C/C++ · MULL · REAL PROJECTS

Mutation Testing

If I introduce a small error into the code, would my tests catch it?

OpenAlt 2026

David Cermak · Embedded developer

Agenda

- Why code coverage isn't enough
- From fault injection to mutation testing
- Mull: how it works and how to wire it up
- Three real projects, three different test libraries: Catch2, Unity, and check

your test is a code too ...and it can be wrong



Only two hard problems in CS:

- Cache invalidation,
- Naming things,
- and off-by-1 errors.

-- Leon Bambrick



Only two hard problems in CS:

- Cache invalidation,
- Naming things,
- and **off-by-1 errors**.
  - 👆 boundary mutations

PART 1 · THE PROBLEM

> “If I were to introduce a small error into the code, would my tests catch it?”

AI-assisted programming produces a lot more code

- Review: *is this code correct?*
- Tests: *does this code still work?*
- Mutation testing: asks: *would a test notice if this code were wrong?*

Many tests, weak suite → Tests exercise code, happy paths, but not the boundaries

Coverage measures execution, not detection

Line coverage → Was this line executed?
Branch coverage → Was this branch taken?
Mutation score → Would a test fail if this changed?

> 100% coverage is not 100% confidence.

High coverage, production bug → The line ran, but nothing asserted the result

A real protocol bug

```
/* WebSocket frame length: choose the RFC length form */
if (len <= 125) {           /* 7-bit length  */
    ws_header[header_len++] = (uint8_t)(len | mask_flag);
} else if (len < 65536) {  /* 16-bit, length byte 126 */
    /* ... */
}
```

Mull mutates `cxx_le_to_lt`: `<=` becomes `<`.

- A **125-byte** payload now takes the 16-bit path.
- A 200-byte test already covered the 16-bit branch.
- Nothing asserted the 125-byte boundary → the **mutant survived**.

Coverage was green. The wire format was wrong.

PART 2 · FROM FAULT INJECTION TO MUTATION TESTING

Fault injection: break it to understand it

Fault injection (FI) is the study of how systems behave under unusual stress.

- Induce faults in hardware: voltage drops, clock glitches, laser pulses, EM bursts, DRAM row hammering.
- Observe unintended values, unintended execution, crashes or leaked secrets.
- Used offensively: bypass secure boot, escalate privileges, extract keys.
- Used defensively: measure resilience before an attacker does.

> “Torturing chips until they submit or reveal secrets.”

The FI toolbox

Technique	Mechanism	Cost (est.)	Character
Clock / voltage glitch	Violate timing constraints	$50 – $600	Easy, affects the whole chip
EMFI	EM pulse near the die	$4k – $10k+	Local, no board modification
Laser FI	Ionizing radiation on a spot	$500 – $50k+	Extremely precise, needs decapsulation
RowHammer	Repeated DRAM row activation	Software-only	No physical access required

Reported success rates vary wildly: **~1.4%** for clock/voltage glitching, **~2%** for EMFI, **~100%** for laser with careful tuning (better measured as average time-to-success).

Fault models and abstraction levels

Fault models describe what the fault does:

`bit flip` · `bit set/reset` · `random byte`
`instruction skip` · `execution fault` · `stuck-at`

Abstraction levels describe where it lands:

```
physics  →  gates/registers  →  ISA/bit/byte  →  assembly  →  source code
```

- **bit flip** (circuit designer)
- **instruction skip** (ISA engineer)
- **control-flow hijack** (application developer)

Fault injection vs. Mutation testing

	Fault injection	Mutation testing
Target	Hardware / system security	Test-suite quality
Fault	Voltage, clock, laser, EMFI	One operator changed in code
Observable	Crash, wrong result, leaked key	A test fails. Or not
Goal	Assess resilience	Assess detection

PART 3 · MUTATION TESTING AND MULL

Mutation testing: the flow

1. Take the compiled program.
2. Apply **one** small semantic change: `<=` → `<`, `+` → `-`, `==` → `!=`.
3. Run the test suite once.
4. Repeat for every mutant.

Result	Meaning
Killed	At least one test failed → the suite noticed the change
Survived	Every test passed → the suite did not notice

Mutation testing

- Mutators are **deterministic**: the same code and rules produce the same mutants.
- No random inputs, no fuzzing, no production traffic.

This is fault injection aimed at the test suite instead of the hardware.

Mutation score = killed / (killed + survived)

Example: 7 killed out of 10 mutants → **70%**.

- Your suite detects 70% of the injected semantic changes.

It tells you

- your suite detects 70% of the injected semantic changes.

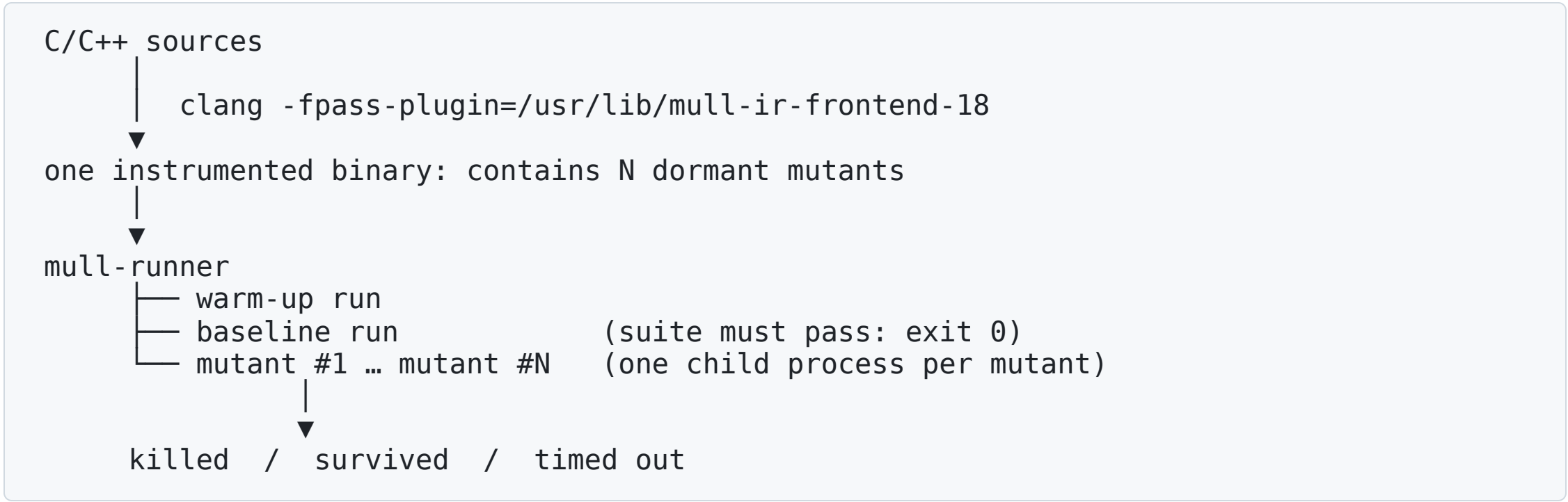
It does not tell you

- **Equivalent mutants** (cannot be killed): change not is observable.
- **Timed-out mutants** (counted as killed): timeout is a hang

Mull: Practical mutation testing for C/C++

- Open-source, LLVM-based mutation testing for **C and C++**.
- Works at the **LLVM IR** level; uses Clang’s AST to keep mutations tied to real source locations.
- A **Clang pass plugin** instruments the program at compile time.
- All mutants live in **one binary**, each hidden behind a conditional flag.
- `mull-runner` activates one mutant per run, in a child process.
- Supports LLVM **13 → 22**; reporters for IDE warnings, SQLite, JSON/HTML, and patches.

Mull: How the run works



- A **failing** test exits non-zero → mutant killed.
- A **passing** test exits zero → mutant survived.

Mull: Mutators

Group	Examples
`cxx_boundary`	`<=` → `<`, `<` → `<=`, `>=` → `>`, `>` → `>=`
`cxx_comparison`	`==` → `!=`, `<` → `>=`, `>=` → `<`, `>` → `<=`
`cxx_increment`	`x++` → `x--`, `--x` → `++x`
`cxx_arithmetic`	`+` → `-`, `-` → `+`, `*` → `/`, `/` → `*`, `%` → `/`
`cxx_bitwise`	`&` → `|`, `|` → `&`, `<<` → `>>`, `^` → `|`
`cxx_calls`	remove a `void` call, replace a scalar call with `42`

Start narrow: **boundary + comparison + increment + arithmetic**.
These map directly to the bugs that hurt in embedded C.

Mull: Configuration

```
# mull.yml
mutators:
  - cxx_boundary
  - cxx_comparison
  - cxx_increment
  - cxx_arithmetic

includePaths:
  - .*/src/core/ipv4/dhcp\.c$

excludePaths:
  - .*/test/.*
  - .*/contrib/.*
  - .*/src/apps/.*
```

Mull: Running it

```
# lwIP: rebuild the check port with the Mull plugin, run one suite
cd test/unit
./scripts/run_mull.sh dhcp

# -> mull_reports/lwip_dhcp.sqlite
sqlite3 mull_reports/lwip_dhcp.sqlite \
  "SELECT line_number, mutator, execution_status
     FROM mutant WHERE execution_status = 2 LIMIT 20;"
```

`execution_status`	Meaning
1	**Killed**: test failed
2	**Survived**: all tests passed
3	**Timed out**: killed by the runner’s timeout

PART 4 · THREE PROJECTS, ONE LOOP

Three projects

three test libraries: Catch2, Unity, and check

- **Catch2** (this demo), tcp_transport, score ~55%
- **Unity** + cmock, mDNS
- **check**, lwIP, DHCP unit test

> Mull integreation is transparent,
> no adaptation needed for different test frameworks

Demo

```
[info] Using config /home/david/repos/0xFFFFE/methods/mull/esp-idf/components/tcp_transport/host_test/mull.yml
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 1ms
[info] Applying file path filter (threads: 32)
       [################################] 232/232. Finished in 4ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 1ms
[info] Running mutants (threads: 32)
       [################################] 232/232. Finished in 80ms
[info] Results can be found at '/home/david/repos/0xFFFFE/methods/mull/esp-idf/components/tcp_transport/host_test/mull_reports/tcp_transport_ws.sqlite'
[info] Surviving mutants: 116
[info] Total execution time: 127ms
Wrote /home/david/repos/0xFFFFE/methods/mull/esp-idf/components/tcp_transport/host_test/mull_reports/tcp_transport_ws_survivors.txt (116 survivors, score 50.0%)

Report: /home/david/repos/0xFFFFE/methods/mull/esp-idf/components/tcp_transport/host_test/mull_reports/tcp_transport_ws.sqlite
Survivors (clickable): /home/david/repos/0xFFFFE/methods/mull/esp-idf/components/tcp_transport/host_test/mull_reports/tcp_transport_ws_survivors.txt
```

Demo (cont'ed)

```
/transport_ws.c:628:85: warning: Survived: Replaced != with == [cxx_ne_to_eq]
        if (data_ptr[0] != 0 || data_ptr[1] != 0 || data_ptr[2] != 0 || data_ptr[3] != 0 ||
                                                                                    ^

/transport_ws.c:639:75: warning: Survived: Replaced > with >= [cxx_gt_to_ge]
    if ((ws->frame_state.opcode & WS_OPCODE_CONTROL_FRAME) && payload_len > 125) {
                                                                          ^

/transport_ws.c:644:26: warning: Survived: Replaced > with >= [cxx_gt_to_ge]
        while (remaining > 0 && len > 0) {
                           ^

/transport_ws.c:644:26: warning: Survived: Replaced > with <= [cxx_gt_to_le]
```

PART 5 · TAKEAWAYS

Tool overview

Tool	Answers	Does not answer
Coverage	Did the line run?	Did we assert the result?
Sanitizers	Did it crash or hit UB?	Is the behavior correct?
Static analysis	Could this be wrong?	Would a test notice?
Mutation testing	Would a test catch this change?	Is the original code correct?

Tool overview (cont'ed)

Tool	Answers	Does not answer
Fuzzing	Can unexpected inputs trigger failures?	Is the behavior correct for all inputs?
Property-based testing	Does the behavior satisfy specified properties across generated inputs?	Are the properties themselves correct and complete?
Cyclomatic complexity	How many independent paths does the control flow have?	Are those paths tested or correct?
Formal verification	Does the implementation satisfy specified properties under stated assumptions?	Are the specification and assumptions correct?

Takeaway

> If I introduce a small error, will a test fail?

- Mutation testing **complements** coverage, sanitizers and static analysis.
- A survivor is a **lead**, not a verdict.
- Timeout are bugs -> unit tests should be fast
- Mutation score -> focused on corner cases

Links

- Mull project https://mull-project.com
  - documentation: https://mull-project.com/getting-started/primer
- mDNS using Mull: https://github.com/david-cermak/esp-protocols/blob/325020525605d1b564181703509fec405e8261cd/components/mdns/tests/host_unit_test/USING_MULL.md
- 0xFFFFE using Mull: https://github.com/david-cermak/0xFFFFE/tree/main/methods/mull
- SoK: A Beginner-Friendly Introduction to Fault Injection Attacks: https://arxiv.org/pdf/2509.18341