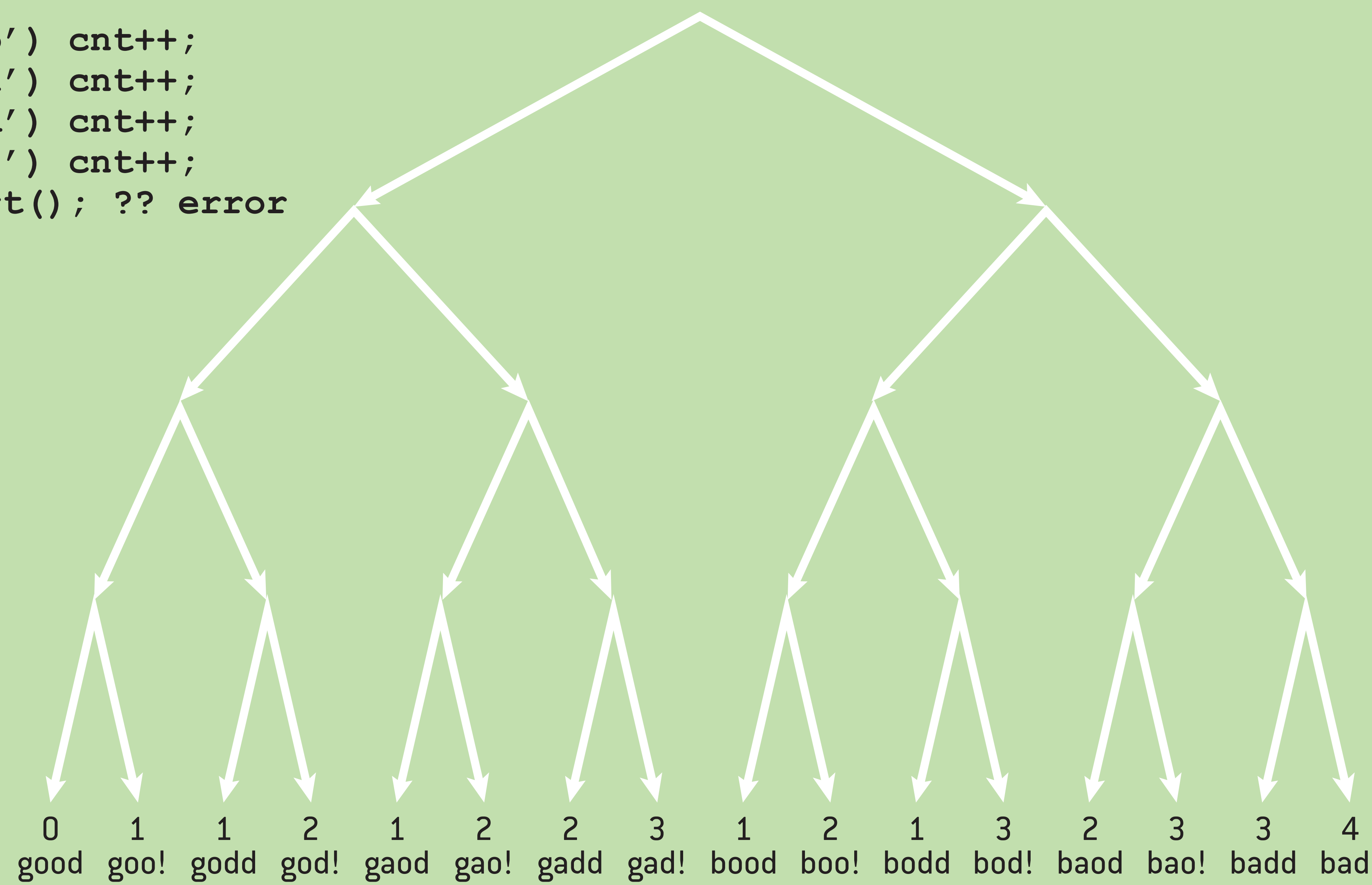
SAGE

1. Start with a set of seed inputs
2. Run program on each input and collect a trace
3. Execute each symbolically *without* forking – follow the same path taken by concrete input
   - This gives a path constraint for each input: (P1, P2, P3, ..., PN)
4. Now systematically negate each path constraint, solve, and use the resulting input as a new test seed
5. GOTO 1

FIGURE 2

Example of Program (Left) and Its Search Space (Right) with the Value of cnt at the End of Each Run

```
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 >= 4) abort(); ?? error
}
```



Source: SAGE: Whitebox Fuzzing for Security Testing

SAGE Success Story

- SAGE has been in use at Microsoft since 2007
- Found 1/3 of all bugs found from file-format fuzzing in Windows 7 before release
  - SAGE ran last – so these were all bugs missed by everything else
- Last year launched as a cloud service: Project Springfield https://www.microsoft.com/en-us/research/project/project-springfield/

Greybox Fuzzing

- In between the two extremes we have greybox fuzzing
- The category was pretty much invented for American Fuzzy Lop (AFL)
- The idea is that we use some feedback to tell us which test cases are most promising
- In the case of AFL, that feedback is *edge coverage*

American Fuzzy Lop

- Currently the most popular greybox fuzzer: very little setup required, achieves strong results

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
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
GnuTLS 1	GnuPG 1 2 3 4	OpenSSH 1 2 3 4 5
PuTTY 1 2	ntpd 1 2	nginx 1 2 3

AFL's Coverage Guidance

- Not full path coverage – edge coverage metric
- These are considered different:
    - A -> B -> C -> D -> E
    - A -> B -> C -> A -> E
- But this path is not:
    - A -> B -> C -> A -> B -> C -> A -> B -> C -> D -> E

AFL's Coverage Guidance

- Coverage tracking does include edge "hit count" divided into buckets: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
- Covered edges are tracked in a bitmap
- Inputs that produce new bitmap values are added to the set of inputs (but do not replace existing items)

Beyond AFL

- Another interesting greybox fuzzer is libfuzzer (part of LLVM project)

- Tougher to get going: you need to modify your program to add a test harness:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)
```

- Benefit: much faster fuzzing, can do more interesting feedback

- New feedback: *array index values, dataflow, division* https://clang.llvm.org/docs/SanitizerCoverage.html#tracing-data-flow

Fuzzing: Instrumentation

- With all of this fuzzing, we will generate millions/billions of inputs
- What do we keep? What are we trying to accomplish?
- We could try to reduce our set of test cases to the minimum number needed to get same coverage
  - This is called *test corpus reduction*
  - We can also try to shrink the size of individual test cases: *test case reduction*
- We could keep only those that cause a problem

Detecting Problems

- Detecting crashes themselves is pretty easy (at least on desktop systems – what about embedded devices?)
- But we may want to detect other errors that don't lead to crashes
  - Memory leaks
  - Out-of-bounds read/write
  - Integer overflow, undefined behavior
- One solution is to use a *sanitizer*: an instrumented version of the program that can flag errors at runtime that may not crash under normal circumstances
- Many sanitizers now: ASAN, TSAN, MSAN, UBSAN