BLACK-BOX FUZZING

CS4239 Software Security

Roland Yap

ryap@comp.nus.edu.sg

National University of Singapore

Acknowledgments: Abhik Roychoudhury, Mauro Pezze’, University of Lugano

OVERVIEW

- Background – Black Box Testing
  - includes Functional Testing
- Black Box Fuzzing
- Measuring Black Box Fuzzing
- White Box Fuzzing
- Grey Box approaches

Finding Bugs / Vulnerabilities

- vulnerability is a bug in program / code / binary / library / executable / service
  - not all bugs are vulnerabilities
- how to find bugs?
  - In particular, bugs which are more likely to be vulnerabilities
    - e.g. memory errors
  - In this part, we focus on **automated vulnerability finding**

Finding Bugs 1

- Verification / Model Checking
  - good: formal proof, clear guarantees
    - systematic / exhaustive
  - minus: very hard (undecidable in worst case), not scalable, automation is limited
- Static Analysis
  - verify certain properties, e.g. memory safety
  - good: automation, clear guarantees
  - minus: false positives, less scalable
- Reverse Engineering / Source Code Auditing
  - good: exploit human ingenuity
  - minus: mostly/partly manual, not scalable

Finding Bugs 2

- Testing Methods
  - Penetration Testing (CS4238 – may be manual)
  - Software Testing
  - Fuzzing
    - Good: can be mostly/fully automated
    - Minus: no guarantees (may find bug), expensive runtimes (use cloud)

Fuzzing Features

- no simple definition of fuzzing
- Some elements:
  - input / interface testing
  - random element to test generation
  - find bugs using crashes / errors / abnormal behavior
  - many frameworks + automation

Limitations of Testing / Fuzzing

- may find some bugs
  - no guarantee bug is found
- cannot prove no bugs
  - does not show correctness
  - does not prove no misbehavior
  - usually no guarantees
- **but many vulnerabilities are successfully found through fuzzing**
  - Linus Torvalds (OSS 2017): "**random attacks is very powerful for finding various bugs**"
    - not purely random – see coverage guided fuzzing

Fuzzing Effectiveness

In practice – very effective. Some fuzzers below

perf_fuzzer

linux root exploit	CVE-2013-4254
linux crash	CVE-2013-2930

quickfuzz

FF integer overflow	CVE-2016-1933
VLC mem error	CVE-2016-3941

cross_fuzz

IE mem error	CVE-2011-0346
FF mem error	CVE-2016-2827

libfuzzer

libxml2 CPU DOS	CVE-2015-5312
OpenSSL mem error	CVE-2016-2108

AFL (numerous)

Android Stagefright 2015	many CVEs (CVE-2015-1538, CVE-2015-1539, ...)

see many at **https://github.com/mrash/afl-cve**

Fuzzing in Google

- Google ClusterFuzz
  - open source scalable fuzzing infrastructure – used for fuzzing Chrome (https://github.com/google/clusterfuzz)
  - Google: runs on over 25,000 machines
  - As of January 2019, ClusterFuzz has found ~16,000 bugs in Chrome and ~11,000 bugs in over 160 open source projects integrated with OSS-Fuzz

History of fuzzing

Developed by Barton Miller, see

http://pages.cs.wisc.edu/~bart/fuzz/

below some history, not our definition

Fuzz testing is a simple technique for feeding random input to applications. The approach has three characteristics.

- *The input is **random**. We do not use any model of program behavior, application type, or system description. This is sometimes called **black box testing**.*
- *The reliability criteria is **simple**: if the application **crashes or hangs**, it is considered to fail the test, otherwise it passes. Note that the application does not have to respond in a sensible manner to the input, and it can even quietly exit.*
- *As a result of the first two characteristics, fuzz testing can be **automated** to a high degree and results can be compared across applications, operating systems, and vendors.*

Random Fuzzing

```
// high level view
while (not reached stopping criteria) {
        I = generate random input;
        run test program on I
        if crashed
                log I
}

// more sophisticated fuzzers possible
```

Simple Fuzzing Assumptions

Assumptions:

- input (test case) encodes the bug
  - not looking at non-deterministic programs
- buggy input causes "crash"
  - **no specs needed**
- avoid (or minimal) analysis of program (or source code/binary)
  - **no source needed**

Fuzzer:

- strategies + algorithm to exercise the assumptions
  - **can be automated**

Testing with (Weak) oracles

```
input x;
if (f(x)){
    out = g(x);
}
else{
     out = h(x);
}
assert(out > 0)
print out
```

When do we say a particular test

Say

x > 0 passes?

Remember:
A fuzzer tool generates such inputs without needing to look at the program source code, and runs it on the program/binary to decide whether the test has passed.

1. Program does **not crash**
2. **No assertion failure**
3. Output printed is as **expected**

Weak Oracle

- Strong Oracle (for testing)
  - some way of determining that execution/output/result is correct
  - but how?
- Weak Oracle (for testing)
  - in absence of strong oracle – weaker proxy
  - don't check for correctness
  - weaker property which **may** or **may not** indicate correctness
    - assertion failure
    - memory error
    - timeout
    - crash
    - ...

Users of Fuzzers

- Broad scope
  - *Software developers*: The company wants to find as many bugs as possible, before shipping it out.
    - e.g. used extensively at Google
  - *Genuine software users*: After the company procures software, they want to stress test it as much as possible, before integrating it into their work-flows
  - *Attackers*: Understand the weak points of a software!
    - also Security researchers: find security bugs / CVEs

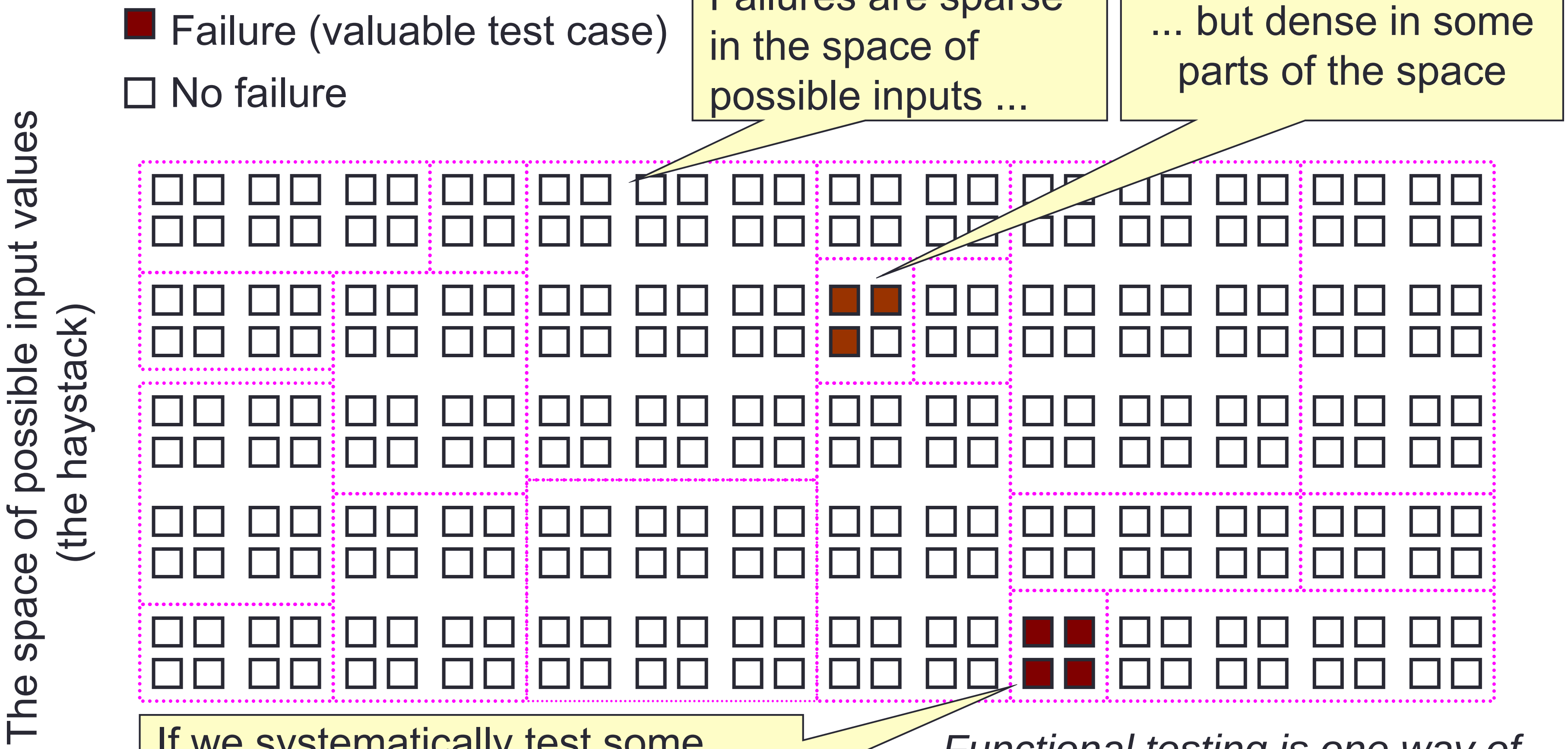
Definition of Functional Testing

- **Functional testing**: Deriving test cases from program specifications
    - *Functional* refers to the source of information used in test case design, not to what is tested
    - eg. unit testing
- *Also known as*:
    - specification-based testing (from specifications)
    - black-box testing (no view of the code)
- Functional specification = description of intended program behavior
    - either formal or informal

Systematic vs. Random

- Random (uniform):
  - Pick possible inputs (uniformly)
  - Avoids designer bias
    - A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
  - But treats all inputs as equally valuable
- Systematic (non-uniform):
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of classes that are apt to fail *often* or *not at all*
- **Functional testing is systematic testing.**
- **Fuzzing is closer to random black-box testing.**

Systematic Partition testing



■ Failure (valuable test case)

□ No failure

Failures are sparse in the space of possible inputs ...

... but dense in some parts of the space

If we systematically test some cases from each part, we will include the dense parts

Functional testing is one way of drawing pink lines to isolate regions with likely failures

Evaluating Functional Testing

- Functional testing is one variety of partition testing, a way of drawing the pink lines so that, when one of the boxes within a pink group is a failure, many of the other boxes in that group may also be failures.
  - Functional testing means using the program specification to draw partitions (pink lines)
  - but depends on how successful the partitions (pink lines) are drawn?

Random testing: Is it effective

- Example
  - A program which breaks a text buffer into 100 chars. lines
    - Random testing can find test buffer with < 100 chars
    - Random testing can find test buffer with > 100 chars
    - Random testing *may not* find test buffer with == 100 chars, and there may be a program error for this corner case.
    - Can be even less effective in certain situations
      - Example to follow: roots program

Using the specification

- Functional testing uses the specification (formal or informal) to partition the input space
- Test each category, and boundaries between categories
  - No guarantees, but experience suggests failures often lie at the boundaries (as in the "roots" program)

    - $x = (-b + \sqrt{(b^2 - 4ac)}) / (2 * a)$
    - $x = (-b - \sqrt{(b^2 - 4ac)}) / (2 * a)$

Square-root program

```
class Roots{
   double root1, root2; int num_roots;
   public roots(double a, double b, double c){
         double q, r;
         q = b*b  – 4*a*c;
         if (q>0 && a!=0){
             num_roots = 2; r = (double)Math.sqrt(q);
             root1 = ((0-b)+r)/(2*a);  root2 = ((0-b)-r)/(2*a);
         } else if (q == 0) {  // look for BUG
              num_roots = 1; root1 = (0-b)/(2*a);  root2 = root1;
         } else{     // equation has no roots if b^2 < 4ac
              num_roots = 0;  root1 = -1;  root2 = root1;
          }
     }
     public int   ….     //  the other methods of the class come here
}
```

Has a bug if

b^2 == 4ac and a == 0

i.e., <a == 0, b == 0>

Such a singular point could be easily missed by random testing, but partition based testing is set up to catch it.

Java sqrt

- special cases:
  - handle all inputs, are outputs numbers?
  - if the argument is **NaN** or less than zero, then the result is **NaN**.
  - If the argument is positive infinity, then the result is positive infinity.
  - If the argument is positive zero or negative zero, then the result is the same as the argument
- see later when we look at arithmetic

Random + Systematic Testing

- It is also possible to combine both – can be a less expensive, effective strategy in practice.
  - 1. Generate some boundary values first to capture corner cases or important relationships.
  - 2. Cover the rest of the input domain by random testing.

  - Example: Three integers a, b , c --- input to some method
    - Possible Corner cases: a= 0, b = 0, and so on
    - Possible Important Relationships: a == b
    - Cover the rest of the input domain by random testing.

Random vs. Systematic

- Completely random testing
- Completely systematic testing
  - Define input partitions
  - Select one test from each partition.
- Combinations
  - Generate special boundary values
  - Cover the rest of the input domain using random testing.

Why Functional Testing?

- The base-line technique for designing test cases
  - Timely
    - Often useful in refining specifications and assessing testability *before* code is written
  - Effective
    - finds some classes of fault (e.g., missing logic) that can elude other approaches
  - Widely applicable
    - to any description of program behavior serving as spec
    - at any level of granularity from module to system testing.
  - Economical
    - typically less expensive to design and execute than structural (code-based) test cases

Why Functional Testing?

- Program code is not necessary
  - Only a description of intended behavior is needed
  - Even incomplete and informal specifications can be used
    - Although precise, complete specifications lead to better test suites
- Early functional test design has side benefits
  - Often reveals ambiguities and inconsistency in spec
  - Useful for assessing testability
    - And improving test schedule and budget by improving spec
  - Useful explanation of specification
    - or in the extreme case (not uncommon), test cases are the spec

- **Fuzzing is (mostly) “random” black-box testing which is geared towards sending “abnormal or mal-formed” inputs.**

OVERVIEW

- Background – Black Box Testing (Functional Testing)
- Black Box Fuzzing
- Measuring Black Box Fuzzing
- White Box Fuzzing
- Grey Box approaches

Salient features of fuzzing 1

- Automated test generation
  - Favor slightly anomalous or malformed or illegal inputs
  - Apart from this issue, (mostly) try to keep test generation random
- Automated test execution
  - Of course
- Automated and weak notion of test oracle
  - No notion of expected output to see if a test is passing
  - no test case needed but can have seed inputs
  - Simply see if the application is hangs / crashes / strange state
  - crash usually = memory error / exception
    - memory defences help find more cases, e.g. ASAN, LowFat

Salient features of fuzzing 2

- Detailed record-keeping
    - For crashing tests, one may find **many crashing tests** by fuzzing
- Independent of any programming language, OS etc.
    - No analysis, only execution!
    - some execution book keeping may be needed, e.g. coverage guided fuzzing

Output of fuzzing

- Lot of crashing test inputs (tests)
  - Voluminous, not directly useful
  - Lot of crashing tests may be a manifestation of the same vulnerability.
  - *Need to **cluster** crashing tests based on why they crash!*
- *What do we do with output from fuzzing*
  - *Check whether attackers can exploit the vulnerability*
  - *Or, it may be easier to just fix the error rather than checking its **exploitability**.*

Cluster Crashing Tests

- Ideal
  - Semantic analysis of test to find a “reason”
  - Capture reason as logical formula, e.g. see white box fuzzing
  - Store a bucket of tests against each logical formula!
- Actual (typical)
  - Cluster tests based on point of failure
    - Failure location / function where failure occurs / path in CFG (control flow graph)
  - Cluster tests based on call-stack similarity
    - *Windows Error Reporting System*
- Why cluster
  - Developers may completely ignore fuzzing output otherwise!

Exploit Generation

```
void func(char *string){
   char buffer[32];
   strcpy(buffer, string);
}

int main(int argc, char **argv){
     func(argv[1]);
     return 0;
}
```

Fuzzing here amounts to finding `argv[1]` which gives an error (memory error → memory exploit → arbitrary code execution/...)

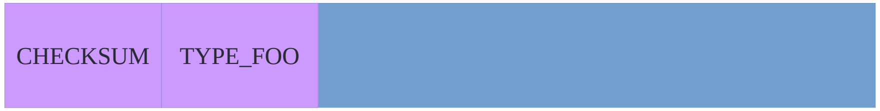
Simple example of Fuzzing

- Fuzz each field
  - Each field is fuzzed once below, can be fuzzed in many ways.
- Sample “correct” input
  - GET /index.html HTTP/1.1

- Fuzzed inputs
  - GGGGG /index.html HTTP/1.1
  - GET / / / / / /index.html HTTP/1.1
  - GET / GGGGG.html HTTP/1.1
  - GET /index.html HTTPPPPPPPP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1

Black-box Fuzzing – Random Permutation

- Randomly permute string as input, try to crash unix program.
- Simple and surprisingly highly effective.
- Problem with code **coverage** and **checksums**?


CHECKSUM TYPE_FOO

Program Input

```
uint8_t header [4] , bad_buf [16] , length ;
read ( input_file , & header , 4) ;
if ( header [0] == checksum ( header ) ) {
    if ( header [1] == TYPE_FOO ) {
        read ( input_file , & length , 1) ;
        read ( input_file , bad_buf , length ) ;
        // ... no more branches ...

    }
} else ...
```

Example Code

Mutational fuzzing

- Inputs
  - Program P
  - Seed input x0
  - Mutation ratio $0 < m \leq 1$
- Next step
  - Obtain an input x1 by randomly flipping m*|x0| bits (mutates x0)
  - Run x1 and check if P crashes or terminates properly.
  - In either case document the outcome, and generate next input.
- End of fuzz campaign
  - When time bound is reached, or N inputs are explored for some N.
  - Always make sure that bit flipping does not run same input twice.

Why depend on mutations?

- Many programs take in structured inputs
  - PDF Reader, library for manipulating TIFF, PNG images
  - Compilers which take in programs as input
  - Web-browsers, ...

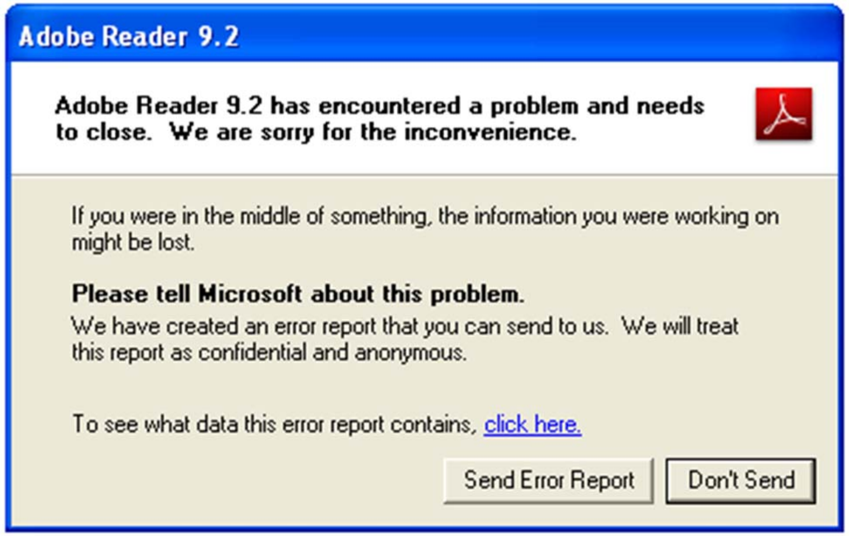
- Generating a completely random input will likely crash the application with little insight gained about the underlying vulnerability.

- Instead take a legal well-formed PDF file and mutate it!

Why depend on mutations?

- Principle of mutation fuzzing
  - Take a well-formed input which does not crash.
  - Minimally modify or mutate it to generate a “slightly abnormal” input
  - See if the “slightly abnormal” input crashes.
- Salient features
  - Does not depend on program at all (nature of BlackBox fuzzing)
  - Does not even depend on input structure.
  - Yet can leverage complex input structure by starting with a well-formed seed and minimally modifying it.

How far can mutations go?



Adobe Reader 9.2

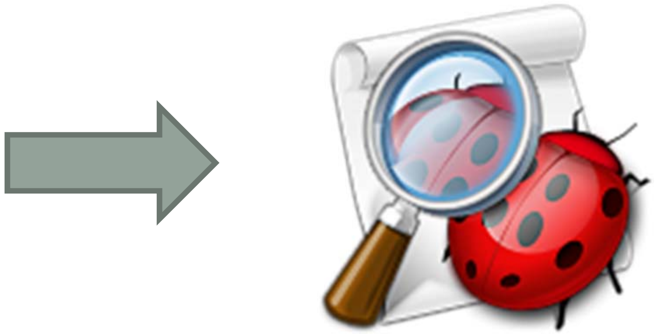
Adobe Reader 9.2 has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.



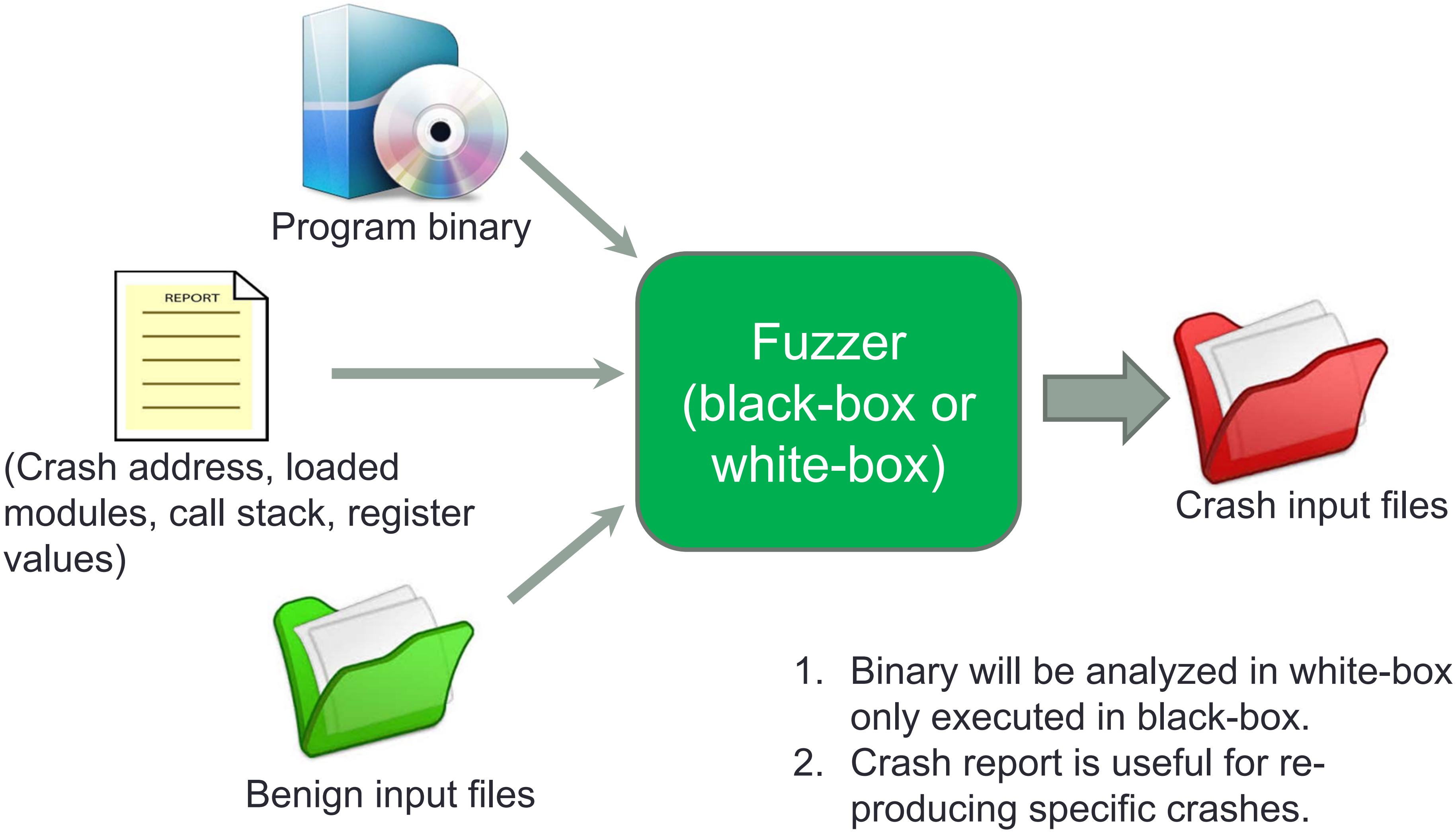
Crash reproducing supports

- In-house debugging and fixing
- Vulnerability checking

Mutate benign PDF files at hand – to produce problematic PDF file

possible to start from 0 length file

Fuzzing infra-structure


Program binary
REPORT
(Crash address, loaded modules, call stack, register values)
Fuzzer (black-box or white-box)
Crash input files
Benign input files

1. Binary will be analyzed in white-box, only executed in black-box.
2. Crash report is useful for re-producing specific crashes.

Generational fuzzing

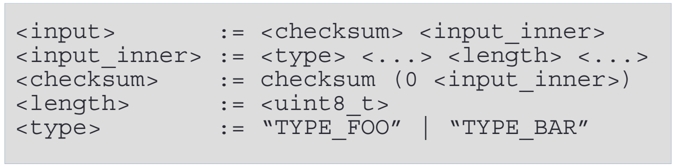
- Mutations can be random
  - Small probability of re-producing a specific crash in a specific byte deep down in the input file.
  - Huge number of mutations to try out.
- Mutations can produce ill-formed inputs
  - A mutated input file with incorrect check-sum
- To avoid these problems
  - Generational fuzzing
  - Generate **well-formed inputs** systematically using a grammar or a specification e.g. generation PDF files from a PDF grammar
  - Minus: need the specification.

Fuzzing: Creating Structure

- Problem: only random input not sufficient
  - need to create (some) structure
  - can think of input as having
    - structured part – may be constant, e.g. part of protocol
    - fuzzing part – could be mutated by fuzzer
- How to get structure?
  - specification
  - learning

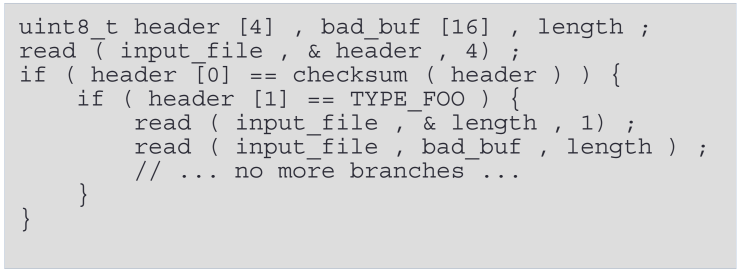
Black-box Fuzzing – Grammar Approaches

- Kaksonen proposed using pre-defined grammar on structure of input to guide the blackbox fuzzing
- Selectively mutate the input based on the grammar producing almost correct input
  - Ensure that checksums are valid
  - Mutating subset of input affecting more interesting paths
  - Higher coverage of the program can be achieved by selectively fuzzing the different portions of the input



```
<input>       := <checksum> <input_inner>
<input_inner> := <type> <...> <length> <...>
<checksum>    := checksum (0 <input_inner>)
<length>      := <uint8_t>
<type>        := “TYPE_FOO” | “TYPE_BAR”
```

Sample Grammar



```
uint8_t header [4] , bad_buf [16] , length ;
read ( input_file , & header , 4) ;
if ( header [0] == checksum ( header ) ) {
    if ( header [1] == TYPE_FOO ) {
        read ( input_file , & length , 1) ;
        read ( input_file , bad_buf , length ) ;
        // ... no more branches ...
    }
}
```

Example Code

Fuzzing tools

- How to get generational fuzzers?
  - Build one for common protocols such as HTTP
  - Generate “well-formed” HTTP queries and see if they crash the web-server you are testing.

- Common (commercial) fuzzing frameworks – look up
  - Spike fuzzer
  - Peach fuzzer
    - can be augmented using grammars or input domain specification.
  - Given a specification, there exists possibility to prioritize tests
    - Starts to move towards white-box approach from black-box approach.

Black-box Fuzzing – Blocks

- Aitel proposed using block-based input generation in the SPIKE framework
- Input is made up of blocks.
- Each block have rules on generation written in API form
- A form of grammar-based fuzzing.

```
s_block_start(“input”)
    s_checksum(“input_inner”)
    s_block_start(“input_inner”)
        s_byte(TYPE_FOO)
        s_byte(0x00)
        s_byte(0x00)
        s_byte_var(0x00)
    s_block_end()
s_block_end()
```

Sample Block

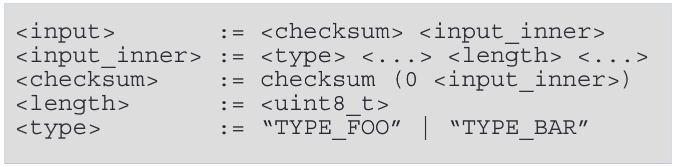
```
uint8_t header [4] , bad_buf [16] , length ;
read ( input_file , & header , 4) ;
if ( header [0] == checksum ( header ) ) {
    if ( header [1] == TYPE_FOO ) {
        read ( input_file , & length , 1) ;
        read ( input_file , bad_buf , length ) ;
        // ... no more branches ...

    }
}
```

Example Code

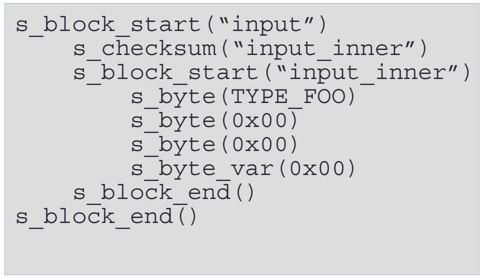
Black-box Fuzzing – Comparing Grammar and Block

- Grammar
  - Provides understanding of the behavior and syntax.
  - Guides fuzzing based on this understanding.
- Block
  - Identifies a rough structure of the input.
  - Aimed at re-using previously known correct structures to generate new inputs.



```
<input>        := <checksum> <input_inner>
<input_inner> := <type> <...> <length> <...>
<checksum>     := checksum (0 <input_inner>)
<length>       := <uint8_t>
<type>         := “TYPE_FOO” | “TYPE_BAR”
```

Sample Grammar



```
s_block_start(“input”)
    s_checksum(“input_inner”)
    s_block_start(“input_inner”)
        s_byte(TYPE_FOO)
        s_byte(0x00)
        s_byte(0x00)
        s_byte_var(0x00)
    s_block_end()
s_block_end()
```

Sample Block in Spike

SPIKE Fuzzer

- Find new vulnerabilities by
  - Making it easy to quickly reproduce a complex (binary) protocol
  - Develop a base of knowledge within SPIKE about different kinds of bug-classes affecting similar protocols
  - Test old vulnerabilities on new programs
  - Make it easy to manually mess with protocols

The SPIKE Datastructure

- A "SPIKE" is a kind of First In First Out Queue (defined thru SPIKE API) or “Buffer Class”
- A SPIKE can automatically fill in “length fields”
  - `s_size_string(“post”,5);`
  - `s_block_start(“Post”);`
  - `s_string_variable(“user=bob”);`
  - `s_block_end(“post”);`

- string variables are fuzzed by Spike

SPIKE Scripts

- Just a way of making API calls using a simple interpreter script
- Strings
  - `s_string(“string”);` // simply prints the string “string” as part of your “SPIKE”
  - `s_string_repeat(“string”,200);` // repeats the string “string” 200 times
- Binary data
  - `s_binary(“\\x41”);` // inserts binary representation of hex 0×41 = ASCII “A”
  - `s_binary_repeat(“\\x41”, 200);` // inserts binary representation of 0×41 200 times

SPIKE Scripts

- Defining blocks
  - `s_block_start(“block1”);` // defines the start of block “block1”
  - `s_block_end(“block1”);` // defines the end of block “block1”
- Block sizes
  - `s_blocksize_string(“block1”, 2);` // adds a string 2 characters long to the SPIKE that represents the size of block “block1”
  - `s_binary_block_size_byte(“block1”);` //adds a 1 byte value to the SPIKE that represents the size of block “block1”

A Sample SPIKE Script:

HTTP Post Request example

```
s_string(“POST /testme.php
HTTP/1.1\r\n”);

s_string(“Host:
testserver.example.com\r\n”);

s_string(“Content-Length: “);

s_blocksize_string(“block1”, 5);

s_string(“\r\nConnection:
close\r\n\r\n”);

s_block_start(“block1”);

s_string(“inputvar=”);

s_string_variable(“inputval”);

s_block_end(“block1”);
```

Output:

```
POST /testme.php HTTP/1.1
Host: testserver.example.com
Content-Length: 17
Connection: close

inputvar=inputval
```

OVERVIEW

- Background – Black Box Testing (Functional Testing)
- Black Box Fuzzing
- Measuring Black Box Fuzzing
- White Box Fuzzing
- Grey Box approaches

When to stop fuzzing?

- When time budget has been exhausted,
- Or
- When you are sure that you have **covered** most of the behavior of the program
  - Statement coverage
  - Branch coverage
  - Path coverage

- **Measuring the adequacy of black-box fuzzing, requires us to look inside the code !!**
  - but can be machine code
  - can measure by instrumenting code (compiler / binary rewriting)

Why coverage?

- Idea
  - what is the difference between executing 100 lines vs 10000 lines in 5000 line program?
  - intuitively: greater line coverage correlated with improved testing / bug finding
    - not true in theory but can be useful in practice
  - can turn this into a "coverage measure"
- granularity of coverage
  - function / line / expression / ...

Structural testing in practice

- Create functional test suite first, then measure structural coverage to identify see what is missing
- Interpret unexecuted elements
  - may be due to natural differences between specification and implementation
  - or may reveal flaws of the software or its development process
    - inadequacy of specifications that do not include cases present in the implementation
    - coding practice that radically diverges from the specification
    - inadequate functional test suites
- Attractive because automated
  - coverage measurements are convenient progress indicators
  - sometimes used as a criterion of completion
    - use with caution: does not ensure ***effective*** test suites

Statement testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\text{ executed statements}}{\#\text{ statements}}$$

- Rationale: a fault in a statement can only be revealed by executing the faulty statement
  - but: assumes no state dependence – statements are independent

Basic Blocks

- basic block
  - sequence of straight line code
  - single entry
  - possibly branching at end
- eg:

```
x = y + 1;
if (x > 5) goto BIG;
```

Control Flow Graph (CFG)

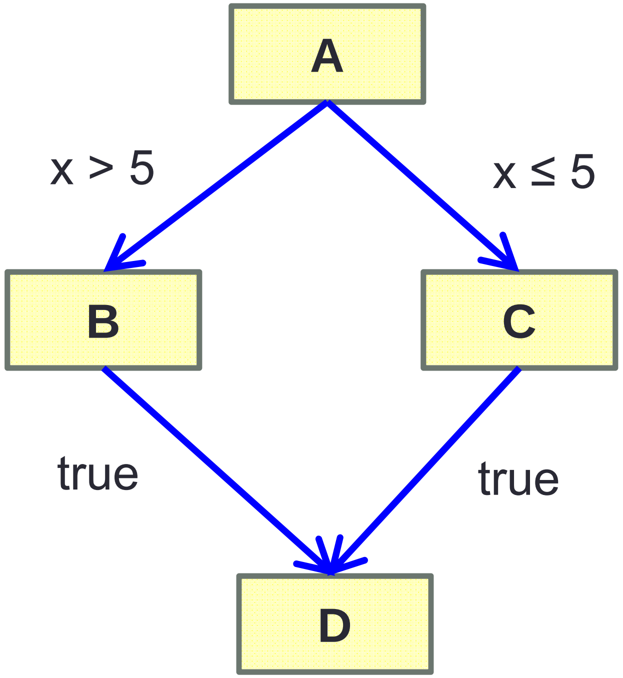
- CFG is a Directed Graph
  - nodes are basic blocks
  - edges are control flow transfer from one block to another
  - nodes can have multiple incoming edges
  - nodes can have multiple outgoing edges
  - representation of control flow of program
- used in compilers, static analysis, ...

CFG Eg 1

```
x = y + 1;
if (x > 5) y = 0;
else y = 1;
a = x + y;
```

CFG Eg 2

```
A:      x = y + 1;
        if (x > 5) goto B;
        else goto C;
B:      y = 0;
        goto D;
C:      y = 1;
        goto D;
D:      a = x * y;
```



Statements or blocks?

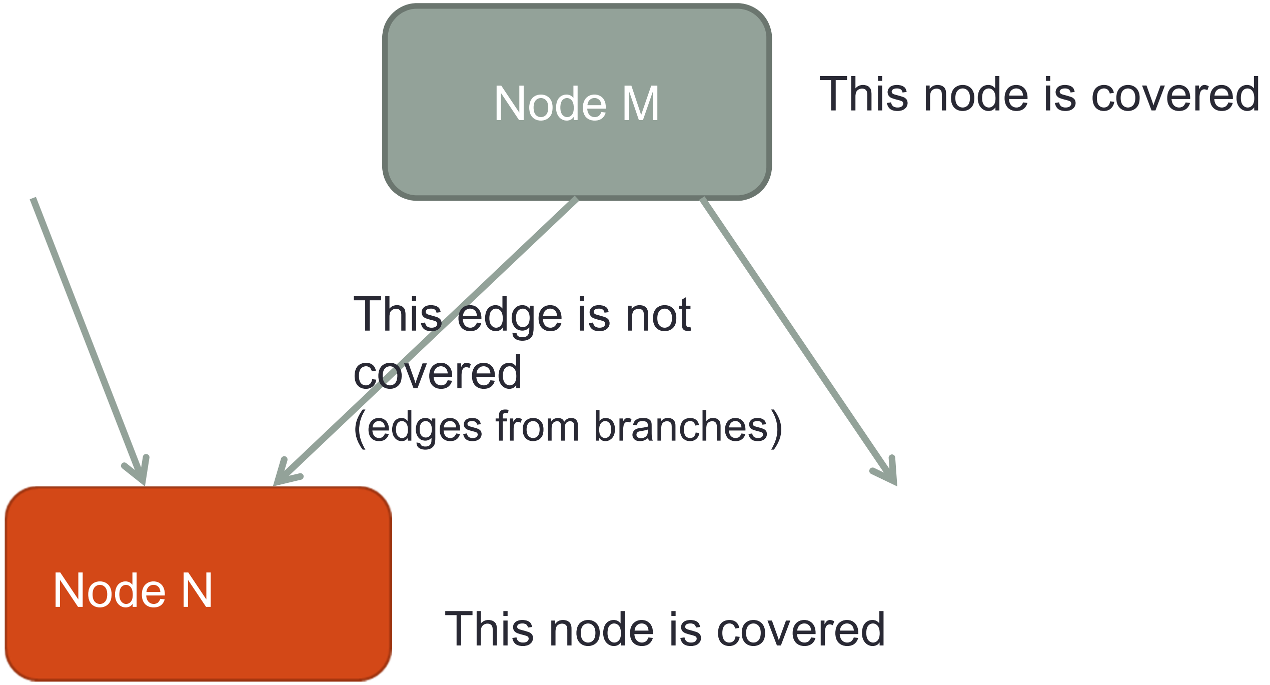
- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to *basic block* coverage or *node coverage*
  - Difference in granularity, not in concept
- No essential difference
  - 100% node coverage = 100% statement coverage
    - but levels will differ below 100%
  - A test case that improves one will improve the other
    - though not by the same amount, in general

Coverage is not size

- Coverage does not depend on the number of test cases
  - $T_0, T_1 : T_1 >_{coverage} T_0$ $T_1 <_{cardinality} T_0$
  - $T_1, T_2 : T_2 =_{coverage} T_1$ $T_2 >_{cardinality} T_1$
- Minimizing test suite size is seldom the goal
  - small test cases make failure diagnosis easier

Exercise: Construct concrete code examples to show that coverage is not size.

Node coverage != Edge coverage



Exercise: Try to construct a program construct whose control flow graph exhibits the above pattern.

Branch testing

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Branch Coverage:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

Statements vs branches

- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

"All branches" can still miss conditions

- Sample fault: missing operator

```
if (digit_high == 1 || digit_low == -1) ...
```

- Branch adequacy criterion can be satisfied by varying only digit_low
  - The faulty sub-expression might never determine the result
  - We might never really test the faulty condition, even though we tested both outcomes of the branch

Example

- Condition h == 1 || l == -1

- Suppose it is buggy
  - Should be h == -1 || l == -1
  - Achieve branch coverage
    - < h == 0, l == 0>   false branch
    - < h == 0, l == -1>   true branch

    - Does not vary the faulty condition at all, and the variables involved!!

Condition testing

- Branch coverage exposes faults in how a computation has been decomposed into cases
  - intuitively attractive: check the programmer’s case analysis
  - but only roughly: groups cases with the same outcome
- Condition coverage considers case analysis in more detail
  - also *individual conditions* in a compound Boolean expression
    - e.g., both parts of digit_high == 1 || digit_low == -1

Basic condition testing

- Adequacy criterion: each basic condition must be executed at least once
- Basic Condition Coverage:

$$\frac{\#\text{ truth values taken by all basic conditions}}{2 * \#\text{ basic conditions}}$$

Basic condition != Branch coverage

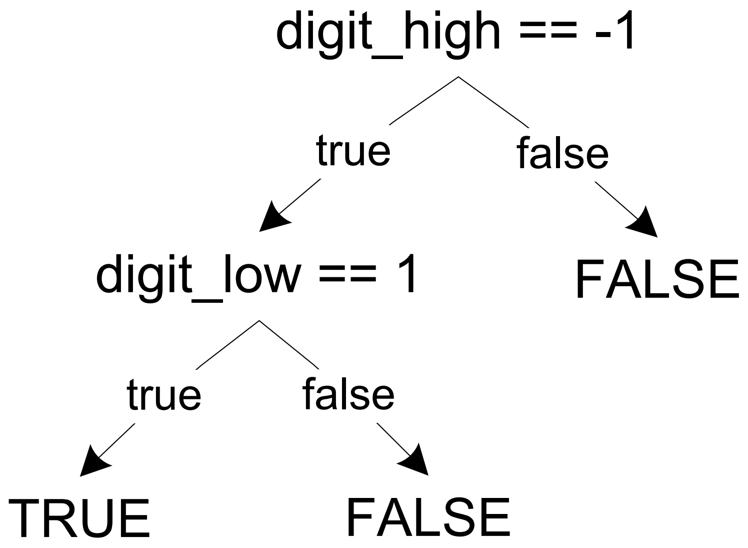
Earlier, we saw an example where branch coverage can be achieved without basic condition coverage.

How can we achieve basic condition coverage without achieving branch coverage?

Exercise: Construct an example of boolean expression and the truth vectors for its constituent conditions – so that the basic condition coverage is achieved – but branch coverage is not achieved.

Covering branches and conditions

- Branch and condition adequacy:
  - cover all conditions and all decisions
- Compound condition adequacy:
  - Cover all possible evaluations of compound conditions
  - Cover all branches of a decision tree



Compound conditions: Exponential complexity

```
(((a || b) && c) || d) && e
```

Test Case	a	b	c	d	e
(1)	T	—	T	—	T
(2)	F	T	T	—	T
(3)	T	—	F	T	T
(4)	F	T	F	T	T
(5)	F	F	—	T	T
(6)	T	—	T	—	F
(7)	F	T	T	—	F
(8)	T	—	F	T	F
(9)	F	T	F	T	F
(10)	F	F	—	T	F
(11)	T	—	F	F	—
(12)	F	T	F	F	—
(13)	F	F	—	F	—

- short-circuit evaluation often reduces this to a more manageable number, but not always

Path adequacy

- Decision and condition adequacy criteria consider individual program decisions
- Path testing focuses consider combinations of decisions along paths
- Adequacy criterion: each path must be executed at least once
- Coverage:

$$\frac{\#\text{ executed paths}}{\#\text{ paths}}$$

Practical path coverage criteria

- The number of paths in a program with loops is possibly unbounded
    - the simple criterion may not be practical to satisfy
- For a feasible criterion:
    - partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
    - the number of traversals of loops
    - the length of the paths to be traversed
    - the dependencies among selected paths

Satisfying structural criteria

- Large amounts of *fossil* code may indicate serious maintainability problems
  - But some unreachable code is common even in well-designed, well-maintained systems
- Solutions:
  - make allowances by setting a coverage goal less than 100%
  - require justification of elements left uncovered
    - RTCA-DO-178B and EUROCAE ED-12B for modified MC/DC

Coverage vs Inputs

- Fuzzing changes inputs
- coverage measure is from execution
- changing coverage requires changing input
    - how to find input?
    - random strategies – black box fuzzing
    - more systematic – white box fuzzing

Summary of coverage criteria

- We defined a number of adequacy criteria
    - **NOT test design techniques – blackbox fuzzing**

- Different criteria address different classes of errors
- Full coverage is usually unattainable
    - Remember that attainability is an undecidable problem!
- …and when attainable, “inversion” is usually hard
    - How do I find program inputs allowing to cover something buried deeply in the CFG?
    - Automated support (e.g., symbolic execution) may be necessary
- Therefore, rather than requiring full adequacy, the “degree of adequacy” of a test suite is estimated by coverage measures
    - May drive test improvement

Coverage Guidance

- Random testing may be too wasteful
- How to know where to focus testing?
- Coverage-based Greybox Fuzzing
  - use code coverage to give execution feedback
  - focus on inputs which can give more coverage
  - Greybox fuzzing
    - not blackbox
    - looks partially into program

Magic Number Eg

```
hard_to_fuzz(char *p)
{
      if (  getchar() == '1') &&
            getchar() == '2) &&
            getchar() == '3') &&
            getchar() == '4'))
                  memory_error(p); // jackpot - the target
      no_bugs();
}
```

// see what happens with random fuzzing

// what happens with coverage guided (later see symbolic execution)

Magic Number: random testing

```
hard_to_fuzz(char *p)
{
      if ((getchar() == '1') && // probability(1)
          (getchar() == '2) &&  // * prob(2)
          (getchar() == '3') && // * prob(3)
          (getchar() == '4'))   // * prob(4) = ?
             memory_error(p);   // jackpot - the target
      no_bugs();
}
```

// see what happens with random fuzzing

// what happens with coverage guided (later see symbolic execution)

Google Testing

- circa 2016
- Fuzzing finds 10x more bugs than unit tests
- several frameworks, 100s fuzzers
- 5000+ cpu cores, continuous fuzzing 24/7
- 5000+ bugs in Chromium, 1200+ bugs in ffmpeg

Practical Fuzzing Issues

- blackbox random testing may not generate useful test cases
  - within timeout limits
- still want advantages of blackbox fuzzing
  - can runtime knowledge be used?
    - form of greybox guzzing
  - what to collect?
  - how to collect?
- can test cases be learnt?

American Fuzzy Lop (AFL)

- Coverage-based **Greybox Fuzzing**
  - instrumentation of CFG edges
- very successful in finding vulnerabilities
- input testing fuzzing
- automated
  - only uses seed inputs
    - can even be empty inputs
- open source
  - developed by Michal Zalewski (lcamtuf), part of Google suite
  - http://lcamtuf.coredump.cx/afl/
  - Notes: http://lcamtuf.coredump.cx/afl/technical_details.txt

AFL Branch Instrumentation

- count CFG edges executed
  - instrumentation adds to every branch instruction

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

  - coverage[] array is a 64 kB shared memory region passed to the instrumented binary by the caller
  - instrumentation adds low overhead – fuzzing execution speed vs power of instrumentation

AFL Coverage Based Greybox Fuzzing

```
Tx = Ø; T = S;        // Seed Inputs S
while (1) {           // till timeout or abort
  t = ChooseNext(T);
  for i=1 to p do
    t' = MutateInput(t);
    run target with input t';
    if t' crashes // memory error or exception
      add t' to Tx;
    else if IsInteresting(t') // interesting transition behavior
      add t' to T
}
output Tx; // the crashing inputs
```

IsInteresting strategy

- IsInteresting(t) ≡ if t creates new tuples in coverage[]
  - Trace #1: A -> B -> C -> D -> E (existing trace t)
  - Trace #2: A -> B -> C -> A -> E

    new tuples: CA, AE
  - Trace #3: A -> B -> C -> A -> B -> C -> D

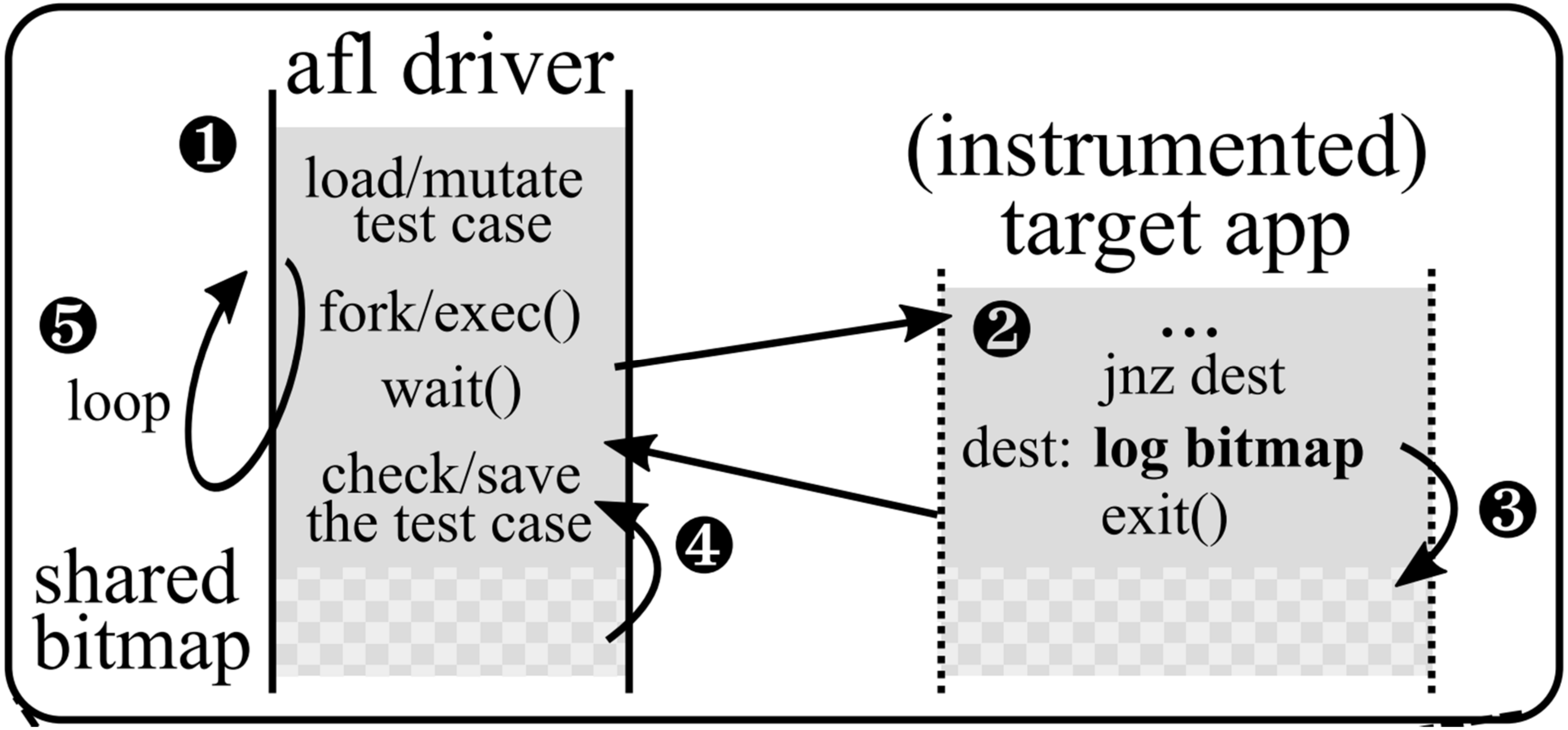
    different trace, no new tuples
- bucket coverage counts
  - buckets: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
  - transitions between buckets considered interesting
- ... may not be exhaustive

AFL Fuzzing Strategy

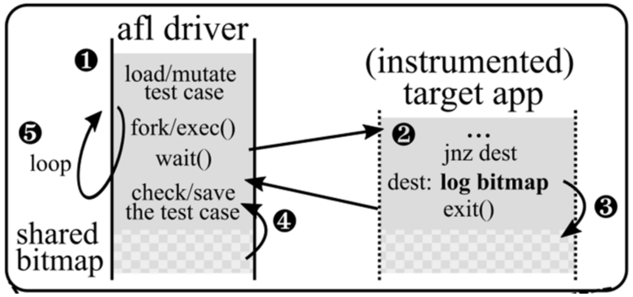
- Deterministic Strategies
  - initial strategy
  - Sequential bit/byte flips with varying lengths and stepovers
  - Sequential addition and subtraction of small integers
  - Sequential insertion of known interesting integers:
    - 0, 1, -1, 256, 1024, INT_MAX-1, INT_MAX, ...
- Non-deterministic Strategies
  - stack operations:

    single bit flips, set interesting bytes, add/subtract small ints,
    random byte sets, block deletion,
    block duplication overwrite/insert, block memset
  - splice test cases

AFL Architecture: Overall



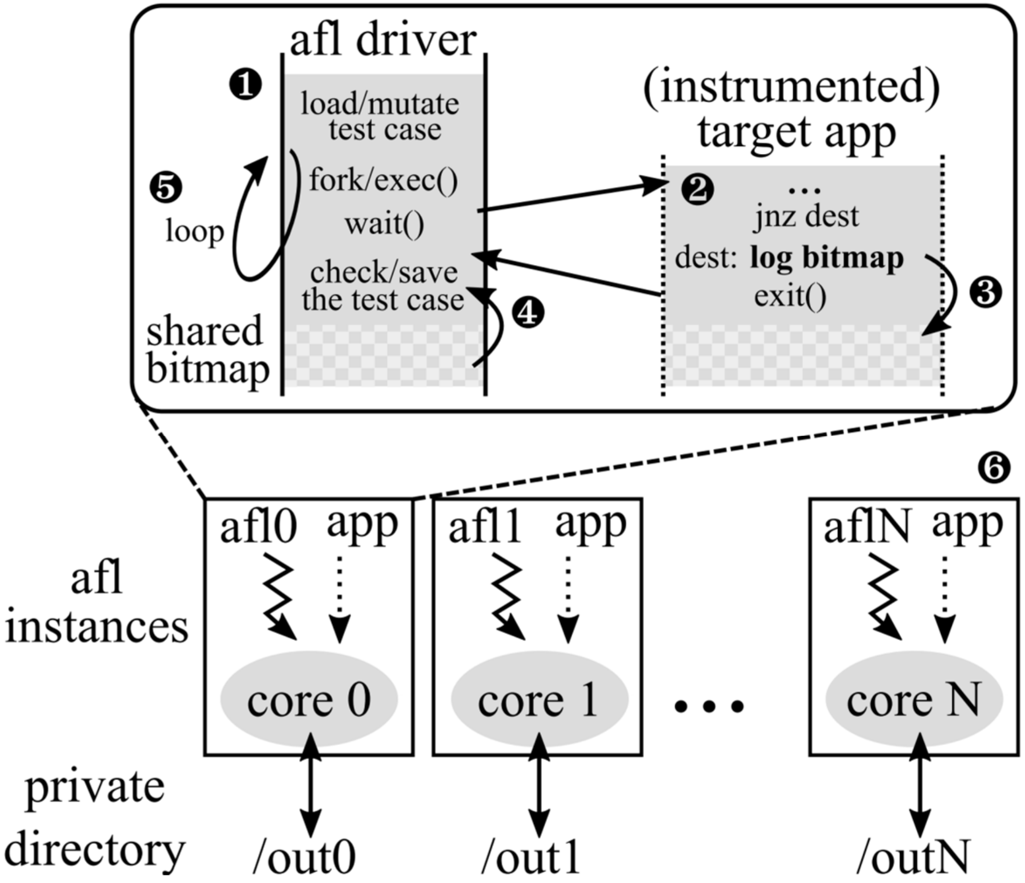
AFL Architecture: Components



1. load/mutate test case
2. execute target app
    1. **Fork Server** forks (clones) new process running target code (similar to Zygote process on Android)
    2. target process runs main() with test case input
    3. driver waits for target termination/exit
    4. target code has AFL instrumentation, records branch coverage bitmap logs coverage bitmap on exit
    5. Fork Server uses shared memory as coverage bitmap
3. driver collects bitmap to decide to if input is interesting

AFL Architecture: Parallel Fuzzing

- scale to N cores in parallel
  - private directories to avoid contention
  - sync test cases with neighbours + check new paths

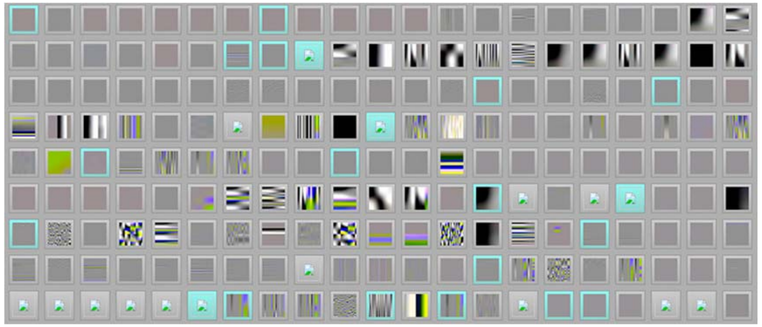


AFL Termination

- Timeout / normal exit
  - no error found
- Error found
  - successful (crashing) input = created/found buggy test case
  - exception / security instrumentation = crash
    - segmentation fault / bus error / illegal instruction ....
    - ASAN memory error – combine AFL + ASAN instrumentation
    - malloc allocator which can exercise errors

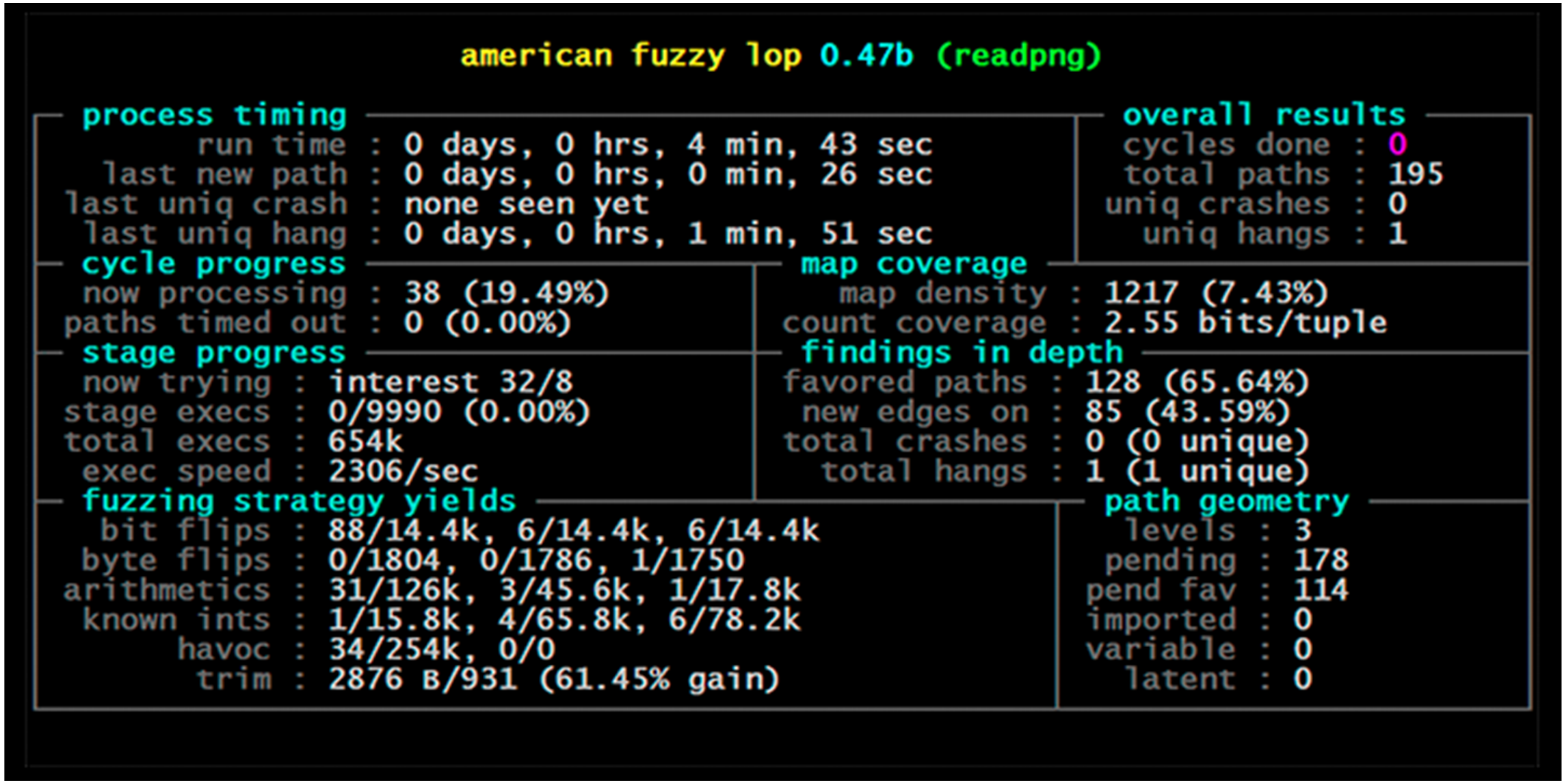
AFL Learning

- mutate input cases to successful crashing inputs
- start with empty input
  - generate JPG



  - generate interesting XML
  - generate SQL queries:
    - input case: `create table t1(one smallint); insert into t1 values(1); select * from t1;`
    - generated: (on SQLite)
      - CREATE VIRTUAL TABLE t0 USING fts4(x,order=DESC);
      - INSERT INTO t0(docid,x)VALUES(-1E0,'0(o');

AFL Statistics



```
                  american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec       │  total paths : 195    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec       │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 38 (19.49%)       │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.55 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8         │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)        │  new edges on : 85 (43.59%)            │
│ total execs : 654k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec              │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k            │    levels : 3          │
│  byte flips : 0/1804, 0/1786, 1/1750                │   pending : 178        │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k             │  pend fav : 114        │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k             │  imported : 0          │
│       havoc : 34/254k, 0/0                          │  variable : 0          │
│        trim : 2876 B/931 (61.45% gain)              │   latent : 0           │
└─────────────────────────────────────────────────────┴────────────────────────┘
```

AFL Enhancements

- AFLFast
  - better use of feedback
  - more effective search strategy – less tests to find bugs
- Driller
  - adds intelligent input generation to AFL
  - symbolic execution
- WinAFL
  - Windows support
  - dynamic instrumentation at runtime

Recap

- What is fuzzing and why do we fuzz software?
- Black-box or functional testing
- Blackbox fuzzing
  - Generational fuzzing
  - Mutational fuzzing
- When do we stop fuzzing
  - Coverage Criteria
- Coverage Guided fuzzing
  - greybox fuzzing
  - AFL