Faults, Errors, and Failures

- **Fault** (also known as a bug): A static defect in software—incorrect lines of code.
- **Error**: An incorrect internal state—not necessarily observed yet.
- **Failure**: External, incorrect behaviour with respect to the expected behaviour—must be visible (e.g. EPIC FAIL).

RIP Fault Model

To get from a fault to a failure:

1. Fault must be *reachable*;
2. Program state subsequent to reaching fault must be incorrect: *infection*; and
3. Infected state must *propagate* to output to cause a visible failure.

Applications of the RIP model: automatic generation of test data, mutation testing.

Dealing with Faults, Errors and Failures

Three strategies for dealing with faults are avoidance, detection and tolerance. Or, you can just try to declare that the fault is not a bug, if the specification is ambiguous.

Testing vs Debugging

Testing: evaluating software by observing its execution.
Debugging: finding (and fixing) a fault given a failure.

About Testing

We can look at testing statically or dynamically.

Static Testing (ahead-of-time): this includes static analysis, which is typically automated and runs at compile time (or, say, nightly), as well human-driven static testing—typically code review.

Dynamic Testing (at run-time): observe program behaviour by executing it; includes black-box testing (not looking at code) and white-box testing (looking at code to develop tests).

Usually the word "testing" means *dynamic testing.*

Definition 1 Observability *is how easy it is to observe the system's behaviour, e.g. its outputs, effects on the environment, hardware and software.*

Definition 2 Controlability *is how easy it is to provide the system with needed inputs and to get the system into the right state.*

Definition 3 *(Coverage Level). Given a set of test requirements TR and a test set T, the coverage level is the ratio of the number of test requirements satisfied by T to the size of TR.*

Exploratory Testing

Scenarios where Exploratory Testing Excels. (from Bach's article)

- providing rapid feedback on new product/feature;
- learning product quickly;
- diversifying testing beyond scripts;
- finding single most important bug in shortest time;
- independent investigation of another tester's work;
- investigating and isolating a particular defect;
- investigate status of a particular risk to evaluate need for scripted tests.

Exploratory Testing Process.

- Start with a charter for your testing activity, e.g. "Explore and analyze the product elements of the software." These charters should be somewhat ambiguous.
- Decide what area of the software to test.
- Design a test (informally).
- Execute the test; log bugs.
- Repeat.

Basic Blocks. We can simplify a CFG by grouping together statements which always execute together (in sequential programs)

Definition 4 *A basic block is a sequence of instructions in the control-flow graph that has one entry point and one exit point.*

Statement and Branch Coverage

Definition 5 *A* test path *is a path p (possibly of length 0) that starts at some initial node (i.e. in N_0) and ends at some final node (i.e. in N_f).*

Definition 6 *Given a set of test requirements* TR *for a graph criterion C, a test set T satisfies C on graph G iff for every test requirement* tr *in* TR*, at least one test path p in path(T) exists such that p satisfies* tr.

Test cases and test paths. We connect test cases and test paths with a mapping path_G from test cases to test paths; e.g. $\text{path}_G(t)$ is the set of test paths corresponding to test case t.

- usually we just write path since G is obvious from the context.
- we can lift the definition of path to test sets T by defining $\text{path}(T) = \{\text{path}(t) | t \in T\}$.
- each test case gives at least one test path. If the software is deterministic, then each test case gives exactly one test path; otherwise, multiple test cases may arise from one test path.

Page Objects

A page object should abstractly represent the actions that a user can take on a page and allow the caller to query the state of the page (if necessary). The Selenium documentation suggests a page object for a sign-in page.

The sign-in page object also exposes the sole functionality of the page, which is to sign in. It then returns the page that gets loaded after sign-in.

Testing State Behaviour of Software via FSMs

Criterion 1 Complete Path Coverage. *(CPC)* TR *contains all paths in G.*

Note that CPC is impossible to achieve for graphs with loops.

We propose the use of graph coverage criteria to test with FSMs.

- nodes: software states (e.g. sets of values for key variables);
- edges: transitions between software states, i.e. something changes in the environment or someone enters a command.

- node coverage: visiting every FSM state = state coverage;
- edge coverage: visiting every FSM transition = transition coverage;
- edge-pair coverage (extension of edge coverage to paths of length at most 2): actually useful for FSMS; transition-pair, two-trip coverage.

The next criteria are mostly not for CFGs.

Definition 7 *A* round trip *path is a path of nonzero length with no internal cycles that starts and ends at the same node.*

Criterion 2 Simple Round Trip Coverage. *(SRTC)* TR *contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.*

Criterion 3 Complete Round Trip Coverage. *(CRTC)* TR *contains all round-trip paths for each reachable node in G.*

Mutation Testing

Using Grammars. Two ways you can use input grammars for software testing and maintenance:

- recognizer: can include them in a program to validate inputs;
- generator: can create program inputs for testing.

Some Grammar Mutation Operators.

- Nonterminal Replacement; e.g.
  dep = `"deposit"` account amount $\Longrightarrow$ dep = `"deposit"` amount amount (Use your judgement to replace nonterminals with similar nonterminals.)
- Terminal Replacement; e.g.
  amount = `"$"` digit$^+$ `"."` digit { 2 } $\Longrightarrow$ amount = `"$"` digit$^+$ `"$"` digit { 2 }
- Terminal and Nonterminal Deletion; e.g.
  dep = `"deposit"` account amount $\Longrightarrow$ dep = `"deposit"` amount
- Terminal and Nonterminal Duplication; e.g.
  dep = `"deposit"` account amount $\Longrightarrow$ dep = `"deposit"` account account amount

Using grammar mutation operators.

1. mutate grammar, generate (invalid) inputs; or,
2. use correct grammar, but mis-derive a rule once—gives "closer" inputs (since you only miss once.)

How Fuzzing Works. Two kinds of fuzzing: *mutation-based* and *generation-based*. Mutation-based testing starts with existing test cases and randomly modifies them to explore new behaviours. Generation-based testing starts with a grammar and generates inputs that match the grammar.

We're generating mutants m for the original program m_0.

Definition 8 *Test case t* kills *m if running t on m gives different output than running t on m_0.*

Mutation testing relies on two hypotheses, summarized from [**?**].

The *Competent Programmer Hypothesis* posits that programmers usually are almost right. There may be "subtle, low-level faults". Mutation testing introduces faults that are similar to such faults. (We can think of exceptions to this hypothesis—if the code isn't tested, for instance; or, if the code was written to the wrong requirements.)

The *Coupling Effect Hypothesis* posits that complex faults are the result of simple faults combining; hence, detecting all simple faults will detect many complex faults.

Definition 9 *Ground string: a (valid) string belonging to the language of the grammar (i.e. a programming language grammar).*

Definition 10 *Mutation Operator: a rule that specifies syntactic variations of strings generated from a grammar.*

Definition 11 *Mutant: the result of one application of a mutation operator to a ground string.*

Some points:

- How many mutation operators should you apply to get mutants? *One.*
- Should you apply every mutation operator everywhere it might apply? *Too much work; choose randomly.*

Killing Mutants. We can also define a mutation score, which is the percentage of mutants killed.

- *strong mutation*: fault must be *reachable*, *infect* state, and **propagate** to output.
- *weak mutation*: a fault which kills a mutant need only be *reachable* and *infect state.*

Uninteresting Mutants. Three kinds of mutants are uninteresting:

- *stillborn*: such mutants cannot compile (or immediately crash);
- *trivial*: killed by almost any test case;
- *equivalent*: indistinguishable from original program.

Integration Mutation. We can go beyond mutating method bodies by also mutating interfaces between methods, e.g.

- change calling method by changing actual parameter values;
- change calling method by changing callee; or
- change callee by changing inputs and outputs.

Summary of Syntax-Based Testing.

	Program-based	Input Space/Fuzzing
Grammar	Programming language	Input languages / XML
Summary	Mutates programs / tests integration	Input space testing
Use Ground String?	Yes (compare outputs)	Sometimes
Use Valid Strings Only?	Yes (mutants must compile)	No
Tests	Mutants are not tests	Mutants are tests
Killing	Generate tests by killing	Not applicable

Notes:

- Program-based testing has notion of strong and weak mutants; applied exhaustively, program-based testing could subsume many other techniques.
- Sometimes we mutate the grammar, not strings, and get tests from the mutated grammar.

Test Design Principles

Unit versus integration tests. Unit tests are more low-level and focus on one particular "class, module, or function". They should execute quickly. Sometimes you need to create fake inputs (or mocks) for unit tests; we'll talk about that too. You should generally not use an entire real input for a unit test.

Flaky tests are terrible. Although your A1Q1 tests should have been deterministic, not all tests are deterministic. Some tests fail a small percentage of the time.

- timeouts can fail when something takes surprisingly long;
- iterators can return items in random order;
- random number generators are, well, random.

Try really hard to make your tests not flaky. There are ways of dealing with them, but they're not good.

Bug Finding

A bug has got to be a violation of some specification. This might be a language-level specification (e.g. don't dereference null pointers), or it may be specific to an API that you are using.

More on Coverity. Coverity is a commercial product which can find many bugs in large (millions of lines) programs; it is therefore a leading company in building bug detection tools. We'll talk a bit more about Coverity in the future. Clients (900+) include organizations such as Blackberry, Yahoo, Mozilla, MySQL, McAfee, ECI Telecom, Samsung, Siemens, Synopsys, NetApp, Akamai, etc. These include domains including EDA, storage, security, networking, government (NASA, JPL), embedded systems, business applications, operating systems, and open source software.

- Contradictions: It attempts to find lies by cross-examining; contradictions indicate errors.
- Deviance: It assumes programs are mostly-correct and tries to infer correct behaviour from that assumption. If 1 person does X, then maybe it's right, or maybe that was just a coincidence. But if 1000 people do X and 1 person does Y, the 1 person is probably wrong.

Crucially: a contradiction constitutes an error, even without knowing the correct belief.

MUST-beliefs versus MAY-beliefs. We differentiate between MUST-beliefs (related to contradictions) and MAY-beliefs (related to deviance).

MUST-beliefs are inferred from acts that imply beliefs about the code. For instance:

```
x = *p / z; // MUST: p not null
            // MUST: z != 0
unlock(l);  // MUST: l acquired
x++;         // MUST: x not protected by l
```

MAY-beliefs, on the other hand, could be coincidental.

A(); // ... B();	A(); // ... B();	A(); // ... B();	A(); // ... B();	// MAY: A() and B() are paired.

We can check them as if they're MUST-beliefs and then rank errors by belief confidence.

MUST-belief examples. Let's look first at a couple of MUST-beliefs about null pointers.

- If I write `*p` in a C program, I'm stating a MUST-belief that `p` had better not be `NULL`.
- If I write the check `p == NULL`, I'm implying two MUST-beliefs: 1) POST: `p` is `NULL` on true path, not-`NULL` on false path; 2) PRE: `p` was unknown before the check.

Redundancy Checking. 1) Code ought to do something. So, when you have code that doesn't do anything, that's suspicious.

Process for verifying MAY beliefs. We proceed as follows:

1. Record every successful MAY-belief check as "check".
2. Record every unsucessful belief check as "error".
3. Rank errors based on "check" : "error" ratio.

Most likely errors occur when "check" is large, "error" small.

Summary: Belief Analysis. We don't know what the right spec is. So, look for contradictions.

- MUST-beliefs: contradictions = errors!
- MAY-beliefs: pretend they're MUST, rank by confidence.

(A key assumption behind this belief analysis technique: most of the code is correct.)

Regression Testing

Regression testing refers to any software testing that uncovers errors by retesting the modified program (Wikipedia). This form of testing often refers to comprehensive sets of test cases to detect regressions:

- of bug fixes that a developer has proposed.
- of related and unrelated other features that have been added.

Regression tests usually have the following attributes:

- **Automated**: no real reason to have manual regression tests.
- **Appropriately Sized**: too small and bugs will be missed. Too large and they will take a long time to run. Optimally, we want to run tests continuously.
- **Up-to-date**: ensure that tests are valid for the version of program being tested.

Industrial Best Practices

- **Unit Tests:** Each class has an associated unit test. If you change a class, you must also modify the unit test.
- **Code Reviews:** Each branch in the central version control system has owners. In order to commit code to that branch, you must have your code reviewed and approved by one of the owners. This ensures code quality.
- **Continuous Builds:** There is often a machine that continuously checks out and tests the latest code. All unit and regression tests are run and the status is made public to the team. The status contains information about the whether the code was built successfully, whether all unit and regression tests passed, and a list of the last few commits that were made to the branch. This ensures developers try to submit good code since if you break something, everyone knows about it :)
- **One-button Deploy:** If all tests have passed, one should be able to deploy to production with one command.
- **Back Button:** Systems should be designed so that it's possible to roll back changes.

State and Behaviour Tests

Let's consider two kinds of tests: state-based tests vs. behaviour-based tests.

- **State:** e.g. object field values. Verify by calling accessor methods.
- **Behaviour:** which calls System Under Test (SUT) makes. Verify by inserting observation points, monitoring interactions.

In state-based tests, we inspect only outputs, and only call methods from SUT. We do not instrument the SUT. We do not check interactions.

You have two options for verifying state:

1. procedural (bunch of asserts); or,
2. via expected objects (stay tuned).

For behaviour verification, we can use a mock object framework (e.g. JMock) to define expected behaviour.

Idea. Observe calls to the logger, make sure right calls happen.

Verifying Behaviour. The key is to observe actions (calls) of the SUT. Some options for doing this:

- procedural behaviour verification (the challenge in that case: recording and verifying behaviour); or
- expected behaviour specification (capturing the outbound calls of the SUT).

How to Improve Your Tests

Reducing Test Code Duplication. Copy-pasting is common when writing tests. This results in duplicate code in test cases, which has some undesirable side effects (bloat, unnecessary asserts). We'll talk about some techniques to mitigate duplication:

- Expected Objects
- Custom Assertions
- Verification Methods

Benefits of Custom Assertions. Writing custom assertions can help with your test design.

- hide irrelevant detail;
- label actions with a good name (names are super important); and
- are themselves testable;

Avoid logic in tests.

Test Doubles

Mock objects are a particular kind of test double. We need test doubles because objects collaborate with other objects, but we only want to test one object at a time. Meszaros categorizes test doubles as follows:

- dummy objects: these are not actually test doubles; they don't do anything, but just take up space in parameter lists. Are like `null`, but get past nullness checks in code.
- fake objects: have actual behaviour (which is correct), but somehow unsuitable for use in production; typical example is an in-memory database.
- stubs: produce canned answers in response to interactions from the class under test.
- mocks: like stubs, also produce canned answers. Difference: mock objects also check that the class under test makes the appropriate calls.
- spies: usually wraps the real object (instead of the mock, which stubs it), and records interactions for later verification.

Flaky Tests

Dealing with flaky tests. Companies with large test suites have found mitigations for the flaky test suite problem. One can label known-flaky tests as flaky and automatically re-run them to see if they eventually pass. One can also ignore or remove flaky tests. But this is unsatisfactory: it takes a long time to re-run failing tests.

Causes of flakiness. Luo et al studied 201 fixes to flaky tests in open-source projects. They found that the three most common causes of fixable flaky tests were:

1. improper waits for asynchronous responses;
2. concurrency; and
3. test order dependency.

Code Review

Formatting. Consistency in formatting helps avoid preventable errors. Positioning of { }s isn't something that necessarily has one right answer. Spaces are probably better than tabs. But the most important thing is to be self-consistent with yourself and within your project.

- **Don't Repeat Yourself**. The usual reason for it being bad is that fixes in one place may remain unfixed in the other place. (Recall: it wasn't always bad when used for forking and templating). For instance, if February actually had 30 days, you'd need to change a lot of code.

- **Fail Fast.** In the language of 6.031, we mean that a defect should be caught closest to when it's written. Static checks, as performed in compilers, catch defects earlier than dynamic checks, which catch defects earlier than letting wrong values percolate in the program state. In this particular example, there are no checks ensuring that a user had not permuted `month` and `dayOfMonth`.

- **Avoid Magic Numbers.** The above code is full of magic numbers. Particularly magical numbers include the `59` and `90` examples, as well as the month lengths and the month numbers. Instead, use names like `FEBRUARY` etc. Enums are a good way to encode months, and days-of-months should be in an array. The `59` should really be `31 + 28`, or better yet, `MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY].`

- **One Purpose Per Variable.** The specific variable that's being re-used in the above example is `dayOfMonth`, but this also applies to variables that you might use in your method. Use different variables for different purposes. They don't cost anything. Best to make method parameters `final` and hence non-modifiable.

Comments and code documentation Code should, ideally, be self-documenting, with good names for classes, methods, and variables. Methods should come with specifications in the form of Javadoc comments.

Reporting Bugs

Properties of Important Bugs

- Bug is sufficiently general to affect many users (easily reproducible).
- Bug has severe consequences (crashes, dataloss).
- Bug is new to most recent version.
- Bug has security implications.

Reproducibility

Developers can't fix problems they can't observe.

- Be maximally specific in describing steps to reproduce.
- Write the steps down as soon as possible, before you forget.
- Try to find a minimal testcase that demonstrates the problem.

Anatomy of a Bug Report

Summary. Perhaps the most critical field: a one-line recap of the bug. Enables searching for and judgement of the bug.

Description. Should be a complete description of the bug, including:

- Overview: expanded summary, e.g. "Drag-selecting any page crashes Mac builds in NSGetFactory".
- Steps to Reproduce (key!)
- Actual Results: what you see when you perform the steps to reproduce.
- Expected Results: what you think is correct
- Build Date and Platform: on development software, helps find the bug; include additional builds and platforms the bug might apply to.

Lifecycle-related fields. Some fields summarize the current state of the bug.

Properties of Good Bug Reports

- Reported in the database.
- Simple: one bug per report.
- Understandable, minimal, and generalizable.
- Reproducible.
- Non-judgemental. "The developers are all morons".
- Not a duplicate.

Bug Triage

The main attributes are type, likelihood, and priority, all evaluated on anchored scales on which staff are calibrated.

Static code analysis: PMD

XPath. Let's start from the fundamentals. You write *selectors* to find nodes. We'll look at a simple XML document. XPath also applies to web programming (in particular the Document Object Model) and also to the Java code we'll be analyzing.

Here is an XML file:

```
<?xml version="1.0" encoding="UTF-8"?>

<bookstore>
 <book>
  <title lang="fr">Harry Potter</title>
  <price>29.99</price>
 </book>
 <book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
 </book>
</bookstore>
```

Consider XPath expression `//price`. The result is the set of price nodes with data `29.99`, `39.95`. So, expression `//price` selects nodes with name `price`; the `//` means any descendants (including self) of the context node (= root node, here)—we asked for all descendants of the root named `price`. And, `count(//price)` counts the number of `price` elements in the tree.

We can also specify an exact path through the tree, say with `/bookstore/book[1]/title`. This starts at the root, visits the `bookstore` element, then its first `book` child, then returns the title. If we omitted `[1]`, then we'd get all of the titles.

We can also select all elements that satisfy some condition, e.g. `/bookstore/book[price>35]/title` selects the titles of books with price greater than 35.

Note the `lang` attribute. We can select elements with a certain value for `lang`: `//title[@lang="fr"]`.

In general, square brackets can contain predicates. We've seen pretty simple ones, but you can put arbitrary tree queries, e.g. `//price[../title[text()="Learning XML"]]`.

You can also combine predicates with `and`, `or`, etc. e.g. `//title[../price < 35 or @lang="en"]`. `*` works as you might expect.

Using XPath to write PMD Rules

Last time, we saw how some notions on how to use XPath in general. Today, we'll talk about XPath and PMD.

Here's a rule ensuring that while statements must not contain directly contain other statements:

```
//WhileStatement[not(Statement/Block)]
```

This rule matches `WhileStatement`s whose `Statement` child does not contain a `Block`.

We can detect `Logger` variables. Note the use of `@Image`. We are matching `VariableDeclarator`s whose type is `Logger`:

```
//VariableDeclarator[../Type/ReferenceType/ClassOrInterfaceType[@Image='Logger']]
```

But we really want class definitions which contain more than one variable of type `Logger`.

```
//ClassOrInterfaceDeclaration
    [count(.//VariableDeclarator
      [../Type/ReferenceType/ClassOrInterfaceType[@Image='Logger']])>1]
```

The Landscape of Testing and Static Analysis Tools

Here's a survey of your options:

- manual testing;
- running a JUnit test suite, manually generated;
- running automatically-generated tests;
- running static analysis tools.

We'll examine several points on this continuum. Some examples:

- Coverity: a static analysis tool used by 900+ companies, including BlackBerry, Mozilla, etc.
- Microsoft requires Windows device drivers to pass their Static Driver Verifier for certification.

Linters

In statically-typed languages, like C, compiler guarantees invariant that if code compiles, all symbols resolve.

For JavaScript and similar languages, need a linter to make that invariant. Cannot say much of anything about symbols resolving without a linter.

Use linters which block deployments if they don't pass!

FindBugs and False Positives

We've seen two tools so far: PMD and jshint. These tools both operate on Abstract Syntax Trees and ensure relatively shallow program properties.

FindBugs. This tool is somewhat deeper than PMD. FindBugs is an open-source static *bytecode* analyzer for Java out of the University of Maryland. A key difference is that it performs static analysis at Java bytecode level rather than AST level. It's therefore harder to write FindBugs rules.

FindBugs finds bug patterns like:

- off-by-one;
- null pointer dereference;
- ignored `read()` return value;
- ignored return value (immutable classes);
- uninitialized read in constructor;
- and more…

False positives. FindBugs, like all static analysis tools, gives some false positives. (The course project has a question about false positives.) In general, they occur because the analysis tool is not powerful enough. Because of the halting problem, there can be no all-powerful tool.

Static versus Dynamic Analysis

`valgrind` detects memory leaks dynamically.

Recall that a *dynamic* analysis monitors program behaviour at runtime, while a *static* analysis reasons about the program text.

- dynamically: You have complete information about program state on observed executions.
- statically: You have partial information about all executions.

So how does that work out on specific examples?

Virtual method call resolution. The question here is: given a virtual method call like `m.foo()`, where the actual runtime type of `m` could vary (due to subclassing), which `foo()` method actually gets called?

Dynamically, it is trivial to answer this question.

Statically, this is quite difficult. The problem is that one needs to know the type of `m`. The easiest answer is by using Class Hierarchy Analysis: using the declared type of `m`, compute the allowed types using the class hierarchy (i.e. subclasses of the declared type). Rapid Type Analysis gives a better answer—it limits the answer to all types that are instantiated somewhere in the program. But that requires an approximation of the reachable code, which in turn requires the answer to the very question we're trying to answer. There are even more accurate algorithms which propagate type constraints through the program.

Checking data structures for cycles. Last time, we saw code to ensure that the tree was actually acyclic. That was fairly straightforward. Here again, static checks are difficult; they require the analysis to verify that the code maintains the acyclicity invariant through all possible executions.

Unreachable code. Not everything is easier to verify dynamically than statically. Consider the question of whether a line of code is reachable or not. This is relatively easy to approximate statically (that's what dead code elimination does), but quite hard to check dynamically, since it depends heavily on finding appropriate program inputs.

Dynamic Analyses

We'll continue talking about generic properties, but this time we'll talk about dynamic verification of these properties.

Memory errors

Let's start by talking about leaks and other memory errors. We saw static detection of leaks with Facebook Infer. Valgrind's Memcheck tool and Clang's Address Sanitizer detect memory errors dynamically.

Valgrind's Memcheck detects the following errors:

- Illegal reads/writes: Memcheck complains about accesses to memory that the program should not be accessing (e.g. not returned from a `malloc`, or below the stack pointer).
- Reads of uninitialized variables: Memcheck tells you about reads of memory that has never been initialized, if the program prints out the resulting values.
- Illegal/wrong frees: Of course, you're not allowed to free memory that you didn't get from a memory allocation function, so Memcheck tells you. It also tells you when you use `free()` on something you got from `new[]`.
- Overlapping source/destination for memory moves.
- Fishy argument values: You probably don't mean to request either -3 bytes or more than 2^{63} bytes.
- Memory leaks: Memcheck tells you when you have memory that didn't get freed but that you no longer have any pointers to.

You will pay a significant performance penalty when using Valgrind; Memcheck typically comes with a $10\times$–$50\times$ slowdown. This is usable for bug diagnosis but obviously not for production.

The AddressSanitizer tool, supported by both clang and gcc, also detects memory errors. These errors are similar to those that Valgrind can detect. The technology is different, but the goals are similar. Because it uses different implementation technology, it runs much more quickly than Valgrind, with a reported typical slowdown of $2\times$.

AddressSanitizer finds the following errors:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks (experimental)

Note that Valgrind may return false positives, while AddressSanitizer has a design goal of returning no false positives. In fact, ASan aborts the program whenever it encounters a memory problem. You're supposed to fix it before proceeding. Of course, ASan might miss some problems.

Implementation Techniques. Valgrind and ASan use different techniques, hence yield different results. Valgrind emulates a CPU and checks, at every memory access, whether that access is legitimate or not. ASan, on the other hand, rewrites relevant memory accesses at compile-time to call a checking library. It turns out that calling a library is cheaper than emulating the CPU.

Conceptually, ASan maintains shadow memory— metadata about where the program is allowed to access (or not). Then, it replaces the `malloc()` and `free()` calls with its own versions; these versions update shadow memory and indicate that allocated memory is OK to access, unallocated memory not OK. Finally, every memory access in the program gets replaced with a checked access:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;  // or: ... = *address;
```

This is not so different from what Java does to check array accesses, for instance, but applies much more generally, to every memory access in the program.

Race detectors

Valgrind also has a race detection tool, Helgrind. This tool dynamically detects memory that is concurrently accessed by two threads. Such accesses are fine as long as they are controlled by a lock. At runtime, Helgrind knows which locks are held. If the program does not hold enough locks, then Helgrind flags a memory error.

```
acquire(lock1);                      acquire(lock2);
read(x);                             write(x); // different lock!
release(lock1);                      release(lock2);
```

One more dynamic tool: Randoop

Key Idea: "Writing tests is a difficult and time-consuming activity, and yet it is a crucial part of good software engineering. Randoop automatically generates unit tests for Java classes."

Randoop generates random sequence of method calls, looking for object contract violations.

To use it, simply point it at a program & let it run.

Randoop discards bad method sequences (e.g. illegal argument exceptions). It remembers method sequences that create complex objects, and sequences that result in object contract violations.

Course Summary

Many of the topics in this course are fairly straightforward. I hope that seeing them all in one place can help you make connections between the different topics.

Introduction

We started by talking about *faults*, *errors*, and *failures*. We also discussed *static* versus *dynamic* approaches, something which recurred throughout the course.

Defining Test Suites

Before defining test suites, I thought it was important for everyone to understand *exploratory testing*. We then moved on to *statement* and *branch* coverage, which require you to understand *control-flow graphs*. Alternatively, you might have a *Finite State Machine* and want to build test suites to cover round-trips in your FSM.

Grammar-based approaches are also important, particularly *fuzzing* for security-based properties. (Don't forget to try out the american fuzzy lop tool). We can also generate inputs from a grammar.

Mutation testing is probably the most difficult concept in the course. Recall that it's indirect: you're trying to make your test suite better by making sure that it can actually detect defects in the code.

We also looked at research which empirically evaluated best-case coverage of well-tested code (JUnit, can reach 93%; 80% is usual benchmark); which evaluated the usefulness of mutation testing (it actually works); and which evaluated the usefulness of coverage (not very, as a goal in itself).

Engineering Test Suites

We then moved on to discuss how to engineer test suites as artifacts. There's a lot more that I would have liked to talk about, like ensuring testability and test smells. But here's what we did discuss.

First, we talked about why you need good tests—it enables you to fearlessly modify your code without worrying about breaking it ("eat your vegetables!") We then talked about some *test design principles*. Moving on to more concrete points, we saw how Selenium let you write tests for webapps. *Regression testing* is also a key use for test suites; they should be fast and automated.

Tests themselves should be *self-checking*. They might verify either *state* or *behaviour* (using *mock objects*). They should be hooked up to a *continuous integration* system and should not be *flaky*.

Tools

A fundamental distinction is between *dynamic* and *static* approaches. Dynamic approaches have perfect information about a limited set of runs; static approaches have approximations which are valid for all runs.

We talked about bug-finding tools somewhat out of sequence to enable you to work on your project. The fundamental idea behind Coverity is to find contradictions and suspicious usage patterns. Your project does the same, but at a simpler level.

On to real tools, the first technique I talked about was still not a tool: *code review*. Along the same lines, *reporting bugs* is also important to talk about, but not strictly speaking a tool either.

We finally continued with real tools: *PMD* and *FindBugs*, which statically detect suspicious code patterns in Java source code and bytecode respectively. We also saw how to use PMD to run queries on your own codebases (using XPath expressions). `jshint` is another tool in the same spirit, but it detects sketchiness in JavaScript (like undefined variables, which you can't even use in sane languages). *Facebook Infer* uses more powerful static analysis to find memory leaks and null pointer dereferences, among others. All of these tools work on significant codebases.

Finally, on the dynamic tool side, we talked about *valgrind* and *Address Sanitizer*, which detect memory errors at runtime by instrumenting the code.