Error Seeding and Mutation Testing

Random Testing

- Based on a description of the legal inputs, generate test cases randomly over the program domain
- Drawbacks
  - Need to have an oracle for each test case
  - May not match the operational profile
- Benefits
  - Easy to generate test cases
  - Serves a a baseline for comparison
    - Using the same number of test cases, does testing criteria X do as well as random testing at detecting faults/finding failure?

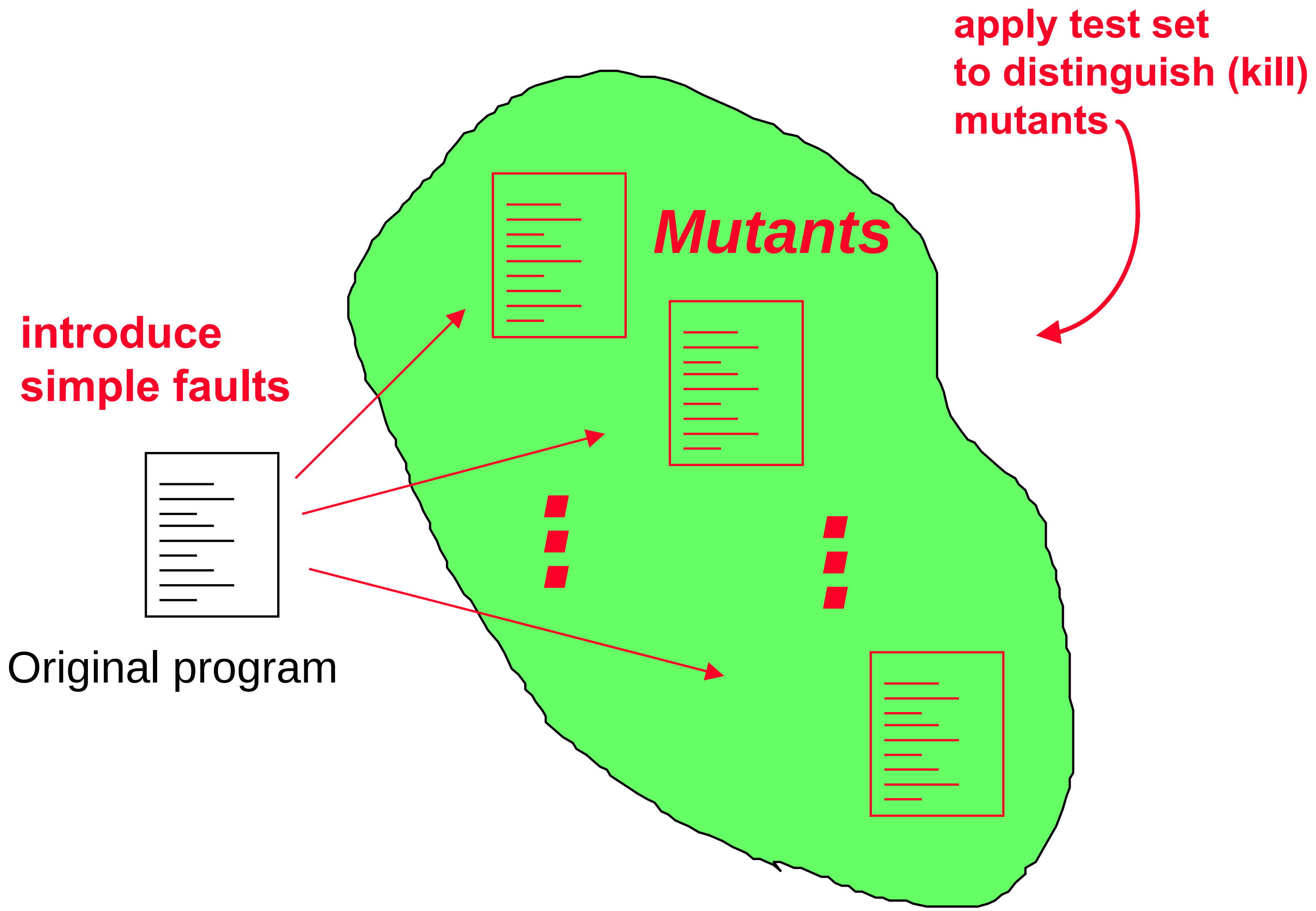
Error Seeding

- Insert “typical” faults into a system
- Determine how many of the inserted faults are found
  - If K of the N faults found, then assume that K/N of actual faults found as well
- Motivates developers/testers
  - Know there is something to find
  - Not looking for their own faults, so more motivated

Mutation Testing

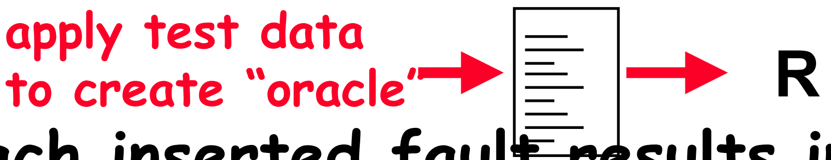
- Systematic method of error seeding
  - originally proposed by Budd, Lipton, DeMillo, and Sayward in the mid 1970s
- Approach: considers all simple (atomic) faults that could occur
  - introduces single faults one at a time to create "mutants" of original program
  - apply test set to each mutant program
  - "test adequacy" is measured by % "mutants killed"

Mutation Testing

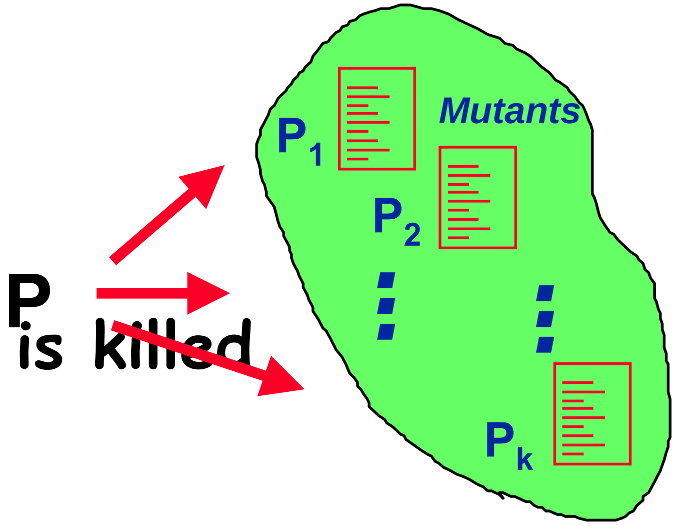


Mutation testing process

- Execute program P on test set T
  - P is considered the "correct" program
  - save results R to serve as an oracle



- Each inserted fault results in a new program
  - Mutant programs = $P_1, \ldots, P_k$



- If $P_i(T) \neq P(T)$ then mutant P_i is killed

Mutation Testing Assumptions

- Competent Programmer Hypothesis
  - programmers write programs that are reasonably close to the desired program
    - e.g., sort program is not written as a hash table
- Coupling Effect
  - detecting simple atomic faults will lead to the detection of more complex faults

Atomic faults: Operand mutations

- Constant replacement

  e.g., x := x + 5; would replace 5 with each constant of the appropriate type that appears in the program

- Scalar variable replacement

  e.g., y := x + 5; would replace x with each scalar variable of the appropriate type that appears in the program

More operand mutations

- scalar variable for constant replacement
- constant for scalar variable replacement
- array reference for constant replacement
- array reference for scalar variable replacement
- constant for array reference replacement

More operand mutations

- scalar variable for array reference replacement
- array reference for array reference replacement
- array index replacement for array index replacement
- data statement alteration

Operator mutations

- arithmetic operator replacement
  - e.g., `x := x + 5;`
  - would replace + with -, *, /, and **
- relational operator replacement
  - e.g., `a > b;`
  - would replace > with >=, <, <=, =, and /=

More operator mutations

- logical connector replacement
- absolute value insertion
- unary operator insertion
- statement deletion
- return statement replacement
- GOTO label replacement
- DO statement end replacement

Example

- consider the assignment:
  A : = X + 1;
- assume:
  - 2 is the only other constant in the program
  - Y is the only scalar variable of the same type as X and A
  - C[I] is the only array with the same type as X and A

Mutating one statement

operand mutations:

```
A := X + 1;  ⇒ A := X + 2
A := X + 1;  ⇒ A := X + Y
A := X + 1;  ⇒ A := X + A
A := X + 1;  ⇒ A := X + C[ I ]
A := X + 1;  ⇒ A := Y + 1
A := X + 1;  ⇒ A := A + 1
A := X + 1;  ⇒ A := C[ I ] + 1
A := X + 1;  ⇒ A := 1 + 1
A := X + 1;  ⇒ A := 2 + 1
A := X + 1;  ⇒ X := X + 1
A := X + 1;  ⇒ Y := X + 1
A := X + 1;  ⇒ C[ I ] := X + 1
```

binary operator replacement:

```
A := X + 1; ⇒ A := X - 1
A := X + 1; ⇒ A := X * 1
A := X + 1; ⇒ A := X / 1
A := X + 1; ⇒ A := X ** 1
```

unary operator insertion:

```
A := X + 1; ⇒ A := -X + 1
A := X + 1; ⇒ A := X + (-1)
A := X + 1; ⇒ A := - ( X + 1)
```

absolute value insertion:

```
A := X + 1; ⇒ A := abs ( X) + 1
A := X + 1; ⇒ A := abs ( X + 1)
```

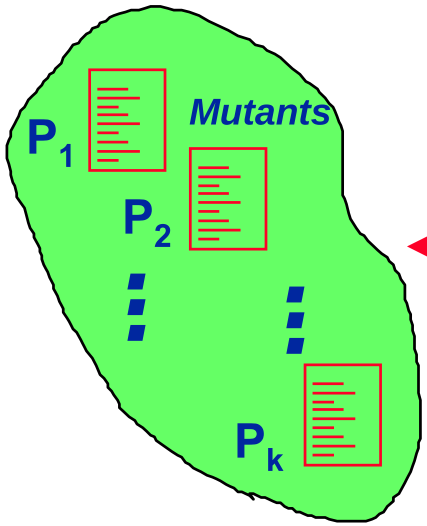
statement replacement:

```
A := X + 1; ⇒ continue
A := X + 1; ⇒ return
A := X + 1; ⇒ go to 100
```

Mutation testing process

- **execute each mutant P_i on T and compare results R_i to R**
  - If $R_i \neq R$ *then mutant is killed*
  - if $R_i = R$ *then either*

    $P_i = P$, *thus it is an equivalent mutant or the test cases do not reveal the error and need to find a new test case that does*



apply test data and compare output with oracle; "kill" distinguished mutants

Mutation System

- Automates the mutation process
    - uses the initial execution to determine the oracle
    - creates the mutants
    - lists the seeded errors that have not been detected

    user states (interactively) if the mutant is equivalent to the original program or finds a test case to kill the mutant

- Mutation system is a test set evaluation system

Techniques to optimize execution cost

- don't actually create and compile all the mutants
  - keep track of the internal state during execution of the original program & start with the statement preceding a mutated statement
  - stop execution if the values computed by the mutant ever become the same as the value computed by the correct program (*report that the mutant was not killed*)
- An alternative approach
  - stop execution if the values computed by the mutant are not the same as the original program (report that the mutant was probably killed)
    - *called weak mutation testing*

The state at a stmt consists of all the variables that are live (will ever be used in the future)

Examples

A := X + 1; mutated stmt A' := X + Y;

Test case:X=5; Y=3

Results: A= 6, A'= 8

Weak mutation testing would stop at this point and report that the mutant is killed

Z := A * (Y - 3);

Strong mutation would continue to be sure the fault propagates to an output

Still can stop and report that the mutant is NOT killed if the state at an intermediate point is the same as the original program

Conclusions

- even with optimization techniques, mutation testing is an expensive way to find faults in a program
- eliminating equivalent mutants is tedious; killing all mutants is hard
  - first 80% are easy, last 20% are hard

Is mutation testing effective at finding real faults in real programs?

- Several analytic studies showed that it "subsumes" other approaches
  - E.g., Subsumes statement and branch coverage
- Some studies showed that it is as effective or almost as effective as other test data selection techniques
  - only a few studies done and on limited size/simple programs
  - Mutation testing usually requires significantly more test cases to be as effective
- For the amount of effort, how does mutation testing compare to random testing?
  - For same number of test cases as mutation testing
    - Random test cases easier to generate
    - Mutation testing assumes the original program is the oracle

Another Mutation-Based Technique:

- Mutating Test Data
  - instead of mutating program, mutate input
- Bart Miller did an experiment where he demonstrated that arbitrary strings caused UNIX to consistently fail
  - wanted to understand why storms caused his connection to go down

C compiler experiment

- conducted by Bill McKeeman
- using the language grammar, generated legal C programs
- ran the generated C programs on n different C compilers and compared the results
- 20% of the time at least one of the compilers generated incorrect code

C compiler experiment-version 2

- weighted the grammar productions to avoid hard cases
- 1 out of 100 times at least one of the compilers generated incorrect code
- compiler maintainers gave a low priority to fixing these errors

Ada validation suite

- intended to demonstrate that a compiler handles all of the constructs
- did not attempt to stress test for each construct, although users have added such cases over the years

Comparison

- mutation testing evaluates the test cases
- mutated test data is "stress" testing the system

  often the generated strings do not correspond to typical cases

  e.g., 007.5

  thus, not the kind of errors most programmers will report

Summary of Mutation testing

- Mutation testing takes error seeding to the absurd,
  but it did simulate some useful research and insight
- Mutated test data is really a form of random testing, but looking at extreme values