mutest-rs: Flexible, Efficient Mutation Analysis Tool for Rust Programs, using Extensive Static Analysis

Zalán Lévai
University of Sheffield

Donghwan Shin
University of Sheffield

Phil McMinn
University of Sheffield

***Abstract*—Determining the adequacy of software tests, and where testing gaps might lie, is crucial for improving and maintaining the strength of test suites. Mutation analysis facilitates this by evaluating tests against generated program faults; however, no mature mutation analysis tooling exists for the safety-focused Rust systems programming language, to date. This paper introduces mutest-rs, a mature, end-to-end mutation analysis tool for Rust programs that is based on extensive static program analysis, and integrates directly with the `rustc` Rust compiler. Our tool overcomes the numerous challenges of generating valid Rust code mutants, and does so efficiently through a Rust-specific meta-mutant approach. Our open-source tool, mutest-rs, is available online at https://mutest.rs. A video demonstrating mutest-rs is available at https://youtu.be/8yEYAU6P63I.**

I. Introduction

Testing for programs written in the safety-focused Rust programming language may have unknown gaps due to a lack of mature mutation analysis tooling. Mutation analysis reveals the strength of test suites by running generated faulty programs, known as mutants, against them, assessing whether the tests detect the differences in behavior [3], [4], [5]. This lack of tooling leaves Rust developers without important insights. Generating valid, compilable code mutations for Rust programs is complex, and requires extensive program analysis. For practical applicability, mutation analysis must be implemented using efficient methods, which is made more challenging due to the strictness of the Rust language.

In this paper, we introduce mutest-rs, a flexible, efficient, mature mutation analysis tool for Rust programs. Unlike existing, simplistic solutions based purely on potentially invalid syntactical changes, we designed mutest-rs to use the `rustc` Rust compiler for program analysis, to ensure the generation of valid, compilable code mutations (Section III-C). These existing Rust solutions only support a very limited subset of mutation analysis, while mutest-rs's dynamic mutation operator model is flexible and extensible by design, supporting the implementation of complex mutation operators (Section III-C1). Our tool avoids the prohibitively expensive overheads associated with compiling individual mutant programs by combining all mutations into a single meta-mutant [6], [7] using a novel Rust-specific approach (Section III-C2). Our tool is able to omit mutations not reachable from tests entirely, and run only the relevant test cases for each mutation, by building a comprehensive static call graph of the program (Section III-B). With this, mutest-rs can also aid users, showing them the relevant snippets of code for each undetected mutation.

To demonstrate the effectiveness of our tool, we perform an empirical evaluation comparing mutest-rs with cargo-mutants on top 8 Rust crates (i.e., packages), discussing the differences in generated mutations, showing a two orders of magnitude reduction in the time taken to run mutation analysis (Section IV).

We encourage the use of mutest-rs for Rust-based empirical mutation analysis studies. We have conducted our own empirical mutation analysis studies using mutest-rs on the open-source Rust ecosystem in our previous papers [8], [9], [10], as well as using it for ongoing research. Our open-source [1] Rust mutation analysis tool, mutest-rs, is publicly available online for use by researchers and developers alike at https://mutest.rs. We make a demo repository of subjects available online [2].

Zalán Lévai is supported by the EPSRC grant EP/W524360/1. Donghwan Shin is supported by the EPSRC grant EP/Y014219/1. Phil McMinn is supported, in part, by the EPSRC grant "Test FLARE" (EP/X024539/1).

II. Background

A. Mutation Analysis and the Rust Programming Language

Fault detection is a fundamental requirement of rigorous software testing. Mutation analysis is the process of using intentionally faulty programs, known as mutants, with generated program faults, known as mutations, against a program's test suite to determine its fault-detecting ability, and to find areas of improvement [3], [4], [5].

Rust is a safety-focused, memory-safe systems programming language seeing increasing adoption throughout the industry, including in safety-critical settings [11], [12]. The language has built-in support for testing, and tests can be declared by annotating functions with the `#[test]` attribute. The build system of Rust, Cargo, supports this by providing a `cargo test` command to run tests in Cargo projects.

There are numerous challenges to applying mutation analysis to Rust programs. First, the strictness of the language, through its unique borrowing rules and its affine type system, means that many syntactical mutations are caught by the compiler, and makes the generation of valid program mutations difficult [9]. Second, Rust being a compiled language with extensive code analysis capabilities, effective mutation analysis for it requires minimizing the overhead of repeated compilations of the program. Therefore, placing mutations behind conditionally branching code in a single mutant program that can be controlled at runtime, similar to Untch et al.'s meta-mutant technique [6], [7], is essential. Uniquely, Rust requires developers to delineate operations that could cause Undefined Behavior (UB) with an `unsafe` block (or other `unsafe` context). This gives an opportunity for mutation analysis to

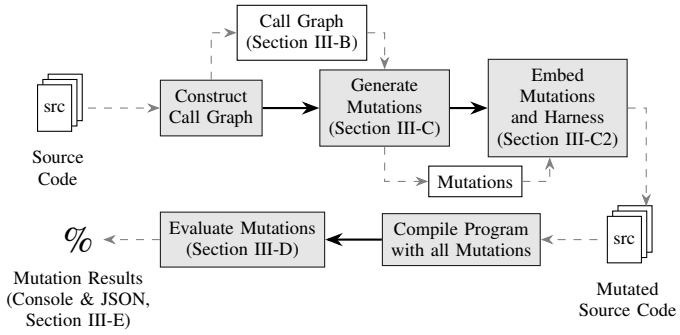

Fig. 1. High-level overview of the sequence of steps taken by mutest-rs to generate and evaluate mutations for a Rust crate.

consider how mutations influence existing program behavior, potentially introducing Undefined Behavior into the mutant.

B. Related Tools: mutagen and cargo-mutants

To date, two relatively limited mutation analysis tools exist for the Rust language: mutagen and cargo-mutants.

mutagen: Bogus's mutagen [13] was an early attempt at mutation analysis for Rust programs. It generates mutations purely based on the Abstract Syntax Tree (AST) of the Rust code, ignoring program semantics, such as types. Therefore, mutations are not guaranteed to be valid Rust code. Mutations are implemented as runtime functions rather than direct code modifications, and can alter the behavior of binary and unary expressions, remove function calls, and change literal values.

cargo-mutants: Pool's cargo-mutants [14] is a maintained mutation analysis tool for Rust programs based on AST transformations. Similar to mutagen, cargo-mutants does not perform program analysis, and operates purely on syntax, therefore generating many invalid Rust code mutants. Unlike mutagen, however, cargo-mutants directly modifies Rust code based on its AST, and generates every mutant as individual copies of the original source code. The primary mutations produced by cargo-mutants are analogous to extreme mutation analysis, replacing entire function bodies with set values based on their return type annotation. In addition, cargo-mutants supports mutating binary and unary operations, and removing `match` arms (analogous to `case` sections in `switch` statements). Because cargo-mutants makes copies of the source code to generate and build every mutant individually, it incurs significant compilation overheads that grow with the number of mutations, and with program size. To evaluate the mutations, cargo-mutants runs each built mutant as a separate process, and runs the entire test suite for each mutant until a test fails. It does not support building a mutation detection matrix that details the detection outcomes for each test case, and only considers the overall detection of each mutation.

III. Design and Implementation

A. Overview

We designed mutest-rs to perform mutation analysis based on extensive static analysis from the `rustc` Rust compiler, and to do so efficiently using only a single compilation encompassing all generated mutations. This both greatly improves the utility of the generated mutations for test improvement, and reduces the time it takes to perform mutation analysis substantially. For user familiarity, we designed mutest-rs to be analogous to the built-in `cargo test` command to build and run all unit and integration tests in a Cargo project. Thus, we ship mutest-rs as a command-line utility that can be invoked using the `cargo mutest run` command, supporting all of the regular Cargo flags, as well as many more options that are detailed throughout Section III. Running mutest-rs performs end-to-end mutation analysis: it analyzes the program and its tests (Section III-B) and generates mutations (Section III-C), but it also compiles all mutations into a single Rust program (Section III-C2), and uses its mutation harness runtime to evaluate the test suite against the mutations (Section III-D). The console output of mutest-rs provides information about each mutation as it is being evaluated, with an overview at the end, but mutest-rs also writes detailed JSON output files for the entire process (Section III-E). Figure 1 shows an overview of the steps mutest-rs takes to generate and evaluate mutations for a Rust crate (Rust's notion of a library or binary unit). The mutest-rs project consists of three main components, each written in Rust itself:

- `mutest-driver`: The code generation component of mutest-rs that performs code analysis, mutation generation, and meta-mutant binary generation, making use of the `rustc` Rust compiler APIs to drive code analysis, generation, and compilation to perform mutant generation. It stands in place of the regular `rustc` Rust compiler when using the `cargo mutest` commands.
- `mutest-runtime`: A library that implements a generic mutation harness based on metadata `mutest-driver` generates for each meta-mutant. The Rust meta-mutants generated by `mutest-driver` rely on it to drive the test and mutant evaluation.
- `cargo-mutest`: A Cargo-compatible command line interface that understands Cargo projects and their layouts, and is the main user interface used to invoke mutest-rs. It runs `mutest-driver` for each selected Rust crate.

B. Building a Call Graph for Reachability Analysis

The first step of mutest-rs, after identifying the test cases in the Rust crate, is to build a detailed, program-wide static call graph starting from the tests. This call graph is important, as it informs the rest of mutest-rs's mutation analysis. The generated call graph is fundamental to mutest-rs's efficiency: mutest-rs only generates mutations that are reachable from at least one test case (all other functions are marked as not covered by tests), and it only evaluates the test cases relevant to any given mutation. This call graph can also inform test development, however: users can use this information to focus on specific call traces within the program between a particular test case and a mutation, helping them understand why a particular mutation was not detected. The key to constructing a static call graph is tracking monomorphized instantiations of generic definitions, carrying over type information from the calling context to resolve subsequent function calls.

TABLE I
IMPLEMENTED MUTATION OPERATORS.

Mutation Operator	Description
ArgDefaultShadow	Replace function argument with default value[1]
BitOpOrAndSwap	Replace bitwise OR with bitwise AND, and vv.
BitOpOrXorSwap	Replace bitwise OR with bitwise XOR, and vv.
BitOpShiftDirSwap	Replace bitwise LSH with bitwise RSH, and vv.
BitOpXorAndSwap	Replace bitwise XOR with bitwise AND, and vv.
BoolExprNegate	Negate Boolean expression
CallDelete	Delete function call, replace with default value[1]
CallValueDefaultShadow	Replace function call result with default value[1]
ContinueBreakSwap	Replace `continue` with `break`, and vv.
EqOpInvert	Invert equality operator
LogicalOpAndOrSwap	Replace logical `&&` with logical `||`, and vv.
OpAddMulSwap	Replace addition with multiplication, and vv.
OpAddSubSwap	Replace addition with subtraction, and vv.
OpDivRemSwap	Replace division with modulo, and vv.
OpMulDivSwap	Replace multiplication with division, and vv.
RangeLimitSwap	Change inclusivity of range's upper bound
RelationalOpEqSwap	Change relation operator's bound wrt. equality
RelationalOpInvert	Invert relational operator

[1]The default value for the type as defined by the implementation of the `Default` trait; i.e., for type `T` the value of `<T as Default>::default`.

C. Generating Mutations

The goal of mutest-rs is to inform users about where more testing needs to take place. Therefore, because our call graph already gives information about function-level test coverage, mutest-rs only generates mutations in functions that are reachable by at least one test case. For unreachable functions, mutest-rs still informs the user about their lack of test coverage, but does not pollute the analysis with mutations that are known to be undetectable by the current test suite. To generate mutations, mutest-rs uses an approach that scales to the largest variety of mutations by design: a dynamic, extensible set of mutation operators, applying them to every possible AST node within reachable function definitions. To ensure that all generated mutations remain valid Rust code, mutest-rs makes use of the extensive compiler analysis information that comes from it being based on the `rustc` Rust compiler. For example, addition cannot be changed to a subtraction for all operands, and requires the corresponding `Sub` operator trait (i.e., interface) to be defined for the operand types. It must also return a value of the same type as the original operation.

1) Built-in and Custom Mutation Operators: Mutation operators are the set of rules that define what mutations are generated for any given pattern of code. In mutest-rs, we model mutation operators as simple functions representing code transformation rules: taking in a location in the program's AST as input (alongside accessors to static analysis information), and outputting one or more corresponding mutations made out of replacement AST nodes, if applicable. To date, mutest-rs implements 18 mutation operators. Table I lists these mutation operators, most of which are standard mutation operators derived from other mutation analysis tools, though adapted to Rust's language rules and idioms. Each mutation operator can be enabled or disabled, or otherwise configured.

We designed mutest-rs, with its generic mutation operator model, to support custom mutation operators. Developers can define new, custom mutation operators by declaring a new type that implements the `Operator` trait (Rust's notion of an interface). Mutation operators have full access to the compiler's analysis, and can generate code replacements not just for the AST node they were applied against, but any other node in the body of the function. This includes generating higher-order mutations by generating replacements for multiple different AST nodes under the same mutation. Figure 2 shows an example definition of a mutation operator: a simplified version of the built-in BoolExprNegate mutation operator.

```
// Operator, optionally with configuration options.
struct BoolExprNegate;
// Implement the `Operator` interface.
impl Operator for BoolExprNegate {
    fn try_apply(&self, mcx: &MutCtxt) -> Mutations {
        // Access original node and compiler analysis.
        let MutCtxt { location, tcx, .. } = mcx;
        // Filter to expression nodes.
        let MutLoc::FnBodyExpr(expr, f) = location else {
            return Mutations::none();
        };
        // Get type information for the expression.
        let typeck = tcx.typeck_body(f);
        let expr_ty = typeck.expr_ty(expr);
        // Ignore non-boolean expressions.
        if expr_ty != tcx.types.bool {
            return Mutations::none();
        }
        // Create negated expression.
        let negated_expr = mk_not_expr(expr);
        // Return mutation replacing original expression.
        Mutations::new_one(vec![
            Substitution(
                SubstLoc::Replace(expr),
                Substitute::AstExpr(negated_expr),
            )
        ])
    }
}
```

Fig. 2. Custom mutation operators can be defined as simple Rust functions mapping original AST nodes (`location`) to their mutated counterparts, if applicable. Mutation operators have access to program analysis through the `mcx: MutCtxt` argument. Some expressions are simplified for brevity.

2) Embedding Mutations into a single Rust program: While compiling each mutation as a separate, standalone mutant is simple, it incurs a significant overhead that grows with the number of mutations. Instead, mutest-rs combines all mutations into a single Rust meta-mutant program that can be compiled once, placing code replacements from mutations behind conditional expressions. This is only possible because all mutations generated by mutest-rs are valid Rust code. Otherwise, a single invalid mutation would prevent the compilation of all mutations in the Rust meta-mutant.

Figure 3 shows an example of mutest-rs combining the code of multiple mutations (and the original expression) into a single conditionally branching expression. The conditional code paths generated by mutest-rs make use of Rust's `match` expressions, which are analogous to the switch-case statements found in C-like programming languages, but can appear in any expression or statement position, including in deeply nested contexts. A mutation harness can control the code path to take at runtime using the `ACTIVE_MUTANT_HANDLE` global variable, which mutest-rs transparently injects into the generated

```
        mem::size_of::<f32>() as u32 * 8
                          ↓
match ACTIVE_MUTANT_HANDLE.subst_at(293) {
    Some(subst) if subst.mutation.id == 116 =>
        mem::size_of::<f32>() as u32 + 8,
    Some(subst) if subst.mutation.id == 117 =>
        mem::size_of::<f32>() as u32 / 8,
    _ => mem::size_of::<f32>() as u32 * 8,
}
```

Fig. 3. Example of a `match`-based conditional expression used to embed substituted expression nodes alongside the original. Derived from code generated for `hashbrown`, illustrating the mutations of a memory offset calculation.

program. To make mutation switching possible with a generic harness, `mutest-driver` generates these expressions, and corresponding metadata that describes each mutation and the appropriate configuration of each generated conditional expression. It can then pass these as data inputs to the generic mutation harness we implemented in `mutest-runtime`.

3) Mutations and Unsafe Code: Rust, being a systems programming language, has low-level constructs that could result in Undefined Behavior, if used improperly. However, uniquely amongst systems programming languages, it also requires that these operations appear in explicitly annotated `unsafe` blocks. Because these operations are, by definition, not compiler-checked, it makes them fragile against code modifications, especially from faulty mutations. One such mutation could crash or corrupt the mutation analysis process. However, these mutations could still be important to analyze. To solve this, we designed mutest-rs to distinguish mutations that could influence the behavior of any `unsafe` code section either directly by changing the unsafe code, or indirectly by changing its context or the code it calls out to. The mutation harness always executes these "unsafe" mutations and their corresponding test cases in separate processes to protect from Undefined Behavior. In addition, users can choose to exclude them from the mutation analysis entirely.

4) Supporting Integration Tests: Rust, specifically its build system Cargo, supports defining integration tests, which are placed next to the crate source files in the `tests/` directory. Cargo compiles each such integration test separately from the main target library or binary crate they are testing. Because these integration tests are compiled as standalone units, independently from the corresponding program crate, mutest-rs has to analyze them separately, and needs to assemble information from the test and the program crate to generate its integration test-specific meta-mutant. To do this, mutest-rs first compiles the program crate normally for the purposes of cross-crate code analysis later. Then, when it is processing an integration test crate, mutest-rs inspects its test cases, and builds a cross-crate call graph, selecting functions in the "remote" program crate for mutating. Afterwards, for the specific integration test crate, mutest-rs recompiles the program crate based on this call graph information into an integration test crate-specific meta-mutant. Finally, it links the integration test crate with the specific meta-mutant, thereby generating the complete testable meta-mutant binary. It then executes this, as normal, to perform the mutation analysis for the integration tests.

D. Evaluating Mutations

Once `mutest-driver` generates and builds the Rust meta-mutant program out of its generated mutations, it executes it to perform the evaluation of the test suite against the mutants. The generated binary uses the generic `mutest-runtime` mutation harness, which drives the evaluation based on the generated mutant and test metadata. The mechanism used by `mutest-runtime` to evaluate mutations is simple by design: for each mutation, the runtime can set the `ACTIVE_MUTANT_HANDLE` to the corresponding mutant descriptor, thus enabling the appropriate mutated code paths. Then, it can run the relevant tests according to the configured options. This approach makes it suited for implementing more complex execution techniques, such as Dynamic Mutation Scheduling [10]. The mutest-rs runtime by default evaluates mutations and their test cases efficiently by using threads within the process. Alternatively, users can configure mutest-rs to run all mutated test cases in isolated processes using the `--isolate=all` flag. As mentioned previously in Section III-C3, "unsafe" mutations, which could cause Undefined Behavior, are always executed in isolated processes, one for each test case. To reduce review effort and analysis times, mutest-rs only runs tests that can reach the mutations. In addition, by default, it stops evaluating tests at the first failing test case for each mutation, as that is enough to confirm mutation detection. The mutest-rs runtime also supports exhaustive mutation analysis, where all relevant test cases are evaluated regardless of a test case already detecting the mutation. Users can enable this using the `--exhaustive` flag, producing a complete test–mutation detection matrix. As the evaluation proceeds, mutest-rs prints output to the console for each mutation, with an overview at the end.

E. JSON Output

Besides the console output, mutest-rs writes extensive JSON output for each tested crate, with individual files detailing the discovered tests (`tests.json`), the test call graph (`call_graph.json`), the mutated functions and the generated mutations (`mutations.json`), and the evaluation and detection of the mutations broken down for each test case (`evaluation.json`). In addition, mutest-rs can output a "stream" of testing events throughout mutation evaluation (`evaluation.json`**`l`**). The JSON output includes extensive metadata about the generated mutations, with source location information and replacement code snippets. This machine-readable output is especially useful to researchers and practitioners building custom tooling around mutest-rs.

F. Current Technical Limitations

Because the `rustc` compiler APIs that mutest-rs builds on do not hold stability guarantees, we always have to build mutest-rs against a very specific version of the Rust compiler toolchain, such as `nightly-2025-11-26`, and have to make occasional upgrades manually. In addition, mutest-rs's code generation uses some custom code for use cases that are currently not well-supported by the compiler APIs.

TABLE II
COMPARISON OF THE GENERATED MUTATIONS, AND THE TOTAL TIME TAKEN TO PERFORM MUTATION ANALYSIS.

		Mutations			Total Time Taken (Speedup)					
		cargo-mutants		mutest-rs	cargo-mutants	mutest-rs				
Crate	Tests	Total	Valid	Reachable[1]	Time	Processes (`--isolate=all`)		Threads (default)		Parallel Mutants [10] (`--parallel-mutants`)	
`alacritty`	79	3,670	3,365	1,808	6,694 s	71.3 s	(94×)	19.0 s	(352×)	**13.5 s**	(496×)
`bevy_math`	276	5,784	4,737	5,109	6,380 s	255.3 s	(25×)	**8.3 s**	(769×)	8.6 s	(742×)
`chrono`	274	5,011	3,538	3,239	14,550 s	243.3 s	(60×)	35.6 s	(409×)	**35.2 s**	(413×)
`hashbrown`	110	1,224	887	291	5,405 s	9.0 s	(601×)	2.6 s	(2079×)	**2.4 s**	(2252×)
`rand`	90	762	653	1,024	790 s	52.8 s	(15×)	33.2 s	(24×)	**12.9 s**	(61×)
`regex-syntax`	147	1,820	1,576	1,916	4,430 s	234.7 s	(19×)	71.7 s	(62×)	**63.3 s**	(70×)
`rustls`	227	3,616	2,506	1,825	17,620 s	179.3 s	(98×)	26.8 s	(657×)	**19.0 s**	(927×)
`url`	66	1,321	1,006	1,753	819 s	106.0 s	(8×)	**7.6 s**	(108×)	8.0 s	(102×)

[1]Mutations in mutest-rs are all valid, compilable Rust code, and are only generated for parts of the program code that are reachable by tests.

The mutest-rs project is extensively and continuously tested, similar to the compiler itself, and has been used in large empirical studies across the open-source Rust ecosystem [8], [9], [10], but some mutest-rs specific bugs might still remain.

IV. EVALUATION

To empirically evaluate the effectiveness and the efficiency of our mutest-rs Rust mutation analysis tool, we ran mutest-rs and cargo-mutants — the other actively maintained Rust mutation analysis tool — on a sampling of top 8 open-source Rust crates, comparing their generated mutations, the fidelity of their result outputs, and the total time taken to perform mutation analysis. We ran mutest-rs with three different configurations: using process isolation for all mutant test cases, using the default option of threaded test evaluation, and using our Dynamic Mutation Scheduling technique [10]. Table II lists the 8 crates, as well as the results of our evaluation.

There are two key differences between the two tools' approach. First, the existing cargo-mutants tool generates a considerable number of invalid mutations caught by the compiler, while all of mutest-rs's mutations are valid and reachable by tests. Second, cargo-mutants compiles each mutant separately, while mutest-rs only performs a single compilation regardless of the number of mutations. From the results, we can see that both tools generate numbers of mutations in the same order of magnitude; however, about 8% to 31% (median of 21%) of cargo-mutants' mutations are not valid. In comparison, all of mutest-rs's mutations are valid, reachable by tests, and also come from a more diverse set of mutation operators applied to more locations in the program. When looking at the total time taken to perform mutation analysis, we see a reduction of two orders of magnitude in all cases with mutest-rs over cargo-mutants. The longest measured runtime of cargo-mutants is 4.9 hours, while in the case of mutest-rs the maximum is only *4.3 minutes*, with the default approach only taking a maximum of *1.2 minutes*. Over all of the tested crates, cargo-mutants takes a median of 1.6 hours to complete, while mutest-rs takes *22.9 seconds* by default, *13.2 seconds* with Dynamic Mutation Scheduling, and *2.4 minutes* with process isolation. The stark improvement comes from the lack of repeated compilations, and the filtering of irrelevant tests and unreachable mutations.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce mutest-rs, our flexible, efficient Rust mutation analysis tool. Our tool improves on the status quo by generating valid, reachable, and diverse mutations based on a flexible and extensible mutation operator model and compiler-based program analysis, by compiling all mutations into a single Rust program, by filtering unreachable mutations and irrelevant test cases, and by providing a comprehensive set of options to configure the mutation analysis. Planned future work includes the addition of new mutation operators, tools for gaining insights from the results of mutation analysis, and an alternative mutation harness that works with embedded microcontrollers.

REFERENCES

[1] Z. Lévai, "mutest-rs," 2026. [Online]. Available: https://github.com/zalanlevai/mutest-rs

[2] ——, "mutest-rs demo," 2026. [Online]. Available: https://github.com/rust-mutation-testing/mutest-rs-demo

[3] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, Apr. 1978.

[4] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation Analysis." Tech. Rep., Sep. 1979.

[5] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE TSE*, Sep. 2011.

[6] R. H. Untch, "Mutation-based software testing using program schemata," in *Proc. ACMSE*, Apr. 1992.

[7] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proc. ISSTA*, Jul. 1993.

[8] Z. Lévai and P. McMinn, "Batching Non-Conflicting Mutations for Efficient, Safe, Parallel Mutation Analysis in Rust," in *Proc. ICST*, Dublin, Ireland, Apr. 2023.

[9] Z. Lévai, D. Shin, and P. McMinn, "A Comprehensive Empirical and Theoretical Analysis of Batching Algorithms for Efficient, Safe, Parallel Mutation Analysis in Rust," *ACM TOSEM*, Jan. 2026.

[10] ——, "Dynamic Mutation Scheduling: Highly Parallel, Efficient Evaluation of Mutations for Rust Programs through Program Splitting," in *Proc. ICST*. Daejeon, South Korea: IEEE, 2026.

[11] The Rust Project Developers, "Rust: Production users," 2024. [Online]. Available: https://www.rust-lang.org/production/users

[12] Rust Foundation Team, "Announcing the Safety-Critical Rust Consortium," Jun. 2024. [Online]. Available: https://rustfoundation.org/media/announcing-the-safety-critical-rust-consortium/

[13] A. Bogus, "mutagen," 2022. [Online]. Available: https://github.com/llogiq/mutagen

[14] M. Pool, "cargo-mutants," 2025. [Online]. Available: https://github.com/sourcefrog/cargo-mutants