L13: Code Coverage & Mutation Testing

Software Analysis

IIT Guwahati · September 1, 2026

Where We Are

- We know how to write a **specification** — types, typestates, pre- and post-conditions, invariants.

Where We Are

- We know how to write a **specification** — types, typestates, pre- and post-conditions, invariants.
- We know how to **infer** some of them.

Where We Are

- We know how to write a **specification** — types, typestates, pre- and post-conditions, invariants.
- We know how to **infer** some of them.
- Using software testing, we can write **test cases** that check the program against those specifications.

Where We Are

- We know how to write a **specification** — types, typestates, pre- and post-conditions, invariants.
- We know how to **infer** some of them.
- Using software testing, we can write **test cases** that check the program against those specifications.
- Consider: all the tests pass. **Are we done?**

Where We Are

- We know how to write a **specification** — types, typestates, pre- and post-conditions, invariants.
- We know how to **infer** some of them.
- Using software testing, we can write **test cases** that check the program against those specifications.
- Consider: all the tests pass. **Are we done?**

How Good Is Your Test Suite?

- **Too few tests?** A bug fails none of them and escapes.

How Good Is Your Test Suite?

- **Too few tests?** A bug fails none of them and escapes.
- **Too many tests?** Test-suite bloat — slow builds, redundant tests, maintenance cost.

How Good Is Your Test Suite?

- **Too few tests?** A bug fails none of them and escapes.
- **Too many tests?** Test-suite bloat — slow builds, redundant tests, maintenance cost.

How do we systematically measure the quality of our test cases?

Code coverage

what fraction of the code
did the tests *execute*?

How Good Is Your Test Suite?

- **Too few tests?** A bug fails none of them and escapes.
- **Too many tests?** Test-suite bloat — slow builds, redundant tests, maintenance cost.

How do we systematically measure the quality of our test cases?

Code coverage

what fraction of the code
did the tests *execute*?

Code Coverage

A **code coverage metric** quantifies the extent to which a program's code is exercised by a given test suite.

Naik, *Software Specifications*, CIS 5470, Penn · Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Code Coverage

A **code coverage metric** quantifies the extent to which a program's code is exercised by a given test suite.

$$\text{coverage score} = \frac{\#\text{ test objectives covered by the suite}}{\#\text{ test objectives in the program}} \in [0\%, 100\%]$$

Naik, *Software Specifications*, CIS 5470, Penn · Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Code Coverage

A **code coverage metric** quantifies the extent to which a program's code is exercised by a given test suite.

$$\text{coverage score} = \frac{\#\text{ test objectives covered by the suite}}{\#\text{ test objectives in the program}} \in [0\%, 100\%]$$

A **coverage criterion** is what fixes the set of *test objectives*: functions, statements, branches, conditions, paths, ...

Naik, *Software Specifications*, CIS 5470, Penn · Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Code Coverage

A **code coverage metric** quantifies the extent to which a program's code is exercised by a given test suite.

$$\text{coverage score} = \frac{\#\text{ test objectives covered by the suite}}{\#\text{ test objectives in the program}} \in [0\%, 100\%]$$

A **coverage criterion** is what fixes the set of *test objectives*: functions, statements, branches, conditions, paths, ...

Naik, *Software Specifications*, CIS 5470, Penn · Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Coverage Criteria

Function Coverage

Objectives: the entry point of every function.

```
int  parse(const char *s);            called by test 1
int  evaluate(int *ast);              called by test 1
void report_error(const char *msg); never called
void dump_ast(int *ast);              never called
```

Function Coverage

Objectives: the entry point of every function.

```
int  parse(const char *s);            called by test 1
int  evaluate(int *ast);              called by test 1
void report_error(const char *msg); never called
void dump_ast(int *ast);              never called
```

$$\text{function coverage} = 2/4 = \mathbf{50\%}$$

Function Coverage

Objectives: the entry point of every function.

```
int  parse(const char *s);            called by test 1
int  evaluate(int *ast);              called by test 1
void report_error(const char *msg); never called
void dump_ast(int *ast);              never called
```

$$\text{function coverage} = 2/4 = \mathbf{50\%}$$

Statement Coverage

Objectives: every statement of the program. Test suite: a single call `foo(1,0)`.

```
int foo(int x, int y) {
    int z = 0;              executed
    if (x <= y)             executed (false only)
        z = x;              NOT executed
    else
        z = y;              executed
    return z;               executed
}
```

Statement Coverage

Objectives: every statement of the program. Test suite: a single call `foo(1,0)`.

```
int foo(int x, int y) {
    int z = 0;              executed
    if (x <= y)             executed (false only)
        z = x;              NOT executed
    else
        z = y;              executed
    return z;               executed
}
```

statement coverage $= 4/5 =$ **80%**

Statement Coverage

Objectives: every statement of the program. Test suite: a single call `foo(1,0)`.

```
int foo(int x, int y) {
    int z = 0;              executed
    if (x <= y)             executed (false only)
        z = x;              NOT executed
    else
        z = y;              executed
    return z;               executed
}
```

statement coverage $= 4/5 = \mathbf{80\%}$

green executed · **yellow** a decision that went only one way · **red** never executed

Decision (Branch) Coverage

Objectives: for each decision, the *true* branch and the *false* branch. Same suite: **foo(1,0)**.

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y)    true:  never     false:  foo(1,0)
        z = x;
    else
        z = y;
    return z;
}
```

Decision (Branch) Coverage

Objectives: for each decision, the *true* branch and the *false* branch. Same suite: **foo(1,0)**.

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y)    true:  never     false:  foo(1,0)
        z = x;
    else
        z = y;
    return z;
}
```

$$\text{decision coverage} = 1/2 = \mathbf{50\%}$$

Decision (Branch) Coverage

Objectives: for each decision, the *true* branch and the *false* branch. Same suite: **foo(1,0)**.

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y)    true:  never     false:  foo(1,0)
        z = x;
    else
        z = y;
    return z;
}
```

$$\text{decision coverage} = 1/2 = \mathbf{50\%}$$

Quiz: Compute the Coverage

Test suite: one test, `foo(1,0)`.

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y)
        z = x;
    else
        z = y;
    return z;
}
```

1. What is the **statement coverage**?
2. What is the **decision coverage**?
3. Add *one* call `foo(x,y)` that raises both to 100%. Which `x`, `y`?

Solution: Compute the Coverage

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y)          false only
        z = x;           missed
    else
        z = y;
    return z;
}
```

1. statement coverage $= 4/5 =$ **80%**
2. decision coverage $= 1/2 =$ **50%**

Solution: Compute the Coverage

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y)          false only
        z = x;           missed
    else
        z = y;
    return z;
}
```

1. statement coverage $= 4/5 =$ **80%**
2. decision coverage $= 1/2 =$ **50%**
3. any arguments with `x <= y` — for instance `foo(1,1)` — takes the true branch, executes `z = x`, and brings both metrics to **100%**.

Decisions Are Not Atomic

```
int fire_alarm(int smoke, int heat, int manual) {
    if (smoke && (heat || manual))
        return 1;
    return 0;
}
```

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Decisions Are Not Atomic

```
int fire_alarm(int smoke, int heat, int manual) {
    if (smoke && (heat || manual))
        return 1;
    return 0;
}
```

One **decision**, three **atomic conditions**: `smoke`, `heat`, `manual`.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Decisions Are Not Atomic

```
int fire_alarm(int smoke, int heat, int manual) {
    if (smoke && (heat || manual))
        return 1;
    return 0;
}
```

One **decision**, three **atomic conditions**: `smoke`, `heat`, `manual`.

Two tests — `(1,1,0)` and `(0,0,0)` — already give **100% statement** and **100% decision** coverage.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Decisions Are Not Atomic

```
int fire_alarm(int smoke, int heat, int manual) {
    if (smoke && (heat || manual))
        return 1;
    return 0;
}
```

One **decision**, three **atomic conditions**: `smoke`, `heat`, `manual`.

Two tests — `(1,1,0)` and `(0,0,0)` — already give **100% statement** and **100% decision** coverage.

Yet `manual` was never evaluated at all. Swap it for anything — `!manual`, `0` — and both tests still pass.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Condition-Level Criteria

CC

condition coverage

each atomic condition
takes both *true* and *false*

CC alone does *not* imply DC: for `a && b`, the tests `(T,F)` and `(F,T)` cover every condition both ways, yet the decision is false in both.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Condition-Level Criteria

CC
condition coverage
each atomic condition takes both *true* and *false*

DCC
decision/condition
DC *and* CC together

CC alone does *not* imply DC: for `a && b`, the tests `(T,F)` and `(F,T)` cover every condition both ways, yet the decision is false in both.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Condition-Level Criteria

CC
condition coverage
each atomic condition takes both *true* and *false*

DCC
decision/condition
DC *and* CC together

MCC
multiple condition
every combination of truth values in a decision

CC alone does *not* imply DC: for `a && b`, the tests `(T,F)` and `(F,T)` cover every condition both ways, yet the decision is false in both.

MCC needs 2^n tests for n conditions — Exhaustive, and in practice unaffordable.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

MC/DC

Modified condition/decision coverage

Each atomic condition must be shown, **on its own**, to change the outcome of the whole decision.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Chilenski & Miller, SEJ 9(5), 1994

MC/DC

Modified condition/decision coverage

Each atomic condition must be shown, **on its own**, to change the outcome of the whole decision.

1. every statement executed (SC)

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Chilenski & Miller, SEJ 9(5), 1994

MC/DC

Modified condition/decision coverage

Each atomic condition must be shown, **on its own**, to change the outcome of the whole decision.

1. every statement executed (SC)
2. every decision takes both outcomes (DC)

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Chilenski & Miller, SEJ 9(5), 1994

MC/DC

Modified condition/decision coverage

Each atomic condition must be shown, **on its own**, to change the outcome of the whole decision.

1. every statement executed (SC)
2. every decision takes both outcomes (DC)
3. every condition takes both values (CC)

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Chilenski & Miller, SEJ 9(5), 1994

MC/DC

Modified condition/decision coverage

Each atomic condition must be shown, **on its own**, to change the outcome of the whole decision.

1. every statement executed (SC)
2. every decision takes both outcomes (DC)
3. every condition takes both values (CC)
4. **and:** for each condition, a pair of tests that differ *only* in that condition and produce *different* decision outcomes — its **independence pair**

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Chilenski & Miller, SEJ 9(5), 1994

MC/DC

Modified condition/decision coverage

Each atomic condition must be shown, **on its own**, to change the outcome of the whole decision.

1. every statement executed (SC)
2. every decision takes both outcomes (DC)
3. every condition takes both values (CC)
4. **and:** for each condition, a pair of tests that differ *only* in that condition and produce *different* decision outcomes — its **independence pair**

Cost: $n + 1$ tests suffice for n conditions — *linear*, against 2^n for MCC.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Chilenski & Miller, SEJ 9(5), 1994

MC/DC on the Example

`smoke && (heat || manual)` — four tests, three independence pairs. (– = short-circuited, never evaluated.)

#	`smoke`	`heat`	`manual`	decision
1	F	–	–	**F**
2	T	F	F	**F**
3	T	F	T	**T**
4	T	T	–	**T**

MC/DC on the Example

`smoke && (heat || manual)` — four tests, three independence pairs. (– = short-circuited, never evaluated.)

#	`smoke`	`heat`	`manual`	decision
1	F	–	–	**F**
2	T	F	F	**F**
3	T	F	T	**T**
4	T	T	–	**T**

`smoke`: pair **(1,3)** · `heat`: pair **(2,4)** · `manual`: pair **(2,3)**

MC/DC on the Example

`smoke && (heat || manual)` — four tests, three independence pairs. (– = short-circuited, never evaluated.)

#	`smoke`	`heat`	`manual`	decision
1	F	–	–	**F**
2	T	F	F	**F**
3	T	F	T	**T**
4	T	T	–	**T**

`smoke`: pair **(1,3)** · `heat`: pair **(2,4)** · `manual`: pair **(2,3)**

Four tests for three conditions ($n + 1$), against eight for MCC.

Multiple Condition Coverage (MCC)

Objective: exercise every possible combination of truth values of the atomic conditions.

```
int authorize(int user, int admin, int active) {
    if (user && admin && active)
        return 1;
    return 0;
}
```

user	admin	active	decision
F	F	F	F
F	F	T	F
F	T	F	F
F	T	T	F
T	F	F	F
T	F	T	F
T	T	F	F
T	T	T	T

Multiple Condition Coverage (MCC)

Objective: exercise every possible combination of truth values of the atomic conditions.

```
int authorize(int user, int admin, int active) {
    if (user && admin && active)
        return 1;
    return 0;
}
```

user	admin	active	decision
F	F	F	F
F	F	T	F
F	T	F	F
F	T	T	F
T	F	F	F
T	F	T	F
T	T	F	F
T	T	T	T

MCC $= 8/8 =$ **100%** $\quad 2^3 = 8$ combinations

Multiple Condition Coverage (MCC)

Objective: exercise every possible combination of truth values of the atomic conditions.

```
int authorize(int user, int admin, int active) {
    if (user && admin && active)
        return 1;
    return 0;
}
```

user	admin	active	decision
F	F	F	F
F	F	T	F
F	T	F	F
F	T	T	F
T	F	F	F
T	F	T	F
T	T	F	F
T	T	T	T

MCC $= 8/8 =$ **100%** $\quad 2^3 = 8$ combinations

But do we need all combinations to show that *each condition independently affects the decision*?

Modified Condition/Decision Coverage (MC/DC)

A standard requirement for safety-critical systems, in avionics and in the automotive domain (DO-178B/C, ISO 26262). MC/DC requires **CC** and **DC**, and in addition that every condition in a decision be shown to **independently affect** the outcome of that decision.

```
C = A || B
```

Test	A	B	C
T_1	T	T	T
T_2	T	F	T
T_3	F	T	T
T_4	F	F	F

Condition coverage (CC)	$\{T_2, T_3\}$
Decision coverage (DC)	$\{T_1, T_4\}$
Condition/decision coverage (C/DC)	$\{T_1, T_4\}$

Modified Condition/Decision Coverage (MC/DC)

A standard requirement for safety-critical systems, in avionics and in the automotive domain (DO-178B/C, ISO 26262). MC/DC requires **CC** and **DC**, and in addition that every condition in a decision be shown to **independently affect** the outcome of that decision.

```
C = A || B
```

Test	A	B	C
T_1	T	T	T
T_2	T	F	T
T_3	F	T	T
T_4	F	F	F

Condition coverage (CC)	$\{T_2, T_3\}$
Decision coverage (DC)	$\{T_1, T_4\}$
Condition/decision coverage (C/DC)	$\{T_1, T_4\}$
MC/DC	$\{T_2, T_3, T_4\}$

Modified Condition/Decision Coverage (MC/DC)

A standard requirement for safety-critical systems, in avionics and in the automotive domain (DO-178B/C, ISO 26262). MC/DC requires **CC** and **DC**, and in addition that every condition in a decision be shown to **independently affect** the outcome of that decision.

```
C = A || B
```

Test	A	B	C
T_1	T	T	T
T_2	T	F	T
T_3	F	T	T
T_4	F	F	F

Condition coverage (CC)	$\{T_2, T_3\}$
Decision coverage (DC)	$\{T_1, T_4\}$
Condition/decision coverage (C/DC)	$\{T_1, T_4\}$
MC/DC	$\{T_2, T_3, T_4\}$

$T_2 \leftrightarrow T_4$ isolates A; $T_3 \leftrightarrow T_4$ isolates B; $n + 1 = 3$ tests suffice.

From MCC to MC/DC — Example

Objective: show that each atomic condition can independently affect the decision outcome.

```
if (user && admin && active)
    return 1;
```

Test	user	admin	active	decision	independence
T_1	F	T	T	F	
T_2	T	T	T	T	user
T_3	T	F	T	F	admin
T_4	T	T	F	F	active

From MCC to MC/DC — Example

Objective: show that each atomic condition can independently affect the decision outcome.

```
if (user && admin && active)
    return 1;
```

Test	user	admin	active	decision	independence
T_1	F	T	T	F	
T_2	T	T	T	T	user
T_3	T	F	T	F	admin
T_4	T	T	F	F	active

$T_1 \leftrightarrow T_2$ only `user` changes, decision changes
$T_2 \leftrightarrow T_3$ only `admin` changes, decision changes
$T_2 \leftrightarrow T_4$ only `active` changes, decision changes

From MCC to MC/DC — Example

Objective: show that each atomic condition can independently affect the decision outcome.

```
if (user && admin && active)
    return 1;
```

Test	user	admin	active	decision	independence
T_1	F	T	T	F	
T_2	T	T	T	T	user
T_3	T	F	T	F	admin
T_4	T	T	F	F	active

$T_1 \leftrightarrow T_2$ only `user` changes, decision changes
$T_2 \leftrightarrow T_3$ only `admin` changes, decision changes
$T_2 \leftrightarrow T_4$ only `active` changes, decision changes

MC/DC: 4 tests vs. MCC: $2^3 = 8$ tests

Path Coverage

Objectives: every execution path through the control-flow graph.

```
if (a) s1;        /* 2 paths        */
if (b) s2;        /* x 2 = 4        */
if (c) s3;        /* x 2 = 8        */

while (p) { ... }   /* unbounded */
```

Naik, *Software Specifications*, CIS 5470, Penn · Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Path Coverage

Objectives: every execution path through the control-flow graph.

```
if (a) s1;        /* 2 paths       */
if (b) s2;        /* x 2 = 4       */
if (c) s3;        /* x 2 = 8       */

while (p) { ... }   /* unbounded */
```

k sequential `ifs` $\Rightarrow 2^k$ paths; a single loop with a data-dependent bound $\Rightarrow$ **unboundedly many**.

Naik, *Software Specifications*, CIS 5470, Penn · Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Path Coverage

Objectives: every execution path through the control-flow graph.

```
if (a) s1;        /* 2 paths       */
if (b) s2;        /* x 2 = 4       */
if (c) s3;        /* x 2 = 8       */

while (p) { ... }   /* unbounded */
```

k sequential `ifs` $\Rightarrow 2^k$ paths; a single loop with a data-dependent bound $\Rightarrow$ **unboundedly many**.

Full path coverage is unachievable for real programs. Bounded variants for example k-path

Naik, *Software Specifications*, CIS 5470, Penn · Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021)

Coverage in Safety-Critical Software

Standard	Domain	Required at the highest level
DO-178C	avionics	**MC/DC** + DC + SC
ISO 26262	automotive	**MC/DC**
EN 50128	railway	**MC/DC**
IEC 62304	medical	structural coverage, class-dependent

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Naik, *Software Specifications*, CIS 5470, Penn

Coverage in Safety-Critical Software

Standard	Domain	Required at the highest level
DO-178C	avionics	**MC/DC** + DC + SC
ISO 26262	automotive	**MC/DC**
EN 50128	railway	**MC/DC**
IEC 62304	medical	structural coverage, class-dependent

DO-178C Level A = failure is **catastrophic**.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Naik, *Software Specifications*, CIS 5470, Penn

Coverage in Safety-Critical Software

Standard	Domain	Required at the highest level
DO-178C	avionics	**MC/DC** + DC + SC
ISO 26262	automotive	**MC/DC**
EN 50128	railway	**MC/DC**
IEC 62304	medical	structural coverage, class-dependent

DO-178C Level A = failure is **catastrophic**.

Bardin et al., *Specify and Measure, Cover and Reveal*, SCP 207 (2021) · Naik, *Software Specifications*, CIS 5470, Penn

Why 100% Is Hard

- **Infeasible objectives.** Some objectives cannot be covered by *any* input — defensive code, dead branches, unreachable states. Deciding which is undecidable in general.

Why 100% Is Hard

- **Infeasible objectives.** Some objectives cannot be covered by *any* input — defensive code, dead branches, unreachable states. Deciding which is undecidable in general.
- **Coverage is not correctness.** Executing a line is not checking it:

```
int r = divide(a, b);   /* 100% covered ... */
/* ... and no assertion, so any wrong r passes */
```

Why 100% Is Hard

- **Infeasible objectives.** Some objectives cannot be covered by *any* input — defensive code, dead branches, unreachable states. Deciding which is undecidable in general.
- **Coverage is not correctness.** Executing a line is not checking it:

```
int r = divide(a, b);   /* 100% covered ... */
/* ... and no assertion, so any wrong r passes */
```

High coverage is **necessary**, not **sufficient**.

Clang and `llvm-cov`

How Source-Based Coverage Works

Runtime counters are mapped back to source regions.

Step 1–3: Compile, Run, Merge

```
$ clang -O0 -g -fprofile-instr-generate -fcoverage-mapping \
       foo.c test_foo.c -o foo

$ LLVM_PROFILE_FILE=t1.profraw ./foo 1 0
foo(1,0) = 0

$ llvm-profdata merge -sparse t1.profraw -o foo.profdata
```

Step 1–3: Compile, Run, Merge

```
$ clang -O0 -g -fprofile-instr-generate -fcoverage-mapping \
       foo.c test_foo.c -o foo

$ LLVM_PROFILE_FILE=t1.profraw ./foo 1 0
foo(1,0) = 0

$ llvm-profdata merge -sparse t1.profraw -o foo.profdata
```

`foo.c` holds the function under test; `test_foo.c` is the driver, so one run of `./foo x y` is one test case.

Step 1–3: Compile, Run, Merge

```
$ clang -O0 -g -fprofile-instr-generate -fcoverage-mapping \
       foo.c test_foo.c -o foo

$ LLVM_PROFILE_FILE=t1.profraw ./foo 1 0
foo(1,0) = 0

$ llvm-profdata merge -sparse t1.profraw -o foo.profdata
```

`foo.c` holds the function under test; `test_foo.c` is the driver, so one run of `./foo x y` is one test case.

Step 4: The Summary, After One Test

```
$ llvm-cov report ./foo -instr-profile=foo.profdata foo.c

Filename   Regions  Missed   Cover   Functions Missed Executed   Lines Missed   Cover   Branches Missed  Cover
foo.c            4       1  75.00%           1      0  100.00%       8      1  87.50%          2      1 50.00%
```

Step 4: The Summary, After One Test

```
$ llvm-cov report ./foo -instr-profile=foo.profdata foo.c

Filename   Regions  Missed   Cover   Functions Missed Executed   Lines Missed   Cover   Branches Missed  Cover
foo.c            4       1  75.00%           1      0  100.00%       8      1  87.50%          2      1 50.00%
```

Four criteria: **region**, **function**, **line**, **branch** coverage.

Step 4: The Summary, After One Test

```
$ llvm-cov report ./foo -instr-profile=foo.profdata foo.c

Filename   Regions  Missed   Cover   Functions Missed Executed   Lines Missed   Cover   Branches Missed  Cover
foo.c            4       1  75.00%           1      0  100.00%       8      1  87.50%          2      1 50.00%
```

Four criteria: **region**, **function**, **line**, **branch** coverage.

The branch column is the one we computed by hand: **50%** — the `if` only ever went false.

Step 5: Line by Line

```
$ llvm-cov show ./foo -instr-profile=foo.profdata --show-branches=count foo.c

    1|       |/* The code under test: five executable statements, one decision. */
    2|      1|int foo(int x, int y) {
    3|      1|    int z = 0;
    4|      1|    if (x <= y)
  ------------------
  |  Branch (4:9): [True: 0, False: 1]
  ------------------
    5|      0|        z = x;
    6|      1|    else
    7|      1|        z = y;
    8|      1|    return z;
    9|      1|}
```

Step 5: Line by Line

```
$ llvm-cov show ./foo -instr-profile=foo.profdata --show-branches=count foo.c

    1|       |/* The code under test: five executable statements, one decision. */
    2|      1|int foo(int x, int y) {
    3|      1|    int z = 0;
    4|      1|    if (x <= y)
  ------------------
  |  Branch (4:9): [True: 0, False: 1]
  ------------------
    5|      0|        z = x;
    6|      1|    else
    7|      1|        z = y;
    8|      1|    return z;
    9|      1|}
```

The middle column is an **execution count**— line 5 ran **zero** times, and the branch went *False* once, *True* never.

Step 6: Add the Second Test and Merge

```
$ LLVM_PROFILE_FILE=t2.profraw ./foo 1 1
foo(1,1) = 1
$ llvm-profdata merge -sparse t1.profraw t2.profraw -o foo.profdata
$ llvm-cov report ./foo -instr-profile=foo.profdata foo.c

Filename   Regions  Missed   Cover   Functions Missed Executed   Lines Missed   Cover   Branches Missed  Cover
foo.c            4       0 100.00%           1      0  100.00%       8      0 100.00%          2      0 100.00%

    4|      2|    if (x <= y)
  |  Branch (4:9): [True: 1, False: 1]
    5|      1|        z = x;
```

Step 6: Add the Second Test and Merge

```
$ LLVM_PROFILE_FILE=t2.profraw ./foo 1 1
foo(1,1) = 1
$ llvm-profdata merge -sparse t1.profraw t2.profraw -o foo.profdata
$ llvm-cov report ./foo -instr-profile=foo.profdata foo.c

Filename   Regions  Missed   Cover   Functions Missed Executed   Lines Missed   Cover   Branches Missed  Cover
foo.c            4       0 100.00%           1      0  100.00%       8      0 100.00%          2      0 100.00%

    4|      2|    if (x <= y)
  |  Branch (4:9): [True: 1, False: 1]
    5|      1|        z = x;
```

The suite is the **merge** of the runs.

Regions, Lines, Statements

Region — a source range with a single execution counter.

```
if (x > 0) {
    y = 1;          // then region
} else {
    y = -1;         // else region
}
```

Regions, Lines, Statements

Region — a source range with a single execution counter.

```
if (x > 0) {
    y = 1;          // then region
} else {
    y = -1;         // else region
}
```

Line — a physical source line; covered if code mapped to that line executes.

```
if (x > 0) {      // line
    y = 1;        // line
}
```

Regions, Lines, Statements

Region — a source range with a single execution counter.

```
if (x > 0) {
    y = 1;          // then region
} else {
    y = -1;         // else region
}
```

Line — a physical source line; covered if code mapped to that line executes.

```
if (x > 0) {      // line
    y = 1;        // line
}
```

Statement — a language-level executable construct.

```
y = 1;              // statement
return y;           // statement
```

Regions, Lines, Statements

Region — a source range with a single execution counter.

```
if (x > 0) {
    y = 1;          // then region
} else {
    y = -1;         // else region
}
```

Line — a physical source line; covered if code mapped to that line executes.

```
if (x > 0) {      // line
    y = 1;        // line
}
```

Statement — a language-level executable construct.

```
y = 1;              // statement
return y;           // statement
```

Region = instrumentation unit **Line** = source location **Statement** = language construct

MC/DC with Clang

```
$ clang -O0 -g  -fprofile-instr-generate -fcoverage-mapping \
       -fcoverage-mcdc alarm.c -o alarm
$ LLVM_PROFILE_FILE=w.profraw ./alarm
fire_alarm(1,1,0) = 1
fire_alarm(0,0,0) = 0
$ llvm-profdata merge -sparse w.profraw -o w.profdata
$ llvm-cov report ./alarm -instr-profile=w.profdata -show-mcdc-summary alarm.c

Filename    Regions Missed   Cover   Lines Missed    Cover   Branches Missed  Cover   MC/DC Missed  Cover
alarm.c          12      1  91.67%      12      0 100.00%          8      3 62.50%       3      2 33.33%
```

Clang, *Source-Based Code Coverage*

MC/DC with Clang

```
$ clang -O0 -g  -fprofile-instr-generate -fcoverage-mapping \
       -fcoverage-mcdc alarm.c -o alarm
$ LLVM_PROFILE_FILE=w.profraw ./alarm
fire_alarm(1,1,0) = 1
fire_alarm(0,0,0) = 0
$ llvm-profdata merge -sparse w.profraw -o w.profdata
$ llvm-cov report ./alarm -instr-profile=w.profdata -show-mcdc-summary alarm.c

Filename    Regions Missed   Cover   Lines Missed    Cover   Branches Missed  Cover   MC/DC Missed  Cover
alarm.c          12      1  91.67%      12      0 100.00%          8      3 62.50%       3      2 33.33%
```

Clang, *Source-Based Code Coverage*

Using llvm-cov

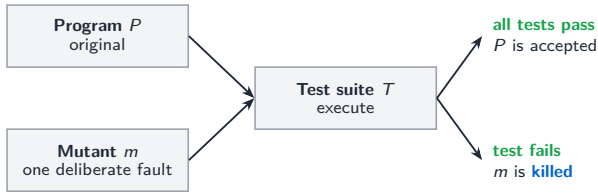
```
$ llvm-cov show ./alarm -instr-profile=w.profdata --show-mcdc alarm.c

    5|      2|    if (smoke && (heat || manual))
  |---> MC/DC Decision Region (5:9) to (5:34)
  |  Number of Conditions: 3
  |     Condition C1 --> (5:9)      [smoke]
  |     Condition C2 --> (5:19)     [heat]
  |     Condition C3 --> (5:27)     [manual]
  |
  |  Executed MC/DC Test Vectors:
  |     C1, C2, C3    Result
  |  1 { F,  -,  -  = F      }
  |  2 { T,  T,  -  = T      }
  |
  |  C1-Pair: covered:  (1,2)
  |  C2-Pair: not covered
  |  C3-Pair: not covered
  |  MC/DC Coverage for Decision: 33.33%
```

Mutation Testing

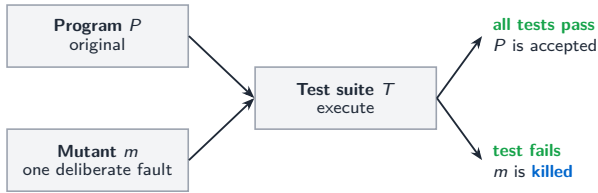
What Is Mutation Testing?

Idea: introduce one small, realistic fault into a program and check whether the test suite detects it.



What Is Mutation Testing?

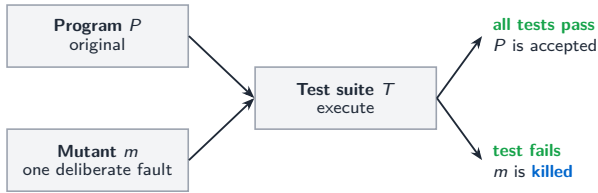
Idea: introduce one small, realistic fault into a program and check whether the test suite detects it.



All tests pass on $m \Rightarrow$ mutant **survives** $\Rightarrow$ the suite did not detect the fault.

What Is Mutation Testing?

Idea: introduce one small, realistic fault into a program and check whether the test suite detects it.



All tests pass on $m \Rightarrow$ mutant **survives** $\Rightarrow$ the suite did not detect the fault.

Program and Mutant, Same Test

The program under test, and a test case that passes.

```
int do_something(int x, int y) {
    if (x < y)
        return x+y;
    else
        return x*y;
}
```

```
int a = do_something(5, 10);
assert(a == 15);
```

```
a == 15
test passes
```

Program and Mutant, Same Test

The program under test, and a test case that passes.

```
int do_something(int x, int y) {
    if (x < y)
        return x+y;
    else
        return x*y;
}
```

```
int a = do_something(5, 10);
assert(a == 15);

        a == 15
        test passes
```

A single fault is inserted, giving a **mutant**; the same test is executed again.

```
int do_something(int x, int y) {
    if (x < y)
        return x-y;
    else
        return x*y;
}
```

```
int a = do_something(5, 10);
assert(a == 15);

        a == -5
        test fails:  mutant killed
```

Program and Mutant, Same Test

The program under test, and a test case that passes.

```
int do_something(int x, int y) {
    if (x < y)
        return x+y;
    else
        return x*y;
}
```

```
int a = do_something(5, 10);
assert(a == 15);

        a == 15
        test passes
```

A single fault is inserted, giving a **mutant**; the same test is executed again.

```
int do_something(int x, int y) {
    if (x < y)
        return x-y;
    else
        return x*y;
}
```

```
int a = do_something(5, 10);
assert(a == 15);

        a == -5
        test fails:  mutant killed
```

The test fails on the mutant: the suite checks against a fault of this kind at this location.

What Counts as a Mutant

- a **syntactic** change to the original program, which must still compile;

What Counts as a Mutant

- a **syntactic** change to the original program, which must still compile;
- a **simple** change — the kind of glitch a programmer makes, not a rewrite of the logic;

What Counts as a Mutant

- a **syntactic** change to the original program, which must still compile;
- a **simple** change — the kind of glitch a programmer makes, not a rewrite of the logic;
- derived mechanically by a **mutation operator**, one rule per class of typical fault;

What Counts as a Mutant

- a **syntactic** change to the original program, which must still compile;
- a **simple** change — the kind of glitch a programmer makes, not a rewrite of the logic;
- derived mechanically by a **mutation operator**, one rule per class of typical fault;

A Test Suite at 100% Coverage

A function and a suite of two tests, both of which pass.

```
int shipping_cost(int weight, int express) {
    int cost = 5;
    if (weight > 10)
        cost = cost + 3;
    if (express)
        cost = cost * 2;
    return cost;
}

T1: shipping_cost( 5, 0) == 5          T2: shipping_cost(20, 1) == 16
```

```
Filename       Regions  Missed   Cover   Lines Missed    Cover   Branches Missed    Cover
shipping.c           5       0 100.00%      11      0 100.00%          4      0 100.00%
```

A Test Suite at 100% Coverage

A function and a suite of two tests, both of which pass.

```
int shipping_cost(int weight, int express) {
    int cost = 5;
    if (weight > 10)
        cost = cost + 3;
    if (express)
        cost = cost * 2;
    return cost;
}

T1: shipping_cost( 5, 0) == 5          T2: shipping_cost(20, 1) == 16
```

```
Filename       Regions  Missed   Cover   Lines Missed    Cover   Branches Missed    Cover
shipping.c           5       0 100.00%      11      0 100.00%          4      0 100.00%
```

Every statement executed, every branch taken both ways.

Injecting Small Faults

Three mutants, each a single-token change to the program.

```
    if (weight > 10)        Δ1  if (weight >= 10)      ROR
        cost = cost + 3;    Δ2  cost = cost - 3;       AOR
    if (express)
        cost = cost * 2;    Δ3  cost = cost + 2;       AOR
```

```
$ ./mutate.sh                                    suite: T1, T2
MUT1  suite passed                                 --> SURVIVED
MUT2  FAIL  shipping_cost(20,1) = 4,  expected 16  --> killed
MUT3  FAIL  shipping_cost(20,1) = 10, expected 16  --> killed

mutation score = 2/3 = 66%
```

Injecting Small Faults

Three mutants, each a single-token change to the program.

```
    if (weight > 10)        Δ1   if (weight >= 10)      ROR
        cost = cost + 3;    Δ2   cost = cost - 3;       AOR
    if (express)
        cost = cost * 2;    Δ3   cost = cost + 2;       AOR
```

```
$ ./mutate.sh                                   suite: T1, T2
MUT1  suite passed                                --> SURVIVED
MUT2  FAIL  shipping_cost(20,1) = 4,  expected 16  --> killed
MUT3  FAIL  shipping_cost(20,1) = 10, expected 16  --> killed

mutation score = 2/3 = 66%
```

The suite distinguishes the program from Δ_2 and Δ_3, but not from Δ_1: a program that charges the surcharge at a weight of exactly 10 passes every test.

The Test the Surviving Mutant Identifies

Δ_1 and the original program differ on exactly one input class, `weight == 10`; the suite contains no such test.

Adding it:

```
T3: shipping_cost(10, 0) == 5          /* the boundary */
```

```
$ ./mutate.sh                                suite: T1, T2, T3
MUT1  FAIL  shipping_cost(10,0) = 8,  expected 5   --> killed
MUT2  FAIL  shipping_cost(20,1) = 4,  expected 16  --> killed
MUT3  FAIL  shipping_cost(20,1) = 10, expected 16  --> killed

mutation score = 3/3 = 100%
```

The Test the Surviving Mutant Identifies

Δ_1 and the original program differ on exactly one input class, `weight == 10`; the suite contains no such test.

Adding it:

```
T3: shipping_cost(10, 0) == 5          /* the boundary */
```

```
$ ./mutate.sh                                suite: T1, T2, T3
MUT1  FAIL  shipping_cost(10,0) = 8,  expected 5   --> killed
MUT2  FAIL  shipping_cost(20,1) = 4,  expected 16  --> killed
MUT3  FAIL  shipping_cost(20,1) = 10, expected 16  --> killed

mutation score = 3/3 = 100%
```

- Both suites have **100% coverage**.
- Regions, lines, and branches are covered equally.
- Coverage cannot distinguish the two suites.
- Mutation testing reveals the missing test.

Mutation Testing

- **Mutation operators** apply small syntactic modifications to the program under test, producing **mutants**. Mutants must compile; they are not tests, but a means of obtaining tests.

Mutation Testing

- **Mutation operators** apply small syntactic modifications to the program under test, producing **mutants**. Mutants must compile; they are not tests, but a means of obtaining tests.

- A test **kills** a mutant m of a program P if the output of the test on P differs from its output on m.

Mutation Testing

- **Mutation operators** apply small syntactic modifications to the program under test, producing **mutants**. Mutants must compile; they are not tests, but a means of obtaining tests.

- A test **kills** a mutant m of a program P if the output of the test on P differs from its output on m.

- Two underlying hypotheses:

  competent programmer — faults introduced by experienced programmers are small syntactic deviations from a correct program;

  coupling effect — tests that detect simple faults also detect the more complex faults that those simple faults couple to.

Kim, *Mutation Testing*, CS40508, KAIST · Ammann & Offutt, *Introduction to Software Testing*, 2nd ed., Ch. 9

Mutants and the Mutation Score

The original program, with three of its mutants shown inline.

```
int Min(int A, int B) {
    int minVal;
    minVal = A;                    Δ1  minVal = B;
    if (B < A) {                   Δ2  if (B > A)
        minVal = B;                Δ3  if (B < minVal)
    }
    return minVal;
}
```

Mutants and the Mutation Score

The original program, with three of its mutants shown inline.

```
int Min(int A, int B) {
    int minVal;
    minVal = A;                    Δ1  minVal = B;
    if (B < A) {                   Δ2  if (B > A)
        minVal = B;                Δ3  if (B < minVal)
    }
    return minVal;
}
```

Each Δ_i is a separate program. The **mutation score** of a test suite T is

$$MS(T) = \frac{\text{mutants killed by } T}{\text{mutants} - \text{equivalent mutants}}$$

Mutants and the Mutation Score

The original program, with three of its mutants shown inline.

```
int Min(int A, int B) {
    int minVal;
    minVal = A;                    Δ1  minVal = B;
    if (B < A) {                   Δ2  if (B > A)
        minVal = B;                Δ3  if (B < minVal)
    }
    return minVal;
}
```

Each Δ_i is a separate program. The **mutation score** of a test suite T is

$$MS(T) = \frac{\text{mutants killed by } T}{\text{mutants} - \text{equivalent mutants}}$$

Equivalent mutant: a mutant whose behavior is identical to the original program for all possible inputs.

When Is a Mutant Killed?

A test t **kills** a mutant m when the output on P differs from the output on m.

test t	`Min`	Δ_1 `minVal=B`	Δ_2 `B>A`	Δ_3 `B<minVal`
t_1: `(A=1, B=2)`	1	**2**	**2**	1
t_2: `(A=2, B=1)`	1	1	**2**	1
t_3: `(A=2, B=2)`	2	2	2	2

When Is a Mutant Killed?

A test t **kills** a mutant m when the output on P differs from the output on m.

test t	`Min`	Δ_1 `minVal=B`	Δ_2 `B>A`	Δ_3 `B<minVal`
t_1: `(A=1, B=2)`	1	**2**	**2**	1
t_2: `(A=2, B=1)`	1	1	**2**	1
t_3: `(A=2, B=2)`	2	2	2	2

- **Killed:** some test produces a different output.
- **Survived:** no test in the current suite kills it.

When Is a Mutant Killed?

A test t **kills** a mutant m when the output on P differs from the output on m.

test t	`Min`	Δ_1 `minVal=B`	Δ_2 `B>A`	Δ_3 `B<minVal`
t_1: `(A=1, B=2)`	1	**2**	**2**	1
t_2: `(A=2, B=1)`	1	1	**2**	1
t_3: `(A=2, B=2)`	2	2	2	2

- **Killed:** some test produces a different output.
- **Survived:** no test in the current suite kills it.
- **Equivalent:** no possible test can distinguish it from the original.

When Is a Mutant Killed?

A test t **kills** a mutant m when the output on P differs from the output on m.

test t	`Min`	Δ_1 `minVal=B`	Δ_2 `B>A`	Δ_3 `B<minVal`
t_1: `(A=1, B=2)`	1	**2**	**2**	1
t_2: `(A=2, B=1)`	1	1	**2**	1
t_3: `(A=2, B=2)`	2	2	2	2

- **Killed:** some test produces a different output.
- **Survived:** no test in the current suite kills it.
- **Equivalent:** no possible test can distinguish it from the original.

t_1 kills Δ_1 and Δ_2; Δ_3 is equivalent.

When Is a Mutant Killed?

A test t **kills** a mutant m when the output on P differs from the output on m.

test t	`Min`	Δ_1 `minVal=B`	Δ_2 `B>A`	Δ_3 `B<minVal`
t_1: `(A=1, B=2)`	1	**2**	**2**	1
t_2: `(A=2, B=1)`	1	1	**2**	1
t_3: `(A=2, B=2)`	2	2	2	2

- **Killed:** some test produces a different output.
- **Survived:** no test in the current suite kills it.
- **Equivalent:** no possible test can distinguish it from the original.

t_1 kills Δ_1 and Δ_2; Δ_3 is equivalent.

$$MS(T) = \frac{\text{mutants killed}}{\text{total mutants} - \text{equivalent mutants}} \times 100\%$$

Survived, or Equivalent?

A mutant that no test in the suite kills is either a **missing test** or an **equivalent mutant**. Consider Δ_3:

```
    minVal = A;                          if (B < minVal)
    if (B < A)          Δ3
```

Survived, or Equivalent?

A mutant that no test in the suite kills is either a **missing test** or an **equivalent mutant**. Consider Δ_3:

```
    minVal = A;
    if (B < A)          Δ3   if (B < minVal)
```

Infection would require `(B < A) != (B < minVal)`. The preceding statement assigns `minVal = A`, so this reduces to `(B < A) != (B < A)`, a contradiction: no input infects the state, and no test can kill Δ_3.

Kim, *Mutation Testing*, CS40508, KAIST · Ammann & Offutt, *Introduction to Software Testing*, 2nd ed., Ch. 9

Survived, or Equivalent?

A mutant that no test in the suite kills is either a **missing test** or an **equivalent mutant**. Consider Δ_3:

```
minVal = A;                          minVal = A;
if (B < A)          Δ3    if (B < minVal)
```

Infection would require `(B < A) != (B < minVal)`. The preceding statement assigns `minVal = A`, so this reduces to `(B < A) != (B < A)`, a contradiction: no input infects the state, and no test can kill Δ_3.

Equivalent mutants are excluded from the denominator of the mutation score. Deciding equivalence is undecidable in general, so survivors are triaged by hand — the principal cost of the method in practice.

Kim, *Mutation Testing*, CS40508, KAIST · Ammann & Offutt, *Introduction to Software Testing*, 2nd ed., Ch. 9

Mutation Operators

Operators either mimic typical programmer mistakes or force expressions towards values that tests commonly probe. A representative selection for a method-level analysis:

Operator	Modification	Example
ABS	absolute value insertion	`m*(o+p)` → `abs(m*(o+p))`
AOR	arithmetic operator replacement	`m*(o+p)` → `m+(o+p)`
ROR	relational operator replacement	`X <= Y` → `X > Y`
COR	conditional operator replacement	`X && Y` → `X || Y`
UOI	unary operator insertion	`(o+p)` → `-(o+p)`
SVR	scalar variable replacement	`m*(o+p)` → `o*(o+p)`

Kim, *Mutation Testing*, CS40508, KAIST · Ammann & Offutt, *Introduction to Software Testing*, 2nd ed., Ch. 9

Strong and Weak Killing: the RIP Model

For a test to kill a mutant, three conditions must hold in sequence:

1. **Reachability** — the test reaches the mutated location;

Strong and Weak Killing: the RIP Model

For a test to kill a mutant, three conditions must hold in sequence:

1. **Reachability** — the test reaches the mutated location;
2. **Infection** — execution of that location yields an incorrect internal state;

Kim, *Mutation Testing*, CS40508, KAIST · Ammann & Offutt, *Introduction to Software Testing*, 2nd ed., Ch. 9

Strong and Weak Killing: the RIP Model

For a test to kill a mutant, three conditions must hold in sequence:

1. **Reachability** — the test reaches the mutated location;
2. **Infection** — execution of that location yields an incorrect internal state;
3. **Propagation** — the incorrect state reaches the output.

Kim, *Mutation Testing*, CS40508, KAIST · Ammann & Offutt, *Introduction to Software Testing*, 2nd ed., Ch. 9

Strong and Weak Killing: the RIP Model

For a test to kill a mutant, three conditions must hold in sequence:

1. **Reachability** — the test reaches the mutated location;
2. **Infection** — execution of that location yields an incorrect internal state;
3. **Propagation** — the incorrect state reaches the output.

Weak killing requires reachability and infection; **strong killing** requires propagation as well.

Kim, *Mutation Testing*, CS40508, KAIST · Ammann & Offutt, *Introduction to Software Testing*, 2nd ed., Ch. 9

Strong and Weak Killing: the RIP Model

For a test to kill a mutant, three conditions must hold in sequence:

1. **Reachability** — the test reaches the mutated location;
2. **Infection** — execution of that location yields an incorrect internal state;
3. **Propagation** — the incorrect state reaches the output.

Weak killing requires reachability and infection; **strong killing** requires propagation as well.

Kim, *Mutation Testing*, CS40508, KAIST · Ammann & Offutt, *Introduction to Software Testing*, 2nd ed., Ch. 9

RIP Example

Δ_1 replaces `minVal = A` by `minVal = B`. Trace the state under t_2: `(A=2, B=1)`.

	Min	Δ_1	
`minVal = A / minVal = B`	`minVal = 2`	`minVal = 1`	infected
`if (B < A)`	`true`	`true`	
`  minVal = B;`	`minVal = 1`	`minVal = 1`	difference erased
`return minVal`	`1`	`1`	outputs agree

RIP Example

Δ_1 replaces `minVal = A` by `minVal = B`. Trace the state under t_2: `(A=2, B=1)`.

	Min	Δ_1	
`minVal = A / minVal = B`	`minVal = 2`	`minVal = 1`	infected
`if (B < A)`	`true`	`true`	
`minVal = B;`	`minVal = 1`	`minVal = 1`	difference erased
`return minVal`	`1`	`1`	outputs agree

Reachability and infection hold, propagation does not: t_2 kills Δ_1 **weakly** but not **strongly**.

RIP Example

Δ_1 replaces `minVal = A` by `minVal = B`. Trace the state under t_2: `(A=2, B=1)`.

	Min	Δ_1	
`minVal = A / minVal = B`	`minVal = 2`	`minVal = 1`	infected
`if (B < A)`	`true`	`true`	
`minVal = B;`	`minVal = 1`	`minVal = 1`	difference erased
`return minVal`	`1`	`1`	outputs agree

Reachability and infection hold, propagation does not: t_2 kills Δ_1 **weakly** but not **strongly**.

Under t_1: `(A=1, B=2)` the guard is false in both programs, the infected value survives to the `return`, and the outputs are 1 and 2 — a strong kill.

Mutation-Guided LLM-based Test Generation at Meta

Christopher Foster
Product Compliance and Privacy team, Meta Platforms
Menlo Park, USA

Abhishek Gulati
Product Compliance and Privacy team, Meta Platforms
Menlo Park, USA

Mark Harman
Product Compliance and Privacy team, Meta Platforms
London, UK

Inna Harper
Developer Infrastructure team, Meta Platforms
London, UK

Ke Mao
WhatsApp team, Meta Platforms
London, UK

Jillian Ritchey
Messenger team, Meta Platforms
New York, USA

Hervé Robert
Product Compliance and Privacy team, Meta Platforms
Menlo Park, USA

Shubho Sengupta
FAIR, Meta Platforms
Menlo Park, USA

Foster et al., Mutation-Guided LLM-based Test Generation at Meta, FSE, 2025