Software Security

Program Analysis Techniques （4）

Fuzzy Testing (Fuzzing)

Guosheng Xu,

guoshengxu@bupt.edu.cn

Outline

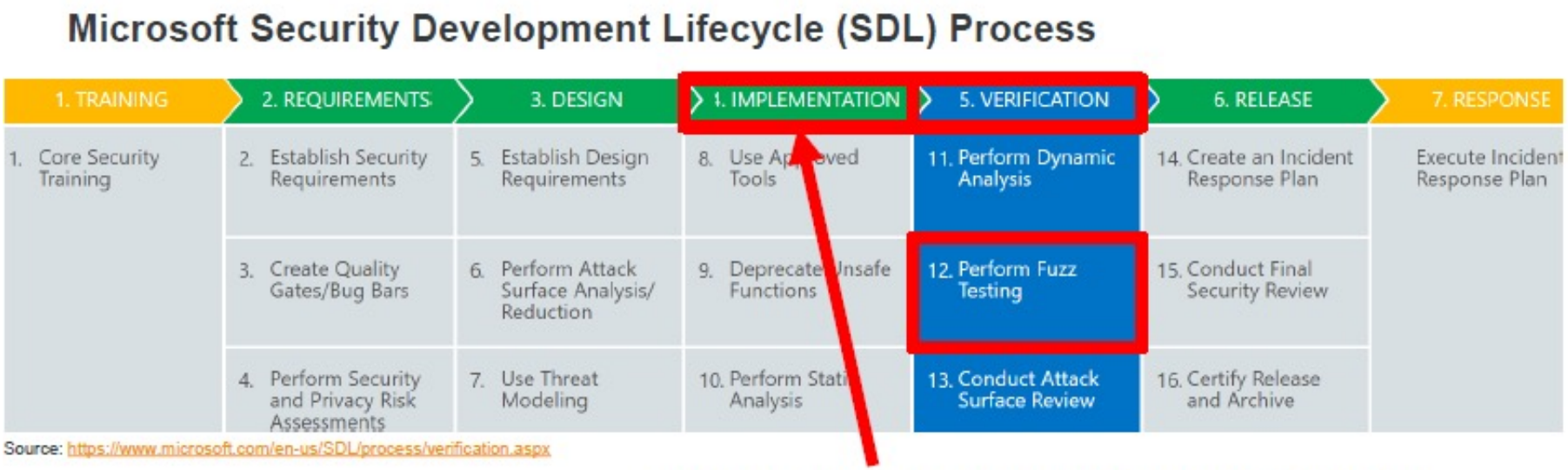
- Techniques used in Security Analysis
- Basic Program Analysis
  - Control flow analysis
  - Data flow analysis
- Taint Analysis
- Symbolic execution
- **Fuzzing**

Fuzzing

Definition of fuzzing (source Wikipedia):

Fuzzing or fuzz testing is an **automated software testing technique** that involves providing **invalid, unexpected, or random data as inputs** to a computer program. **The program** is then **monitored for exceptions such as crashes,** or failing built-in code assertions or for finding potential memory leaks.

Why do we need Fuzzing?



Source: https://www.microsoft.com/en-us/SDL/process/verification.aspx

I also recommend fuzzing during implementation

Example: You finished a complex task and you are not sure if it behaves correctly and is secure

➔ Start a fuzzer over night / the weekend ➔ Check corpus

Why do we need Fuzzing?

SDL Phase 4 *Security Requirements*

Where input to file parsing code could have crossed a trust boundary, **file fuzzing must be performed on that code**. […]

- **An Optimized set of templates must be used.** Template optimization is based on the maximum amount of **code coverage** of the parser with the minimum number of templates. Optimized templates have been shown to double fuzzing effectiveness in studies. **A minimum of 500,000 iterations, and have fuzzed at least 250,000 iterations since the last bug found/fixed that meets the SDL Bug Bar.**

Source: https://msdn.microsoft.com/en-us/library/windows/desktop/cc307418.aspx

0days Are a Hacker Obsession

- An 0day is a vulnerability that's not publicly known
- Modern 0days often combine multiple attack vectors & vulnerabilities into one exploit
  - Many of these are used only once on high value targets
- 0day statistics
  - Often open for months, sometimes years

Market for 0days

- Sell for $10K-1M

How to Find a 0day?

- **Manual audit**
- **(semi)automated techniques/tools**
  - E.g, Program Analysis
  - Fuzzy testing (focus of this lecture)

Solution 1: Code Auditing

iOS和OS X中的SSL/TLS 重大安全漏洞

```
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

Solution 2: Static Analysis (on binary)

- Reverse Engineering (e.g., IDA)

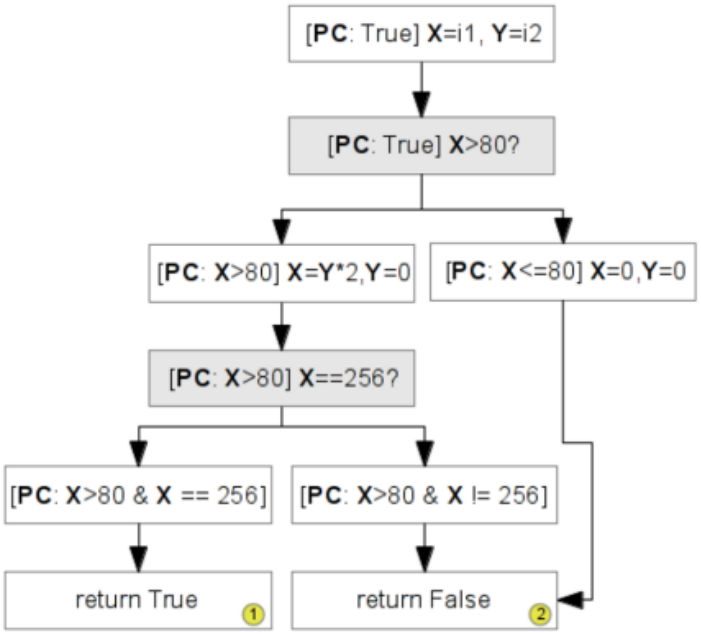
Problem: Too Complex (e.g., browser)

Two Popular Directions

- Symbolic Execution (also static)
- Fuzzing (dynamic)

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```



return True 1 — **PC**: **i1**>80 & (**i2** * 2)==256

return False 2 — **PC**: **i1**<=80 | (**i1**>80 & (**i2** * 2)!=256)

Problem: State Explosion

- Too many path to explore
- Too huge state space (e.g., browser? OS?)
- Solving constraints is a hard problem

Today's Topic: Fuzzing

- Two key ideas
  - Reachability is given (since we are executing!)
  - Focus on quickly exploring the path/state
    - How? mutating inputs
    - How well? e.g., coverage

Example: How well fuzzing can explore all paths?

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

Game Changing Fact: Speed

- **In this example**
  - Symbolic Execution explores/checks just two conditions
  - Fuzzing requires 256 times (by scanning values from 0 to 256)
- **But what if fuzzer is an order of magnitude faster (say, 10K times)?**

Fuzzing is really bad at exploring paths

```
x = int(input())
if x > 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Let's fuzz it!

1 ⇒ "You lose!"

593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

498 ⇒ "You lose!"

48 ⇒ "You win!"

Fuzzing is really bad at exploring paths

```
x = int(input())
if x > 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Let's fuzz it!

1 ⇒ "You lose!"

593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

498 ⇒ "You lose!"

42 ⇒ "You lose!"

3 ⇒ "You lose!"

.....

57 ⇒ "You lose!"

Fuzzing vs. Symbolic Execution

Fuzzing

- Good at finding solutions for general conditions
- Bad at finding solutions for specific conditions

Symbolic Execution

- Good at finding solutions for specific conditions
- Spends too much time iterating over general conditions

Fuzzing vs. Symbolic Execution

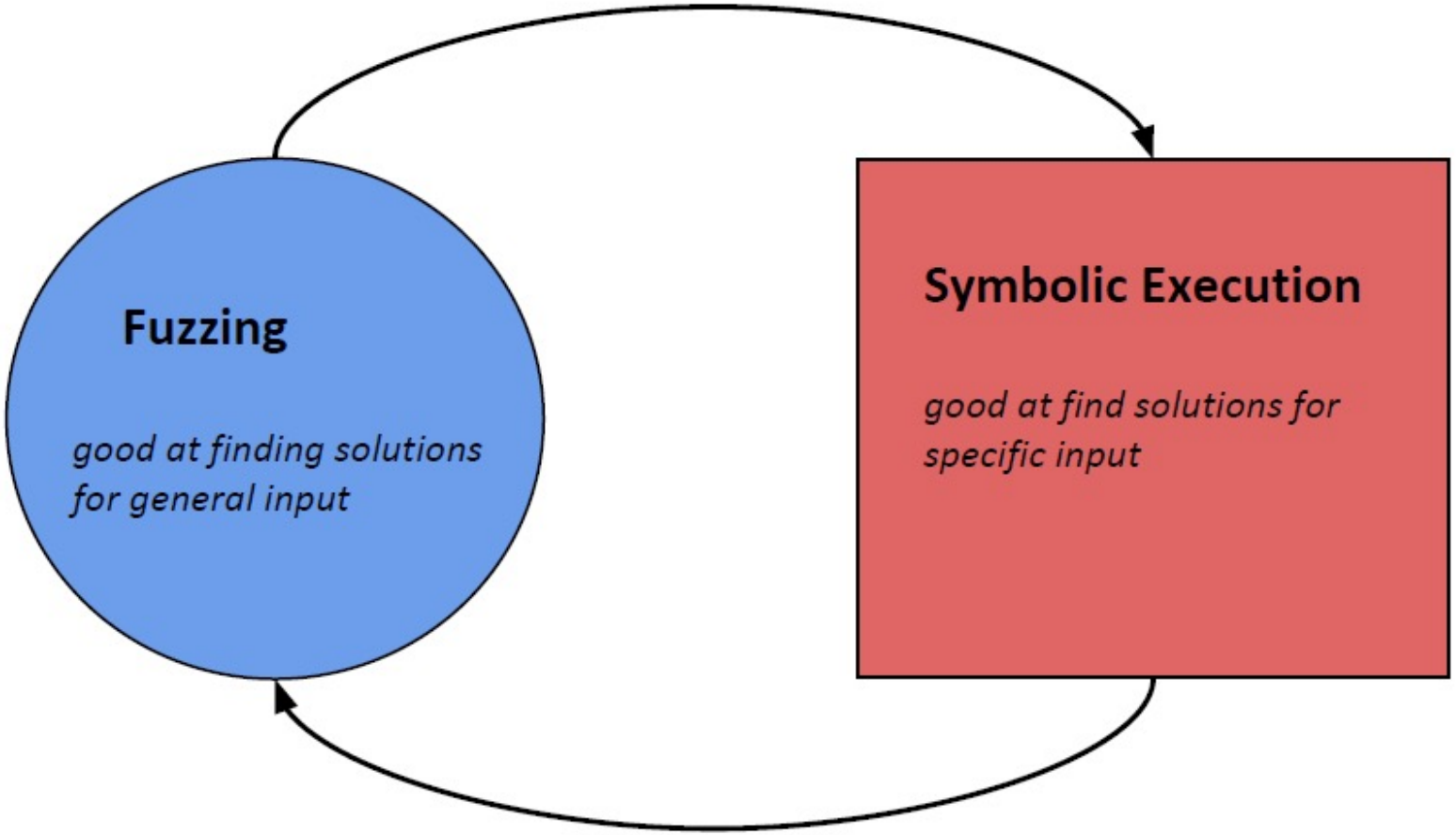
```
x = input()

def recurse(x, depth):
  if depth == 2000
    return 0
  else {
    r = 0;
    if x[depth] == "B":
      r = 1
    return r + recurse(x
[depth], depth)

if recurse(x, 0) == 1:
  print "You win!"
```

Fuzzing Wins

```
x = int(input())
if x >= 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Symbolic Execution Wins

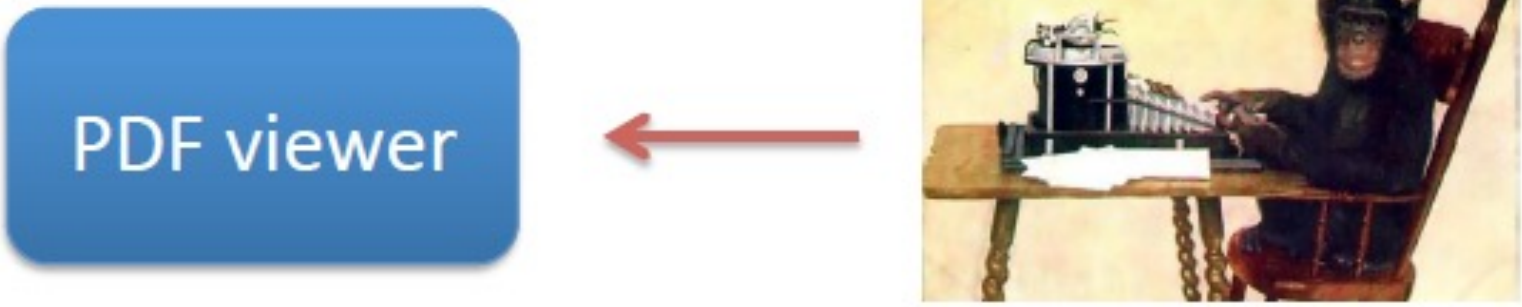
Fuzzing vs. Symbolic Execution



Importance of High-quality Corpus

- In fact, fuzzing is really bad at exploring paths
  - e.g., if (a == 0xdeadbeef)
- So, paths should be (or mostly) given by corpus (sample inputs)
  - e.g., pdf files utilizing full features
  - but, not too many! (do not compromise your performance)
- A fuzzer will trigger the exploitable state
  - e.g., len in malloc()

Finding bugs in a PDF viewer

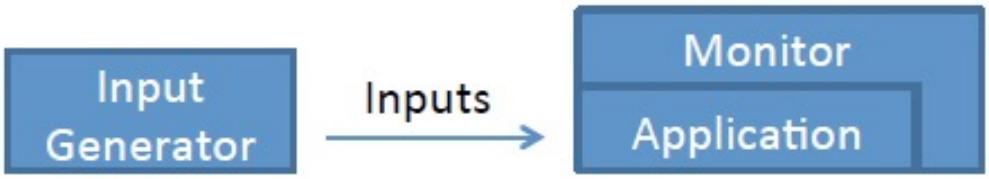


Black-box Fuzzy Testing

- Given a program, simply feed it random inputs, see whether it crashes
- Advantage: really easy
- Disadvantage: inefficient
  - Input often requires structures, random inputs are likely to be malformed
  - Inputs that would trigger a crash is a very small fraction, probability of getting lucky may be very low

Fuzzing

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, .mpg)
- Or network based…
  - http, SNMP, SOAP

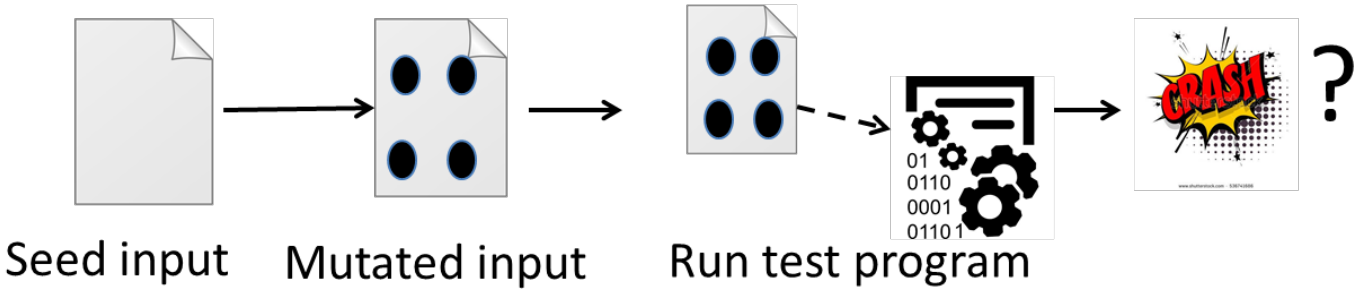


Regression VS. Fuzzing

	Regression	Fuzzing
Definition	Run program on many **normal** inputs, look for badness.	Run program on many **abnormal** inputs, look for badness.
Goals	Prevent **normal users** from encountering errors (e.g. assertion failures are bad).	Prevent **attackers** from encountering **exploitable** errors (e.g. assertion failures are often ok).

Mutation-based Fuzzing/基于变异的模糊测试

- **Take a well-formed input, randomly perturb (flipping bit, etc.)** 取一个格式良好的输入，随机扰动（翻转位等）
- **Little or no knowledge of the structure of the inputs is assumed**
- **Anomalies are added to existing valid inputs**
  - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- **Examples: ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.**



For example, ZZUF

- **http://www.freebuf.com/news/83737.html**

convert example.png example.gif
convert example.png example.xwd
convert example.png example.tga

```
for i in {1000..3000}; do for f in example.*; do zzuf -r 0.01 -
s $i < "$f" > "$i-$f"; done; done
```

从example中创建出大量的畸形文件

Example: fuzzing a PDF viewer

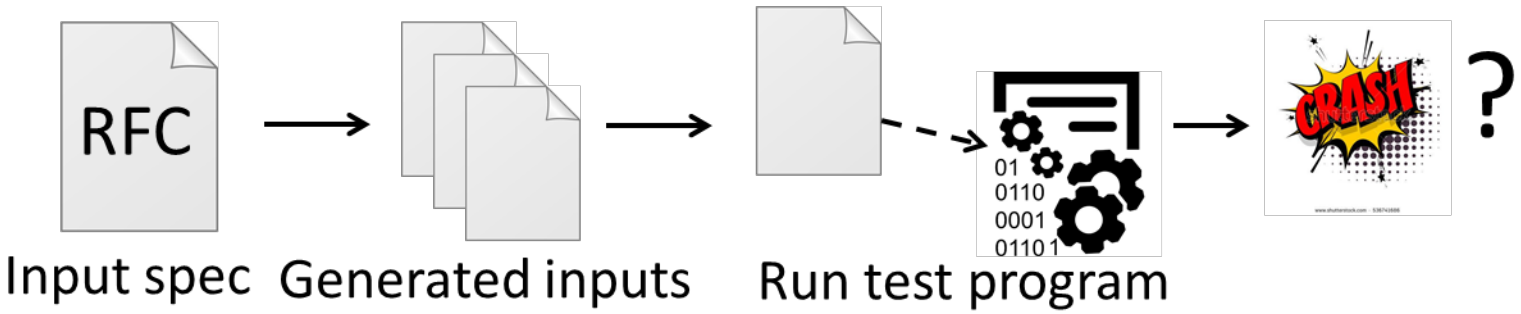
- **Google for .pdf (about 1 billion results)**
- **Crawl pages to build a corpus**
- **Use fuzzing tool (or script)**
  - Collect seed PDF files
  - Mutate that file
  - Feed it to the program
  - Record if it crashed (and input that crashed it)

Mutation-based fuzzing

- Super easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus
- May fail for protocols with checksums, those which depend on challenge

Generation-Based Fuzzing

- **Test cases are generated from some description of the input format (e.g., documentation)**
  - Using specified protocols/file format info
- **Anomalies are added to each possible spot in the inputs**
- **Knowledge of protocol should give better results than random fuzzing**



Generation-Based Fuzzing

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);     // Width
                s_push_int(0x14, 1);     // Height
                s_push_int(0x8, 3);      // Bit Depth - should be 1,2,4,8,16, base
                s_push_int(0x3, 3);      // ColorType - should be 0,2,3,4,6
                s_binary("00 00");       // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);      // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Sample PNG spec

Mutation-based vs. Generation-based

- **Mutation-based fuzzer**
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fall for protocols with checksums and other hard checks
- **Generation-based fuzzers**
  - Pros: Completeness, can deal with complex dependncies (e.g, checksum)
  - Cons: writing generators is hard, performance depends on the quality of the spec

How much fuzzing is enough?

- Mutation-based fuzzers may generate an infinite number of test cases.
  - **When has the fuzzer run long enough?**
- Generation-based fuzzers may generate a finite number of test cases.
  - **What happens when they're all run and no bugs are found?**

Code Coverage

- Some of the answers to these questions lie in ***code coverage***
- Code coverage is a metric that can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. gcov, lcov

e.g., gcov, lcov

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - hello - test.c (source / functions)			
Test:	main_test.info			
Date:	2017-10-22			
Lines:		7	9	77.8 %
Functions:		1	1	100.0 %

	Line data	Source code
1	:	#include <stdio.h>
2	:	
3	1 :	void test(int count)
4	:	{
5	:	int i;
6	10 :	for (i = 1; i < count; i++)
7	:	{
8	9 :	if (i % 3 == 0)
9	3 :	printf ("%d is divisible by 3\n", i);
10	9 :	if (i % 11 == 0)
11	0 :	printf ("%d is divisible by 11\n", i);
12	9 :	if (i % 13 == 0)
13	0 :	printf ("%d is divisible by 13\n", i);
14	:	}
15	1 :	}

Generated by: LCOV version 1.10

Line coverage

- Line/Statement coverage: Measures how many lines of source code have been executed.
- For the code on the right, how many test cases (values of pair (a,b)) needed for full(100%) line coverage?

```
if( a >  2  )
    a =  2;
if( b >2  )
    b =  2;
```

Branch coverage

- Branch coverage: Measures how many branches in code have been taken (conditional jmps)
- For the code on the right, how many test cases needed for full branch coverage?

```
if( a >  2  )
    a =  2;
if( b >2  )
    b =  2;
```

Path Coverage

- Path coverage: Measures how many paths have been taken

```
if( a >  2  )
    a =  2;
if( b >2  )
    b =  2;
```

- For the code on the right, how many test cases needed for full path coverage?

Example

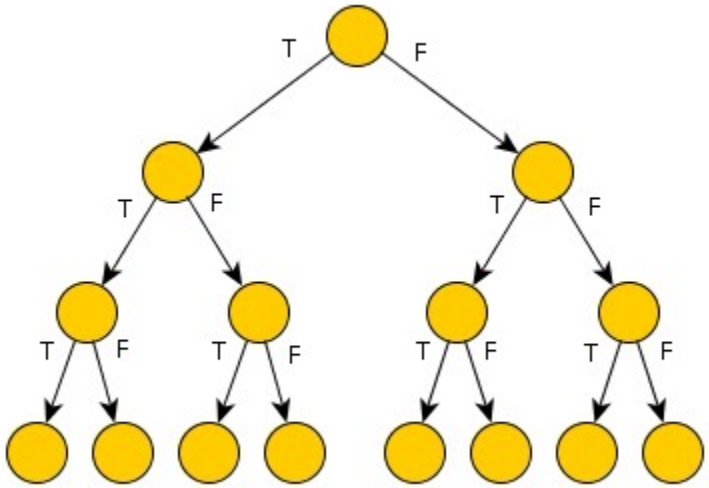
```
public int
returnInput(int input,
boolean condition1,
boolean condition2,
boolean condition3) {
  int x = input;
  int y = 0;
  if (condition1)
    x++;
  if (condition2)
    x--;
  if (condition3)
    y=x;
  return y;
}
```

(x, true, true, true) - 100% statement covered

In order to cover 100% of branches we would need to add following test case:
(x, false, false, false)

Example

Path Coverage



there are 8 separate paths:

1-2-3

t -t -t - covered with testcase 1

t -t -f

t -f -t

t -f -f

f -t -t

f -t -f

f -f -t

f -f -f - covered with testcase 2

Benefits of Code coverage

- **Can answer the following questions**
  - How good is an initial file?
  - Am I getting stuck somewhere? 我会卡在某个地方吗？

```
if (packet[0x10] < 7) { //hot path
} else { //cold path }
```

  - How good is fuzzer X vs. fuzzer Y
  - Am I getting benefits by running multiple fuzzers?

Problems of code coverage

- For:

```
mySafeCopy(char *dst, char* src) {
    if(dst && src)
        strcpy(dst, src);
}
```

- Does full line coverage guarantee finding the bug?
- Does full branch coverage guarantee finding the bug?

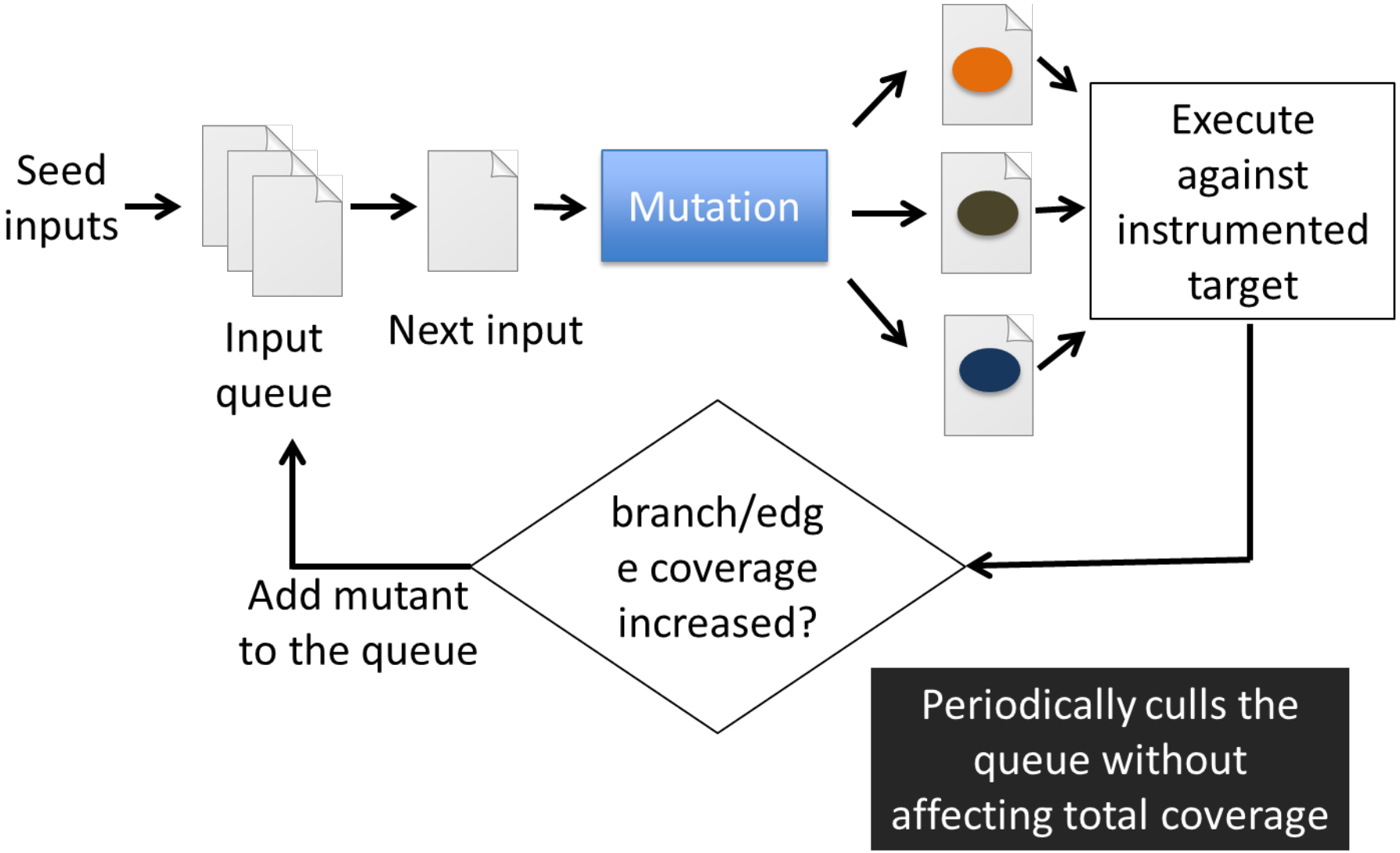
Coverage-guided gray-box fuzzing

- **Special type of mutation-based fuzzing**
  - Run mutated inputs on instrumented program and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use genetic/遗传 algorithms, i.e., try random mutations on test corpus and **only add mutants to the corpus if coverage increases**
  - Examples: AFL, libfuzzer

American Fuzzy Lop (AFL)

- American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.
- This substantially improve the functional coverage for the fuzzed code. The compact synthesized corpora produced by a tool are also useful for seeding other, more labor-or resource-intensive testing regimes down the road.

American Fuzzy Lop (AFL)



AFL - Example

```
void test(char *buf)
{
    int n = 0;
    if(buf[0] == 'b') n++;
    if(buf[1] == 'a') n++;
    if(buf[2] == 'd') n++;
    if(buf[3] == '!') n++;

    if(n == 4) {
        crash();
    }
}
```

Very hard to trigger the crash!
Millions of times!

AFL - Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void test (char *buf) {
    int n = 0;
    if(buf[0] == 'b') n++;
    if(buf[1] == 'a') n++;
    if(buf[2] == 'd') n++;
    if(buf[3] == '!') n++;

    if(n == 4) {
        raise(SIGSEGV);
    }
}

int main(int argc, char *argv[]) {
    char buf[5];
    FILE* input = NULL;
    input = fopen(argv[1], "r");
    if (input != 0) {
        fscanf(input, "%4c", &buf);
        test(buf);
        fclose(my_file):
    }
    return 0;
}
```

./afl-gcc crasher.c -o crash

mkdir testcase
echo ‘test’ > testcase/file

./afl-fuzz -i testcase -o output/ ./crash @@

Further Reading Materials

AFL,
http://lcamtuf.coredump.cx/afl/

Try it yourself!

http://lcamtuf.coredump.cx/afl/README.txt

Fuzzing: The State of the Art

www.dtic.mil/dtic/tr/fulltext/u2/a558209.pdf