The University of Melbourne
SWEN90006: Software Testing and Reliability
Learning Outcomes

This document outlines the learning outcomes for SWEN90006. By the end of this subject, a student should be able to do the following (categorised by chapters in the course notes):

Introduction to Software Testing

- Discuss the purpose of software testing.
- Present an argument for why you think software testing is useful or not.
- Discuss how software testing achieves its goals.
- Define faults and failures.
- Describe the purpose of the different levels of testing: unit, integration, system, end-to-end, user acceptance, and regression.
- Specify the input domain of a program.
- Develop an oracle for a program from a specification.
- Describe the problems of observability and controllability, and their impact on software testing.

Input Partitioning

- Describe the purpose of random testing.
- Define an operational profile for a program.
- Implement a test driver for random testing of a program, given an operational profile.
- Describe the purpose of equivalence partitioning.
- Produce equivalence classes for testing a program based on its functional specification.
- Produce equivalence classes for testing a program based on its source code.
- Combine equivalence classes from different domains/variables.

Boundary-Value Analysis

- Explain why boundary-value analysis is valuable compared to standard equivalence partitioning.
- Given a set of equivalence classes for a program, identify the boundaries of those equivalence classes, including the on points and off points.
- From a set of equivalence classes, select test inputs that adequately cover the boundaries of a program.

Coverage-Based Testing

- Identify the different parts of a control-flow graph, such as branches, conditions, and paths.
- Derive a control-flow graph for a program from its source code.
- Select test inputs for a program based on its control-flow graph, using the various control-flow coverage criteria.
- Identify when a variable in a program is defined, referenced, and undefined.
- Select test inputs from a control-flow graph annotated with variable information, using the various data-flow coverage criteria.
- Identify static data-flow anomalies from the variable information in a control-flow graph.
- Compare and contrast the different control-flow and data-flow coverage criteria.
- Describe the process of mutation analysis.
- Present an argument as to why you think mutation analysis is useful or not.
- Describe the coupling effect, and its impact on mutation analysis.
- Describe the program of equivalence mutants, and their impact on mutation analysis.
- Describe the importance of and motivate the use of coverage metrics in testing.

Test oracles

- Define the term “test oracle”.
- Contrast the different types of test oracle.
- Choose the correct type of test oracle for a given problem.
- Develop an oracle for a program from a specification.

Testing Modules

- Discuss the impact that observability and controllability have when testing modules.
- Construct a finite state automaton from the specification of a module.
- Derive a set of test sequences that achieve the different coverage criterion of a finite state automaton.
- Derive a set of test sequences from a finite state automaton that intrusively and non-intrusively test the module.
- Describe the pros and cons of polymorphism and inheritance in testing object-oriented programs.
- Derive a test suite that takes advantage of inheritance and counteracts the problems of polymorphism.

Security Testing

- Discuss the purpose of security testing.
- Explain the potential security implications (in terms of integrity, confidentiality, availability, arbitrary code execution etc.) of security faults
- Explain the relative strengths and weaknesses of random fuzzing vs mutation based fuzzing vs generation based fuzzing
- Choose (and justify) which security testing technique (random fuzzing, mutation based fuzzing, generation based fuzzing, or some combination) is most appropriate for testing a particular software component
- Explain the security implications of undefined behaviour in C code
- Discuss the advantages and disadvantages of using a memory debugger to check for memory errors, and undefined behaviour sanitizers to check for undefined behaviour
- Predict the impact that using a memory debugger will have on security testing (i.e. the liklihood of finding different kinds of security faults)
- Explain the difference between a covert storage channel and covert timing channel
- Discuss the use of security testing for finding these kinds of confidentiality faults
- Explain in general terms how general-purpose greybox fuzzers like AFL or libFuzzer operate and their advantages and disadvantages as compared to random, mutation and generation-based fuzzing.
- Explain in general terms how code execution (injection) attacks that exploit stack buffer overflows operate

Software Reliability Theory and Practice

- Explain the difference between testing for reliability and testing for functional correctness.
- Construct a reliability block diagram from a system architecture.
- Given reliability estimates for the sub-systems in a reliability block diagram, calculate an estimate for the overall system using reliability and failure logic.
- Construct a Markov model from a system description.
- Calculate the Markov state of a model after a specified number of transitions.
- Explain reliability growth.
- Critique the basic execution time model.
- Given some initial failure data, use the formula of the basic time execution model to provide reliability and failure estimates.
- Describe the process of error seeding for fault estimation.
- 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

- Explain the concepts of symbolic execution and dynamic symbolic execution, and discuss their strengths and weaknesses.
- Compare and contrast (dynamic) symbolic execution with other forms of testing, such as random testing and equivalence partitioning.
- Manually perform symbolic execution on a small program, deriving the path condition for that program.
- Compare and contrast symbolic execution and dynamic symbolic execution.