Finding the Gaps your Rust Tests Miss: Mutation Testing with mutest-rs

Presented at Rust Manchester, 2026-02-24

Zalán Lévai • @zalanlevai • https://mutest.rs

Who am I? What is this talk about?

- I am Zalán, a PhD student at The University of Sheffield working in the Testing Research Group.
- Creator and developer of mutest-rs (4+ years), a mutation testing tool for Rust programs.
- This talk will cover:
  - What mutation testing is, and how you can make use of it to improve your tests;
  - mutest-rs: how I created it, how it works, and the cool and interesting techniques I used to make it.

Who uses mutation testing?

- Google, Meta employ very limited form of mutation testing in CI.
- Peace of mind over tests.
- Required by some safety-critical certifications.

Learning outcomes

By the end of this talk, you will be able to:

1. Describe the fundamental principles of mutation testing;
2. Explain the benefits of using mutation testing for identifying functional testing gaps;
3. Use mutest-rs to efficiently perform mutation testing on your Rust test suites; and
4. Analyse the results of mutation testing:
    i. to find places that could benefit most from testing improvement, and
    ii. to find stale tests that could be pruned.

How do we deal with bugs in our code?

- All software has bugs.
- Rust eliminates some of these by design.
- Rust saves you from writing some tests, but semantic bugs remain.
- Rust does not save you from deleting important files, executing malicious code, or taking down an important service with a crash.
- Some problems can be caught with code reviews, fuzzers, traceability, but…
- Testing is still the answer in many cases.

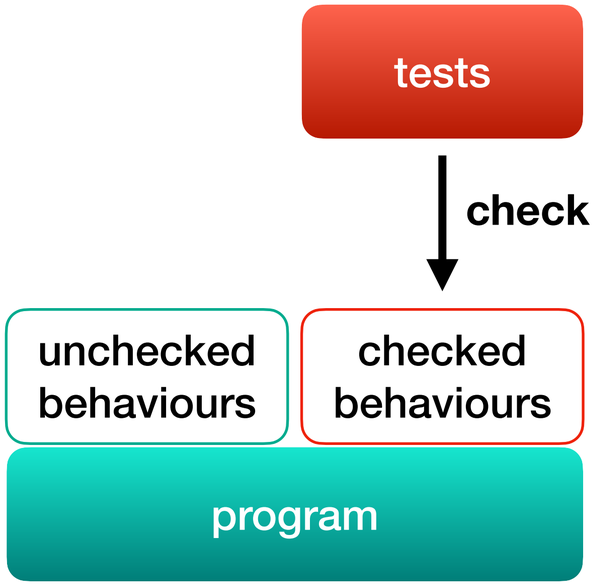
How can bugs slip past our tests?

- Because the tests are weak:
  - Focus on the happy path, thus missing the edge cases.
  - Forgot to check something about the result of the tested code.
  - Code changes can change the code paths and edge cases.
  - Tests need to change with the code.

What behaviours do our tests check?

The typical testing scenario

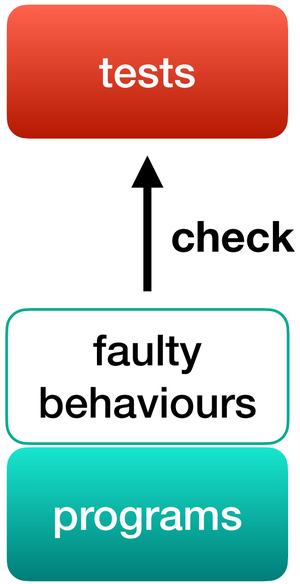
- Only successful if:
  - tests capture every requirement, AND
  - tests would fail if the program was incorrect.
- Unchecked behaviors exist without our tests telling us.
  - They could contain bugs now.
  - They can evolve to contain bugs in the future.



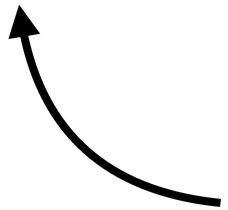
Finding behaviours that our tests do NOT check

The inverted testing scenario

- Invert the checking between tests and programs.
- Given various program behaviours, which ones do our tests allow through, and which ones does it flag as unwanted?
- Current behaviours all passing tests: derive faulty ones through changes.
- Use faulty program behaviours against our tests.



Mutation Testing



Intentional changes resulting in
faulty program behaviours

Mutations: small, subtle changes

- Mutations mimic the mistakes of experienced developers.
- Smaller bugs compound into bigger ones, so detecting the smaller bugs means we should detect the bigger ones.
- Our tests should ideally reject faulty behavior of mutations.
- In summary: Mutation testing is the process of using intentionally buggy programs as “test cases” for our tests, in order to find out test weaknesses.

Doesn’t coverage already do this?

- This function is entirely covered by tests, but no one checked whether some of the configuration options are set.

undetected **mutation 14: swap bitwise assignment operator `|=` for `&=`** in alacritty/src/cli.rs

```
 93      pub fn override_config(&mut self, config: &mut UiConfig) {
 94          #[cfg(unix)]
 95          if self.socket.is_some() {
 96              config.ipc_socket = Some(true);
 97          }
 98
 99          config.window.embed = self.embed.as_ref().and_then(|embed| parse_hex_or_decimal(embed));
100          config.debug.print_events |= self.print_events;
101          config.debug.log_level = max(config.debug.log_level, self.log_level());
102          config.debug.ref_test |= self.ref_test;
             config.debug.ref_test &= self.ref_test;
103
104          if config.debug.print_events {
105              config.debug.log_level = max(config.debug.log_level, LevelFilter::Info);
106          }
107
```

Applying Mutation Testing to Rust

Mutating Rust code: mutest-rs

- Build on rustc for compiler analysis.
  - Only “interesting” mutations.
- Call graph from test functions to “program” functions.
  - Only run relevant tests for each mutation.
- Compile into a single binary.
- Timeouts: non-terminating mutations.

Some mutations are unsafe…

```
+ let buffer = [0_u8; 1024];
- let buffer = [0_u8; 0];

  let el = unsafe { buffer.get_unchecked(1) };
```

How fast is mutest-rs?

	Tests	Mutations
alacritty	79	1,808

Non-exhaustive evaluation: mutations are skipped past as soon as any test detects them. 10 core machine, with RUST_TEST_THREADS=16.

How fast is mutest-rs?

	Tests	Mutations	Processes (`--isolate=all`)
alacritty	79	1,808	64.1 s

Non-exhaustive evaluation: mutations are skipped past as soon as any test detects them. 10 core machine, with RUST_TEST_THREADS=16.

How fast is mutest-rs?

	Tests	Mutations	Processes (`--isolate=all`)	Threads (default)
alacritty	79	1,808	64.1 s	11.8 s

Non-exhaustive evaluation: mutations are skipped past as soon as any test detects them. 10 core machine, with RUST_TEST_THREADS=16.

How fast is mutest-rs?

	Tests	Mutations	Processes (`--isolate=all`)	Threads (default)	`--parallel-mutants`
alacritty	79	1,808	64.1 s	11.8 s	6.1 s

Non-exhaustive evaluation: mutations are skipped past as soon as any test detects them. 10 core machine, with RUST_TEST_THREADS=16.

How fast is mutest-rs?

	Tests	Mutations	Processes (`--isolate=all`)	Threads (default)	`--parallel-mutants`
alacritty	79	1,808	64.1 s	11.8 s	6.1 s
rustls	227	1,825	171.5 s	19.0 s	11.0 s

Non-exhaustive evaluation: mutations are skipped past as soon as any test detects them. 10 core machine, with RUST_TEST_THREADS=16.

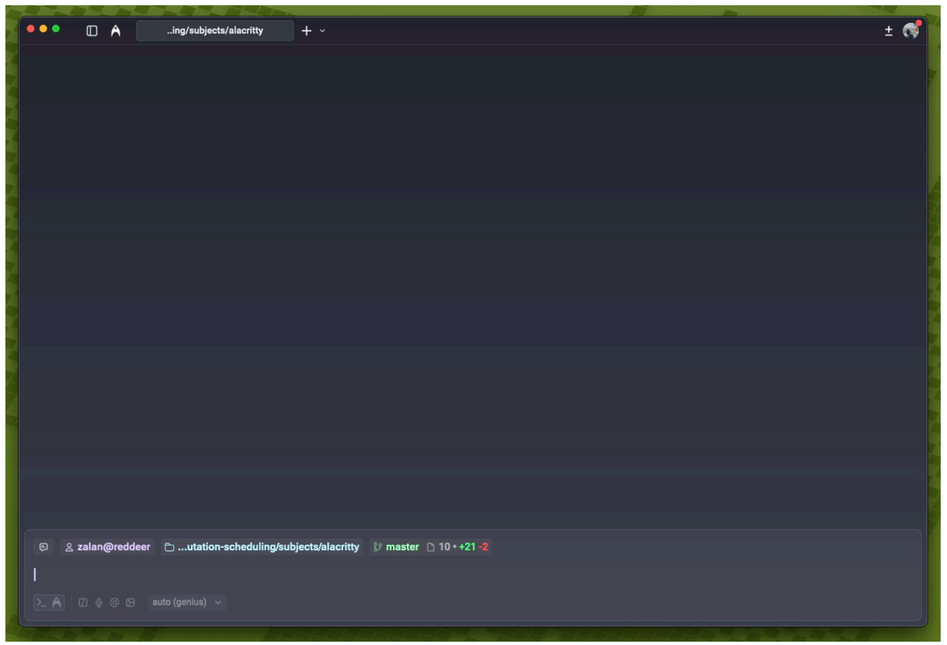
How fast is mutest-rs?

	Tests	Mutations	Processes (`--isolate=all`)	Threads (default)	`--parallel-mutants`
alacritty	79	1,808	64.1 s	11.8 s	6.1 s
rustls	227	1,825	171.5 s	19.0 s	11.0 s
gleam-core	4,285	4,556	*t/o*	154.2 s	141.0 s

Non-exhaustive evaluation: mutations are skipped past as soon as any test detects them. 10 core machine, with RUST_TEST_THREADS=16.

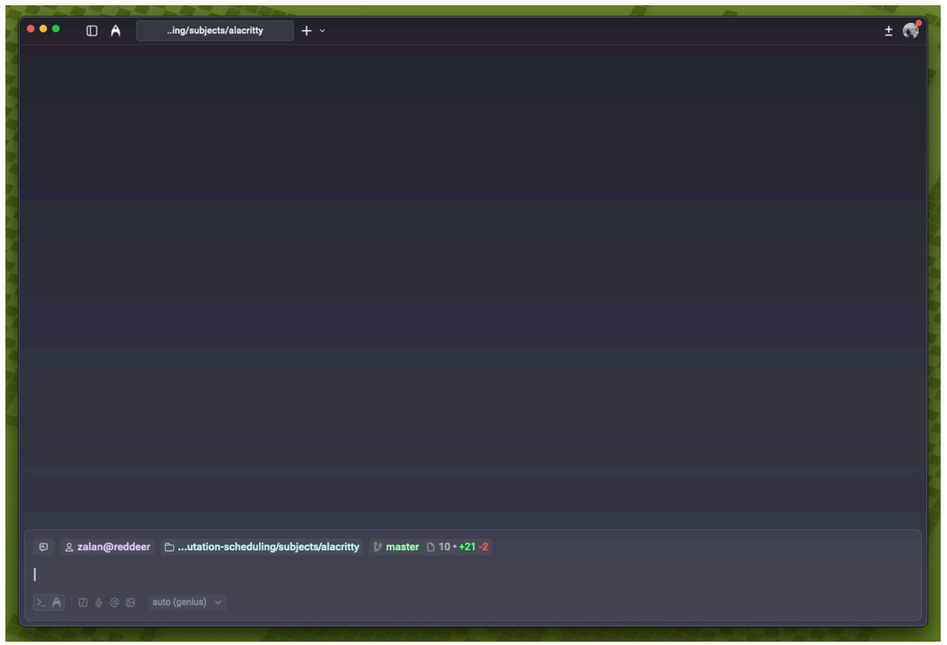
Using mutest-rs

- mutest-rs is an end-to-end mutation testing tool for Rust projects.
- Run `cargo mutest run` like you would `cargo test`.
- mutest-rs generates JSON metadata for analysis later.
  - Web-based inspector: `cargo mutest inspect`.



Using mutest-rs

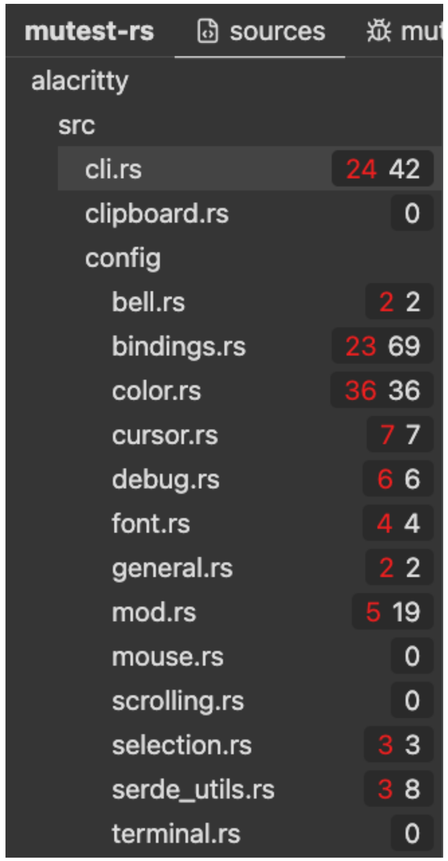
- mutest-rs is an end-to-end mutation testing tool for Rust projects.
- Run `cargo mutest run` like you would `cargo test`.
- mutest-rs generates JSON metadata for analysis later.
  - Web-based inspector: `cargo mutest inspect`.



Inspecting Mutations using the Web Inspector

How to start improving test coverage?

- Start with whatever you deem the most critical components of your system.
- Consider picking an important source file with undetected mutations, and writing new tests or adjusting existing ones.
- Make sure that you do not overfit your tests to mutations: mutations are only a guide for test development.
- Mutation testing does not have to be an all or nothing approach, and can be applied iteratively to consistently improve testing.



trace of 2 calls from `cli::tests::dynamic_title_ignoring_options_by_default` to `<config::cursor::Cursor as std::default::Default>::default` with mutation 102

alacritty/src/cli.rs: `cli::tests::dynamic_title_ignoring_options_by_default` calls `<config::ui_config::UiConfig as std::default::Default>::default` in 1 place

```
441     fn dynamic_title_ignoring_options_by_default() {
442         let mut config = UiConfig::default();
```

alacritty/src/config/ui_config.rs: `<config::ui_config::UiConfig as std::default::Default>::default` calls `<config::cursor::Cursor as std::default::Default>::default` in 1 pl...

```
43  #[derive(ConfigDeserialize, Serialize, Default, Clone, Debug, PartialEq)]
44  pub struct UiConfig {
45      /// Miscellaneous configuration options.
46      pub general: General,
47
48      /// Extra environment variables.
49      pub env: HashMap<String, String>,
50
51      /// How much scrolling history to keep.
52      pub scrolling: Scrolling,
53
54      /// Cursor configuration.
55      pub cursor: Cursor,
```

alacritty/src/config/cursor.rs: `<config::cursor::Cursor as std::default::Default>::default` with mutation 102: negate boolean expression

```
29      fn default() -> Self {
30          Self {
31              thickness: Percentage::new(0.15),
32              unfocused_hollow: true,
                unfocused_hollow: !true,
```

How can you find and prune stale tests?

- Mutation testing is not only about adding tests.
- Good indicators of stale tests that “do not carry their own weight”:
  - tests that do not detect mutations, i.e., fault coverage of zero, or otherwise very low;
  - tests that detect the same mutations as other tests do: some of these are likely redundant.

mutest-rs | sources | mutations 1808 | tests 79 | traces

test	reachable mutations	ran	coverage	undetected	detected	timed out	crashed
input::tests::triple_click_failed ›	538	499	2.81%	485	14	0	0
message_bar::tests::add_newline_for_button ›	150	48	0.00%	48	0	0	0
message_bar::tests::appends_close_button ›	150	50	0.00%	50	0	0	0
message_bar::tests::empty_with_shortterm ›	150	59	1.69%	58	1	0	0
message_bar::tests::hide_button_when_too_narrow ›	150	145	24.83%	109	36	0	0
message_bar::tests::hide_truncated_when_too_narrow ›	150	66	16.67%	55	11	0	0
message_bar::tests::multiline_close_button_first_line ›	150	74	2.70%	72	2	0	0
message_bar::tests::pop ›	7	1	100.00%	0	1	0	0
message_bar::tests::remove_duplicates ›	7	6	100.00%	0	6	0	0
message_bar::tests::remove_target ›	13	7	85.71%	1	6	0	0
message_bar::tests::splits_on_length ›	150	56	0.00%	56	0	0	0
message_bar::tests::splits_on_newline ›	150	111	7.21%	103	8	0	0
message_bar::tests::strip_whitespace_at_linebreak ›	150	72	6.94%	67	5	0	0
message_bar::tests::truncates_long_messages ›	150	55	16.36%	46	9	0	0
message_bar::tests::wrap_on_words ›	150	49	0.00%	49	0	0	0
message_bar::tests::wrap_with_unicode ›	150	102	28.43%	73	29	0	0
migrate::tests::migrate_empty ›	1	1	0.00%	1	0	0	0
migrate::tests::move_values ›	0						
renderer::text::builtin_font::tests::builtin_line_drawing_glyphs_coverage ›	843	812	5.54%	767	41	4	0
renderer::text::builtin_font::tests::builtin_powerline_glyphs_coverage ›	843	840	3.69%	809	31	0	0
string::tests::into_shortened_with_shortener ›	47	47	80.85%	9	38	0	0
string::tests::into_shortened_without_shortener ›	47	10	30.00%	7	3	0	0

When should I use mutation testing?

- Mutation testing is general-purpose.
- If you are modifying software built against a strict specification, most code modifications would have undesired consequences.
- Having a strong, strict test suite can guide developers towards the right code changes.

What’s next for mutest-rs?

- Review mode for inspector: help prioritise testing.
- Experimental support for mutation testing embedded firmware on-device.
- Improve reliability by working together with the Rust Compiler Team.

Get mutating!

- mutest-rs is open source and available at https://mutest.rs.
- Experiment → Improve → Consider CI
- I would love to hear:
  - about any issues,
  - about the bugs and test gaps you have discovered,
  - your ideas and questions.