Introduction

1. **Purpose of software testing**
    a. software testing methods are used to
        - Evaluate and assure that a program meets all of its requirements, both functional and non-functional

    b. The aim of most testing methods is to systematically and actively find these discrepancies in the program.
    c. testing vs. debugging
        - The aim of testing is to demonstrate that there are faults in a program
        - The aim of debugging is to locate the cause of these faults, and remove or repair them.

2. **Why use software testing**
    a. For system requires high reliability on safety, faults can lead to serious consequences if not been discovered
    b. It is hard to measure to what extent the software meets its specification without testing (Avoid deviation from specification
3. **Different types of testing**
    a. Unit testing
    Unit testing measures the correctness, completeness and consistency of units such as **functions** and classes.
    b. Integration testing
    Integration testing tests collections of modules **working together** to confirm that the modules works together properly.
    c. System testing
    System testing tests the entire system against **requirements** specification and design goals.
    d. End-to-end
    e. Performance testing
    Non-functional testing on the system
    f. User acceptance
    User acceptance tests is a set of tests that the software must pass before it is accepted by the **clients**.
    g. Regression testing
    Regression tests runs the entire test suite of a system **after any change**.
4. **Input domain**
    a. The set of values in the input type that are accepted by the program as inputs
    b. Not all elements of an input domain are relevant to the specification
    c. Input domain can vary in different machines
    d. Deterministic: every input has a unique output
    e. Non-deterministic: return one of a possible number of outputs for a given input
5. **Faults, errors, and failures**
    a. Faults
        i. An incorrect step, process, or data definition in a computer program.
        ii. In systems it can be more complicated and may be the result of an incorrect design step or a problem in manufacture

b. Failures
    i. A failure occurs when there is a deviation of the observed behaviour of a program, or a system, from its specification.
    ii. A failure can also occur if the observed behaviour of a system, or program, deviates from its intended behaviour which **may not be captured** in any specification
c. Errors
    i. An incorrect internal state that is the result of some fault.
    ii. An error may not result in a failure – it is possible that an internal state is incorrect but that it does not affect the output.
d. A fault in a program can trigger a failure and/or an error under the right circumstances.
e. Coincidental correctness
    i. trigger an error not results in a failure
f. Failure and error are results of faults
g. Example: square(3) results in 6
    i. A failure – because our specification demands that the computed answer should be 9.
    ii. The fault leading to failure occurs in the statement return x*2
    iii. The first error occurs when the expression x*2 is calculated

6. Software testing work process

a. Test strategy
b. Test case selection
c. Test execution
d. Test evaluation
e. Test reporting
f. Try to fix the fault to improve the overall quality of the program.

7. Testability

a. Controllability
    - The controllability of a software artifact is the degree to which a tester can provide test inputs to the software.
b. Observability
    - The observability of a software artifact is the degree to which a tester can observe the behaviour of a software artifact, such as its outputs and its effect on its environment.
c. Impact
    i. Use interface may reduce the testability (embedded software)

8. Discuss how software testing achieves its goals.
9. Specify the input domain of a program
10. Develop an oracle for a program from a specification.

Input partitioning

1. **Faults**
    a. computational fault: correct path but incorrect computation
    b. domain fault: correct computation but incorrect path
2. **Equivalence class**
    a. A set of values from input domain that are considered to be as likely as each other to produce failures of the program
    b. Each element in the EC should execute the same statements in the program in the same order
    c. input condition
        i. a single variable in the input domain (combination of variables)
    d. valid input
        i. The input in the input domain is the expected value from the program
        ii. non-error value
    e. invalid input
        i. An element of the input domain that is not expected
        ii. testable/untestable
    f. binary search function
        i. input condition: sorted array
        ii. invalid input: unsorted array
    g. Property of EC
        i. EC1^EC2 = 0 -> disjoint
        ii. Combine EC = ID -> testable ID is covered
3. **Domain testing**
    a. White-box testing for the execution of statement
    b. Determine the actual EC of the program
    c. Terminology
        i. relational expression
        ii. predicates(diverge in program)
            1. simple predicates
            2. compound predicates
            3. linear simple predicates
        iii. path condition
4. **Equivalence partitioning**
    a. Black-box testing for the functionality
    b. Determine the expected EC from the function requirements
    c. Assumption
        i. All members in one EC behaves in the same way with respect to failure
        ii. Not entirely correct
    d. **Purpose**
        i. Test case selection based on the input and output domain
        ii. To minimise the number of test cases required to cover all of the equivalence classes
    e. Methods to choose EC
        i. identify initial EC

ii. eliminate overlapping EC

iii. select test cases

5. **EC selection guidelines(testable ID)**
    a. G1-given a range of value
        i. partition 1 valid EC and 2 invalid ECs
        ii. [1,99] -> EC *3
    b. G2-A set of possible input
        i. Every kind of input in the set + one out of scope EC
        ii. {A,B} -> EC_a,EC_b, EC_x
    c. G3-specify the input number N
        i. 1 valid EC, 2 invalid EC (N<>number of input)
    d. G4-varying size of collection as input
        i. valid EC(size=0), valid EC(size=1),valid EC(size>1)
        ii. eg:maximum number of input = 10 -> 0/1/5/10/11(invalid)
    e. G5-“Must-be”
        i. one valid & one invalid
        ii. {first s is number}
    f. G6-different manner

6. Produce equivalence classes for testing a program based on its functional specification.
7. Produce equivalence classes for testing a program based on its source code.
8. **Test template trees**
    a. Mitigating tree explosion
    b. Variable interaction
9. Combine equivalence classes from different domains/variables.

Boundary-value analysis

1. **Definition**
    a. boundary-value analysis
        - selects test cases to explore **conditions** on, and around, the edges of equivalence classes obtained by input partitioning
    b. boundary conditions
        - apply on/above/beneath the boundaries of input
2. **Is valuable than equivalence partitioning**
    - Although we assume that every test case in the same EC performs the same path
    - Man made-error in the logic of program
        - some inputs execute incorrect paths
        - more likely at the boundary between EC
3. **Value and boundary**
    a. path condition: the condition that input data to execute the path
    b. domain: the set of input data satisfying path condition
    c. domain boundary: boundary of domain
    d. Type of boundary
        i. closed boundary (with =)
        ii. open boundary (no =)
        iii. on point (x>10 ,op = 10 but not in this EC)
        iv. off point
            - (open x<10): off_p = 9
            - [closed x<=10]: off_p = 11
4. **Test cases selection guidelines**
    a. == : 选择on point 和2个off point
    b. <> : 选择on point 和1个off point(符合条件的)
    c. unordered types枚举：选择1个on 1个off
    d. 分析condition
    e. Intersection points -> reduce the number of test cases
    f. Boundary shift
        i. 不相等 on_p*2 (far from each other) + off_p
        ii. 相等 on_p*2,off_p*2
        iii. 相等 off_p距离为d detect > d

5. Given a set of equivalence classes for a program, identify the boundaries of those equivalence classes, including the on points and off points.
6. From a set of equivalence classes, select test inputs that adequately cover the boundaries of a program.

Coverage-based testing

1. **Control-flow-graph**
    a. branch/decision
    b. condition
    c. path
2. Derive a control-flow graph for a program from its source code.
3. **Control-flow coverage criteria**
    a. statement coverage
    b. branch coverage (each branch value in T/F)
    c. condition coverage (each condition value in T/F)
    d. decision/condition coverage
    e. multiple-condition coverage
    f. path coverage (not possible in loops)
        i. execute 0 time
        ii. execute once -> loop can be entered
        iii. execute twice -> different iterations
        iv. execute N times -> arbitrary iterations
        v. execute N+1 times -> after N, still "correct"(upper bound)
4. **Problem of CFA**
    a. If we generate test inputs automatically only based on CFA, it may not related to the specification
    b. May not get good coverage
5. **Data-flow testing**
    a. Why introduce data-flow testing
        i. better test cases selection
        ii. produce additional information for analysis
        iii. creation and use of data information
    b. static data-flow analysis
        i. define a variable (x=5)
        ii. reference a variable
            1. l-value (on the left side of assignment)
            2. r-value (y=3*x)
        iii. undefine a variable(unknown->the scope of local variable ends
    c. Aim of static data-flow analysis
        - Trace through the program's control-flow graph and detect data-flow anomalies.
        - Not tell what the fault actually is but potentially something wrong
    d. data-flow anomalies(indicate possibility of program faults)
        i. u-r
           uninitialized variable is used
        ii. d-u
           variable not be used before undefined
        iii. d-d
           same variable is defined twice
           no reference between two defined variables
6. Select test inputs from a control-flow graph annotated with variable information, using the various data-flow coverage criteria.

7. Dynamic data-flow testing

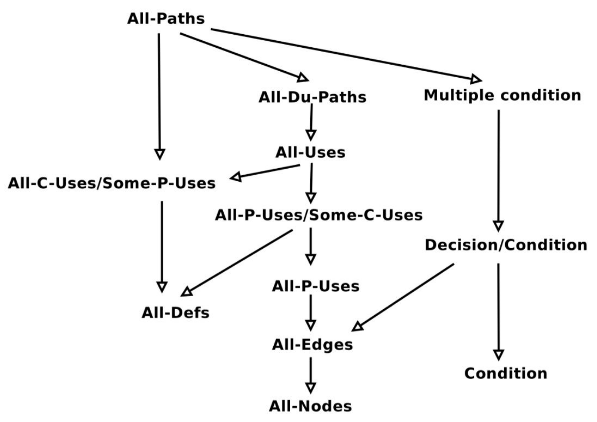
a. definition
   i. C-use(computational use y=2*x)
   ii. P-use(predicate use y<2)
   iii. D: assigned/initialized to a value n
   iv. U: variable is used/referenced
   v. K: variable is killed/undefined
   vi. A definition clear path
      - from head node define to undefine
   vii. A loop-free path segment
      - every node at most is visit once
   viii. A definition reaches a use
      - head to end

8. Data-flow path selection criteria

a. All-Defs
   i. Sub-path for all defined variables
   ii. A path contains one variable define-to-use
b. All-Uses
   i. All definitions of a variable
   ii. All uses from this definition of a variable
c. All-Du-Paths
   i. All definitions to all uses and every successor after use
   ii. if from u1 to u2 has “Y”, includes both paths
d. All C-use, some P-use
e. All P-use, some C-use
f. All P-use

9. Coverage vs. data-flow criteria

a. better coverage score
b. Adv of data-flow
   i. “d-u” anomaly can be revealed by DFA but may not be revealed by testing due to equivalent behaviour
   ii. Static data-flow analysis
      + anomaly will always be detected whereas using testing, it requires certain cases to be selected
c. Disadv
   i. Be careful about anomaly in table-analysis in loops
d. subsume relation
   The subsumes relation is transitive, therefore, if A subsumes B, and B subsumes C, then A subsumes C.



10.Mutation-based analysis

a. definition
   i. mutant:a copy of the program, but with one slight syntactic change
   ii. select input not test case(output)
b. process of mutation
   i. Systematically introduce some slight syntactic change to create multiple mutants about the original program.
   ii. Run test suites on all those mutants and see how much of the mutants has been killed.
   iii. This infers whether the test suites are adequate or not.
   iv. Output difference means "kill"
c. The meaning of mutation-based analysis
   i. The program is close to correct but not totally correct
   ii. Faulty program do not produce error by running insufficient test cases
   iii. A method for measuring the effectiveness of test suites(add new)
   iv. Handle coupling effect:
   Create mutations easily by introducing some simple faults to check whether test suites are adequate or not.

11.Coupling effect

a. what is coupling effect
   i. A test case that distinguishes a small fault in a program by identifying unexpected behaviour is so sensitive that it will distinguish more complex faults
   ii. That is, complex faults in programs are coupled with simple faults.
b. Impact on mutation analysis
   i. Mutation analysis creates mutations easily by introducing some simple faults to check whether test suites are adequate or not.
   ii. Original program test cases can be evaluated by modifying the program

12.Systematic Mutation Analysis via Mutant Operators

a. Mutant operators: a transformation rule to generate mutants
   i. relational replacement
   ii. value insertion/positive or negative
   iii. conditional
   iv. shift << >>
   v. logical & | ^
   vi. assignment += -=

    vii. unary operator insertion/deletion !
    viii. scalar variable replacement
    ix. bomb replacement
  b. mutation score
    - dividing the number of killed mutants by the number of total mutants

13.Equivalent mutants problem

  a. equivalent mutants
    - for every input, the program and the mutant produce the same output
    - cannot be killed by any test case due to the equivalence
  b. impact
    - Make tester harder to find which mutants have not been killed due to the inadequate of test suites
    - An adequate test suite that one that kills every mutant is unachievable
    - Need manual analysis to compare whether two programs are equivalent or not

14.Comparing coverage criteria

  a. Subsume relation
    i. Multiple condition coverage and the criteria that it subsumes do not subsume any of the data-flow criteria
    ii. Decision/condition coverage clearly subsumes both branch and condition coverage, also subsumes statement cover- age,
    iii. Multiple-condition coverage subsumes decision/condition coverage
    iv. Path coverage (if possible) subsumes branch coverage

Test oracles

1. **Definition**
   a. test case
      i. inputs
      ii. expected output
      iii. environment
   b. test oracle
      i. program
      ii. process
      iii. a body of data
      iv. to determine if the actual output is failed or not
2. **Different types of test oracle**
   a. active oracle(generate)
      i. given an input for a program-under-test, can generate the expected output of that input
      + always exists active oracles
      - no false positive
   b. passive oracle(verify)
      i. given an input for a program-under-test, and the actual output produced by that program-under-test, verifies whether the actual output is correct.
      + preferred than active one because of less replication
      + can handle non-deterministic programs(1-input m-output)
      - not suitable for some hard problems (math equation)
3. **Test oracle types**
   a. formal,executable specification (equations)
   b. solved examples (handmade computation)
   c. metamorphic oracles
      i. testing sorted list
         - input : permutation of the list
         - output: same results
      ii. testing shortest path in a graph
         - check subpath points' shortest path
   d. alternative implementation
      i. partial, simplified version of implementation
      ii. sorted list: check boundary value index
   e. heuristic oracles
      i. provide approximate results for input
      ii. find patterns between inputs and outputs
      iii. testing database insert: check record +1 ?
      iv. testing GPS shortest path: <1.5 times of straight distance
   f. the golden program (previous program implementation)
4. **Why and how to use test oracles?**
   a. Assist random testing
      i. cannot know the input in advance so have no idea on the output evaluation

    b. Test oracles usually return Yes/No
    c. Steps
        i. Identify the failure in the program
        ii. Constraints of inputs and outputs
        iii. Violating the domain & requirement
5. Choose the correct type of test oracle for a given problem
6. Develop an oracle for a program from a specification

Test Modules

1. **OO testing**
    a. test module
    instead of allowing any part of a program to manipulate any part of data, if access to data was provided via a clear and unambiguous interface made up of functions on that data, then complex program could be understood more readily
    b. state : data
    c. module : a collection of operations on the state
2. **Testability**
Testability of a program is defined by the **observability** of the program, and the **controllability** of the program.
3. **Discuss the impact that observability and controllability have when testing modules**
For example: A stack with push/pop/view/isEmpty
    a. observability
        - can be seen in the
    b. controllability
        - The program controlled directly
    c. Intrusively test
    Modify program to gain access to hidden data types
    However there is **not much observability** since it requires the programmer to change the code to some extent in order to get the internal state which:
        - Does not test the implementation being used.
        - If the underlying data types changes, the tests must change as well
        - Operations acting on the same data as inherently linked, and must be tested together
    d. **Non-intrusively test**
    Requires a set of operations defined by module states to be included in the test case which increases the observability but decrease the controllability.
    eg: use another function to test this function

4. **Construct a finite state automaton from the specification of a module**
    a. finite state automaton
    A finite state automaton (FSA) or finite state machine is a model of behaviour consisting of a **finite set of states** with actions that move the automata from one state to another
    b. construct the FSA
        i. identify states
        ii. identify operations in each states
        iii. identify the source and target states between each operation
    c. deriving test cases from a FSA
        i. a sequence of transitions from state 0 to state n
        ii. generate test sequences
5. **Transversal criteria**

a. state coverage: Not enough since not all transactions are covered (not operations can be tested)
b. transition coverage: Practical, feasible and straightforward to achieve. Subsume state coverage.
c. path coverage:not possible if there is a path circle

6. **Intrusively and non-intrusively test**
   - Intrusively: Requiring extra code to the module for accessing Internal state
   - Non-intrusively: Not requiring extra code to the module.
     - All observer operations could be used to observe the state when an FSA state is being visited for the first time.

7. **Problem with state-based testing**

8. **Polymorphism and inheritance in OOP**
   a. Key elements
      i. Classes and objects which provide the main units for structuring.
      ii. Inheritance, which provides one of the key structuring mechanisms for object-oriented design and implementation.
      iii. Dynamic Binding or Polymorphism, which provides a way of choosing which object to use at run-time
   b. Polymorphism and inheritance improve the developer's ability to design, maintenance and reuse codes much more easier.
   c. Inheritance in OOP
      i. class=object state + methods (same idea of state and a set of operations)
      ii. Must be tested in subclasses(new test cases)
         1. directly modified methods
         2. methods that reference it
      iii. Building and testing inheritance approaches
         1. flattening the inheritance hierarchy (redundant cases)
         2. incremental inheritance-based testing
            + reduce test cases
            - overhead of analysis
      iv. A form of regression testing
         - Aims to minimise the number of test cases needed to test a class modified by inheritance
   d. Polymorphism in OOP
      i. Procedural language is statically bound(which function will be called at compile time)
      ii. Each possible binding of a polymorphic class requires a separate set of test cases
   e. Adv of polymorphism and inheritance in OOP
      i. Natural framework of test module due to the same idea of state + operations
   f. Disadv of polymorphism and inheritance in OOP
      i. Subclass testing -> derive new test cases
      ii. Need to explore the effects of object-oriented testing in the presence of inheritance and polymorphism.

g. Derive a test suite that takes advantage of inheritance and counteracts the problems of polymorphism.
    - Determine the number of possible bindings through a combination of static and dynamic program analysis

Security testing

1. **Discuss the purpose of security testing**
    - Aims to determine how secure is our system by finding issues such as vulnerability.
    - Not to uncover program faults but uncover security vulnerabilities(allow attackers to do something unwanted)
2. **Vulnerability**
    - A fault in a program or system that allows an attacker to subvert its security
    - Unintended functionality
        - SQL injection
        - Buffer overflow
3. **Explain the potential security implications of security faults**
    - steal confidential information (confidentiality violation)
    - modify/corrupt critical data (integrity violation)
    - crash the system or make it unavailable to others (availability violation)
    - (arbitrary code execution)
    - gain unauthorized access
4. **Fuzzing(randomly)**
    a. What is fuzzing
        - Fuzzer generates inputs
        - Input is fed to the program
        - Program execution is monitored for crashes, exceptions, memory leaks

    b. Fuzzing testing

    Simplest form of fuzzing: inputs generated randomly (from a fixed distribution), aka "dumb fuzzing"

    c. Advantages of fuzzing
        - Generally fast to generate inputs and cheap
        - Unbiased because the inputs are randomly generated

    d. Disadvantages of fuzzing
        - Huge number of inputs might need to be generated
            - Large number of test inputs may need to be generated to cover input domain.
        - Might miss vulnerabilities (Pareto-Zipf: 80% bugs lie in 20% of the code)
            - Distribution of random inputs misses the program faults.
        - Highly unlikely to achieve good coverage

    ❖ **Mutation based fuzzing**
        ➢ Starting with a well-formed input and randomly modifying (mutating) parts of that input
        ➢ Adv
            ■ It generally achieves higher code coverage than random testing
                - produce valid checksum
            ■ Produce the superset of test cases than random inputs
            ■ Identify certain patterns of inputs
                - Figure out the target part in the protocol stack
        ➢ Disadv

        - The success is highly dependent on the **valid inputs** that are mutated.
            - good quality inputs to mutate
            - good heuristics to choose how to mutate inputs
        - It still suffers from low code coverage due to unlikely cases
- **Generation based fuzzing**
    - Using some specification of the input data, such as a grammar of the input.(with knowledge)
        - Web interface: the requirement of HTTP format
    - Adv
        - Generally giving higher coverage
        - identify certain patterns of inputs
    - Disadv
        - It requires some knowledge about the input protocol.
        - The setup time is much higher since knowledge are required.

5. Choosing different kinds of fuzzer

- Random fuzzing: Small input domain
- Mutation based fuzzing: Larger input domain without much knowledge.
- Generation based fuzzing: With enough knowledge on the input protocol with larger input domain.
- Consider some aspects
    - coverage
    - implementation
    - knowledge

6. Memory debuggers(sanitizers)

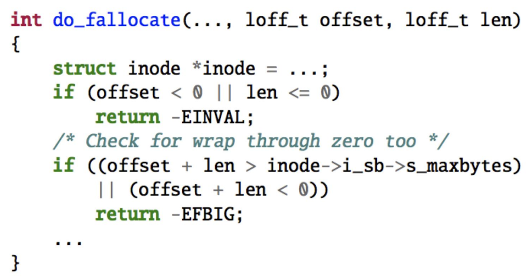
a. **Turns all sorts of common errors that would not cause the program to crash into crashes**
    i. Eg: overwrite the value of adjacent variable without a crash
b. Catching errors in
    i. Out-of-bounds accesses to memory (heap, stack, globals)
    ii. Use-after-free
    iii. Double-free and invalid-free
c. Memory leaks
d. Accesses to uninitialised memory
e. Out-of-bounds
    i. stack
       program overwrites the value of x
    ii. heap
       program defines the size of input array then use another one

7. UNDEFINED BEHAVIOUR

a. definition
   something that a program does that causes its future behaviour to be unknown.
b. List *5
    - divide by 0
    - use of NULL pointer(dereference/comparison/segmentation fail)
    - overflowing a **signed** integer
        - wrap around (2^32 - 1 + 1 is a negative number)

        - signed int -2^31 - 2^31-1
        - unsigned int 0-2^32 -1
    - underflowing a signed integer
    - Use of misaligned pointers
    - Loading bool values that are neither true nor false
    - buffer overflow
        - hacker can inject some data to the overflow area to get administrator authorization.

(`loff_t` is a signed integer)



```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
    ...
}
```

Compiler aims to do fast and delete some redundant statement which may allow signed overflow never be checked.

8. Memory debugger vs. undefined behaviour sanitizer

a. memory debugger
   - + Find anomalies such as buffer overflows that are difficult to observe using system behaviour.
   - - High performance cost

b. impact of memory debugger on testing
   - likelihood of finding different faults
   - Uninitialised memory / Freed memory / Memory overflows /Memory leaks

c. undefined behaviour sanitizer
   - + find some bugs that compiler may omit to check
   - - cost?

9. Data security

a. C.I.A.
   - Integrity: prevention of unauthorised data modification.
   - Confidentiality: prevention of unauthorised data disclosure.
   - Availability: ensuring attackers cannot deny legitimate access to data.

b. buffer overflow
   - Violate memory **integrity** and may cause program crashes, affecting availability

c. SQL injection
   - SQL injection violates the **integrity** of the database interface

d. Integrity -> directly observable

10.Overt channel

a. definition -> confidentiality
   An overt channel is a mechanism (whether intended or not) for directly transferring data in a system

b. mechanism

- Exist due to missing access checks
- Allow direct leakage of information
- Can be tested for by testing for missing checks (if you know what they are)
- example:

public variable

```
public = secret;
```

secret variable

```
print(secret);
```

```
public = secret + 4;
```

11.Covert channel

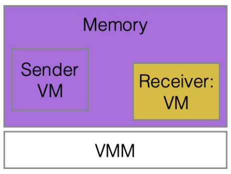
a. definition
   - The value of output depends on the sensitive data
   - Depends on output value, jumping into different piece of code

b. A and B communicate in a channel
   1. inaccessible data leakage (server sends back the whole size of buffer)
      - packetA.length = 200
      - packetA.data = "abc"
      - echo content is 200 length of data(leak info near packet in the memory)
   2. A's data sent by B (without free memory of A used)
      - packetA = "secret_A"
      - packetB = ""
      - send back A's data

c. Attack of memory info

To send the 1-bit (boolean) value **`secret`**:

```
void *p;
if (secret){
  p = malloc(HEAP_SIZE);
}
sleep(10);
```



To read the value that was sent (in receiver VM):

```
sleep(5);
/* allocate_memory returns 0 on success */
if (allocate_memory(100) == 0){
    secret = false;
}else{
    secret = true;
}
```

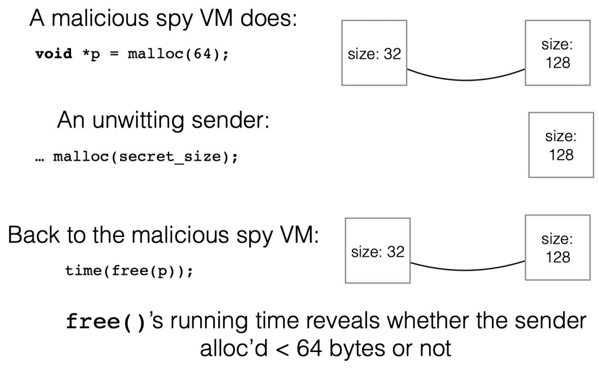
- The purple VM allocates all memory depends on the value of **secret**
- The yellow VM monitors the remind memory to infer secret

d. Implicit leaking: timing
   - update publicly observed data with time
   - timing behavior

```
public = public - 1;
while (secret > 0)
  secret = secret - 1;
print(public);
```

- The answer is produced depends on secret

e. inadvertent无意的 leakage


A malicious spy VM does:

```
void *p = malloc(64);
```

An unwitting sender:

```
… malloc(secret_size);
```

Back to the malicious spy VM:

```
time(free(p));
```

`free()`'s running time reveals whether the sender alloc'd < 64 bytes or not

- The information leakage by timing
    - The sorted blocks sequence may resort after one memory block is used
    - According to the timing -> infer which block is used

f. Covert storage channel

- leakage exists when varying the secret causes the public output to change

g. Covert timing channel

- executing time depends on the times of loop/value of secret

12.How to use security testing to find confidentiality faults?

13.How general-purpose greybox fuzzers operate?

a. definition

Greybox fuzzer are coverage-based evolutionary fuzzer that suitable for testing vulnerabilities within large programs

b. principle

It can generate new test case by slightly mutating a set of seed inputs. Once new coverage has been covered by that test case, it will be added to the set of seed inputs.

c. Adv

+ higher coverage score than random and mutation based fuzzing
+ with some time may even be able to beat generation based fuzzing
+ No need to analyze program itself

d. Disadv

- requires long time evolving

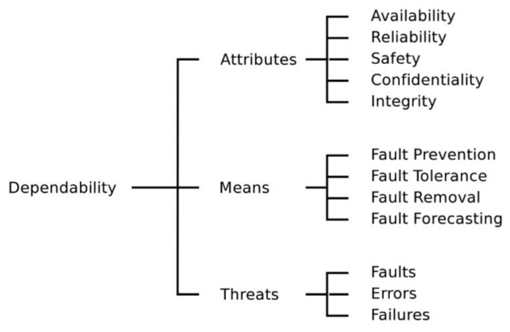
14.How code execution (injection) attacks operate?

Overwrite the return address of specific function and inject new address to guide the program execute in specific part

Reliability - Random Testing

1. Definition

a. Dependability



b. Reliability

Reliability is the probability that a system will operate **without failure** for a **specified time in a specified environment**

c. Failure

A failure is the departure of the results or execution of a program from its requirements

- program crash
- incorrect output
- response time too slow

d. Operational Profile (description of the inputs in the real world)

i. Probability distribution of program inputs, used for random testing to measure reliability

ii. Captures the program's expected environment Includes:

- The type of the input
- How often new inputs arrive / are generated

2. Reliability can be used to do ?

- Evaluate new system
- Evaluate development status
  - Compare current failure rates and desired failure rate
- monitor operational profile
  - evaluate new changes for the system
- Understand quality Attributes for the system

3. Random testing

a. Advantages of random testing

- Good simulation of reality
- Unbiased, free of human bias
- Cheap to generate inputs
- Black-box technique

b. Disadvantages of random testing

- Difficult to achieve high coverage
- Difficult to see low-probability failures
- Require a good operational profile

c. Purpose of random testing
    - Mainly for reliability measurement and security testing, because it gives a good statistical spread of the inputs to a program.
    - Could be also used for testing functional correctness.

d. Why use random testing to do reliability measurement
  - Not control-flow testing
    - More expensive to cover all branches
    - Assume each path can be executed in reality with equal probability
      - Eg: File system
      - Data in disk, error in disk is in low probability
      - Not capture reality (rare path)

4. Operational profile

a. What is operational profile
    - The operational profile of a program is the probability distribution of its input domain when the program is used in actual operation.
    - Generally, random testing inputs are generated according to the operational profile.
    - The distribution of data collected from operational profile affect reliability prediction.

b. ATM operational profile
  - Interaction with system may be different from the new one and the old one
  - Data collection from bank

Reliability Block Diagrams

Quantitative : how reliable it is (estimate the reliability)

1. **Reliability**
    a. Explain the difference between testing for reliability and testing for functional correctness.
    - Reliability
      Probability that component doesn't fail (over some period)
      F(C): failure probability
      R(C): reliability
2. **Reliability block diagram**
    a. Construct a reliability block diagram from a system architecture.
    - Serial Block: All of them are reliable
    - Parallel block: Any one of them is reliable
    - Independence
    - Inevitability of failure
    - Imagine super-reliable machines: MTBF of 30 years
    - Deployed at large scale (e.g. Google, Amazon, etc.): 10,000 of them
    - What is the probability that any of them fails on a given day?
        - 1 machine fails every 30 years, i.e. ~ 10,950 days
        - So F(M) = 1 / 10,950, **i.e. R(M) = 0.9999086758**
        - But we have 10,000 of them, so: **R(System) = 0.9999086758^{10000} = 0.401**
        - F(System) = **1 - R(System) = 0.598**
    - So expect roughly one failure every other day
    - **NOTE**: real hardware fails way more often than this (MTBF of *months*)

    b. Usability of block diagram
    - Most useful when we have good estimates of component reliability.

    i. Where do those come from?
    - (lots of) random testing
    - logs of usage over long periods in high volumes
    - vendor information

    ii. Why block diagrams?
    - Cost
    - how reliable does component X need to be for overall reliability?
    - what if we changed the architecture?
3. Given reliability estimates for the sub-systems in a reliability block diagram, calculate an estimate for the overall system using reliability and failure logic
4. **Random variables in the reliability model**
    - the number of failures at time T which is discrete, or an integer-valued, random variable
    - the time at which the first failure occurs, which is a continuous, or real-valued, random variable
5. **Construct a Markov model from a system description**
    - Model

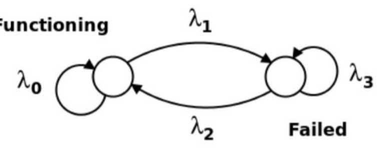


Figure 7.9: The two state model — functioning and failed.

For one state, the sum of out-point transition equals to one.

- Matrix

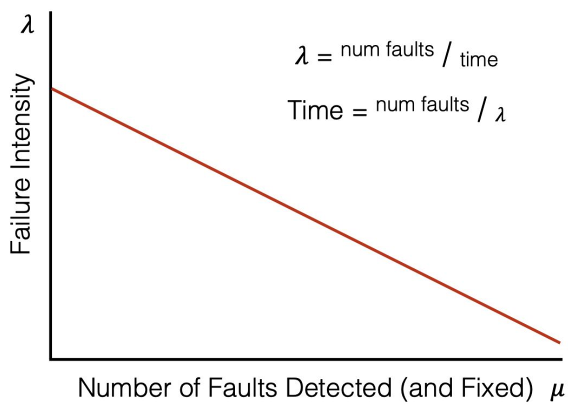
6. Calculate the Markov state of a model after a specified number of transitions.

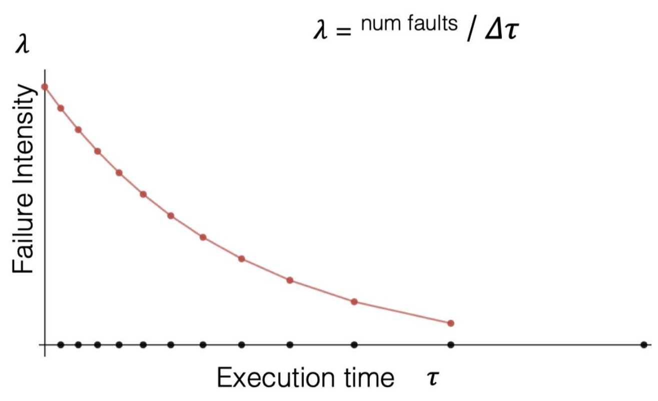
7. **Explain reliability growth**
   a. Reliability growth
      i. A collection of techniques for estimating reliability as a program is being developed and tested
   b. Failure rate
      i. The number of failures observed per unit time
   c. Measure reliability -> random testing
      i. Not aim to uncover faults but provide random samples of inputs to estimate reliability

8. **Reliability growth model**
   a. As development and testing continue the failure rates experienced should decrease.
   b. If the failure rates are decreasing then the reliability should be increasing.

9. **Critique the basic execution time model**
   a. Exponential decay
      i. Assumptions
         - Based on execution time
         - Program execution stops when a fault occurs and is restarted once the fault is fixed
         - Removal of each fault lowers the failure intensity by a constant amount
           - The model assumes that the achieved rate of failure reduction is constant
         - Uniform operational profile

- Representation
  - Failure intensity $\lambda = \text{num faults} / \text{time}$
  - Number of faults we have found μ
  - Expected time for next fault $= 1 / \lambda$





- Execution time τ
- Noisy in failure intensity
    - Average failure intensity
    - Group them together -> make the line more smoothly
- Time = 0, failure intensity = prediction from the equation
- Failure intensity = 0, the maximum number of faults you can find
    - The faults in the software (total faults the model predicts)
- Slope (a steeper gradient)
    - exponential decay(FI drop rapidly over time)
    - execution time(spent to reach target is less)
    - experience the estimated total number of failure in a shorter time
    - uncover the faults leads to a greater number of failure
        - Every fault removed gives rise to more failures
        - Every failure requires to remove more faults
        - The first data point skewed the regression line

10. Given some initial failure data, use the formula of the basic time execution model to provide reliability and failure estimates.

Failure intensity w.r.t faults: $\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\nu_0}\right)$

Failure intensity w.r.t time: $\lambda(\tau) = \lambda_0 \, e^{\left(-\frac{\lambda_0}{\nu_0}\tau\right)}$

Difference in failures: $\Delta\mu = \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F)$

Total failures w.r.t time: $\mu(\tau) = \nu_0 \left(1 - e^{\left(-\frac{\lambda_0}{\nu_0}\tau\right)}\right)$

Reliability: $R(\tau) = e^{-\lambda(\tau)}$

Difference in time: $\Delta\tau = \frac{\nu_0}{\lambda_0} \ln \frac{\lambda_P}{\lambda_F}$

11. Describe the process of error seeding for fault estimation.
    a. Definition
        i. It is an approach for estimating the total number of failures remaining in a program given a small seeded sample of errors.
    b. Goal
        i. Estimate N, the total number of faults in the program
    c. Steps
        i. Random seed the program with some number of faults E
        ii. Test the program to uncover the faults
            1. Find P seeded faults
            2. Find Q unseeded faults
        iii. Assumptions
            1. Both kinds of faults are equally to be found
        iv. P/E = Q/N
12. Motivate the use of the four models (reliability block diagrams, Markov models, basic execution time model, error seeding model) in this chapter in engineering software and systems.

Symbolic verification

1. **Symbolic execution**
   The key idea behind symbolic execution is to replace the concrete state with a symbolic state, in which the variables map to symbolic values, and to perform calculations on those symbolic value
2. **Definition**
   a. Constraint solver
      The question that can be answered
      - Satisfiable?
      - Solution?
      - Entail?
   - Constraint solving is an approach to generating solutions for mathematical problems expressed as constraints over a set of objects
   - Each constraint to be solved has three parts:
     - a set of variables;
     - a domain for each variable (e.g. the integers, characters, or real numbers);
     - the set of the constraints.

   b. Entailment
      - If constraint C entails constraint D, written C ⊢ D. Which means any solution for C also a valid solution for constraint D. Or, C is stronger than D.
3. **Strength & Limitation**
   a. Strength
      i. Good at small programs
      ii. Super effective, and can detect subtle faults.

   b. Limitation
      i. Path explosion
         - cover all possible paths in the program
         - If paths grow exponentially (lots of "if" conditions)

      ii. Constraint solving
         - Not work well in large program (too large constraints)
         - Loops
         - Cannot proof the upper bound

      iii. Unsolvable paths
         - Unsolvable constraints
           - Not-linear (real number)
         - External calls of library functions
           - Don't know how the library function behavior and how it can affect the current constraint states
4. **Dynamic symbolic execution**
   a. Definition
      i. Fuzzer-> coverage limitation -> internal knowledge
      ii. Follow the path taken by a concrete test input, collect the relevant constraints along the way
      iii. Use these constraints to generate new inputs that take new paths in the programs

        iv. DSE is a test input generation technique that is a cross between random testing and symbolic execution, with the aim of overcoming the weaknesses of each of these.
        v. Essentially, a DSE tool runs a **concrete input** on a program, and simultaneously **performs symbolic execution on the path** that this input executes
    b. From an initial state to symbolic
        i. Example: initial string: good   target string: bad!
        ii. If not feasible path -> throw away
    c. When use DSE
        i. Eg: (x==hash(s)) ------complicate expression
        ii. Symbolic execution cannot meet the internal constraints and may generate lots of paths

5. Characteristics of Dynamic symbolic execution

a. Steps
    i. Simultaneously execute both concretely and symbolically
    ii. Then negate(否定) path constraints obtained symbolically to generate new inputs to take different paths
b. Combines the advantages of both
    i. random testing
        - not so random, much better coverage
    ii. symbolic execution
        - using concrete value when encountering an unsolvable path
        - solve the path explosion problem
    iii. However, is best for finding bugs rather than for doing proofs
c. Disadvantages
    i. All of the SE limitations
    ii. Divergence(发散): when a new test we generate takes a different path to the one we expect
d. Sage : TOOL from Microsoft
    i. Trigger + patch vulnerabilities

6. Compare and contrast SE vs. DSE

a. The concrete input is executed simultaneously with the symbolic input, instead of generating the test after symbolically executing a path. The first input is chosen arbitrarily (perhaps randomly).
b. Path constraints are modified to generate a new test after each path is executed.