CS 4501 Software Analysis Final Study Guide

Sabrina Fuller

December 2021

1 Dynamic Analysis

1.1 Overview

Analysis that requires execution of a program to determine certain properties of that program. Most dynamic analysis includes the insertion of probes within the source code or binary file that will be triggered when the code is executed

1.2 Challenges

Requires insertion of the probes, may increase the execution time of the program Requires execution of the code which could be time consuming especially with various test suites. Can mostly only tell you the properties of that particular execution with the given inputs b. Differences between static and dynamic Static analysis tells the general overview of the program and the general properties, while dynamic analysis can only tell the particular properties of a certain execution. Cannot find the infeasible paths with dynamic execution Dynamic analysis can test runtime environment and concurrency of runtime, while static analysis can only tell about certain properties such as cycles etc.

1.3 Instrumentation

Once you determine that property that you want to analyze you create probes that record that information. This could be the recording certain branch paths What lines of code that gets executed What objects are created

1.4 Parts of dynamic instrumentation

Determine properties that you want to study Create instrumentation to gather that information Create test cases / test suite to use as input for your program input Run your program and look for the property you want to study

1.5 Gathering Information

1.5.1 Execution Coverage

Details what portions of the program are run. Such as what statements are executed, what statements functions/ branches are traversed, or whether a variable changed.

1.5.2 Execution Profiling

Records the sequence of statements run, the sequence of program states and sequence of memory usage. Can trace through, basic blocks, control flow edges etc.

1.5.3 Execution Tracing

Traces the sequence of the program, counts the occurrence of events for the program, this can be the code or other program graphs. Can look through def-use pairs to monitor data.

1.6 Heisenberg (Heisenbuggy) Principle

Adding instrumentation into your program will change some properties of that program and may affect runtime and not fully emulate the runtime environment of the program. For example, the instrumentation may change how the values in the stack/ registers change

Logging lots of information might be very memory intensive, so sampling from those logs to analyze can save time, but might not uncover "rare" bugs.

1.7 Dynamic vs. Static Analysis

.

Dynamic Analysis

- Analyze specific executions
- Requires specific instrumentation
- Finds memory/performance/concurrence/security issues
- Abstracts part of a program
- No confusion on path taken
- Can be used to test environment and average test case.
- Will not generate any false positives or false negatives

Static Analysis

- Does not require the code to be run
- Systemically follows an abstracted program
- Can provide generalized information regarding a specific program
- Focused on looking at validation/correctness

1.8 Challenges for Instrumentation

- Helpful if lots of test cases
- Are those tests indicative of common behavior
- You can use specific inputs to replicate issues
- Can use logs from live sessions

2 Fuzzing

2.1 Goal

The objective of fuzzing is to mutate the input to a program to find faults in the program

2.1.1 Feedback fuzzing

When you fuzz a program you provide a handful of seed inputs (that do not crash the program) and then mutate those seed inputs to try to create unique crashes/hangs for the program. Depending on how the program responds to those inputs, you mutate based on those. Provides a feedback loop that hopefully discovers more bugs as time goes on

2.1.2 Comparison with mutation testing

Mutation testing mutates the actual program and tests the test suite, while fuzzing mutates the input and tests how the program responds to those inputs

2.1.3 Differences between fuzzing and symbolic execution

Symbolic Execution

SE looks at what inputs cause which parts of the program to execute. For example, which inputs cause certain branches/ functions of the program to execute, symbolic execution can guarantee high coverage of the program, however it requires more overhead to actually implement.

Fuzzing

While fuzzing is less intelligent and just mutates the input to try to find how it crashes the program, however it is much easier to implement and use out of the box Fuzzing is an automated or semi automated way to find inputs while symbolic execution is based on symbolic representations etc.

2.2 Fuzzing Challenges

- When do you theoretically stop?
- How to mutate the input to find interesting inputs?
- The fuzzer may not find all paths or may not find interesting faults in the program in a timely manner
- AFL (American Fuzzy Lop) can take a simple text file as input to a program that was supposed to parse jpegs and eventually the fuzzer created inputs that were also jpegs (but after over 8 hours of running)

2.2.1 Interesting Data

Fuzzers mutate the input and save the inputs that cause unique crashes/hangs The user then can look at those faults and use to determine if there are possible security issues or more development is needed for the product

2.3 Phases of Fuzzing

- Identify seed inputs
- Generate fuzzed data (mutate those seed inputs
- Execute fuzzed data
- Monitor for faults in the program
- Determine exploit-ability/ if the faults are unique repeat

2.4 Types of bugs

- Security bugs
- Memory leaks
- Assertion errors
- Input errors or recursion errors etc.

3 Mutation Testing

3.1 Purpose

Mutate the program and run against the test suite to determine the quality of the test suite. The goal is that mutating the program will cause the program to crash with the test suites. Mutation testing can be used to determine what the test suite could be missing, the strength of certain test cases, or when to stop testing

3.2 Assumptions

- Programmers are competent and will produce programs that are correct or very close to a correct program
- An incorrect program can be corrected with just some minor changes to the program. Thus the given program and the "ideal" program are very similar.
- Mutants are similar to real faults that the program will face
- If the a given test suite can find mutant programs, the test suite will be good at catching real faults

3.3 Challenges

- Run time cost to generate the mutant programs P‘ and to run those programs against the given test suite
- Determining if Pa‘ and Pb‘ are equivalent is NP-complete

3.4 Mutating a Program

3.4.1 Creating Mutant Programs

Programs can be mutated with:

- Inserting a code statement
- Deleting a code statement
- Modifying values of variables
- Modifying/ Inverting the logic of a conditional/loop

3.4.2 Neutralizing Mutants

When a mutant program P‘ is run on the test suite and causes a fault that mutant is neutralized (traditionally referred to as killing mutant, but that's a bit violent) The goal is that there will be at least 1 input in the test suite $t \in T$ such that $P‘(t) \neq P(t)$

3.4.3 Equivalent mutants

A NP complete problem

Listing 1: P

```
int x = i;
if(x >=4){
x++}
```

Listing 2: P‘

```
int x = i;
if(! (x <3){
x++}
```

3.5 Mutation Testing Procedure

Given a test suite S and a program P

1. Mutate that program P to create P‘
2. Run against the test suite

   for each $t \in T$

   Determine if P(t) != P‘(t)
3. Calculate the number of mutants neutralized versus total mutations

If a $P(t)! = P‘(t)$, then that test case t is an adequate test case

3.6 Mutation Testing and Real Faults

- Mutation testing assumes that real faults will be found with a mutated program
- May not fully cover the test suite
- Papers suggest that around 17% of faults can not be found with mutation testing. Suggesting that mutation testing has too broad assumptions.

4 Regression Testing

4.1 Purpose

Regression testing is used to modify/update a test suite for a program that has some sort of modification. Useful in professional development environments to ensure that obsolete tests are discarded and the remaining tests are still relevant/useful for the program.

By completing regression testing you can save lots of time on bug fixes as well as ensuring functionality is carried on throughout the modified code

4.2 Types of Test cases After Modification

4.2.1 Obsolete Test Cases

Obsolete test cases, these cases are no longer useful to test certain properties of the program either with updated functionality of the program etc.

4.2.2 Reusable test cases

these test cases can be reused, and will ensure that those properties of the program are still working properly. However they don’t need to be executed because of the modifications of the program

4.2.3 Retestable New Test Cases

, are test cases that must be re-run and written to handle new functionality of the program

4.3 Test Case Selection

Find a test set on P‘ that ensures confirms to test specifications S Find new test cases that also meets test specifications S‘

4.3.1 Test Minimization

Challenges to test minimization is trying to find the test cases that are relevant and not redundant Because you want to ensure that functionality is carried over and no new bugs are introduced

4.3.2 Test Selection Techniques

:

- Test all $T - T_{obsolete}$ not practical mostly
- Random selecion, randomly select tes cases but might not select tests that tests modified code
- Selecting modification Traversing tests, guarantee selecting tests that run through modified code

4.3.3 Dejavu algorithm

Construct CFG for both P, P‘, G and G‘ are entry nodes

Compare G and G‘

- Compare N, N‘
- Mark each nodes as modified

  For each sucessor of N and N‘

  check to see if $N = N$‘

  If so then Compare the successors

  If not then add those edges to test cases

Essentially, compare the CFGs on a node by node basis and find the edges that are “interesting” and then select test cases that pass those edges

5 Test Prioritization

Purpose: Test minimization and prioritization, we want to reduce the number of tests to reduce run time and redundancy

5.1 Ordering prioritization

- Sequential/ Reverse Ordering
- Random Ordering
- Intelligent Ordering (assumes a-priori knowledge)
- ordering based on runtime and likelihood of finding faults
- Greedy Selection, finding the next best test case to to run based on some metric such as run time etc.
- Choosing test cases that test x amount of faults in x amount of time
- Fault matrix, matrix of test cases and faults that found
- Want to select smallest subset which test all cases
- Genetic Algorithmn, select test cases perform fitness function keep tests that perform well on the fitness function

5.2 Greedy Prioritization Algorithm

Follows the principle of making the locally optimal choice at each stage of execution. Other algorithms include creating genetic algorithm that select the best test cases

- Find $t \in T_{rt}$ that covers the most test cases
- set $T_{min} = T_{min} \cup t$
- Remove t from T_{rt}
- Repeat

6 Delta Debugging

6.1 Purpose

The purpose of delta debugging is to find the minimal input that causes the given program to crash, where any other smaller subset of that input works.

6.1.1 Input

The input into this delta debugging algorithm is the crash producing input Then implement a binary search to find the smallest test case that causes the crash Set the granularity of the program initially 2, if the binary search does not yield a crashing program / assertion error, increase the granularity and repeat

6.1.2 Elementary changes

The paper that created this technique shows that by sequencing elementary changes together (ie composing different changes together) that modify the input will produce a minimal crashing put Delta debugging allows you to fine tune comb through the input to find a minimal input useful for finding that tiny change that created a given bug

7 Program Repair

7.1 Purpose

The goal of program repair is to automate fixing of bugs within a program A native technique to do this would to be take every statement in a program and insert into the bug causing line in hopes that it fixes the bug

7.2 Assumptions

Assumptions that the program is mostly correct and that the solution to finding the bug lies in the source code/ database of correct programs Assumes that a small change in the program will fix the bug

7.3 Overfitting

If the bug is more complex and is not well generalized beyond the test cases used to create the repaired program

In the paper we looked at we saw that the technique actually overfitted with program repair and that was not able to generalize . Because training and input set are the same

7.4 Semantic Repair

The technique discussed in lecture assumes that bugs are within blocks of code and there is a correct version of the program in a program database The program then searches the database to find a block of code to insert into the program and then uses a genetic algorithm to determine fitness.

7.5 Genetic Search

Mutates the program, tests it compared to the test suite, determine if more useful than current test suite. If so, replace those tests in the test suite. This allows the inputs to evolve in a way that hopefully creates a repaired program.

7.5.1 Fault Localization

The assumption that statements that fail test cases should be weighted more heavily when repairing the program.

We also favor nodes that were not visited

8 DNN Testing

8.1 Purpose

While traditional software testing techniques have been well studied deep neural networks (DNN) are types of programs that are not well understood, thus presents challenges when trying to test the software.

8.2 DNN Challenges

The characteristics of DNN networks can create challenges The network is expected to generalize valid input distribution Current DNN look for valid/invalid inputs Generating invalid inputs increase the cost Invalid inputs inflate coverage Time needed to debug failures Higher coverage != better test suite Assumptions for traditional software Input validation logic is distinct from functional logic Test suites with higher coverage are better

8.3 Methods deployed for coverage

The coverage criteria for DNN focus on neuron coverage The ratio of number of neurons whose output is greater than a threshold value compared to total number of neurons Activation traces are captured during training and measure lower/upper bounds for each neuron Whether a test case falls into a major functional area or an edge case DNN or adversarial networks

8.4 Training Data and Deployed Data Distribution

8.5 Out of Distribution Problems

- Out of distribution refers to outliers/ anomaly detection in the given inputs
- Generative models learn the distribution of data and predict how likely a test input will fall within certain boundaries
- This technique can be used to determine invalid test inputs

8.6 Coverage

KMNC dividing interval Neuron boundary coverage Strong neuron activation modified condition/ decision coverage

8.7 Variational autoencoder

- Variational Autoencoder is a type of unsupervised learning
- This technique increases test coverage and produces inputs that cause the model make incorrect predictions
- Stochastic encoder maps inputs to latent space
- The decoder generates the data by sampling from the latent space
- The program then reconstructs the probabilities for both valid/invalid inputs
- In the case of MNIST certain areas of the images are more "interesting" to the network which can be used to figure out what number the image contains