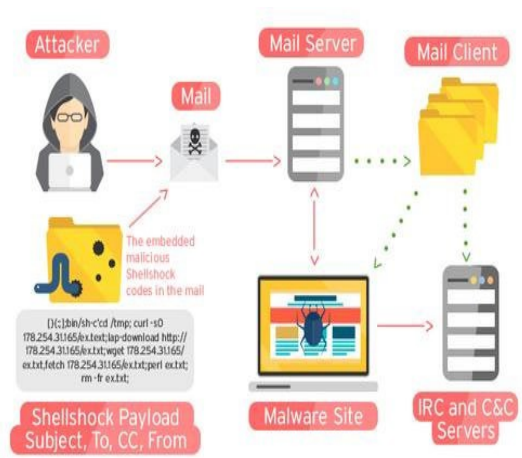
Software Testing (Advanced Part II)

Week 10

Shellshock not only steal data, but may also gain control over the system and run arbitrary commands. Heartbleed returns the memory of the server with long length and short payload. The bound check can be enforced to prevent this bug. Stagefright causes buffer overflow of 3gpp metadata fields and is exposed to arbitrary code execution.

Security is *often* a software issue.

In Deloitte's *2007 Global Security Survey*, 87 percent of survey respondents cited poor software development quality as a top threat in the next 12 months.



CVE-2014-0160

Heartbleed

CVE-2014-6271

Shellshock

Multiple CVE

STAGEFRIGHT

A Simple Vulnerability

```
while ((read = getch()) != '\0')
{
    name[id++] = read;
    .....
}
```

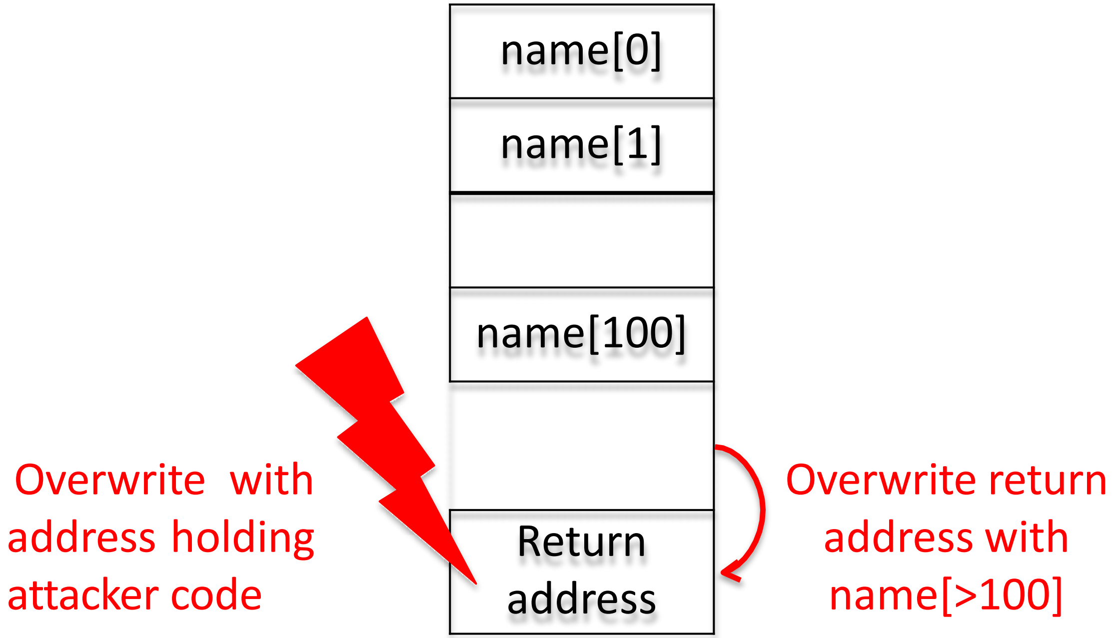
taking input character from user, end loop only when null character has been given.

User can give infinite number of characters, none being the null character thus leading to overflow.

getch() reads input from standard input terminal

A Simple Vulnerability

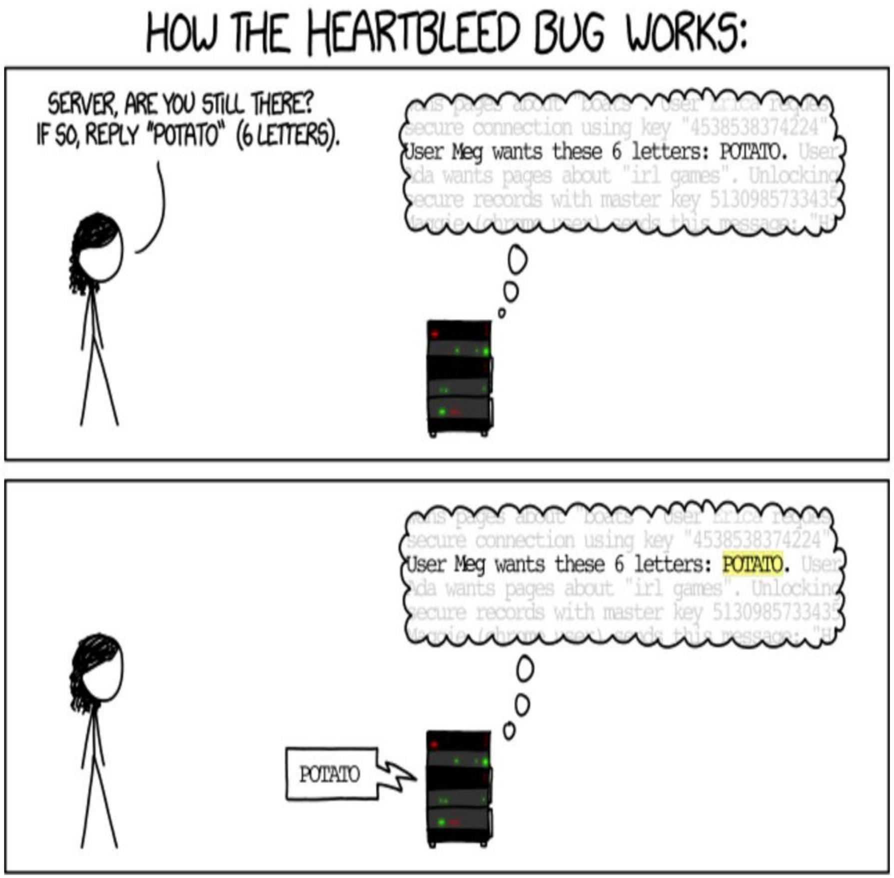
```
char name[100];
while ((read = getch()) != '\0')
{
    name[id++] = read;
        .....
}
```



getch() reads input from standard input terminal

Overflow Example: Heartbleed

- Heartbleed is a [security bug](#)in the [OpenSSL cryptography](#)library, which is a widely used [implementation of the Transport Layer Security (TLS) protocol.](#)
- Details can be found at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160



Example: Heartbleed

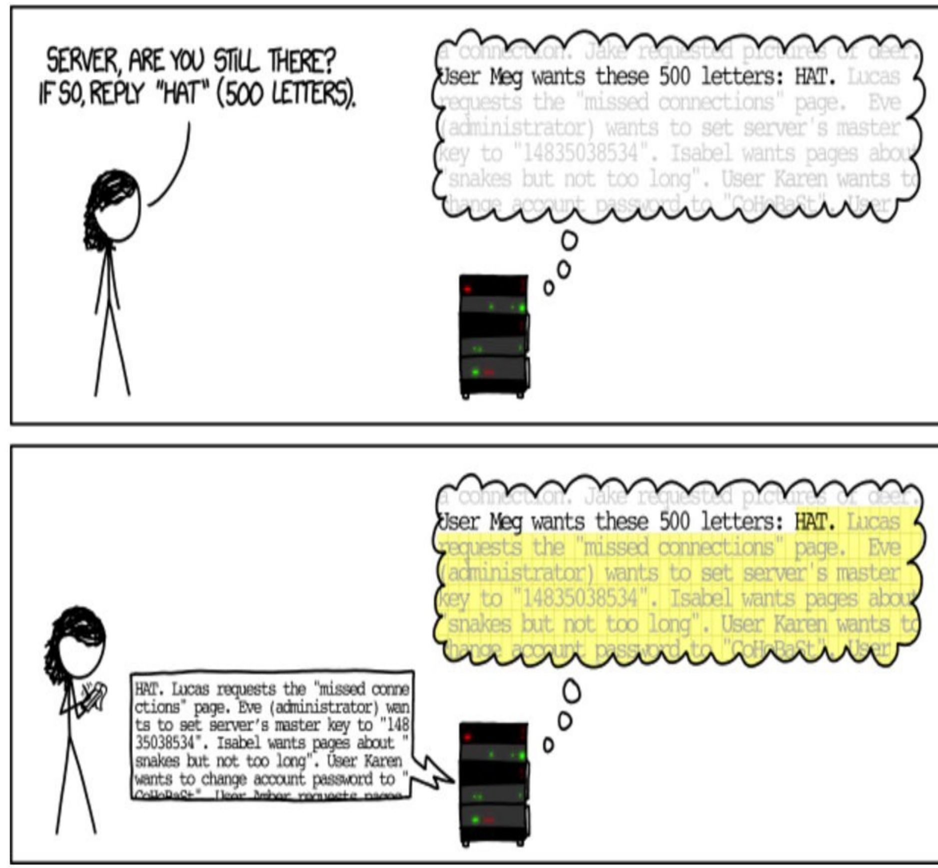
The Bug:

```
memcpy(bp, pl, payload);
```

where **bp** is a pointer, **pl** is where the data the client sent as a heartbeat is, and **payload** is a number that says how big **pl** is.

The Fix:

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
return 0; /* silently discard per RFC 6520 sec. 4 */
```



How Does Testing Work?

Questions

- How do we run tests?
- How do we know whether the output is correct or not? (The oracle problem)
- What inputs do we test and how do we generate them? (The test-generation problem)
- When have we tested enough?

Sample Answers

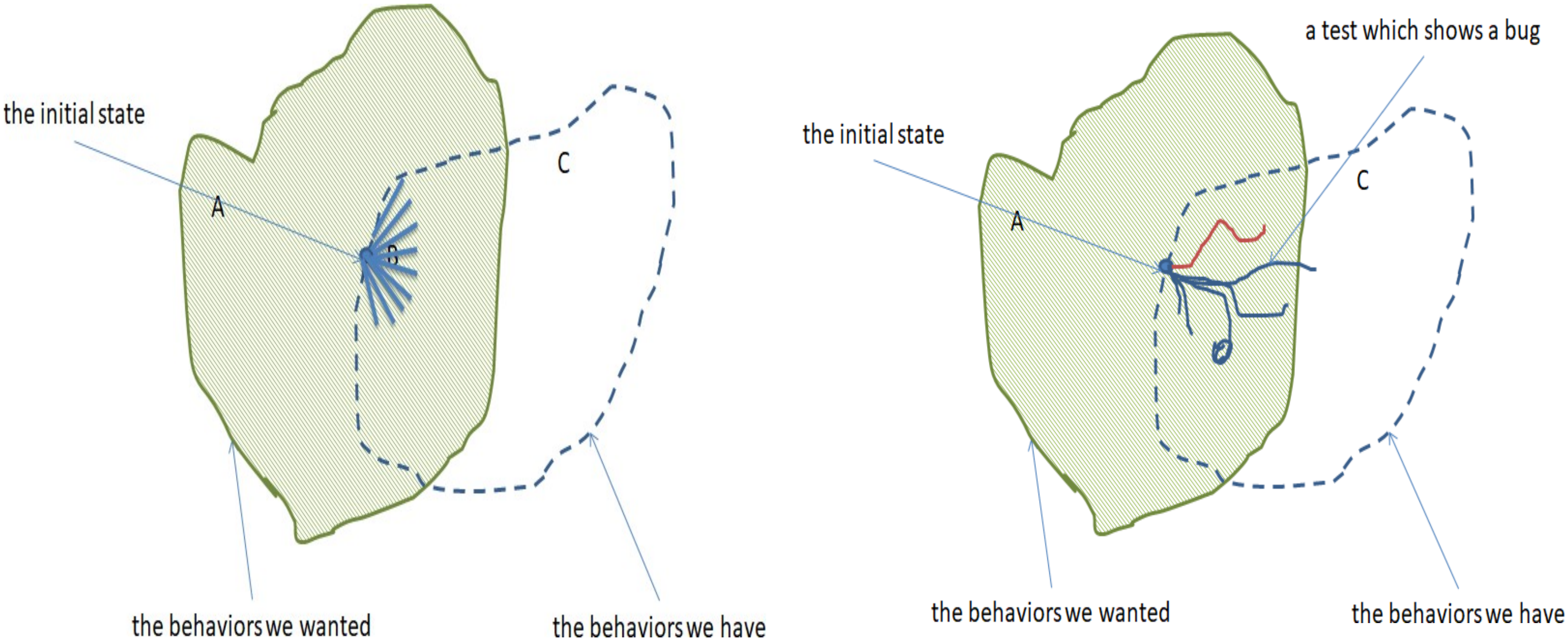
- jUnit
- Selenium driver

Security Relevant Testing

Testing remains an effective way of checking functional correctness and security of systems.

- Penetration testing
- Fuzzing
- Systematic testing
- ...

Testing: a Big Picture



Fuzzing

Fuzzing or fuzz testing is an automated [software testing](#) technique that involves providing invalid, unexpected, or [random data](#) as inputs to a computer program.

- Very long or completely blank strings
- Maximum and minimum values for integers
- Null characters, new line characters, semi-colons
- Format string values (%n, %s, etc.)

Fuzzing aims to identify test inputs which reveal exploitable vulnerabilities.

Programmers often think in term of valid inputs!

Why Fuzzing

- A study found that one-quarter to one-third of all utilities on every UNIX system that the evaluators could get their hands on would crash in the presence of random input.
- A study that looked at 30 different Windows applications found that 21% of programs crashed and 24% hung when sent random mouse and keyboard input, and every single application crashed or hung when sent random Windows event messages.
- A study found that OS X applications, including Acrobat Reader, Apple Mail, Firefox, iChat, iTunes, MS Office, Opera, and Xcode, were even worse than the Windows ones.

Fuzzing: Pros and Cons

Pros

- Can provide results with little effort
- Can reveal bugs that were missed in a manual audit
- Provides an overall picture of the robustness of the target software

Cons

- Will not find all bugs
- The crashing test cases that are produced may be difficult to analyse, as the act of fuzzing does not give you much knowledge of how the software operates internally
- Programs with complex inputs can require much more work to produce a smart enough fuzzer to get sufficient code coverage

Fuzzing Phases

- Identify target
- Identify inputs
- Generate fuzzed data
- Execute fuzzed data
- Monitor for exceptions/errors/crashes
- Determine exploitability

The central question: how to smartly generate fuzzed data?

Fuzzer Output

```
`,a=~F]8b'<Dks}jG[BCO:U65~3+hAO[(qs=z!X?|G_>Ia3<yNm\hO6#R;
C-Fkmo\U$5l2qpm"$#QM7',bI{x^B$MXW`JxdguN@Cz2m=]*-
T2_IfWJo(&3+QPz j?w+FX:iif
ey$~6WykYgC^(GZ[d*Qd6M+O>Gh*TLThD\Sxk`;8J2g'1bPH2bb1O`^
LGRZ?MNt>2trkvJ Gm`W|(+4@/\W/ByT7HAsZ#_4}abq)50ghBfs
```

simple-fuzzer.c

Reasons for Bombing

- Arrays and Pointers
- Not checking return code of a function
- …

Safe Coding Practices

- Check all array bounds
- Apply bounds on program inputs
- Check all return values of a function
- Do not trust third-party inputs

All supported by modern programming languages

Fuzzer Output

Z(cG*mOGaQ%%SWbFUXVGZin.,Kg5x)a0fM{,3+{Pd=X#s-
'^*\m<rG%~Z)"ZWqeJJox'w|dV7XbK)"xl@5=8LW`VFZAqsk,Aaxm0;6bh6H]S+
pj=\#M3CwSy\$Ko?`t.tR!)WTmWE_gHxLKS*cT$\:[A
Z01Uv;/LB,Z?1cb]\|*^EL L?<iBt<Q'\oOFDUy]7)-
6e2e>PyNkU7'\!HKiD}H/ P33ly$i/(74c$Ntq`r!|`ioyW
!1!&:c`JZsAD3CkZ!J?@aF /+T_<NFU!!Ab'U{zvV(g:\N+.
X&5i:}?85r]
t4#Te?;T~]YVTehejfqY=\} FQd5:"X|?P#Z"&VD#l&ysw
frNZo(\igNKFf{S&2`l.7O/ ISK,\9huTKuZX"5OB=Fl:O0#
&|db/uY&WKm^HB7}xt' 6@?^;2hV]'WrKIAb`WY"C
G+fM.npbOzoZ9JAJ@PS
SxnyVvb({2Ja]b?.1*Y!S`vTRnms$Er9e9'U"+_0<gq>WS*J:I1XJzt\.Dh%C:ePg`7oc[
E+db9-
?TI.&49O9Z#WNvPmFhUgJmL8v+WSN@o/[;f(RIPkoBabRzDNV$77MBw4\hpkUI
)4+

Most Programs reject invalid inputs

How to get fuzzer to generate valid input?

Idea

ISTDisApillarInSUTDbutItsNameiSGoin
gToChange

Leverage existing VALID inputs

Mutation

ISTDisApillarInSUTDbutItsNameiSGoingToChange

Trim at a random position

ISTDisApillarInSUTDbutIts

Mutation

ISTDisApillarInSUTDbutItsNameiSGoingToChange

Select a position randomly

ISTDisApillar**I**nSUTDbutItsNameiSGoingToChange

Flip a random bit of the selected position

ISTDisApillar**Y**nSUTDbutItsNameiSGoingToChange

mutation-fuzzer.c

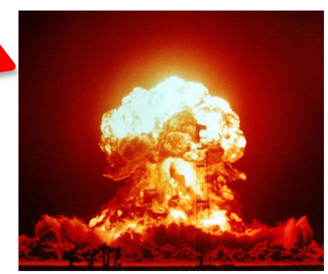
Mutation Operators

- Flipping a bit
- Trimming
- Swapping characters
- Inserting characters

Mutation operators are chosen at random in every iteration

Idea

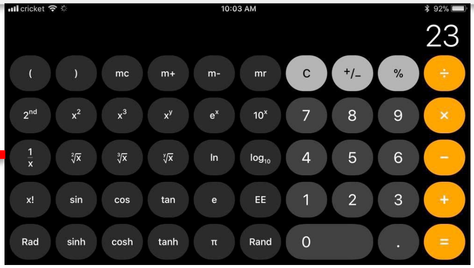
Parser



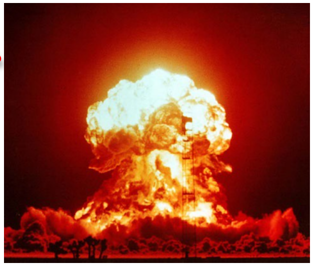
Get a format of the valid inputs

Idea

Calculator



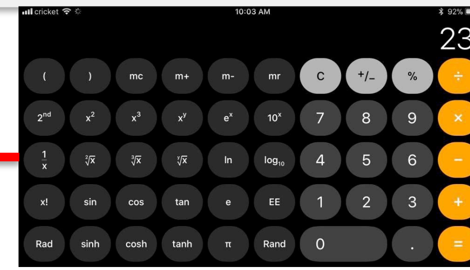
Parser



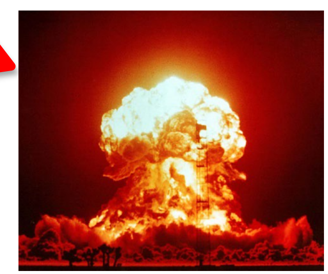
Calculator supports arithmetic expressions

Idea

Calculator



Parser



1+3, 3+1/2, 4*(5+1)-1 are valid inputs

1/*1, +-21/12, 2---- are invalid inputs

Idea

S := Expr

Expr := Expr + Term | Expr – Term | Term Term

:= Term * Factor | Term / Factor | Factor

Factor := -Integer | (Expr) | Integer | Integer.Integer

Integer := Digit | IntegerDigit

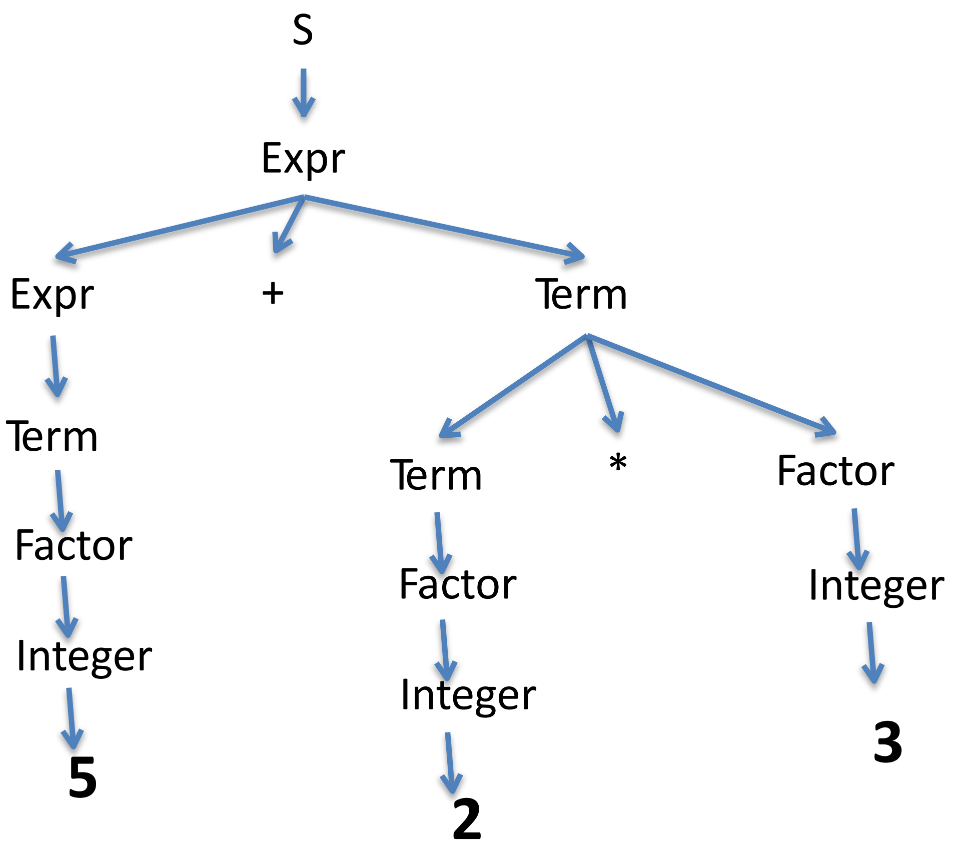
Digit := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Rules

Integer can split into digit or Integer & digit
eg 53 (integer)
5 (Integer)   3(digit)
5 (Digit)   3
5

Capture valid inputs by a grammar

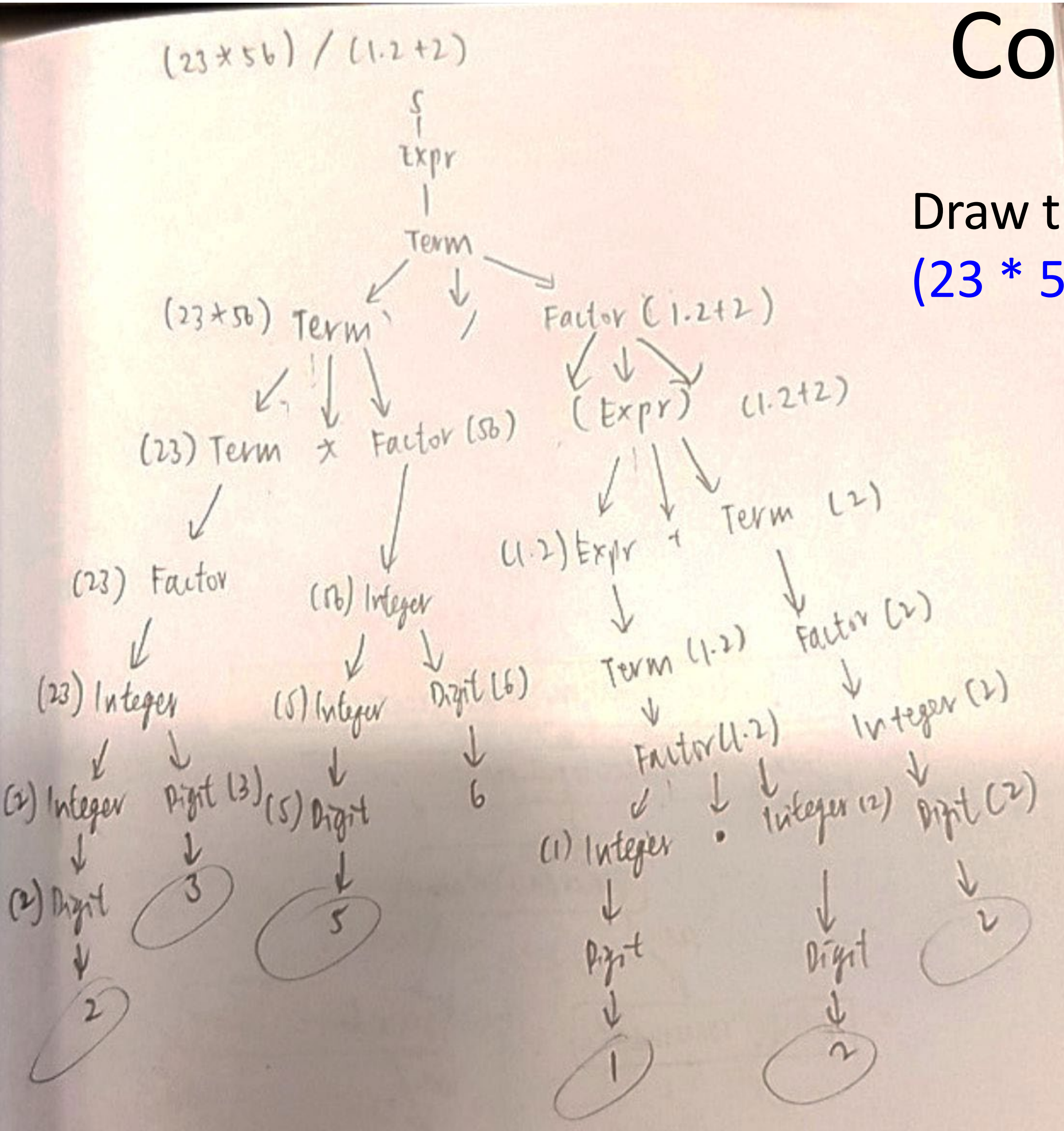
Derivation of inputs from grammar



Generates 5+2*3

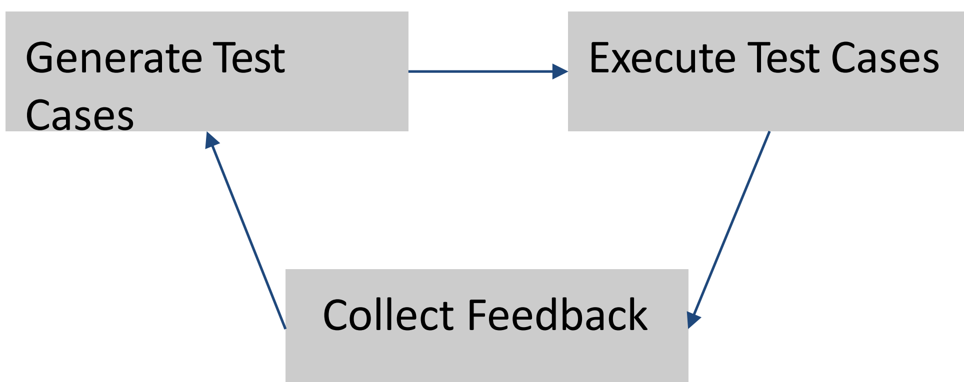
Capture valid inputs by a grammar

Cohort Exercise 2



Draw the derivation tree for input (23 * 56) / (1.2 + 2)

Feedback-based Fuzzing



The idea is to tune the test case generation problem into an optimization problem.

Example

AFL (American Fuzz Lop) is a well-known fuzzer which is responsible for finding significant software bugs in dozens of major free software projects, including PHP, OpenSSL, Firefox, etc.

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

Collect Feedback

One way to measure the effectiveness of a set of test cases is code coverage criteria.

We can instruct the program to collect coverage measurement. For instance, for branch coverage, we instructment each branch so as to know whether it is covered by a test case.

Example

```
int func (int[] B, int N) {
    int i = 0;
    int s = 0;
    if ( i != N) {
        //print: "branch coverage";
        i = i+1;
        s = s +B[i];
    }
    return s;
}
```

Code Instrumentation

Useful for

- Code coverage measurement
- Memory and performance profiling and runtime tracing
- Runtime verification

Considerations

- Runtime overhead

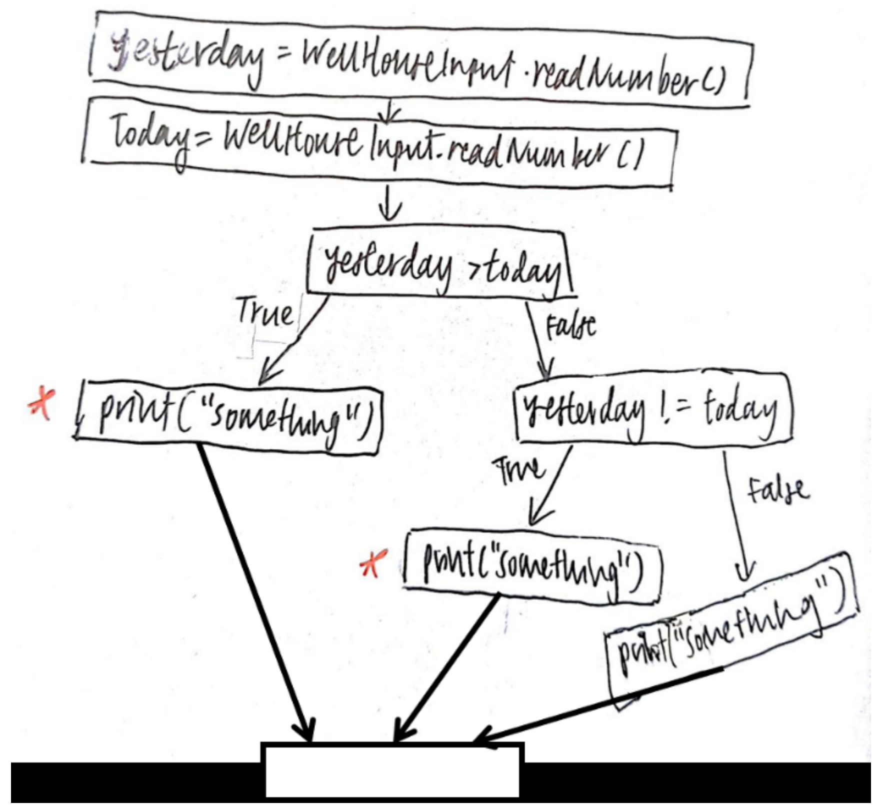
There are existing tools for systematic code instrumentation (e.g., Soot).

Cohort Exercise 4

Assume that we need to instrument the code on the right to obtain statement coverage measure.

Instrument the code in a way such that we can obtain the statement coverage of any test case while keeping the overhead minimum.

```
public static void foo () {
    Float yesterday=WellHouseInput.readNumber();
    float today=WellHouseInput.readNumber();
    if (yesterday > today) {
        System.out.println("something");
    }
    else {
        if (yesterday != today) {
                System.out.println("something");
        }
        else {
                System.out.println("something");
        }
    }
}
```



Collect Feedback: Crash Detection

If you cannot accurately determine when a program has crashed, you will not be able to identify a test case as triggering a bug.

- Attach a debugger: most accurate, significantly overhead
- Runtime-monitoring through code instrumentation: e.g. AddressSanitizer
- Timeout: to know whether a deadlock or infinite loop has been triggered.

Example

```
void func (char *str) {
        char buff[12];
        //add: if (strlen(str) > 12)
        //              printf("overflow")
        strcpy(buff, str);
}
```

Generate New Test Cases

The fuzzer gradually evolves a set of test cases that improves code coverage.

Example:

Genetic Algorithms

Simulated Annealing

History

Genetic Algorithms were invented to mimic some of the processes observed in natural evolution.

The idea with GA is to use this power of evolution to solve optimization problems.

Genetic Algorithm

Overall Idea

GAs simulate the survival of the fittest among individuals over consecutive generation for solving a problem.

Each generation consists of a population of test inputs that are analogous to the chromosome that we see in our DNA.

Each individual represents a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution.

Algorithm

randomly initialize population(t)
determine fitness of population(t)
Repeat
    select parents from population(t);
    perform crossover for population(t+1)
    perform mutation of population(t+1)
    determine fitness of population(t+1)
until best individual is good enough

Example: GA directory

Genetic Algorithm

Selection Operator

- key idea: give preference to better individuals, allowing them to pass on their genes to the next generation.
- The goodness of each individual depends on its fitness.

Example:

```
static int getFitness(Individual individual) {
   int fitness = 0;
   for (int i = 0; i < individual.size(); i++) {
        fitness-=Math.abs(individual.getGene(i)-solution[i]);
   }
   return fitness;
}
```

The fitness is defined based on the accumulated difference of each character.

Genetic Algorithm

Crossover Operator

- Two individuals are chosen from the population through the selection operator.
- The values of the two individuals are exchanged.
- The new offspring created from this mating are put into the next generation of the population.

Example:

```
private static Individual crossover(Individual indiv1, Individual indiv2) {
    Individual newSol = new Individual();
    for (int i = 0; i < indiv1.size(); i++) {
        if (Math.random() <= uniformRate) {
            newSol.setGene(i, indiv1.getGene(i));
        } else {
            newSol.setGene(i, indiv2.getGene(i));
        }
    }
    return newSol;
}
```

The offspring is a random combination of the parents.

Genetic Algorithm

Mutation Operator

- With some probability, a portion of the new individuals will have some of their genes mutated.
- Its purpose is to maintain diversity within the population and inhibit premature convergence.
- Mutation alone induces a random walk through the search space.

Example:

```
private static void mutate(Individual indiv){
    for (int i = 0; i < indiv.size(); i++){
        if (Math.random() <= mutationRate) {
            Random r = new Random();
            char c = (char)(r.nextInt(95) + 32);
            indiv.setGene(i, c);
        }
    }
}
```

Pick a random gene and mutate it randomly.

Cohort Exercise 5

1. Study the implementation of the Genetic Algorithm provided.

2.Modify the provided genetic algorithm so that it generates any palindrome string with 64 characters.

- How do you define the fitness function?
- How do you define the selection/crossover/mutation operator?

Modified fitness function

```
fitness-= Math.abs(individual.getGene(i) -
individual.getGene(individual.size()-i-1));
```

The selection/crossover/mutation operator were not modified.

As seen above, the Harder class took a longer time to test while the Easier class took a much
shorter time to test. In the Harder class, there were a lot of “==” conditions which made it harder to test because the probability of coming up with a test that satisfies these strict conditions is lower. On the other hand, in the easier class, the conditions were mostly “<” and passing some conditions will automatically allow you to pass the subsequent conditions. The conditions were not strict and thus the probability of coming up with a test that satisfies the conditions were much higher, leading to a much shorter time taken. In the Example class, the conditions were mostly “>” then “<”, meaning that the inputs x, y, z have to be in a range in order to satisfy the conditions. Hence, these conditions were stricter than the Easier class but more relaxed than the Harder class. Unsurprisingly, the time taken to test the Example class falls in between the Harder and Easier classes.

White-Box Fuzzing

Fuzzing is a form of random testing, which has its limitations.

Fuzzing is likely to find the bug in the following code.

```
public static void example(int x, int y)
    { int[] array = new int[10];
    array[x] = y; //x must be [0..9]
}
```

Fuzzing is unlikely to find the bug in the following code.

```
public static void example(int x, int y) {
    int[] array = new int[10];

    if (y == 42342531) {
        array[x] = y; //x must be [0..9]
    }
}
```

We Do Logical Reasoning

```
public static void example(int x, int y) {
        int[] array = new int[10];

        if (x > 0) {
                assert(x>=0);
                Array[x] = 5; (x>9 && x>0) || (x<0 &&x>0)
        }
}
```

not possible
$x < 0$ && $x > 0$ will not happen

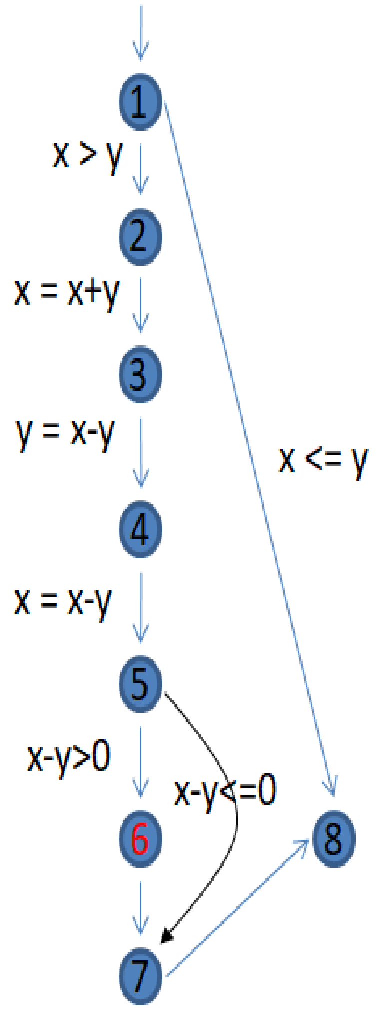
but array[x] = 5 might lead to overflow

assert(x<=9)

Will assertion failure occur?

We Do Logical Reasoning

```
public static void func (int x, int y) {
1.  int[] array = new array [10];
2.  if (x>y) {
2.        x = x + y;
3.        y = x – y;
4.        x = x – y;
5.        if (x-y>0) {
6.            array[10]=0;
7.        }
8. }
```



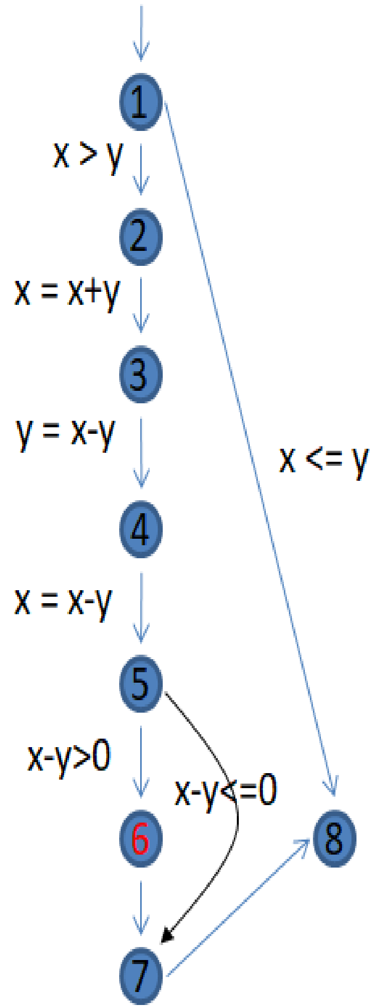
program has a bug if and only if x-y>0 condition can be true

We Do Logical Reasoning

Error occurs if and only if the following path condition is satisfiable:

x1 > y1 &&
x2=x1 && y2 = y1 &&
x3=x2+y2 && y3 = y2 &&
x4=x3 && y4=x3-y3 &&
x5=x4-y4 && y5=y4 &&
x5-y5>0

xi = value of "x" immediately before node "i"
yi = value of "y" immediately before node "i"



Constraint Solving

How do we efficiently know whether the following constraint is satisfiable or not?

x1 > y1 &&
x2=x1 && y2 = y1 &&
x3=x2+y2 && y3 = y2 &&
x4=x3 && y4=x3-y3 &&
x5=x4-y4 && y5=y4 &&
x5-y5>0

We use automatic constraint solvers.

Symbolic Execution

- Rather than executing a program with concrete input value, execute it with symbolic variables representing the inputs.
- Proposed in 1976: “A System to Generate Test Data and Symbolically Execute Programs”, IEEE Transactions on Software Engineering by L. A. Clarke.
- Popularized only in recent years due to advancement in constraint solving techniques.
- Used for white-box fuzzing, e.g. Microsoft SAGE.

Symbolic Execution Engines

- KLEE based on LLVM
- Pex from Microsoft for .NET
- JPF (Java Path Finder) and JDart for Java programs
- Jalangi2 for JavaScript
- Oyene for smart contracts

- Purpose of Symbolic Execution
    - Provided enough time, it can explore all execution paths of a program
        - Often infeasible in practice.

Symbolic Execution Tree

- What is Symbolic Execution
  - Executing a program with un-instantiated values for certain variables
    - E.g. input variables
  - Since some values are un-instantiated the execution is NOT a sequential trace any more
  - The symbolic execution forms a tree where each path from the root of the tree to its leaf corresponds to a unique execution path
    - Symbolic Execution Tree

Symbolic Execution: Limitation

Path Explosion

```
int x = input();
while (x > 0) {
    x++;
    assert(x < Integer.MAX_VALUE);
}
```

How do we handle loops?

- check all paths which reach the assertion in one iteration.
- … in two iterations.
- … in three iterations.

When does it end?

(Typical solution: we check up to certain number of iterations, or we find out somehow all possible x values).

Incompleteness

```
public static void func (int x, int y, int z) {
    if (x*x*x*x + y*y*y < z*z) {
        assert(false);
    }
}
```

SMT solver is no magic

- Existing SMT solvers supports theories on linear integer arithmetic, bit vectors, string, etc.
- Existing SMT solvers are not particularly scalable or efficient for certain theories.