Table of Contents

[Week 1 Lecture 2 Chapter 1 An Introduction to Software Testing 2](#_Toc52630974)

[Week 2 Lecture 3 Chapter 2 Input partitioning 3](#_Toc52630975)

[Week 2 Lecture 4 Chapter 3 Boundary-Value Analysis 5](#_Toc52630976)

[Week 4 Lecture 5 Chapter 4 Coverage-Based Testing 7](#_Toc52630977)

[Week 5 Lecture 5 Chapter 5 Testing Modules 10](#_Toc52630978)

[Week 7 Lecture 7 Chapter 8 Security Testing 12](#_Toc52630979)

[Week 8 Lecture 8 Chapter 8 Security Testing 15](#_Toc52630980)

SWEN90006 Security and Software Testing

# Week 1 Lecture 2 Chapter 1 An Introduction to Software Testing

What is software testing and why do we do it?

* Show correctness
* Show that it tmeets requirements
* Increase confidence
* Find faults

What is a successful test case?

* One that fails
* Regression testing: To make sure we did not introduced new faults when we modify

Input domain vs. output domain

* Input domain: The set of values in the input type
* Output domain: The set of values in the output type

Fault vs. failure vs. error

* Fault: A fault is an incorrect step, process, or data definition in a computer program.
* Error: An incorrect internal state that is the result of some fault. An error may not result in a failure.
* Failure: A failure occurs when there is a deviation of the observed behaviour of a program, or a system, from its specification.
* Failures and errors are the result of faults
* In testing we can only ever detect failures

A test case consists of three essential pieces of information

* A set of test inputs, or if the program under test is non-terminating, a set of sequences of test inputs
* The expected results when the inputs are executed
* The execution conditions or execution environment in which the inputs are to be executed

Principles of software testing psychology

* Principle 1: A necessary part of a test case is a definition of the expected output or result.
* Principle 2: A programmer should avoid attempting to test his or her own program.
* Principle 8: The probability of the existence of more errors in a section of a program is proportional to the numbers already found in that section.

Black-box testing vs. white-box testing

* Black-box testing: Test cases are derived from the functional specification of the system
* White-box Testing: Test cases are derived from the internal design specifications or actual code for the program
  + Check all the paths that a program can execute

# Week 2 Lecture 3 Chapter 2 Input partitioning

2.2 Equivalence Classes

Equivalence class: An equivalence class is a set of values from the input domain that are as good as each other to find failures in a program. Relating

* Can be considered a description of test
* Defines a whole set of tests, only one of which we need to execute (not entirely)

Properties

* Coverage: The union of all equivalence classes = input domain
* Disjoint: A single test belongs to only one equivalence class

Valid input vs. invalid inputs

* A valid input to a program is an element of the input domain that is the value expected from the program specification.
* An invalid input is an input to a program is an element of the input domain that is not expected or an error value that as given by the program specification.

Input domain

* Valid input domain
* Invalid input
  + Testable invalid input domain (satisfies precondition)
  + Untestable invalid input domain (violates precondition)

2.4 Equivalence Partitioning

Guideline 1

* If an input condition specifies a range of values
  + One valid equivalence class in the range
  + Two invalid equivalence classes; one below the range and one above the range

Guideline 2

* If an input condition specifies a set of possible input values and each is handled differently
  + One valid equivalence class for elements in the set
  + One invalid equivalence class for the elements that are not in the set

Guideline 3

* If the input condition specifies the number (say N) of valid inputs
  + One valid equivalence class for that number
  + Two invalid equivalence classes, one for values < N and one for values > N

Guideline 4 (zero-one-many rule)

* If the input condition specifies that an input is a collection of items, and the collection can be of varying size
  + One valid equivalence class for a collection of size 0
  + One valid equivalence class for a collection of size 1
  + One valid equivalence class for a collection of size > 1

Guideline 5

* If an input condition specifies a “must be” situation
  + One valid equivalence class
  + One invalid equivalence class

Guideline 6

* If there is any reason to believe that elements in an equivalence class are handled in a different manner than each other by the program
  + Split the equivalence class into smaller equivalence classes.
  + (binary search, left side of list or right side of list)

2.5 Test Template Trees

When we partition each leaf node, we ensure that the partitioning is

* Disjoint: The partitions do not overlap
* Coverage: The partitions cover their parent, that is, if we combine the new partitions, that combination will be equivalent to their parent.

2.6 Combining Partitions

3 criteria

* All combinations: The all combinations criterion specifies that every combination of each equivalence class between blocks must be used.
* Each choice combinations: The each choice criterion specifies that just one test case must be chosen from each equivalence class.
* Pair-wise combinations: The pair-wise combinations criterion specifies that an equivalence class from each block must be paired with every other equivalence class from all other blocks.

Lecture Notes:

Testing binary search

* Target not in list
* Target in list
* Duplicates, list = [1,1,1,2], t = 1, expected output = 1 or 2
* Empty list
* Target out of range
* We won't test unsorted list, as "list must be sorted" is input condition

Test template trees

E.g. sorted list,

Coverage: Size = 0, size = 1, size > 1, add them up, we will get all sorted list

Disjoint: no list which is both size 0 or size 1

All combination: Apply guideline to all node

Each combination: Apply guideline to one node. E.g. we only test leap year on num of days in Feb

# Week 2 Lecture 4 Chapter 3 Boundary-Value Analysis

3.1 Chapter Introduction

Boundary shift vs. computation fault

* Computational fault: A computational fault that occurs during a computation in a program
* Boundary shift: A boundary shift is when a predicate in a branch statement is incorrect, effectively ‘shifting’ the boundary away from its intended place.

3.2 Values and Boundaries

Boundary test

* On point
  + An on point is a point on the boundary of an equivalence.
  + For a closed boundary, an on point will be a member of the equivalence class, and for an open boundary, it will not.
* Off point
  + An off point is a point just off the boundary of an equivalence class.
  + For a closed boundary, the off point will fall outside of the equivalence class, and for an open boundary, it will fall inside the equivalence class.

Lecture Notes:

How boundary shift is detected?

* When boundary shift happens, it should be in EC1, but it ends up in the EC2

Example 1 - one dimension

* x = 0, when boundary shifts to -1, it should return 0, but it actually returns 100/0
* x = 1
  + when boundary shifts to 1, it should return 100/x, but it actually returns 0
  + when boundary shifts to 2, it should return 100/x, but it actually returns 0

Example 2 - two dimensions

* Choose two on points that are very far away
* Choose one off point that is close to the boundary

Example 3 - three dimensions

* Choose three on points on the plain
* Choose one off point just above the plain

SWEN90006 Security and Software Testing

Week 3 Lecture 5 Chapter 4 Coverage-Based Testing

4.2 Control-Flow Testing

4.2.2 Coverage-Based Criteria

Coverage criteria

* Statement coverage (or node coverage)
  + Every statement of the program should be exercised at least once.
* Branch coverage (or decision coverage)
  + Every possible alternative in a branch (or decision) of the program should be exercised at least once.
  + E.g. if (a), a must evaluate to true and false
* Condition coverage
  + Each condition in a branch is made to evaluate to true and false at least once.
  + E.g. if(a&&b), a must evaluate to true and false at least once, and so must b.
* Decision/Condition coverage
  + Each condition in a branch is made to evaluate to both true and false and each branch is made to evaluate to both true and false.
  + E.g. if(a&&b), a must evaluate to true and false, b must evaluate to true and false, a&&b must evaluate to true and false.
* Multiple-condition coverage
  + All possible combinations of condition outcomes within each branch should be exercised at least once.
  + E.g. if(a&&b), a must evaluate all combinations of a and b: true true true, false true false, true false false, false false false
* Path coverage
  + Every execution path of the program should be exercised at least once.

4.2.3 Measuring Coverage

* Coverage score = Objectives met / Total objectives

# Week 4 Lecture 5 Chapter 4 Coverage-Based Testing

4.3 Data-Flow Testing

4.3.1 Static Data-Flow Analysis

A statement may act on a variable in 3 different ways

* Define a variable
  + A statement defines a variable by assigning a value to the variable
  + E.g. x = 5
* Reference a variable
  + A statement makes a reference to a variable either as a l-value or r-value
    - l-value: A variable into which values can be stored. E.g. x = 5
    - r-value: A variable that is referenced to get its value. E.g. y = 3 \* x
  + (return x is also reference a variable)
* Undefine a variable
  + A statement undefines a variable whenever the value of the variable becomes unknown.
  + E.g. the scope of a local variable ends.

Data-flow anomalies indicate the possibility of program faults

* u-r anomaly
  + An undefined variable is referenced
  + A variable is referenced without it having been assigned a value first
* d-u anomaly
  + A defined variable has not been referenced before it becomes undefined.
* d-d anomaly
  + The same variable is defined twice causing a hole in the scope of the first definition of the variable.

4.3.2 Dynamic Data-Flow Analysis with Testing

Coverage-based criteria

* All-Defs
  + Require some definition-clear sub-path from all definitions of a variable to a single use of that variable.
  + Test at least one path from d1(x)
* All-Uses
  + Requires some definition-clear sub-path from all definitions of a variable to all uses reached by that definition.
  + Test d1(x) to each use and its successor nodes.
* All-Du-Paths (All-Definition-Use-Paths)
  + Requires that a test set traverse all definition-clear sub-paths that are cycle-free or simple-cycles from all definitions to all uses reached by that definition, and every successor node of that use.
  + Test d1(x) to each use through each path.

4.4 Mutation Analysis

Mutant

* Given a program, a mutant of that program is a copy of the program, but with one slight syntactic change.

Coupling effect

* 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.
* Complex faults in programs are coupled with simple faults.

Systematic mutation analysis via mutant operators

A mutant operator is a transformation rule that, given a program, generates a mutant for that program.

* Arithmetic Operator Replacement: Replace each occurrence of an arithmetic operator +, −, \*, /, \*\*, and % with each of the other operators, and also replace this with the left operand and right operand (for example, replace x + y with x and with y).
* Conditional Operator Replacement: Replace each occurrence of a logical operator &&, ||, &, |, and ˆ with each of the other operators, and also replace this entire expression with true, false, the left operand, and the right operand.
* Shift Operator Replacement: Replace each occurrence of the shift operators <<, >>, and >>> with each of the other operators, and also replace the entire expression with the left operand.
* Logical Operator Replacement: Replace each occurrence of a bitwise logical operator &, |, and ˆ with each of the other operators, and also replace the entire expression with the left and right operands.
* Assignment Operator Replacement: Replace each occurrence of the assignment operators +=, −=, \*=, /=, %=, &=, |=, ˆ=, <<=, >>=, and >>>= with each of the other operators.
* Unary Operator Insertion: Insert each unary operator, +, −, !, and ∼ before each expression of the correct type. For example, replace if(x = 5) with if(!(x = 5)).
* Unary Operator Deletion: Delete each occurrence of a unary operator.
* Scalar Variable Replacement: Replace each reference to a variable in a program by every other variable of the same type that is in the same scope.
* Bomb Statement Replacement: Replace every statement with a call to a special Bomb() function, which throws an exception. This is to enforce statement coverage, and in fact, only one call to Bomb() is required in every program block.

Equivalent mutants

* Given a program and a mutation of that program, the mutant is said to be an equivalent mutant if, for every input, the program and the mutant produce the same output.
* An equivalent mutant cannot be killed by any test case.
* (Example of equivalent mutant: return mid -> return mid --)

# Week 5 Lecture 5 Chapter 5 Testing Modules

5.3 Testability of State-Based Programs

Testability

* Testability of a program is defined by the observability of the program, and the controllability of the program.

5.4 Unit Testing and Finite State Automata

Finite state automaton (FSA)

* A model of behaviour consisting of a finite set of states with actions that move the automata from one state to another.

Constructing a FSA

* Step 1: Identify the states of the FSA.
* Step 2: Identify which operations are enabled in a state and which operations are not enabled in a state.
* Step 3: Identify the source and target states of every operation in the model.

5.4.1 Deriving Test Cases from a FSA

Path

* A path from a state s0 to a state sn in the automaton is a sequence of transitions t1t2 . . . tn such that the source of t1 is s0 and the target of tn is sn.

Traversal criteria

* State coverage
  + Each state in the FSA must be reached by the traversal.
* Transition coverage
  + Each transition in the FSA must be traversed.
  + This subsumes state coverage.
* Path coverage
  + Each path in the FSA must be traversed.
  + This subsumes state coverage, but is impossible to achieve for a FSA with cyclic paths.

Intrusively testing the state transition diagram

* Adding testing code to the module to measure and monitor the internal states of the module as it is tested.
* The problem with this kind of testing is that it breaks encapsulation.

Non-intrusive testing the state transition diagram

* Test a module without adding code to a module

Classification of the operations of a module

* Initialisers
  + The operations that initialise the module.
  + E.g. Stack constructor.
* Transformers
  + Operations to change the state of the module.
  + E.g. pop operation in the stack example.
* Observers
  + Operations to observe the module state.
  + E.g. isEmpty and top operations in the stack example.

5.5 Testing Object-Oriented Programs

5.5.1 Object-Oriented Programming Languages

Key elements

* Classes and objects which provide the main units for structuring.
* Inheritance, which provides the one of the key structuring mechanisms for object-oriented design and implementation.
* Dynamic Binding or Polymorphism, which provides a way of choosing which object to use at run-time.

Building and testing inheritance hierarchies

Incremental inheritance-based testing

* Step 1: Test each base class by
  + (1) Testing each method using a suitable test case selection technique
  + (2) Testing interactions among methods by using the techniques discussed earlier in this chapter.
* Step 2: Consider all sub-classes that inherit or use (via composition or association) only those classes that have already been tested.
  + A child inherits the parent’s test suite which is used as a basis for test planning.
  + We only develop new test cases for those entities that are directly, or indirectly, changed.

Implication of polymorphism

* In the case of object-oriented programming, each possible binding of a polymorphic class requires a separate set of test cases.

# Week 7 Lecture 7 Chapter 8 Security Testing

8.1 Chapter Introduction

Penetration testing (Pentesting)

* Penetration testing is the process of attacking a piece of software with the purpose of finding security vulnerabilities.

8.2 Security Vulnerabilities

* A vulnerability is a flaw in a program that allows its security, or that of a larger system, to be subverted.
  + Integrity is violated when an attacker is able to modify data that they are not allowed to modify.
  + Confidentiality is violated when an attacker is able to read (or learn) data or information that is supposed to be kept secret.
  + Availability is violated when an attacker is able to prevent a system from running, or prevent other users from using the system. A denial of service (DoS) attack is a good example of one that violates availability.

8.3 Introduction to Penetration Testing

SQL vulnerability

Stack buffer overflow

Fuzz testing (Fuzzing)

* Fuzz testing is a (semi-)automated approach for penetration testing that involves the randomisation of input data to locate vulnerabilities.
* A fuzz testing tool (or fuzzer) generates many test inputs and monitors the program behaviour on these inputs, looking for things such as exceptions, segmentation faults, and memory leaks; rather than testing for functional correctness.

Three basic techniques for fuzzing

* Random testing: Tests generated randomly from a specified distribution.
* Mutation-based fuzzing: Starting with a well-formed input and randomly modifying parts of that input
* Generation-based fuzzing: Using some specification of the input data, such as a grammar of the input.

8.4 Random Testing

Random testing

* As with random testing for reliability, tests are chosen according to some probability distribution (possible uniform) to permit a large amount of inputs to be generated in an fast and unbiased way.

Advantages

* It is often cheap and easy to generate random tests, and cheap to run many such tests automatically.
* It is unbiased, unlike tests selected by humans. This is useful for pentesting because the cases that are missed during programming are often due to lack of human understanding, and random testing may search these out.

Disadvantages

* A prohibitively large number of test inputs may need to be generated in order to be confident that the input domain has been adequately covered.
* The distribution of random inputs simply misses the program faults
* It is highly unlikely to achieve good coverage.

8.5 Mutation-Based Fuzzing

Mutation-based fuzzing

* Mutation-based fuzzing is a simple process that takes valid test inputs, and mutates small parts of the input, generating (possibly invalid) test inputs.
* The mutation (not to be confused with mutation analysis discussed in can be either random or based on some heuristics.

Advantages

* It generally achieves higher code coverage than random testing. While issues such as the checksum issue discussed earlier still occur, they often occur less of the time if the valid inputs that are mutated have the correct values to get passed these tricky branches. Even though the mutated tests may change these, some will change different parts of the input, and the e.g., checksum will still be valid.

Disadvantages

* The success is highly dependent on the valid inputs that are mutated.
* It still suffers from low code coverage due to unlikely cases (but not to the extent of random testing).

8.6 Generation-Based Fuzzing

Generation-based fuzzing

* Generation-based fuzzers generate their own input from such existing models, rather than mutating existing input.
* Generation-based fuzzers have some information about the format of the input that is required, for example, a grammar of the input language (e.g. SQL grammar), knowledge of the file format. Using this knowledge, a generated-based fuzzer can create inputs that preserve the structure of the input, but it can randomly or heuristically modify parts of the input based on that knowledge.
* Therefore, instead of randomly modifying parts of a string representing an SQL query, it can produce syntactically-correct SQL, but with random data within that structure.

Advantages

* Knowledge the input protocol means that valid sequences of inputs can be generated that explore parts of the program, thus generally giving higher coverage.

Disadvantages

* Compared to random testing and mutation fuzzing, it requires some knowledge about the input protocol.
* The setup time is generally much higher, due to the requirement of knowing the input protocol; although in some cases, the grammar may already be known (e.g. XML, RFC).

8.7 Coverage-Guided Fuzzing

8.8 Memory Sanitisers

Memory sanitiser (memory debugger)

* A memory sanitiser is a tool for finding memory leaks and buffer overflows.

Memory sanitisers detect memory errors, which include

* Memory is accessed out of bounds (e.g. buffer overflows)
* Invalid pointers are dereferenced (e.g. NULL pointer dereference errors)
* Dynamically allocated memory is freed (deallocated) more than once (e.g. double-free vulnerabilities)
* Dynamically allocated memory is used after it was decallocated (e.g. use-after-free vulnerabilities).

8.9 Undefined Behavior

Undefined behavior

* Something that a program does that causes its future behaviour to be unknown.
* (If there is undefined behaviour, compiler can do whatever it wants, does not guarantee all machine will behave the same)

Undefined behaviour in C include

* Dividing by 0
* Dereferencing a NULL pointer
* Overflowing a signed integer
* Underflowing a signed

# Week 8 Lecture 8 Chapter 8 Security Testing

8.10 Confidentiality

8.10.1 Overt Channels

Overt channel

* An overt channel is a mechanism (whether intended or not) for directly transferring data in a system.
* Overt channels exist when necessary access checks are missing.
* To test for the presence of overt channels, one can therefore test whether the program performs the necessary checks.

8.10.2 Covert Channels

Covert storage channel

* This program leaks in the publicly-observable variable public the parity of the secret variable secret.
* This program is an example of a covert storage channel, since secret information is indirectly stored in a public place.

Covert timing channel

* The time at which the publicly observable output “Finished!” becomes available depends on secret information, and so leaks that secret information to an attacker who can measure the passage of time.

Testing for covert channels

* Observe how a program’s public outputs or timing behaviour changes in response to changes to the secret data.