CMPT 473: Software Quality Assurance

A Course Overview

Jeffrey Leung
Simon Fraser University

Spring 2020

Contents

1 Introduction

1.1 Software Quality

- Quality of development process influences quality of resulting software
- Perspectives/roles of the software:
  - Priorities of *end users*:
    * Fulfill its desired purposes
    * Produce reliable results upon consistent input
    * Handles bad input
    * Easy to use
    * Responsive
    * Integrates well with other software
  - Priorities of *operations/deployment*:
    * Secure from attacks
    * Uses appropriate amount of resources
  - Priorities of *developers*:
    * Easily modifiable
    * Comprehensible
    * Has measurable quality
    * Adaptible to other systems
- **ISO/IEC 9126:** Functionality, reliability, usability, efficiency, portability, maintainability
  - **Reliability:** Characteristic of software which, in the context of software faults, involves avoidance, maintenance of performance during, and re-establishment of performance/data afterwards
  - **Usability:** Characteristic of software which is understandable in the context of how it fits the users' needs, easy to learn/operate, and enjoyable
  - **Maintainability:** Characteristic of software which makes defects easy to identify, allows changes unlikely to affect other components, and easy to test
- Defect terminology:
  - **Defect/fault:** Flaw in static software/code
    * **Latent defect:** Unobserved defect in delivered software which was not exposed by testing
  - **Failure:** Observable behaviour which does not match expectations
  - **Error/infection:** Not-yet-observed incorrect state

1.2 Quality Measurement

- **Planning:** Choosing the most important assessment criteria
- Tools:
  - **Synthetic tools:** Quality measurement tools/techniques to create better software
    * E.g. Design methodologies, coding standards, templates, compilers

- **Analytical tools:** Quality measurement tools/techniques to evaluate software quality
    * E.g. Walk-throughs, audits, unit/integration/system testing
- **Manual tools:** Quality measurement tools/techniques which is interactively driven
    * E.g. Design methodologies
- **Automated tools:** Quality measurement tools/techniques which requires no interaction
    * E.g. Compilers, program generators

- Testing is difficult because of dependencies
- Use polymorphism to create a mock to isolate modules during testing
  - Inheritance can be used
  - **Parametric polymorphism:** Applying superclasses or templates of parameters to allow generics for testing
  - **Dependency injection:** Using dependencies by accepting them as arguments upon construction rather than instantiating them directly
- **Test frame:** Plan for a set of test cases based on partitioned inputs
- Coverage effectiveness:
  - For statement coverage, having quantitatively more coverage is not necessarily more effective
  - For mutation testing, test frame size correlates with defect-finding ability

2 Debugging

- **Debugging:** Application of the scientific method to find and eliminate an incorrect behaviour
- Steps:
  - Ignore assumptions
    - Mental model of software may be incorrect
    - Comments may be incorrect
  - Reproduce the behaviour
  - Brainstorm possible reasons why the incorrect behaviour occurred
  - Choose the most testable and likely hypotheses
- Debugging framework features:
  - Breakpoints (conditional)
  - Stepping through/over code
  - State:
    - Print/display
    - Modification
    - Watchpoints
  - Call functions

2.1 Bug Reporting

- Perspectives:
  - Developer: How a bug should be handled
  - Client/teammate: How a bug should be reported/fixed
- Error messages should contain:
  - What is incorrect
  - Where the error occurred
  - When the error occurred
- Good error messages allow you to:
  - Reproduce a failure
  - Find the original creator
  - Combine duplicate error reports
  - Identify causes
  - Prioritize
  - Identify workarounds
  - Create an accurate fix
- Prioritize bugs by:
  - Frequency

  - Risk level or consequence
  - Recency of introduction
- Bug reports should contain:
  - Summary
  - What happened, when, where
  - Expected result
  - Steps to reproduce
  - Product, version, feature
  - Platform and environment
  - Severity/priority
  - Owner(s)
  - Duplicate(s)

3 Types of Testing

- Test cases:
  - Require an input and expected output/state/behaviour
  - **Oracle:** Evaluation of the output/behaviour of a test
  - **Mock:** Entity which is used to measure or examine behaviour
  - **Stub:** Fake entity which is used during testing to replace a component
- If external state is uncontrolled, tests will be nondeterministic
  - Factors causing lack of control:
    * Lack of isolation
    * Asynchronous behavior
    * Remote services
    * Time
    * Resource leaks
- **Coverage/adequacy:** Measurement of how well a test suite addresses quality criteria
- **Test Driven Development (TDD)**: Software testing where unit tests are created first and used to drive development
- **Unit testing**: Software testing of the smallest possible components
  - Principles include component isolation, simplicity, ease of understanding
- **Integration testing**: Software testing based on the connection of multiple components
- **Acceptance testing**: Software testing based on acceptance criteria
- **Black-box testing:** Software testing based on the external input specification of a system
  - Involves input space partitioning
- **White-box testing:** Software testing based on the internal program structure of a system
  - Involves graph coverage
- **Fuzz testing:** Method of exploratory software testing which inputs randomly mutated data into a program to evaluate random inputs
- Test scenarios can be concrete (e.g. $x = 5$) or abstract (e.g. for all x, $x > 0$)
  - Abstract test cases can generate a test and check the oracle, using:
    * Testing with randomly generated values
    * Symbolic execution
- Testing strategies which evaluate the existing test suite for effectiveness:
  - MC/DC
  - Mutation testing

4 Property-Based Testing

- **Property-based testing:** Testing which generates tests to evaluate functional properties/requirements
  - Mathematical representation of an expectation
- Common test strategies:
  - **Symmetry:** Test strategy where operations are performed to return to the original value
  - **Alternative:** Test strategy where a value is compared with a value generated from alternative solutions
  - **Induction:** Test strategy
  - **Idempotence:** Test strategy where performing an operation more than once has no effect
  - **Invariant:** Test strategy where a property of a preprocessed value must be equal to the property of a processed value

5 A/B Testing

- **A/B testing:** Hypothesis testing which provides different services to randomly selected individuals
  - Requires a hypothesis and population to test
- Used to evaluate:
  - Usability improvements
  - Perforamnce improvements
  - Promotion effectiveness
  - Gradual rollouts
- Possible issues:
  - Uncontrolled influencing factors
  - Populations may not be representative
  - False positives/negatives
    - **p-hacking:** Altering results by executing many tests to compound the effect of false positives/negatives, and choosing exactly when to stop based on the results
    - **Regression to the mean:** Tendency for results to return to relatively normal levels after an extreme event
      - E.g. Poorly performing students are placed in a program, after which their grades improve
- Ways to mitigate issues:
  - Calculate significance and test amount beforehand, rather than stopping when significance is reached

5.1 Hypothesis Testing

- T-Test: Comparison between samples of populations
  - Modeled as a distribution
  - Requires the data to have a known variance, independence from other factors
- Sequential testing may have bounding criteria for when to stop early
- **Multi-armed bandit:** Testing technique which determines the best of multiple options based on evidence so far
  - Requirements:
    - Reward probabilities do not change
    - Sampling is singular, instantaneous, and independent
  - ϵ**-greedy strategy:** Multi-armed bandit technique where the greater the previous sample proportion, the more likely the population is sampled
    - Sensitive to variance
  - **Thompson sampling:** Multi-armed bandit technique where the probability of the best arm is chosen

6 Input Space Partitioning

- **Input Space Partitioning:** Division of potential inputs into classes where each input in a class should yield identical output
- **Input Domain Model:** Description of possible test inputs through discrete partitions which are disjoint and cover the entire domain
  - **Interface-based approach:** Choosing inputs for a domain model based on parameters and domains
  - **Functionality/requirements-based approach:** Choosing inputs for a domain model based on behaviours or functionality in the specification
- Process:
  - Identify and isolate the component under test
  - Identify inputs
    - Possible values to be partitioned:
    - Parameters and inputs
      - Object state
      - Global state
      - File contents
  - Identify characteristics of each input to divide into possible values
  - Identify constraints
    - Characteristics to consider:
      - Preconditions and postconditions
      - Relationships to special values
      - Relationships between variables
  - Select representative values from an input block, including:
      - Expected/valid values
      - Special values
      - Invalid values
      - Boundary values
- E.g. Given a command to FIND instances of a PATTERN in a FILE:
  - The component is the FIND command
  - The parameters are the pattern to search for, the filename, and the file contents
  - The characteristics include:
    - Is the pattern empty?
    - Is the length of the pattern contents less than, the same as, or greater than the length of the longest line in the file?
    - Does the pattern have quotation marks enclosing it?
    - Are ther escaped quotation marks in the pattern?

    - Is the filename empty?
    - Does the file exist?
    - Is the file a directory?
    - Is the file blank?
    - Does the file have a blank line?
    - Is there a line in the file matching the pattern multiple times?

6.1 Test Combination Strategies

- * means any value is valid
- **Each Choice:** From each block, use at least one value in at least one test
  - Size: Largest domain
  - Does not cover many possible conflicting states
  - E.g. Given inputs A/B/C and 1/2, an adequate set of tests are:
    - A 1
    - B 2
    - C *
- **Pair Wise:** From each block, choose 1 value and test it at least once with every value from every other block
  - Size (lower bound): $\geq$ the product of the domain sizes of the two largest partitions
  - E.g. Given inputs A/B/C, 1/2, and X/Y, an adequate set of tests are:
    - A 1 X
    - A 2 Y
    - B 1 Y
    - B 2 X
    - C 1 *
    - C 2 *
- **T-Wise:** From each block, test 1 value for each group of T characteristics
  - Size: $\geq$ product of the T largest domain partitionings
- **Base Choice:** Create a base test and create tests by changing only a single value and fixing the others
  - Size: 1 base test plus one for each other unselected block
  - Base case must be a valid positive test
- Hierarchy of test type satisfaction:
  - All combinations includes all T-Wise tests and Multiple Base Choice testing
    - T-Wise testing includes all Pair-Wise tests
      - Pair-Wise tests includes all Each Choice tests
    - Multiple Base Choice testing includes all Base Choice tests
      - Base Choice tests includes all Each Choice tests

7 Graph Coverage

- **Control flow graph (CFG):** Graph where nodes represent code and edges represent paths taken
  - Types of nodes: Entry, decision/branch, join, exit
  - For a while loop, see figure 1
  - For a for loop, see figure 2
  - For a switch statement, see figure 3
  - For a short-circuited if statement, see figure 4

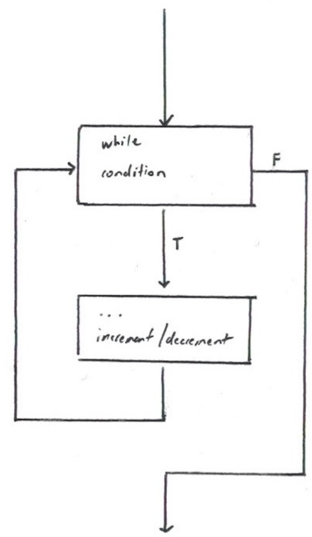

Figure 1: Control Flow Graph of a while loop

- Edge coverage (i.e. all branches used) is a superset of node coverage
- **Complete path coverage:** Coverage of all possible paths through the graph
  - Reason: Permutations/combinations of multiple possible paths can affect results
  - Infeasible/intractible because it would entail combinatorial explosion and inability to efficiently test looping paths
- **Edge pair coverage:** Coverage which includes every path of length $<= 2$
- **Specified path coverage:** Coverage which tests k paths for a given k
- **Reachability**: Property of a piece of code which may or may not be executable based on states
  - **Syntactic reachability:** Analysis of reachability based on the structure of the code
  - **Semantic reachability:** Analysis of reachability based on the meaning of the code (cannot be checked by an automated program)

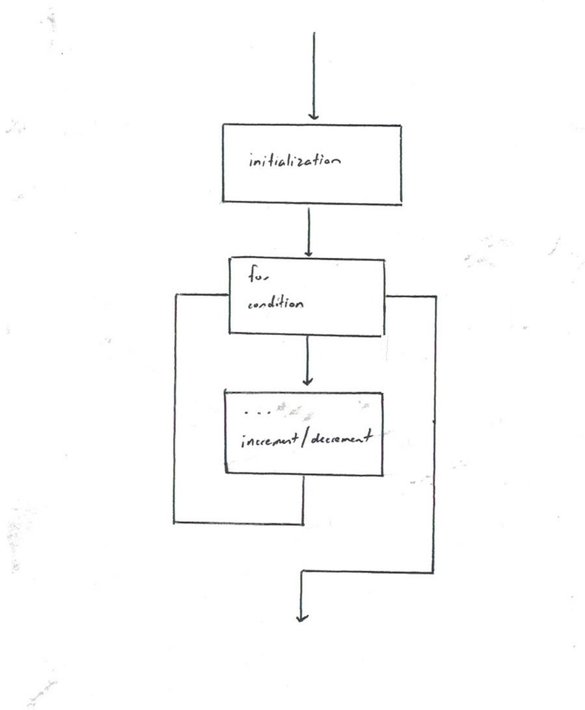

Figure 2: Control Flow Graph of a for loop

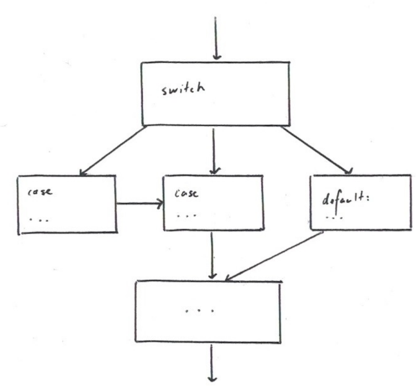

Figure 3: Control Flow Graph of a switch statement

- Testing loops is relevant when each iteration may affect the next
  - **Simple path:** Acyclic path between nodes where no node appears more than once (except first/last)
  - **Prime path:** Simple path which is not a subpath of any other simple path
    - I.e. A simple path which cannot be extended

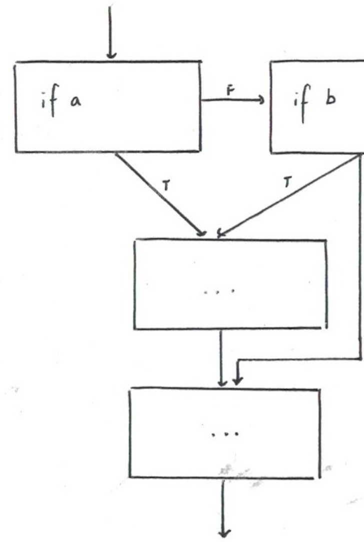

Figure 4: Control Flow Graph of a if statement short-circuited

    * E.g. Simple path which starts and ends at the same node

  - **Tour:** Path which is a subpath of another

    * Mathematical definition: A path p tours path q if q is a subpath of p

    * Tour with sidetrips: Tour where every edge of the superpath appears in the same order in the subpath

    * Tour with detours: Tour where every node of the superpath appears in the same order in the subpath

- **def-use pair:** Definition statement and the next relevant uses of the variable (before reassignment)

  - Used to test along data flow

  - Possible test approaches:

    * All defs coverage: Every def must be covered by at least one test of its uses

    * All uses coverage: Every use must be tested with least one definition

    * All def-use pairs coverage

    * All def-use paths coverage: All simple paths between def-use pairs are covered

- Path/branch coverage is insufficient due to scalability and complex conditions (e.g. non-short-circuiting) which do not have the notion of a path

8 Logic-Based Coverage

- **Predicate coverage:** Each boolean expression must be tested as true and false in at least one test each
- **Clause coverage:** Each clause must be tested as true and false in at least one test
- **Combinatorial/Multiple Condition coverage:** Each possible combination of clauses must be tested
- Clause determines the outcome of a predicate if changing only the value of that clause changes the outcome of the predicate
- **Modified Condition/Decision Coverage (MCDC):** Coverage demonstrating that each entry/exit is used, each decision can take every possible outcome, each clause can take every possible outcome, and each clause independently can impact the outcome
  - Based on the behaviour how one clause affects the entire expression
  - Ensures that each clause has an impact
  - Not effective to generate a test suite as the drive to create minimal tests interferes with MC/DC
  - Used to check test suites generated using other strategies
- Determining the impact of a predicate
  - Process for a given predicate a:
    * Create two clauses, one which replaces a with $\#T$ (true) and the other which replaces a with $\#F$ (false)
    * Set these two clauses as not equal to each other
    * Solve the equation
    * If the equation is not equal, then the predicate has impact
  - Example: Given $(a \wedge b) \vee (a \wedge \neg b)$, prove whether a has impact or not.
    * Let $a = T$ for one version of the clause, and $a = F$ for another version of the clause.

$$(T \wedge b) \vee (T \wedge \neg b) \stackrel{?}{=} (F \wedge b) \vee (F \wedge \neg b)$$
$$b \vee \neg b \stackrel{?}{=} F \vee F$$
$$T \stackrel{?}{=} F$$

    * The expression evalutes to $T \neq F$. Therefore a has impact.
- Process of creating a minimal test suite using MC/DC:
  - Change all compared expressions into clauses, each represented by one predicate
  - Create a minimal set of logical assignments where, for each predicate, there are at least two assignments where:
    * The values of the predicate and result both differ, and
    * The values of all other predicates match
  - Create a test suite with the original inputs set to specific values to satisfy the predicate values
  - Example: Given $a \vee (b \wedge c)$, generate a minimal set of tests to demonstrate MCDC coverage.
    * See figure 5.

* Test entries 1 and 2 show the impact of a.
* Test entries 2 and 3 show the impact of b.
* Test entries 3 and 4 show the impact of c.

a	b	c	Result
T	F	T	T
F	F	T	F
F	T	T	T
F	T	F	F

Figure 5: MCDC Example Test Suite

– Example: Given $(a \wedge b \wedge c) \vee (d \wedge a)$, generate a minimal set of tests to demonstrate MCDC coverage.

* See figure 6.
* Test entries 1 and 2 show the impact of a.
* Test entries 2 and 3 show the impact of d.
* Test entries 3 and 4 show the impact of c.
* Test entries 4 and 5 show the impact of b.

a	b	c	d	Result
F	T	F	T	F
T	T	F	T	T
T	T	F	F	F
T	T	T	F	T
T	F	T	F	F

Figure 6: MCDC Example Test Suite

9 Mutation Analysis and Testing

9.1 Mutant Fundamentals

- **Mutant:** Valid program which behaves differently from the original
  - Involves smallest possible changes
  - Invalid (and not counted in the mutation score) if:
    - Not compilable (still born)
    - Killed by most test cases (trivial)
    - Equivalent to the original program or to other mutants (redundant; can be undecidable)
  - A valid mutant must satisfy the reachability, infection, and propagation model:
    - **Reachability:** Ability of the fault to be executed by a test
    - **Infection:** Ability of a fault to cause the program state to differ
    - **Propagation:** Ability of the differing program state to affect the output
- **Mutation operator:** Systematic change applied to produce a mutant
  - **Intraprocedural mutations:** Modifying the internal values or operators of a procedure
    - E.g. (Optionally negated) absolute value insertion, arithmetic/relational/conditional operator replacement
  - **Interprocedural mutations:** Modifying the inputs of a procedure
    - E.g. Parameter values, call target, incoming dependencies
- **Kill:** Characteristic of a test which produces a different outcome on a mutant than the original program
  - Formal definition: A test t kills a mutant m if t produces a different outcome on m than the original program
  - **Weakly kill:** A mutant test which results in different internal state
    - Satisfies reachability and infection, but not propagation
  - **Strongly kill:** A mutant test which results in different output
    - Satisfies reachability, infection, and propagation
- Difficulties:
  - Managing and executing a large amount of mutants
  - Identifying identical mutants

9.2 Fault Seeding

- **Fault seeding:** Inserting expected faults to be killed
  - Equation:
$$\frac{\text{\# of mutants which killed a bug}}{\text{\# of mutants}}$$
  - Issues:
    - Faults may not be meaningful
    - May forget to remove the faults

9.3 Mutation Testing

- **Mutation analysis:** Ability to find bugs using a mutant
- **Mutation testing:** Process of creating a test suite which covers a representative set of mutants
  - Given an unkilled mutant, improve the test suite by adding a test which kills it
  - **Representative set:** Set of mutants which covers all possible faults
- **Mutation score:** Quantitative score of mutation analysis effectiveness
  - Invalid mutants are not counted
  - Equation:

$$\frac{\text{\# of non-duplicated mutants which kill a bug}}{\text{\# of non-equivalent, non-duplicated mutants}}$$

- Manage scalability by:
  - Filter based on coverage
  - Short circuit tests
  - Testing multiple mutants simultaneously
- Test coverage:
  - **Weak mutation coverage**: For each mutant, the test suite contains a test which weakly kills the mutant
  - **Strong mutation coverage**: For each mutant, the test suite contains a test which strongly kills the mutant

10 Regression Testing

- **Regression testing:** Method of testing which ensures previous functionality is preserved, supporting change
- Unexpected behaviour can be caused by:
  - New environments
  - Modifying other components
- Regression test suite is a subset of the test suite
- Components are tests for:
  - Previously fixed bugs
  - Units
  - System
- Upon a failing test, one or more of the following should occur:
  - Fix the software bug
  - Fix stale test inputs
  - Change expected behaviour

10.1 Managing Test Suite Size

- Limit regression test suite size by preventing redundant tests (e.g. not useful behaviour, not covered by adequacy criteria)
- Choosing a subset of tests:
  - Conservative approach: Run all tests
  - Cheap approach: Run tests which have requirements relating to the modified lines
  - Middle ground approach: Run tests affected by how changes propagate by software
    * **Change impact analysis:** Identification of how a change affects other components

11 Program Analysis

- **Program analysis:** Tools and techniques which automatically analyze software behaviour
- **Dynamic analysis:** Analysis about a single instance of program execution
  - Can be computationally expensive
  - Does not examine all possible executions
- **Static analysis:** Analysis on source code about all possible executions
  - Undecidability prevents some analyses
  - **Abstract interpretation:** Static analysis method which simulates different execution paths
- False positives/negatives may occur
- Examples:
  - Valgrind: Dynamic binary instrumentation tool to check for memory leaks
    * Only works on executables which provide both stack and heap allocated memory
  - Clang sanitizers: Compile-time instrumentation tools to analyze safety of usage

12 Test Planning

- **Test plan:** Documentation of testing goals, concerns, methodology, metrics
  - Guides testing process
- **Attribute-component-capability (ACC) testing:** Analysis of how testing addresses user-focused importance of components
  - Test requirements and case count are sorted into their corresponding cell
  - **Attribute:** High-level nonfunctional property to ensure (e.g. fast, secure)
  - Component: Entity or grouping of software
  - **Capability:** A characteristic of the system which supports a component having a particular attribute (e.g. for a database being secure, passwords should not be stored in plaintext)

13 Automated Test Generation

- **Automated test generation:** Executing program analysis to automatically derive tests
- **Fuzz testing:** Automated test generation method which creates sample program inputs
  - **Generational (model-based) fuzz testing:** Fuzz testing method which creates inputs based on a predefined model
    - Inputs will be valid; cannot test invalid inputs
  - **Mutational (heuristic change based) fuzz testing:** Fuzz testing method which creates inputs based modifying a test suite
    - Given a corpus of inputs, evolve new inputs; if the input tests a new area of the program, add it to the corpus
      - Criteria can be different lines of code, more/less memory, etc.
    - Inputs may not necessarily be valid
- **Symbolic execution:** Automated test generation method which replaces program inputs with symbolic values and calculates inputs based on constraints
  - **Concolic (dynamic symbolic) traversal:** Symbolic execution where values are maintained as symbolic, then calculated at the end to reveal every possible path
- **Execution generated testing:** Symbolic execution where some values are set to be concrete
- **Execution tree:** Graph of the possible paths taken by a program

14 Performance

- Performance areas include:
  - Speed/runtime
  - Resource management
  - Throughput
  - Responsiveness
- Analyzed differently depending on component granularity (e.g. system-level, instruction-level)
- Strategies of measuring performance:
  - Identify area of interest
- Evaluating results:
  - Be aware of:
    * Warm-up time
    * Caching
  - Measure and compare across changes
  - Run many executions and take the average
- Measurement of results:
  - **Arithmetic mean:** Average of measurements which measure the same value
    * Equation:

$$\frac{\sum_{i=1}^{N} r_i}{N} \tag{1}$$

  - **Harmonic mean:** Average of measurements which report rates (e.g. throughput for multiple tasks)
    * Represents the constant rate required for the same amount of time
    * Calculated by dividing the total number of rates by the rate per unit (inversion of the rate)
    * Equation:

$$\frac{N}{\sum_{i=1}^{N} \frac{1}{r_i}} \tag{2}$$

  - **Geometric mean:** Average of measurements which represent different values
    * A change in any benchmark affects the final value proportionally
    * Represents a multiplied score of performance
    * Equation:

$$\sqrt[N]{\prod_{i=1}^{N} r_i} \tag{3}$$

  - **Standard deviation:** Measure of confidence in the mean
    * Large values imply needing more samples or correction of methodological error

15 Security

- **Security:** Maintainance of desired properties against the presence of adversaries
- **CIA model:** Model of classic security properties
  - **Confidentiality:** Security property where information is only available to those authorized to access it
    - E.g. Information leaks violate confidentiality of information
  - **Integrity:** Security property where information can only be modified by authorized entities in permitted ways
    - E.g. Data corruption removes data integrity
  - **Access:** Security property where those authorized for access are not prevented
    - E.g. Denial of service attacks remove access from legitimate users
- Inability to test all points creates an attack surface
- MITRE's categories of security vulnerabilities: Insecure interaction, risky resource management, porous defenses
- Buffer overflows can overwrite other code in the stack
  - **Stack canary:** Indicator of compromised stack memory which exists between the return address and frame pointer, and aborts the program if it is overwritten
  - **Data Execution Prevention:** Technique which only allows execution of code from an allowed area
- **Return to libc attack:** Attack which replaces critical code which must be executed
  - E.g. Replacement of a return address with a pointer to a new, compromised function
  - **Return-oriented programming:** Manipulation of function pointers and stack memory to execute various components of existing functions
- **Address Space Layout Randomization (ASLR):** Randomized placement of function and stack data to prevent data manipulation and execution redirection
- **Control flow integrity:** Technique which restricts program execution to only allowed areas
- Memory safety vulnerabilities:
  - Potential causes:
    - Out-of-bounds pointers
    - Dangling pointers
  - Use tools/abstractions which avoid thse issues
- SQL injections
- CIA can be violated by inferring information
  - **Side channel attack:** Attach which infers system information based on information details
  - Leaks from logs, output, timing, power, sound, light, etc.
  - Difference in behaviour/cache retrieval upon sensitive information can create difference in timing
- **Access control policies:** Rules which specify who can access certain information

– **Discretionary access control:** Access control policies where the owner determines access within their own domain

– **Mandatory access control:** Access control policies where the operating system determines access to resources