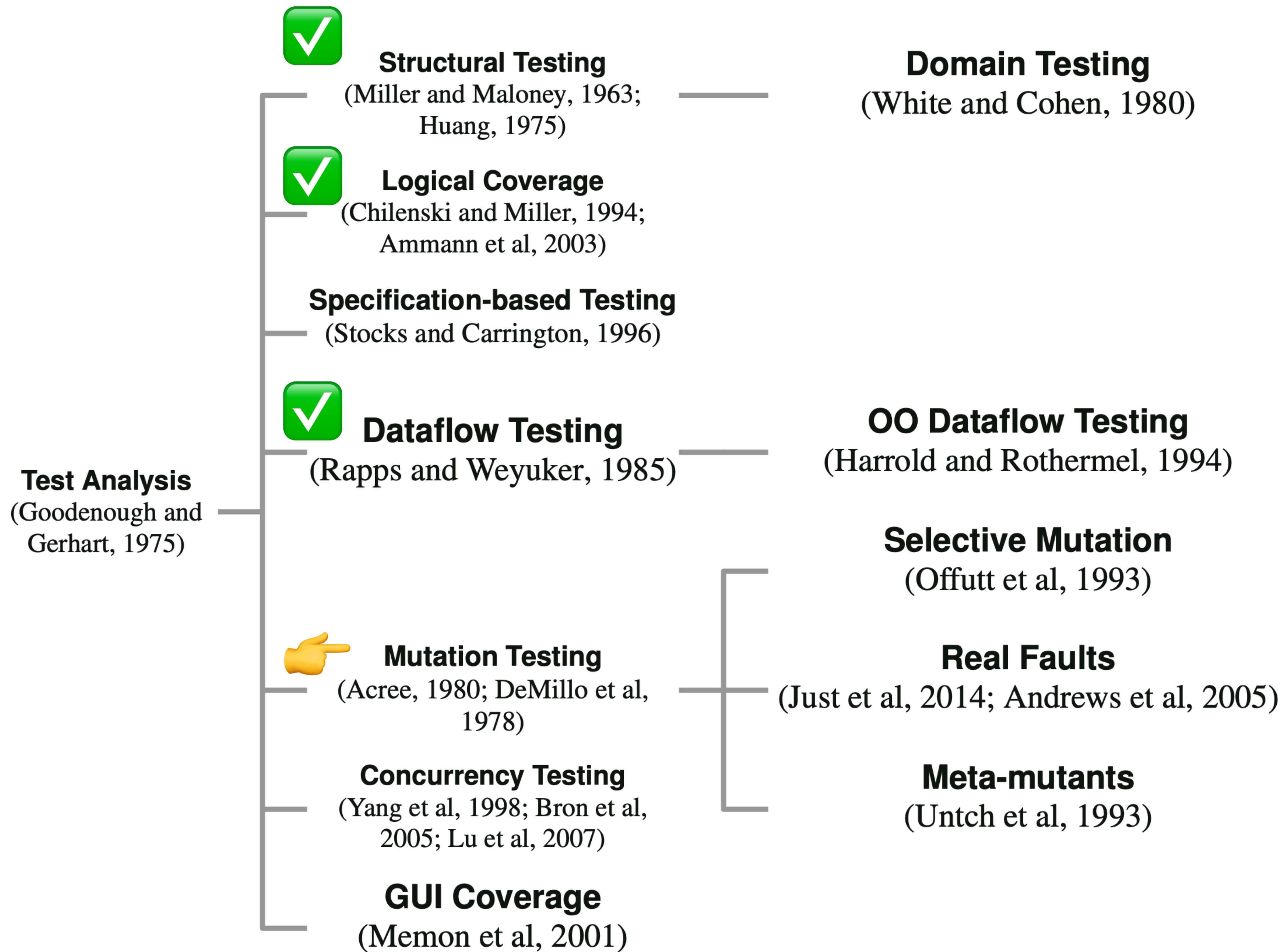
Software Testing, Verification and Validation

November 23, 2022
Week #11— Lecture #8

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Last week, we introduced dataflow testing in particular: All Defs, All Uses, All Def-Uses-Paths, All P-Uses/Some C-Uses, All C-Uses/Some P-Uses, All P-Uses, and All C-Uses as part of our set of white-box techniques. This week we will introduce Mutation Testing.



Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6

Mutation Testing

François Chollet
@fchollet

If you modify a complex piece of code, and your tests pass on the first try, you should immediately proceed to break the code in an obvious way and rerun the tests, to check that you're actually testing what you think you're testing.

7:03 AM · 4/11/21

181 Retweets **22** Quote Tweets **1,690** Likes

Context

In the structural testing module, we discussed line coverage, branch coverage, … In the logical coverage module we discusses MC/DC, … In the model-based testing module, we discussed path coverage. Etc. All these adequacy criteria measure how much of the program is exercised by the tests we designed.

However, these criteria alone might not be enough to determine the quality of the test cases. In practice, we can exercise large parts of the system, while testing very little.

```
public class Division {

    public static int[] getValues(int a, int b) {
1           if (b == 0)
2               return null;
3           int quotient = a / b;
4           int remainder = a % b;
5           return new int[] { quotient, remainder };
    }
}
```

```
public class Division {

    public static int[] getValues(int a, int b) {
1        if (b == 0)
2            return null;
3        int quotient = a / b;
4        int remainder = a % b;
5        return new int[] { quotient, remainder };
    }
}
```

```
@Test
public void testGetValues() {
    int[] values = Division.getValues(1, 1);
}

@Test
public void testZero() {
    int[] values = Division.getValues(1, 0);
}
```

```java
public class Division {

    public static int[] getValues(int a, int b) {
1           if (b == 0)
2               return null;
3           int quotient = a / b;
4           int remainder = a % b;
5           return new int[] { quotient, remainder };
    }
}
```

```java
@Test
public void testGetValues() {
    int[] values = Division.getValues(1, 1);
}

@Test
public void testZero() {
    int[] values = Division.getValues(1, 0);
}
```

While these tests give us 100% condition coverage, 100% decision coverage, 100% line coverage, …, these tests may never fail (i.e., may never reveal any fault). **Why?**

```java
public class Division {

    public static int[] getValues(int a, int b) {
1       if (b == 0)
2           return null;
3       int quotient = a / b;
4       int remainder = a % b;
5       return new int[] { quotient, remainder };
    }
}
```

```java
@Test
public void testGetValues() {
    int[] values = Division.getValues(1, 1);
}

@Test
public void testZero() {
    int[] values = Division.getValues(1, 0);
}
```

While these tests give us 100% condition coverage, 100% decision coverage, 100% line coverage, …, these tests may never fail (i.e., may never reveal any fault) as there are no **assertions**!

Fault Detection Capability

Fault detection capability indicates the test's capability to reveal faults in the system under test. The more faults a test can detect or, in other words, the more faults a test fails on, the higher its fault detection capability. Using this criterion, we can indicate the quality of our test suite in a better way than with just the coverage metrics we have so far.

The fault detection capability does not just regard the amount of production code executed, but also the assertions made in the test cases. For a test to be adequate according to this criterion, it has to have a meaningful **test oracle** (i.e., meaningful assertions).

The fault detection capability, as a test adequacy criterion, is the fundamental idea behind **mutation testing**. In mutation testing, we change small parts of the code, and check if the tests can find the introduced fault.

In the previous example, we created a test suite that was not adequate in terms of its fault detection capability, because there were no assertions, and so the tests would never find any faults in the code. Tests with assertions (aka oracles) check if the result of the method is what we expect.

```java
public class Division {

    public static int[] getValues(int a, int b) {
1       if (b == 0)
2           return null;
3       int quotient = a / b;
4       int remainder = a % b;
5       return new int[] { quotient, remainder };
    }
}
```

```java
@Test
public void testGetValues() {
    int[] values = Division.getValues(1, 1);
}

@Test
public void testZero() {
    int[] values = Division.getValues(1, 0);
}
```

```
public class Division {

    public static int[] getValues(int a, int b) {
1       if (b == 0)
2           return null;
3       int quotient = a / b;
4       int remainder = a % b;
5       return new int[] { quotient, remainder };
    }
}
```

```
@Test
public void testGetValues() {
    int[] values = Division.getValues(1, 1);
    assertEquals(1, values[0]);
    assertEquals(0, values[1]);
}

@Test
public void testZero() {
    int[] values = Division.getValues(1, 0);
    assertNull(values);
}
```

```
public class Division {

    public static int[] getValues(int a, int b) {
1           if (b == 0)
                return null;
3           int quotient = a / b;
4           int remainder = a % b;
5           return new int[] { quotient, remainder };
    }
}
```

```
✅ @Test
   public void testGetValues() {
       int[] values = Division.getValues(1, 1);
       assertEquals(1, values[0]);
       assertEquals(0, values[1]);
   }

✅ @Test
   public void testZero() {
       int[] values = Division.getValues(1, 0);
       assertNull(values);
   }
```

```java
public class Division {

    public static int[] getValues(int a, int b) {
1       if (b == 0)
2           return null;
3       int quotient = a / b;
4       int remainder = a % b;
5       return new int[] { quotient, remainder };
    }
}
```

To see how the values in a test case influence the fault detection capability, let's create two tests, where the denominator (i.e., b) is not 0.

```java
public class Division {

    public static int[] getValues(int a, int b) {
1       if (b == 0)
2           return null;
3       int quotient = a / b;
4       int remainder = a % b;
5       return new int[] { quotient, remainder };
    }
}
```

```java
@Test
public void testGetValuesOnes() {
    int[] values = Division.getValues(1, 1);
    assertEquals(1, values[0]);
    assertEquals(0, values[1]);
}

@Test
public void testGetValuesDifferent() {
    int[] values = Division.getValues(3, 2);
    assertEquals(1, values[0]);
    assertEquals(1, values[1]);
}
```

To see how the values in a test case influence the fault detection capability, let's create two tests, where the denominator (i.e., b) is not 0.

```java
public class Division {

    public static int[] getValues(int a, int b) {
1        if (b == 0)
2            return null;
3        int quotient = a / b;
4        int remainder = a % b;
5        return new int[] { quotient, remainder };
    }
}
```

```java
✅ @Test
   public void testGetValuesOnes() {
       int[] values = Division.getValues(1, 1);
       assertEquals(1, values[0]);
       assertEquals(0, values[1]);
   }

✅ @Test
   public void testGetValuesDifferent() {
       int[] values = Division.getValues(3, 2);
       assertEquals(1, values[0]);
       assertEquals(1, values[1]);
   }
```

```
public class Division {

    public static int[] getValues(int a, int b) {
1           if (b == 0)
                return null;
3           int quotient = a / b;
4           int remainder = a % b;
5           return new int[] { quotient, remainder };
    }
}
```

```
@Test
public void testGetValuesOnes() {
    int[] values = Division.getValues(1, 1);
    assertEquals(1, values[0]);
    assertEquals(0, values[1]);
}

@Test
public void testGetValuesDifferent() {
    int[] values = Division.getValues(3, 2);
    assertEquals(1, values[0]);
    assertEquals(1, values[1]);
}
```

For the fault detection capability, we want to see if the tests detect any faults in the code. This means we have to go back to the source code and introduce an error.

```
public class Division {

    public static int[] getValues(int a, int b) {
1           if (b == 0)
2               return null;
3           int quotient = a / b; a * b;
4           int remainder = a % b;
5           return new int[] { quotient, remainder };
    }
}
```

```
@Test
public void testGetValuesOnes() {
    int[] values = Division.getValues(1, 1);
    assertEquals(1, values[0]);
    assertEquals(0, values[1]);
}

@Test
public void testGetValuesDifferent() {
    int[] values = Division.getValues(3, 2);
    assertEquals(1, values[0]);
    assertEquals(1, values[1]);
}
```

For the fault detection capability, we want to see if the tests detect any faults in the code. This means we have to go back to the source code and introduce an error. We replace the division by a multiplication: a clear bug.

```
public class Division {

    public static int[] getValues(int a, int b) {
1           if (b == 0)
2               return null;
3           int quotient = a / b; a * b;
4           int remainder = a % b;
5           return new int[] { quotient, remainder };
    }
}
```

```
✅  @Test
    public void testGetValuesOnes() {
        int[] values = Division.getValues(1, 1);
        assertEquals(1, values[0]);
        assertEquals(0, values[1]);
    }

❌  @Test
    public void testGetValuesDifferent() {
        int[] values = Division.getValues(3, 2);
        assertEquals(1, values[0]);
        assertEquals(1, values[1]);
    }
```

If we run our tests with the *buggy* code, we see that the `testGetValuesOnes` still passes, but the other test, i.e., `testGetValuesDifferent` fails. This indicates that the second test has a higher fault detection capability.

```
public class Division {

    public static int[] getValues(int a, int b) {
1           if (b == 0)
2               return null;
3           int quotient = a / b; a * b;
4           int remainder = a % b;
5           return new int[] { quotient, remainder };
    }
}
```

```
✅ @Test
   public void testGetValuesOnes() {
       int[] values = Division.getValues(1, 1);
       assertEquals(1, values[0]);
       assertEquals(0, values[1]);
   }

❌ @Test
   public void testGetValuesDifferent() {
       int[] values = Division.getValues(3, 2);
       assertEquals(1, values[0]);
       assertEquals(1, values[1]);
   }
```

Even though both tests exercise the method in the same way and execute the same lines, we see a difference in the fault detection capability. This is because of the different input values for the method and the different test oracles (assertions) in the tests. In this case, the input values and test oracle of the `testGetValues` test can detect the *bug* better.

Hypotheses for Mutation Testing

The idea of mutation testing is to **assess the quality of the test suite**. This is done by manipulating bits of the source code and running the tests with this manipulated source code. If we have a good test suite, at least one of the tests will fail on this changed (i.e., *buggy*) code. Following this procedure, we get a sense of the fault error capability of our test suite.

In mutation testing, we use **mutants**. Mutants are modified programs which contain the defects, or faults that we introduce in the source code to be then used for determining the quality of the test suite.

A big question regarding the mutants is what their size should be. We can change single operations, whole lines or even multiple lines of code. What would work best?

Mutation testing and the answer to this question are based on the following two hypotheses:

- **The Competent Programmer Hypothesis (CPH)**: Here, we assume that the program is written by a competent programmer. More importantly, this means that given a certain specification, the programmer creates a program that is either correct, or it differs from a correct program by a combination of simple errors.
- **The Coupling Effect**: The coupling effect hypothesis states that simple faults are coupled with more complex faults. In other words, test cases that detect simple faults, will also detect complex faults.

Based on these two hypotheses, we can determine the size that the mutants should have. Realistically, following the competent programmer hypothesis, the faults in the actual code will be small. This indicates that the mutants' size should be small as well. Considering the coupling effect, test cases that detect small errors will also be able to detect larger, more complex errors.

Terminology

Before we talk about mutation testing in an in-depth way, we define some terms:

- **Mutant:** Given a program P, a mutant called P' is obtained by introducing a syntactic change to P. A mutant is killed if a test fails when executed with the mutant.

- **Syntactic Change:** A small change in the code. Such a small change should make the code still valid, i.e., the code can still compile and run.

- **Change:** A change in the code that mimics typical human mistakes. We will see some examples of these mistakes later in the following slides.

```
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

The test suite contains two tests for corner cases that throw an exception (i.e., `testInvertZero` and `testInvertMinValue`, and two "happy path" tests (i.e., `testInvert` and `testInvertNegative`).

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

✅
```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}
```

✅
```java
@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}
```

✅
```java
@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}
```

✅
```java
@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

The test suite contains two tests for corner cases that throw an exception (i.e., `testInvertZero` and `testInvertMinValue`, and two “happy path” tests (i.e., `testInvert` and `testInvertNegative`).

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

✅
```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}
```

✅
```java
@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}
```

✅
```java
@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}
```

✅
```java
@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

The test suite contains two tests for corner cases that throw an exception (i.e., `testInvertZero` and `testInvertMinValue`, and two “happy path” tests (i.e., `testInvert` and `testInvertNegative`). We will now determine the quality of our test suite using mutation testing.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

First, we have to create a *mutant* by applying a syntactic change to the original method. Keep in mind that, because of the two hypotheses, we want the syntactic change to be small: one operation/variable should be enough. Moreover, the syntactic change is a *change*, hence it should mimic mistakes that could be made by a programmer.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

First, we have to create a *mutant* by applying a syntactic change to the original method. Keep in mind that, because of the two hypotheses, we want the syntactic change to be small: one operation/variable should be enough. Moreover, the syntactic change is a *change*, hence it should mimic mistakes that could be made by a programmer.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator,  numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

First, we have to create a *mutant* by applying a syntactic change to the original method. Keep in mind that, because of the two hypotheses, we want the syntactic change to be small: one operation/variable should be enough. Moreover, the syntactic change is a *change*, hence it should mimic mistakes that could be made by a programmer.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator,  numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

✅
```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}
```

❌
```java
@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}
```

✅
```java
@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}
```

✅
```java
@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

If we execute the test suite on this mutant, the `testInvertNegative` test will fail, as `result.getFloat()` would be positive instead of negative.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

Another mistake could be made in line 1. When we studied boundary analysis, we saw that it is important to test the boundaries due to off-by-one errors. We can make a syntactic change by introducing such an off-by-one error. Instead of `numerator == 0`, in our new mutant we make it `numerator == 1`.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 1) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

Another mistake could be made in line 1. When we studied boundary analysis, we saw that it is important to test the boundaries due to off-by-one errors. We can make a syntactic change by introducing such an off-by-one error. Instead of `numerator == 0`, in our new mutant we make it `numerator == 1`.

```
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 1) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

❌
```
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}
```

✅
```
@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}
```

❌
```
@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}
```

✅
```
@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

We see that, again, the test suite catches this error. The test `testInvertZero` will fail, as it expects an exception but none is thrown in the mutant. The test `testInvert` will also fail since it has a 1 in the numerator which wrongly triggers an exception.

Automation

Writing the mutations of our programs manually might take a lot of time. Moreover, we probably would only think of the cases that are already tested. Like with test execution, we want to **automate the mutation process**. There are various tools that automatically generate mutants, but they all use the same methodology.

First we need mutation operators. A **mutation operator** is a grammatical rule that can be used to introduce a syntactic change. This means that, if the generator sees a statement in the code that corresponds to the grammatical rule of the operator (e.g., `a + b`), then the mutation operator specifies how to change this statement with a syntactic change (e.g., turning it into `a - b`).

Mutation Operators

We can identify two categories of mutation operators:

- **Real fault based operators**: Operators that are very similar to defects seen in the past for the same kind of code. Such operators look like common mistakes made by programmers in similar code.

- **Language-specific operators**: Mutations that are made specifically for a certain programming language. For example, changes related to the inheritance feature we have in Java, or changes regarding pointer manipulations in C.

Mutation Operators

Most mutation testing tools include various basic mutation operators for real fault based operators. Here are brief descriptions of some common mutation operators:

- **AOR - Arithmetic Operator Replacement**: Replaces an arithmetic operator by another arithmetic operator. Arithmetic operators are +, -, *, /, %.

- **ROR - Relational Operator Replacement**: Replaces a relational operator by another relational operator. Relational operators are <=, >=, !=, ==, >, <.

- **COR - Conditional Operator Replacement**: Replaces a conditional operator by another conditional operator. Conditional operators are &&, ||, &, |, !, ^.

- **AOR - Assignment Operator Replacement**: Replaces an assignment operator by another assignment operator. Assignment operators include =, +=, -=, /=.

- **SVR - Scalar Variable Replacement**: Replaces each variable reference by another variable reference that has been declared in the code.

AOR - Arithmetic Operator Replacement

Replaces an arithmetic operator by another arithmetic operator. Arithmetic operators are +, -, *, /, %.

Original

```
int c = a + b;
```

Mutant example

```
???
```

AOR - Arithmetic Operator Replacement

Replaces an arithmetic operator by another arithmetic operator. Arithmetic operators are +, -, *, /, %.

Original

```
int c = a + b;
```

Mutant example

```
int c = a - b;
```

ROR - Relational Operator Replacement

Replaces a relational operator by another relational operator. Relational operators are <=, >=, !=, ==, >, <.

Original

```
if (c == 0) {
    return -1;
}
```

Mutant example

```
???
```

ROR - Relational Operator Replacement

Replaces a relational operator by another relational operator. Relational operators are <=, >=, !=, ==, >, <.

Original

```
if (c == 0) {
    return -1;
}
```

Mutant example

```
if (c > 0) {
    return -1;
}
```

Fix CVE-2022-3602 in punycode

github.com/openssl/openssl/commit/3b421ebc64c7b52f1b9feb3812bdc7781c784332

Product Solutions Open Source Pricing Search Sign in Sign up

openssl / openssl Public

Sponsor Notifications Fork 8.6k Star 20.2k

Code Issues 1.8k Pull requests 268 Actions Projects 2 Wiki Security Insights

Fix CVE-2022-3602 in punycode decoder.

```
An off by one error in the punycode decoder allowed for a single unsigned int
overwrite of a buffer which could cause a crash and possible code execution.

Reviewed-by: Matt Caswell <matt@openssl.org>
Reviewed-by: Tomas Mraz <tomas@openssl.org>
(cherry picked from commit fe3b639)
```

Browse files

master

paulidale authored and **t8m** committed 22 days ago — 1 parent 89d7231 — commit 3b421ebc64c7b52f1b9feb3812bdc7781c784332

Showing **1 changed file** with **1 addition** and **1 deletion**.

Split Unified

2 crypto/punycode.c

```
@@ -181,7 +181,7 @@ int ossl_punycode_decode(const char *pEncoded, const size_t enc_len,
181  181          n = n + i / (written_out + 1);
182  182          i %= (written_out + 1);
183  183
184      -        if (written_out > max_out)
     184 +        if (written_out >= max_out)
185  185              return 0;
186  186
187  187          memmove(pDecoded + i + 1, pDecoded + i,
```

0 comments on commit 3b421eb

Please sign in to comment.

© 2022 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About

https://github.com/openssl/openssl/commit/3b421ebc64c7b52f1b9feb3812bdc7781c784332 — November 1, 2022

X.509 Email Address 4-byte B…

github.com/advisories/GHSA-8rwr-x37p-mx23

Product Solutions Open Source Pricing Search GitHub Sign in Sign up

GitHub Advisory Database / GitHub Reviewed / CVE-2022-3602

X.509 Email Address 4-byte Buffer Overflow

Critical severity GitHub Reviewed Published 22 days ago • Updated 16 days ago

Vulnerability details Dependabot alerts 0

Package	Affected versions	Patched versions
openssl-src (Rust)	>= 300.0.0, < 300.0.11	300.0.11

Description

A buffer overrun can be triggered in X.509 certificate verification, specifically in name constraint checking. Note that this occurs
after certificate chain signature verification and requires either a CA to have signed the malicious certificate or for the application to
continue certificate verification despite failure to construct a path to a trusted issuer. An attacker can craft a malicious email address
to overflow four attacker-controlled bytes on the stack. This buffer overflow could result in a crash (causing a denial of service) or
potentially remote code execution.

Many platforms implement stack overflow protections which would mitigate against the risk of remote code execution. The risk may be further mitigated based on stack layout for any given platform/compiler.

Pre-announcements of CVE-2022-3602 described this issue as CRITICAL. Further analysis based on some of the mitigating factors described above have led this to be downgraded to HIGH. Users are still encouraged to upgrade to a new version as soon as possible.

In a TLS client, this can be triggered by connecting to a malicious server. In a TLS server, this can be triggered if the server requests client authentication and a malicious client connects.

References

- rustsec/advisory-db#1452
- alexcrichton/openssl-src-rs@ 4a31c14
- https://rustsec.org/advisories/RUSTSEC-2022-0064.html
- https://www.openssl.org/news/secadv/20221101.txt

Severity

Critical 9.8 / 10

CVSS base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	High
Availability	High

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

Weaknesses

CWE-120

CVE ID

CVE-2022-3602

GHSA ID

GHSA-8rwr-x37p-mx23

Source code

alexcrichton/openssl-src-rs

https://github.com/advisories/GHSA-8rwr-x37p-mx23

COR - Conditional Operator Replacement

Replaces a conditional operator by another conditional operator. Conditional operators are &&, ||, &, |, !, ^.

Original

```
if (a == null || a.length == 0) {
  return new int[0];
}
```

Mutant example

```
???
```

COR - Conditional Operator Replacement

Replaces a conditional operator by another conditional operator. Conditional operators are &&, ||, &, |, !, ^.

Original

```
if (a == null || a.length == 0) {
  return new int[0];
}
```

Mutant example

```
if (a == null && a.length == 0) {
  return new int[0];
}
```

AOR - Assignment Operator Replacement

Replaces an assignment operator by another assignment operator. Assignment operators include =, +=, -=, /=.

Original

```
c = a + b;
```

Mutant example

```
???
```

AOR - Assignment Operator Replacement

Replaces an assignment operator by another assignment operator. Assignment operators include =, +=, -=, /=.

Original

```
c = a + b;
```

Mutant example

```
c −= a + b;
```

SVR - Scalar Variable Replacement

Replaces each variable reference by another variable reference that has been declared in the code.

Original

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b == 0 || b == Integer.MIN_VALUE){
      return null;
    }

    int quotient = a / b;
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

Mutant example

```
???
```

SVR - Scalar Variable Replacement

Replaces each variable reference by another variable reference that has been declared in the code.

Original

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b == 0 || b == Integer.MIN_VALUE){
      return null;
    }

    int quotient = a / b;
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

Mutant example

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (a == 0 || a == Integer.MIN_VALUE){
      return null;
    }

    int quotient = b / a;
    int remainder = quotient % a;

    return new int[] {remainder, a};
  }
}
```

Mutation Operators

In Java, there are a lot of language-specific operators. We can, for example, change the inheritance of the class, remove an overriding method, or change some declaration types. Without going into detail about these language-specific mutant operators, here are some examples:

- Access Modifier Change
- Hiding Variable Deletion
- Hiding Variable Insertion
- Overriding Method Deletion
- Parent Constructor Deletion
- Declaration Type Change
- …

Of course, there are many more mutant operators that are used by mutant generators. For now, you should at least have an idea what mutation operators are, how they work and what we can use them for.

Mutation Analysis & Testing

Our goal is to use **mutation testing** to determine the **quality** of our test suite. As we saw before, we have to create the mutants first and it is best to do this in an automated way with the help of mutation operators. Then, we run the test suite against each of the mutants with an execution engine. If any test fails when executed against a mutant, we say that the test suite *kills* the mutant. This is good, because it shows that our test suite has some fault detection capability. If none of the tests fail, the mutant stays alive.

Mutation Analysis & Testing

When performing mutation testing, we count the number of mutants our test suite killed and the number of mutants that are still alive. By counting the number of each of these mutant groups, we can give a value to the quality of our test suite.

We define the **mutation score** as:

$$\frac{\text{\# killed mutants}}{\text{\# mutants}} \times 100\%$$

Assessing the quality of a test suite by computing its mutation score is called **mutation analysis**.

Mutation testing then means using this mutation analysis to improve the quality of the test suite by adding and/or changing test cases.

These concepts are highly interconnected. Before mutation testing is carried out, you have to compute the mutation score. This then indicates whether the test suite should be changed. If the mutation score is low, there are a lot of mutants that are not killed by the test suite. You then have to improve the test suite.

Equivalent Mutants

Calculating the mutation score is challenging. The mutation score increases when less mutants are alive. This suggests that the best scenario is to have all the mutants killed by the test suite. While this is indeed the best scenario, it is often unrealistic because some of the mutants may be impossible to kill.

Mutants that cannot be killed are called equivalent mutants. An **equivalent mutant** is a mutant that always behaves in the same way as the original program. If the mutant behaves like the normal code, it will always give the same output as the original program for any given input. Clearly, this makes this mutant (which is basically the same program as the one under test) impossible to be killed by the tests.

Here, the equivalence is related to the definition of program equivalence. Program equivalence roughly means that two programs are functionally equivalent when they produce the same output for every possible input. This is also the equivalence between the normal code and an equivalent mutant.

Equivalent Mutants

Original

```
public void foo(int a) {
  int index = 10;
  while (...) {
    // ...
    index--;
    if (index == 0)
      break;
  }
}
```

Mutant example

```
public void foo(int a) {
  int index = 10;
  while (...) {
    // ...
    index--;
    if (index <= 0)
      break;
  }
}
```

Equivalent Mutants

Original

```
public void foo(int a) {
  int index = 10;
  while (...) {
    // ...
    index--;
    if (index == 0)
      break;
  }
}
```

Mutant example

```
public void foo(int a) {
  int index = 10;
  while (...) {
    // ...
    index--;
    if (index <= 0)
      break;
  }
}
```

Note how the original code starts with `index` set to 10, decrements it at every iteration of the `while` loop, and breaks out of the loop when `index` equals 0. The mutant works exactly the same, even though the condition is technically different. The `index` is still decremented from 10 to 0. Because `index` will never be negative, the == operator does the same as the <= operator.

The mutant produced by the generator is an equivalent mutant in this case.

Because of these equivalent mutants, we need to change the mutation score formula. We do not want to take the equivalent mutants into account, as there is nothing wrong with the tests when they do not kill these mutants.

Equivalent Mutants

Original

```
public void foo(int a) {
  int index = 10;
  while (...) {
    // ...
    index--;
    if (index == 0)
      break;
  }
}
```

Mutant example

```
public void foo(int a) {
  int index = 10;
  while (...) {
    // ...
    index--;
    if (index <= 0)
      break;
  }
}
```

The new mutation score formula becomes:

$$\frac{\#\ \text{killed mutants}}{\#\ \text{non-equivalent mutants}} \times 100\%$$

For the denominator, we used the number of non-equivalent mutants, instead of the total number of mutants.

To compute this new mutation score automatically, we need a way to determine automatically if a mutant is an equivalent mutant. Unfortunately, we cannot do this automatically, as **detecting equivalent mutations is an undecidable problem**. We can never be sure that a mutant behaves the same as the original program for every possible input.

Application

Mutation testing sounds like a great way to analyze and improve our test suites. However, the question remains whether we can use mutation testing in practice. For example, we can ask ourselves whether a test suite with a higher mutation score actually finds more errors.

A lot of research in software engineering tries to gain some insight into this problem. So far all the studies about mutation testing have shown that mutants can provide a good indication of a test suite's fault detection capability, as long as the mutation operators are carefully selected and the equivalent mutants are removed.

More specifically, a study by Just et al. shows that **mutant detection is positively correlated with real fault detection**. In other words, the more mutants a test suite detects, the more real faults the test suite can detect as well. Even more interesting is that this correlation is independent of the coverage. Furthermore, the correlation between mutant detection and fault detection is higher than the correlation between statement coverage and fault detection. So, the mutation score provides a better measure for the fault detection capability than the test coverage.

The Costs of Mutation Testing

Mutation testing is not without its costs. We have to generate the mutants, possibly remove the equivalent mutants, and execute the tests against each mutant. This makes mutation testing quite expensive, i.e., it takes a long time to perform.

Assume we want to do some mutation testing. We have:
- A code base with 300 Java classes
- 10 test cases for each class
- Each test case takes 0.2 seconds on average

The total test execution time is then:

300 x 10 x 0.2 = 600 seconds (10 minutes)

This execution time is for just the normal code.

The Costs of Mutation Testing

Mutation testing is not without its costs. We have to generate the mutants, possibly remove the equivalent mutants, and execute the tests against each mutant. This makes mutation testing quite expensive, i.e., it takes a long time to perform.

Assume we want to do some mutation testing. We have:
- A code base with 300 Java classes
- 10 test cases for each class
- Each test case takes 0.2 seconds on average

The total test execution time is then:

  300 x 10 x 0.2 = 600 seconds (10 minutes)

This execution time is for just the normal code.

For the mutation testing, we decide to generate on average 20 mutants per class. We will have to execute the entire test suite of a class on each of the mutants. Per class, we need

  20 x 10 x 0.2 = 40 seconds.

In total, the mutation testing will take

  300 x 40 = 12,000 seconds, or 3 hours and 20 minutes.

The Costs of Mutation Testing

Because of this cost, researchers have tried to find ways to make mutation testing faster for a long time. Based on some observations, they came up with the following heuristics.

- The first observation is that a test case can never kill a mutant if it does not cover the statement that changed (also known as the *reachability condition*). Based on this observation, we only have to run the test cases that cover the changed statement. This reduces the number of test cases to run and, with that, the execution time. Furthermore, once a test case kills a mutant, we do not have to run the other test cases anymore. This is because the test suite needs at least one test case that kills the mutant. The exact number of test cases killing the same mutant does not really matter.
- A second observation is that mutants generated by the same operator and inserted at the same location are likely to be coupled with the same type of fault. This means that when we use a certain mutation operator (e.g., Arithmetic Operator Replacement) and we replace the same statement with this mutant operator, we get two mutants that represent the same fault in the code. It is then highly probable that if the test suite kills one of the mutants, it will also kill the others. A heuristic that follows from this observation is to run the test suite against a subset of all the mutants (a technique also known as do fewer). Obviously, when we run the test suite against a smaller number of mutations, the overall testing will take less time. The simplest way of selecting the subset of mutants is by means of random sampling. As the name suggests, we select random mutants to consider. This is a very simple, yet effective way to reduce the execution time.

Tools

To perform mutation testing you can use one of many publicly available tools. These tools are often made for specific programming languages. One of the most mature mutation testing tools for Java is called PiTest (PIT Mutation Testing) https://pitest.org.

- PIT can be run from the command line, but it is also integrated into most popular IDEs (like Eclipse or IntelliJ). Project management tools like Maven or Gradle can also be configured to run PIT. As PIT is a mutation testing tool, it generates the mutants and runs the test suites against these mutants. Then it generates easy to read reports based on the results. In these reports, you can see the line coverage and mutation score per class. Finally, you can also see more detailed results in the source code and check which individual mutants were kept alive.

- Another mutation testing tool is Major http://mutation-testing.org. Major enables efficient mutation analysis of large software systems as well as fundamental research on mutation testing.

References

- Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6.
- R. Just, "The Major mutation framework: Efficient and Scalable Mutation Analysis for Java", in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014.
- R. Just, G. M. Kapfhammer, F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?", in Proceedings of the International Workshop on Mutation Analysis (Mutation), 2012.
- R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, "Are mutants a valid substitute for real faults in software testing?", in Proceedings of the Foundations of Software Engineering (FSE), 2014.
- Mauro Pezz and Michal Young, “Software Testing and Analysis: Process, Principles and Techniques”, Chapter 16.
- G. Petrović, M. Ivanković, G. Fraser, R. Just, “Does mutation testing improve testing practices?”, in Proceedings of the International Conference on Software Engineering (ICSE), 2021.
- J. Perretta, A. DeOrio, A. Guha, J. Bell, “On the use of mutation analysis for evaluating student test suite quality”, in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2022.