**All Combinations (ACoC):** all combinations of blocks from all characteristics must be used. **Each Choice Coverage (ECC):** one value from each block for each characteristic must be used in at least one test case. **Pair-Wise Coverage (PWC):** a value from each block for each characteristic must be combined with a value from every block for each other characteristic. **T-Wise Coverage (TWC):** a value from each block for each group of t characteristics must be combined. **Base Choice Coverage (BCC):** a base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic. ***For triang(): Base:*** A1, B1, C1 – A1, B1, C2; A1, B1, C3; A1, B1, C4; A1, B2, C1; A1, B3, C1; A1, B4, C1; A2, B1, C1; A3, B1, C1; A4, B1, C1. **Multiple Base Choice Coverage (MBCC):** at least one and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic. ***For triang(): Bases:*** A1, B1, C1 – A1, B1, C3; A1, B1, C4; A1, B3, C1; A1, B4, C1; A3, B1, C1; A4, B1, C1. A2, B2, C2 – A2, B2, C3; A2, B2, C4; A2, B3, C2; A2, B4, C2; A3, B2, C2; A4, B2, C2. **ACC, PWC, TWC:** drop the infeasible pairs. **BCC, MBCC:** change a value to another non-base choice to find a feasible combination. **Path:** a sequence of nodes (each pair of nodes is an edge). **Length:** the number of edges (a single node is a path of length 0). **Subpath:** a subsequence of nodes in p is a subpath of p. **Test path:** a path that starts at an initial node and ends at a final node. **SESE graphs:** all test paths start at a single node and end at another node. **Visit:** a test path p visits node n if n is in p. A test path p visits edge e if e is in p. **Tour:** a test path p tours subpath q if q is a subpath of p. **Syntactic reach:** a subpath exists in the graph. **Semantic reach:** a test exists that can execute that subpath. **Structural Coverage Criteria:** defined on a graph just in terms of nodes and edges. **Data flow coverage criteria:** requires a graph to be annotated with references to variables. **Node Coverage (NC):** TR contains each reachable node in G. **Edge Coverage (EC):** TR contains each reachable path of length up to 1, inclusive in G. **Edge-Pair Coverage (EPC):** TR contains each reachable path of length up to 2, inclusive, in G. **Simple Path:** a path from node ni to nj is simple If no node appears more than once. **Prime Path:** a simple path that doesn’t appear as a proper subpath of any other simple path. **Round-trip Path:** a prime path that starts and ends at the same node. **Data Flow Coverage:** augment the CFG, (defs) are statements that assign values to variables, (uses) are statements that use variables. **Mutation Testing:** a mutant is a variation of a valid string, based on “mutation operators.” **Mutant:** a small syntactic change to a programming artifact. **Killing Mutants:** given a mutant m in M for a derivation D and a test t, t is said to kill m iff the output of t on D is different from the output of t on m. **Mutation score:** ratio of mutants killed. **Dead mutant:** a test case has killed it. **Trivial mutant:** almost every test can kill it. **Equivalent mutant:** no test can kill it. **The RIPR Model:** Reachability: the test causes the mutated statement to be reached. *Infection:* the test causes the mutated statement to result in an incorrect state. Propagation: the incorrect state propagates to incorrect output. Revealability: the tester must observe part of the incorrect output. **Premise of Mutation Testing:** if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault. **Metamorphic Relationship:** can be derived from the property if the vertices a and b are swapped, the length of the shortest path will remain unchanged, |f(G,b,a)| = |f(G,a,b)|. **Search Based Software Testing:** allows us to intelligently and heuristically search the space of possible solutions, non-exhaustive. **Regression Testing:** need to retest the software after each change. Have original test cases and test results from existing version of code, rerun the modified version with those tests and confirm passing. **Safe Regression Testing:** selects every test case in the test suite that may reveal a fault in the modified software.

A diagram of a graph

Description automatically generatedA diagram of a structure

Description automatically generatedA diagram of a diagram

Description automatically generated

A diagram of a graph

Description automatically generatedA table with text and images

Description automatically generatedA graph with numbers and lines

Description automatically generated with medium confidence

A diagram of a program mutation

Description automatically generatedA white paper with black text and blue and white text

Description automatically generatedA list of words on a white background

Description automatically generated

**In t-way combinatorial coverage, t represents:** the strength of the coverage, which indicates how many parameters are considered simultaneously in each test case. In pairwise testing (2-way coverage), each test case covers all possible pairs of parameter values. In 3-way testing, each test case covers all possible combinations of three parameters. The higher the value of t, the more comprehensive the coverage but also the larger the number of test case needed. **Prime path coverage subsumes edge-pair coverage:** False, prime path coverage aims to cover all unique paths through the control flow graph of a program, ensuring that each path is traversed at least once. Edge-pair coverage focuses on covering all possible pairs of edges in the CFG. **The TSL tool generates tests to satisfy the following coverage criteria (1) all combinations coverage (2) pair-wise coverage (3) base choice coverage, select strongest criteria it satisfies:** Pair-wise coverage, it aims to cover all possible pairs of parameter values in the test cases. **What is a basic block in a CFG?** A sequence of consecutive instructions with a single-entry point and single exit point. Example: x=5; y=10; z=x + y; print(z); The basic block has 4 instructions, each executed sequentially, and there are no branches or jumps within this sequence.

A screenshot of a computer program

Description automatically generated**Find an input that doesn’t reach the mutants:** *For mutant 1*, an array where the first element is not equal to val, but one of the subsequent elements is equal to val. This way, the loop would iterate over the entire array, ensuring that the mutant condition is never triggered. *For mutant 2*, an array where none of the elements are equal to val. This way, the loop would iterate over the entire array without ever setting findVal to 0, ensuring that the mutant condition is never triggered. **Find an input that infects but doesn’t propagate the mutants:** *For mutant 1*, an input where the first element is equal to val, causing the mutant to execute, but subsequent elements aren’t equal to val, preventing further propagation of the mutant (ex: int[] num = {5,3,7,9,2}; int val=5). *For mutant 2*, an input where at least one element is equal to val, causing mutant to execute, but the last element is not equal to val, preventing further propagation of the mutant (ex: int[] num={2,4,6,8,10,5}; int val=5). **Find an input that infects, propagates and kills the mutants:** *For mutant 1,* input where the first element is not equal to val, causing mutant to execute, and subsequent elements are equal to val, propagating the mutant, but last element is not equal to val, killing the mutant (ex: int[] num={1, 5, 5, 3}; int val=5). *For mutant 2,* input where at least one element is equal to val, causing mutant to execute, and last element is equal to val, propagating mutant, but no other elements equal to val, killing the mutant (ex: int[] num = {1,2,3,4,5}, int val=5). **Define the mutant score of a test suite:** Metric used to evaluate the effectiveness of the test suite in detecting faults (mutations) introduced intentionally into the codebase. It is the ratio of the number of mutants killed by the test suite to the total number of mutants introduced. Higher mutation score indicates that the test suite is more effective in detecting faults, as it successfully identifies and kills a larger proportion of mutants.