A Comprehensive Empirical and Theoretical Analysis of Batching Algorithms for Efficient, Safe, Parallel Mutation Analysis in Rust

ZALÁN LÉVAI, University of Sheffield, UK
DONGHWAN SHIN, University of Sheffield, UK
PHIL MCMINN, University of Sheffield, UK

There is a lack of serious tooling for mutation analysis for Rust, a safety-focused systems programming language seeing increased adoption across the industry. As such, the testing technique has not been widely used on programs written in the language as of yet. Without robust mutation analysis, Rust developers cannot determine test thoroughness. In response to this challenge, we designed a mutation analysis pipeline for Rust, which overcomes the challenges of generating valid mutants caused by the strictness of the language. Our approach accounts for Rust's distinction between safe and unsafe operations, ensuring that safe mutations of valid Rust programs — those with only valid unsafe code sections — can be safely evaluated within the same process, without the potential for crashes or other undefined behavior invalidating the mutation analysis. We introduce mutation batching, our novel technique for efficiently evaluating multiple mutations simultaneously, while guaranteeing they do not interact. Batching maximizes thread usage, by executing significantly more test cases in parallel. As batching is NP-hard, we present multiple fast approximation algorithms for grouping mutations. We implemented our techniques into a mutation analysis tool, mutest-rs, which we used in our empirical evaluation on a diverse set of 22 Rust libraries and programs. We found that mutation batching reduces the overall runtime of mutation analysis by up to 52.3% and also saving 73.8 seconds in one case, and that unsafe mutations are detected at a lower rate, of up to 53.4%. Our mutation analysis tool is available at https://mutest.rs.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Packing and covering problems*; *Graph algorithms analysis*; • **Mathematics of computing** → Probabilistic algorithms.

Additional Key Words and Phrases: mutation analysis, mutation testing, cost reduction, mutation batching, rust, static analysis

ACM Reference Format:
Zalán Lévai, Donghwan Shin, and Phil McMinn. 2026. A Comprehensive Empirical and Theoretical Analysis of Batching Algorithms for Efficient, Safe, Parallel Mutation Analysis in Rust. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2026), 53 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Rust is a systems programming language that guarantees memory and thread safety statically, and places a large emphasis on automated testing. These qualities make it ideal for programs that require both performance and safety. However, while the language provides built-in support for writing and running automated test suites, it lacks mature mutation analysis techniques, which leaves Rust developers without a way of ensuring thorough testing of programs written in the language.

Authors' Contact Information: Zalán Lévai, University of Sheffield, Sheffield, UK, zblevai1@sheffield.ac.uk; Donghwan Shin, University of Sheffield, Sheffield, UK, d.shin@sheffield.ac.uk; Phil McMinn, University of Sheffield, Sheffield, UK, p.mcminn@sheffield.ac.uk.

Being a compiled language used primarily for writing low-level, often safety-critical systems software (e.g. operating systems, kernels, device drivers, networking tools, compilers), Rust faces a multitude of unique, additional challenges when it comes to applying techniques based on code generation, such as mutation analysis. Rust programs are compiled into native binaries and are not executed by a high-level virtual machine. This makes dynamic approaches to mutation analysis impractical, and as such, static optimization approaches become necessary. In addition, the strict static analysis of the language — both in terms of type safety, and memory restrictions — means that reliably generating valid Rust programs that compile comes with particular challenges. This is highlighted by the two existing research works on test case generation [48, 51], both of which place significant constraints on their code generation related to object lifetimes, which Rust has strict rules around. For mutation analysis, this means that mutation operators have to be constrained using tailored type checking and data flow analysis, which is not required for less strict languages such as Java, C, or C++. For example, mutations must ensure that types remain exactly identical to the original program's (e.g., the various range types `a..b`, `a..=b`, etc. are all different, incompatible types), and that the resolution of complex type-relative names (e.g., in method calls) remains unchanged, often requiring the addition of explicit type annotations. Because of tracked memory lifetimes unique to Rust, mutation operators must also be careful of creating short-lived values that might escape their scope. Furthermore, the presence of explicitly-annotated *unsafe* code sections means that special care has to be taken to avoid any possibility of introducing undefined behavior into the program through mutations. This is an important consideration for any in-process mutation evaluation technique, including sequential evaluation of mutations, as undefined behavior from one mutation can influence the correctness of the rest of the mutation evaluation, either by introducing subtle errors (e.g., memory corruption), or by crashing it entirely. Our distinction between safe and unsafe mutations alleviates the need for evaluating every single mutation's respective test cases in individual processes each to protect from such occurrences. We discuss this fundamental *safety* property — defined by the Rust project developers [56] as the guarantee that code cannot cause undefined behavior — throughout this paper. Finally, mutation analysis approaches also have to be efficient in terms of their total runtime, both to make their use possible on very large projects, and to make the widespread application of the approach more practical in general. Mutation analysis approaches must scale well to large numbers of tests, mutations, and input source code. This is especially important, as mutation analysis is widely considered to be a computationally expensive technique.

The research in this paper, along with its original conference paper version [36], is the first to consider mutation analysis for the Rust programming language. To tackle the unique challenges facing mutation analysis and Rust, we devise a robust, efficient mutation analysis pipeline for the language. We define five new, additional operators to cover some of the most often used language features that are unique to Rust. We also define a set of rules to identify mutations which could introduce undefined behavior, which we call "unsafe" mutations (corresponding to the definition of safety in Rust). To tackle the high costs of performing mutation analysis, specifically the time required to evaluate mutations, we devise a novel mutation evaluation cost reduction technique that we refer to as mutation "batching".

Mutation Batching. Mutation batching addresses the inefficiency of evaluating mutations sequentially on modern, parallel processors. While test cases may be run in parallel, if the mutations are evaluated one after another, then the amount of parallelism that can be introduced will be bounded by the number of independent test cases corresponding to each mutation. For example, if on a computer capable of running 10 concurrent threads, a mutation m_1 with only 3 corresponding test cases is evaluated, then only a maximum of 3 out of the 10 concurrent threads can be utilized to evaluate the mutation, one for each of the test cases. This means that we effectively only utilize 30% of the compute resources available to us. However, if we could also evaluate a mutation m_2 with 7 corresponding test cases alongside

m_1's 3 test cases, then we could utilize all 10 of the concurrent threads to evaluate the two mutations. This means that we can effectively utilize 100% of the processors' resources, and most importantly, we could reduce the overall runtime. It is easy to see that mutations with few test cases, like m_1, are the common case, and that mutations reached by a large number of test cases are significantly less common, especially as the number of available concurrent threads increases. As such, it stands to reason that such an optimization can have a large impact on the runtime of mutation analysis. For such an evaluation to be correct however, we must ensure that mutations m_1 and m_2 have no way of interacting with each other, or altering each other's behavior.

Thus, mutation batching is our novel process of grouping individual, "non-conflicting" mutations together into optimal batches, while ensuring the correctness of the resulting mutation analysis. Non-conflicting mutations are sets of mutations that cannot influence whether the other mutations in the set are killed or not, due to them appearing in different, distinct parts of a program. Our technique determines which pairs of mutations are non-conflicting based on an extensive, conservative static analysis of the functions potentially reachable from each of the test suite's test cases, based on their call graph. Mutations are then "classified" based on which function they mutate, and the set of test cases their respective function is reachable from. If two mutations m_1 and m_2 appear in two different functions exclusively reached by two different sets of test cases — $\{t_1, t_2, t_3\}$ and $\{t_4, t_5, \ldots, t_{10}\}$, respectively — then m_1 and m_2 are candidates for being placed in the same batch, and thus being evaluated together. By ensuring that mutations in a batch are each exclusively reached by two different sets of test cases, we can also uniquely determine which mutation in a batch was killed based on which test case failed. For example, if during the evaluation of the batch $\{m_1, m_2\}$, t_1, t_2 or t_3 fails, then we know that only m_1 could have caused the failure, and as such, m_1 is known to have been killed. Consequently, if t_4, t_5, …, or t_{10} fails, then m_2 is known to have been killed.

During evaluation, the test cases relevant to each individual mutation batch can be executed in parallel, evaluating more concurrent test cases in fewer iterations, resulting in the increase in the utilization of the available parallel processors previously described. This is in contrast to existing approaches for parallelizing mutation analysis [18, 32, 50, 57], which all use multiple, forked processes for individual mutations. Our approach is fundamentally different, and instead focuses on increasing the amount of parallelizable work. Compared to running mutations in parallel processes, mutation batching, with its in-process parallelism, has the advantage of introducing significantly less overhead, and having much more control over the entire evaluation.

The conflicts between mutations that form the basis of mutation batching can be described in terms of graph theory (Section 3.4.1). Since mutation batching is an NP-hard problem, we present fast, greedy approximation algorithms for producing batches of mutations based on the compatibilities between them: a fully-deterministic greedy algorithm, and a partially-deterministic algorithm that we call epsilon-greedy (ϵ-greedy) batching (Section 3.4.2).

At the end of the process, our pipeline creates a single meta-mutant program [60] to represent the generated mutations, alongside the mutation batches created up front. This meta-mutant program statically embeds all mutations, building on the idea of conditional mutation [30]. Our injected test harness is then able to dynamically enable sets of mutations while evaluating individual batches.

Original Conference Paper and the mutest-rs Project. To accompany this paper and our future research work, we also develop the first mature mutation analysis tool for Rust, called mutest-rs [34], our ongoing tool and mutation analysis project which implements our pipeline and techniques. This paper's description and evaluation of mutest-rs is an extension of the original conference paper [36] that was presented at the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST) 2023. We used an early version of mutest-rs to evaluate our approach on 10 critical and commonly-used Rust programs and libraries (referred to in Rust as "crates"). Our empirical results showed that mutest-rs is applicable to a range of Rust subjects, and can reliably generate mutants, while also demonstrating that batching is effective for reducing mutation analysis runtimes. The contributions of the original ICST conference paper are as follows:

(1) A set of mutation operators suitable for Rust programs, including adaptations of thirteen existing operators, and five new, additional operators specifically designed for the language (Section 3.2).
(2) An algorithm for batching mutations for simultaneous, parallel, and efficient mutation evaluation (Section 3.4).
(3) A definition of mutation safety based on Rust's distinct safe and unsafe scopes, allowing for the mutation of system programs without the fear of introducing undefined behavior (Section 3.5).
(4) The results of an empirical evaluation of the reduction in testing time possible with mutation batching in practice, revealing a reduction in the overall mutation analysis runtime for a diverse set of commonly-used Rust subject programs [36].

While the empirical evaluation of that early version of mutest-rs and mutation batching had positive results — warranting further research and development of the tool and our technique — we only presented and evaluated a single approximation algorithm for mutation batching, with many of the choices in this algorithm unevaluated, and further experimentation left as an item of future work.

Changes to mutest-rs and Additional Contributions. Since our original evaluation of mutest-rs, we have made changes to the tool, and its pipeline, to improve it further. First, unsafe mutations (Section 3.5) — which mutest-rs was not able to safely evaluate before, can now be evaluated in isolated child processes (alongside safe mutations running in the main process), ensuring that they do not affect the main mutation analysis process (**RQ2** in Section 5). Second, we have made changes to how mutations are activated in mutest-rs during mutation evaluation[1], which results in significantly improved runtimes across all subjects both with and without mutation batching applied (Section 3.1). It is important to note that this means that reported runtimes are not directly comparable to those present in the original conference paper, as runtimes for both batched, and unbatched runs have improved significantly. Third, and most notable for this paper, we have implemented a number of additional mutation batching algorithms into mutest-rs, including randomized batching, and a new, partially probabilistic algorithm called epsilon-greedy batching (Section 3.4). In this paper, we evaluate mutest-rs on a much larger variety of Rust programs than previously, using over double the amount of subject programs going from 10 to 22, further strengthening our results. The additional contributions that this paper makes over the original ICST 2023 conference paper, therefore, are as follows:

(5) A theoretical analysis of mutation batching and its complexity in terms of graph theory (Section 3.4.1).
(6) A set of additional mutation batching algorithms, including alternative ordering heuristics for the original greedy algorithm, and a new, partially probabilistic algorithm that we call epsilon-greedy batching. (Section 3.4.2).
(7) The results of an empirical evaluation (Section 4) evaluating:
    (a) the validity of our five new, additional mutation operators, showing that their mutations are detected at a similar or lower rate to those of traditional mutation operators, by up to 24.3% (**RQ1** in Section 5);

[1]Specifically, we have removed all lock-based synchronization around the central, generated data structure that is used to enable and disable mutations in the meta-mutant program, referred to as the active mutant handle (Section 3.1). This structure is accessed very frequently during mutation evaluation, especially with mutations in "hot" code paths. As the locking ultimately did not add to or change the safety guarantees of the meta-mutant execution, we removed it from mutest-rs.

(b) the prevalence of unsafe mutations, and their adverse effects which our approach mitigates, showing that 5 out of the 22 subjects resulted in unsafe mutations, and that they were detected at a lower rate, by up to 74.6% less (**RQ2** in Section 5);

(c) the reduction in testing time possible with mutation batching on an updated, much wider set of commonly-used Rust subject programs, showing an overall reduction in the mutation analysis runtime of up to 52.3%, and 73.8 seconds saved in the case of one subject (**RQ3** in Section 5);

(d) the original greedy mutation batching algorithm against randomized mutation batching, and the new epsilon-greedy mutation batching algorithm, using a variety of greedy ordering heuristics (**RQ3** in Section 5).

Our tool, empirical data, and our full replication package including scripts to reproduce our experiments are available online [34, 35]. The latest version of our tool, mutest-rs, alongside documentation and examples, is available at https://mutest.rs.

2 Background

2.1 Mutation Analysis

Mutation analysis is a software testing technique that aims to measure the quality of a program's test suite through its ability to recognize faults automatically injected into the program [1, 11, 21, 27, 37]. In mutation analysis, these faults are based on common faults competent programmers might mistakenly introduce, according to the "Competent Programmer Hypothesis" [1, 11]. Such faults are generated automatically based on the original program by adding small syntactical changes. The resulting faulty programs are referred to as mutants. To perform mutation analysis, the original test suite is evaluated against these mutant programs. If any test case in the test suite fails in the presence of mutations, then that mutation is considered to have been "killed" by the test suite. One of the results of mutation analysis is the mutation score, which is the percentage of mutations that have been killed by the test suite.

Mutation analysis is considered to be a computationally expensive technique, as it requires multiple evaluations of the test suite proportional to the number of mutants that are being evaluated.

2.2 The Rust Programming Language — Testing, and Mutation Analysis

Rust is a relatively new, emerging programming language that aims to bring compiler-proven memory safety to low-level systems programming, making it a performant, safer alternative to C, and C++. Because of its safety characteristics and low-level control, the language has seen increased adoption throughout many parts of the industry, especially in safety- and performance-critical applications [53]. However, despite its success in the industry, to date, Rust has received little attention from software testing research, and research on applying testing techniques to the language is sparse, with only two notable papers attempting test case generation [48, 51]. Specifically, the practice of mutation analysis — the use of automatically generated code defects to measure the quality of a program's test suite in recognizing faults [1, 11, 21, 27, 37] — has not been attempted by any research dedicated to the language to the authors' knowledge, to date.

Rust is a safety-focused programming language with both static analysis and testing at its core. It has built-in automated testing facilities: functions can be marked with the `#[test]` attribute and are then evaluated by the included test harness provided by the rustc reference Rust compiler. Each unit of code, referred to in Rust as a "crate", can be tested individually through its unit tests — test functions defined "next to" program code, and integration tests —

test modules defined outside of program code. These built-in testing tools are used by the majority of Rust's thriving ecosystem of library crates, which become common dependencies of most projects.

Despite the testability advantages of Rust, a full-featured mutation analysis technique does not exist, hitherto, for the language, although two relatively limited solutions exist — `mutagen` [7], and `cargo-mutants` [46]. Bogus's `mutagen` is able to apply simple AST transformations to Rust programs to produce mutants. These mutants are not guaranteed to be valid programs, and are all compiled separately. Pool's `cargo-mutants` only implements a form of extreme mutation [5, 6] — a simplified form of mutation analysis that involves replacing entire function bodies, and more recently, a limited form of binary operator mutation. However, it shares the same limitations as `mutagen` around the validity of mutants, and separate compilation of mutated programs. Both of these existing approaches generate a large number of invalid mutants because they do not consider the semantics of the program (e.g. types, lifetimes, control flow). They both operate purely on program syntax, rather than semantics, and as a result are also highly limited in the kinds of mutations they can produce, are generally inefficient at both mutation generation and evaluation, and are difficult to apply in practice. In our research of open-source Rust subject programs, we have not found evidence of the usage of either `mutagen` or `cargo-mutants` in the forms of either relevant configuration files or code annotations, which are required for the usage of both of these tools. This can be seen as an indicator for the lack of widespread application of mutation testing in the Rust ecosystem.

Meanwhile, although not explicitly built for Rust programs, the research of Denisov and Pankevich [12] into the mutation analysis of LLVM bytecode — the instruction language Rust compiles to by default — mentions its potential applicability to Rust. However, a number of problems arise from bytecode-based mutation analysis over source-based approaches. First, mutations may be easily introduced in external library code less relevant to the tested program, such as the standard library, which the programmer has no control over. Mutation analysis should ideally only test the program's own code, and stop at external API boundaries. Many of these cases can be avoided by augmenting the bytecode with source location information, which is widely available, as shown by Chekam et al.'s [9] work on `Mart`. However, issues such as inlining of the code of external functions can still result in, for example, the loss of mutable locations in the program, such as the site of the inlined call to the external library itself. This issue remains an inherent limitation of bytecode-based approaches, due to the lossy nature of source code to bytecode conversion. Second, many bytecode mutations, while representable in bytecode, do not have source code counterparts, and thus could not have been written by a programmer. These mutations are thus not relevant to the programmer. The results seen with generic LLVM bytecode mutation [9, 12, 22, 23] reinforce the need for research based on language-specific, source-based analysis.

2.3 Considerations Towards Implementing Mutation Analysis for Rust

Multiple considerations have to be made when implementing mutation analysis for Rust towards the specific features of the language. As highlighted by the two existing mutation analysis tools above, efficient, and reliable mutation analysis has its challenges. Due to the extensive static analysis performed by the language, it is important for mutation analysis tools to reliably generate valid programs. This is for efficiency, to avoid unnecessary compilation, and analysis of mutant programs.

Most notable is Rust's safety paradigm, which requires special attention when applying mutation analysis. Rust statically proves the memory safety of programs, although this requires developers to abide by its safety rules, which is defined as the guarantee that code cannot cause undefined behavior [56]. Unlike languages such as C, C++, or Java, Rust restricts unsafe operations to blocks specifically annotated as `unsafe`. As such, when more flexibility is required, dedicated unsafe code sections may be introduced to perform these unsafe operations, like dereferencing a raw pointer, or calling an external C function through a foreign function interface (FFI) to interact with existing code. These unsafe code sections have to be manually audited to ensure that they uphold safety contracts; i.e., guarantee that the code cannot cause undefined behavior. The use of unsafe operations is discouraged, but is sometimes necessary, for example, for interfacing with foreign code. Evans et al. [15] analyzed common Rust libraries, and found that around 30% of libraries contained unsafe code, mostly through the use of `unsafe` function calls from library dependencies. Astrauskas et al. [3] conducted a survey of Rust libraries, and found that unsafe code was used to achieve three main goals: “overcoming aliasing restrictions” such as in data structures with complex sharing, or to overcome certain incompleteness issues, “emphasizing contracts, and invariants”, for example annotating functions which require an invariant the compiler cannot prove as unsafe; and “accessing a lower abstraction layer”, like foreign functions, or compiler intrinsics. This clear divide between safe and unsafe operations as a language feature is, in systems programming, unique to Rust. The complex interactions between safe and unsafe code have to be considered for mutation analysis to retain these guarantees [56]. This is especially important for mutation analysis, where we are trying to introduce generated program mutations while retaining the existing safety of the code.

Rust has an expressive expression-based syntax, which lends itself well to Untch et al.’s meta-mutant [58–60], and Just et al.’s conditional mutation [30] approaches. These approaches minimize compilation time needed for mutants by embedding mutations into a single program, called a meta-mutant, which results in only a single program needing to be compiled. This meta-mutant program can then act as any of the constituent mutants based on an argument upon invocation.

Among the safety goals of Rust is increased testability. As test cases are free functions, i.e., standalone functions that are not methods associated to a type, Rust’s testing paradigm has no explicit support for setup and teardown functions commonly found in testing frameworks for other languages. This simple approach to test cases may give way for opportunities to optimize test running strategies further. In addition, the built-in test runner does not make any guarantees about the execution order of test cases. These help avoid the introduction of flaky, order-dependent tests [14, 44], and are, again, helpful when optimizing alternative testing approaches.

3 Approach

This section details our process of mutation analysis for Rust by first providing an overview of our mutation analysis pipeline (Section 3.1), followed by sections explaining each aspect of our mutation analysis approach: our representation of mutations in Rust and our additional mutation operators (Section 3.2); our construction of call graphs suitable to represent complex Rust programs (Section 3.3); our technique for finding compatible mutations and batching them together (Section 3.4); our mutation safety technique for avoiding undefined behavior (Section 3.5); and finally our parallelized test evaluation method (Section 3.6). We implemented our pipeline and techniques into our mutation analysis tool, mutest-rs.

3.1 Overview

At a high-level, our mutation analysis pipeline performs mutation analysis on a given Rust crate in two stages. The first stage takes the input Rust source code, and produces the source code for the meta-mutant program, which represents every generated mutation. The second stage takes this mutated source code, and produces the results of mutation analysis. Figure 1 shows an overview of our mutation analysis pipeline, which we implemented in mutest-rs.

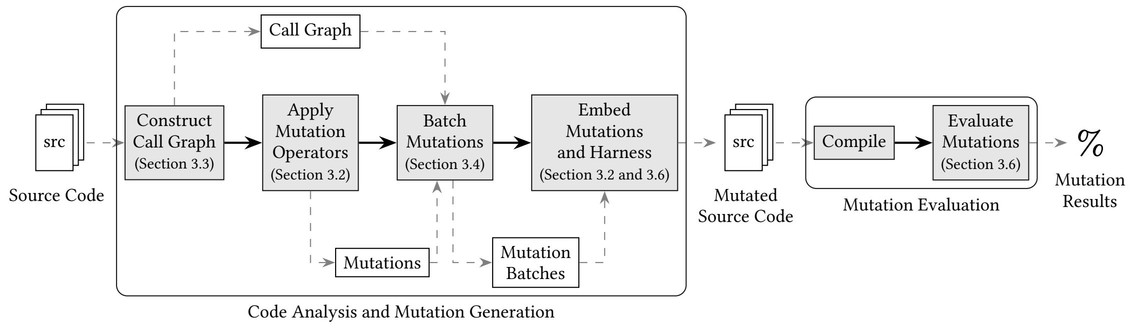

Call Graph
src
Source Code
Construct Call Graph (Section 3.3)
Apply Mutation Operators (Section 3.2)
Batch Mutations (Section 3.4)
Embed Mutations and Harness (Section 3.2 and 3.6)
Mutations
Mutation Batches
Code Analysis and Mutation Generation
src
Mutated Source Code
Compile
Evaluate Mutations (Section 3.6)
Mutation Evaluation
%
Mutation Results

Fig. 1. A high-level overview of our mutation analysis pipeline with mutation batching, which we implemented in our mutation analysis tool, mutest-rs. The gray boxes and black arrows represent the linear pipeline of steps taken by mutest-rs to perform mutation analysis, while the white boxes and gray, dashed arrows represent the data produced and used by each of the steps.

In the first stage (*Code Analysis and Mutation Generation* in Figure 1), our tool uses an augmented version of the Rust compiler, rustc, to analyze the original source code and generate mutations from it. First, our tool generates a call graph (Section 3.3) starting from the crate's test functions (which we treat as entry points in the context of testing). This ensures that our tool only mutates functions that are reachable by the test suite, and allows our tool to build a mapping between mutations, and the tests that may reach them. This mapping is then used during the batching, and later during the evaluation of the generated mutations. Second, our tool applies a set of mutation operators (Section 3.2) to every possible location in the body of the previously selected functions. Third, the generated mutations are tested for compatibility, and are batched by one of our mutation batching algorithms (Section 3.4). Finally, the tool inserts the conditionally mutated code fragments which make up the mutations into a copy of the source code of the original program, alongside the mutation and mutant metadata; and an injected global variable, `ACTIVE_MUTANT_HANDLE`, used to control the currently active mutant. The tool replaces the entry point of the program (i.e., the `main` function) with a call to the generic mutation test harness, which drives the evaluation of the mutation analysis. The output of the first pass is the generated source code of the mutated program with injected code to drive the mutation analysis.

In the second stage (*Mutation Evaluation* in Figure 1), our tool compiles the generated source code of the mutated program into a binary. This custom test binary is then executed to perform the evaluation of the mutations, and to produce the results of mutation analysis.

3.2 Mutation Operators for Rust

We model mutation operators as a mapping from a source code location, a node in the original program's abstract syntax tree (AST), to a set of substitutions required to reproduce the mutation, if the mutation operator is applicable. Every substitution is a pair of an existing syntax node, and a new, replacement syntax node. Substitutions may replace existing expression nodes, or insert statements before or after an existing statement. The operators may make substitutions at any node in the body of the function, not just the input location. (This means that we can use mutest-rs to generate both first-, and higher-order mutations [25, 26] using our approach, although we only consider first-order mutations in this paper.) First-order mutations can be represented as a singleton set of a single substitution.

Our implementation embeds the generated substitutions that make up the mutations using in-place conditional expressions over an injected global state. This state represents the active mutations at runtime, and is managed by the test harness. Figure 2 shows an example of such a conditional expression. First, substitutions of the various mutations are

```
        mem::size_of::<f32>() as u32 * 8
                              ↓
match subst!(ACTIVE_MUTANT_HANDLE @ rep_13466) {
    Some(subst) if subst.mutation.id == 1116 =>
        mem::size_of::<f32>() as u32 + 8,
    Some(subst) if subst.mutation.id == 1117 =>
        mem::size_of::<f32>() as u32 / 8,
    _ => mem::size_of::<f32>() as u32 * 8,
}
```

Fig. 2. Example of a `match`-based conditional expression used to embed substituted expression nodes alongside the original. Taken from code generated for `hashbrown` (a subject used in our empirical study in Section 4.1), illustrating the mutations applied to a memory offset calculation.

grouped by the location in the original source code they apply to. Then, for each substituted location, our tool replaces the original node with a `match`-based conditional expression, with a branch for each substituted node, and a default branch for the original expression. For inserted statements, this default branch is an empty block. This representation of the substitutions supports nesting. By replacing subexpressions first, their corresponding substitution expression gets placed into the default branch of any outer substitution expression.

We designed a set of fundamental mutation operators for programs written in Rust. These mutation operators filter equivalent mutants directly, using a rule-based approach, which we describe in further detail for each mutation operator. Thirteen of these mutation operators are adaptations of common operators used in almost all languages (e.g., by Major, and PIT for Java), that are applicable to Rust. We refer to them as traditional mutation operators. Five are additional operators that are tailored specifically to language features commonly used in Rust programs that are not covered by the more common traditional operators. While most of these are not entirely new, they are all highly adapted (in some cases restricted due to the strict rules of the language, while in other cases more generalized) to the commonly found, unique code patterns in Rust programs. They are primarily intended to produce mutations with similar behavior to mutations generated by other common mutation operators for other programming languages. Table 1 lists these eighteen mutation operators, with the traditional and additional mutation operators introduced in this paper as follows:

The eight **OpAddMulSwap**, **OpAddSubSwap**, **OpDivRemSwap**, **OpMulDivSwap**, and the bitwise **BitOpOrAndSwap**, **BitOpOrXorSwap**, **BitOpShiftDirSwap**, and **BitOpXorAndSwap** traditional mutation operators target binary operation expressions. These mutation operators, that replace existing operators with related counterparts, all have to determine whether the trait required for the new operator is implemented by the operand types. For example, the `Instant` timestamp type implements the `Sub` trait for subtraction but not the `Add` trait for addition, which limits the applicable valid mutations. These mutation operators filter out such non-applicable mutations. For these mutations, minimal equivalent mutant filtering is required, as either of the operands would have to be the identity of both the original and the substituted binary operation for equivalence (e.g., `i + 0` with OpAddSubSwap, where `0` is the identity of both the addition and subtraction operations). The resulting mutation would be equivalent if the resulting value was unused, but the Rust compiler already warns against such cases of dead code. This caveat applies to most of our mutation operators more generally. It is worth noting that these mutation operators are more generic than most of their original counterparts, as they apply to all types that implement the corresponding binary operator traits (i.e., they implement the operation).

Table 1. Mutation operators for Rust implemented in mutest-rs. The five mutation operators in **bold** are highly-adapted versions of existing mutation operators, newly introduced in this paper. We refer to them as additional mutation operators.

Mutation Operator	Description
ArgDefaultShadow	Replace function argument with default value[1]
BitOpOrAndSwap [32]	Replace bitwise OR with bitwise AND, and vice versa
BitOpOrXorSwap [32]	Replace bitwise OR with bitwise XOR, and vice versa
BitOpShiftDirSwap [32]	Replace bitwise LSH with bitwise RSH, and vice versa
BitOpXorAndSwap [32]	Replace bitwise XOR with bitwise AND, and vice versa
BoolExprNegate [32]	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 vice versa
EqOpInvert [32]	Invert equality operator
LogicalOpAndOrSwap [32]	Replace logical `&&` with logical `||`, and vice versa
OpAddMulSwap [32]	Replace addition with multiplication, and vice versa
OpAddSubSwap [32]	Replace addition with subtraction, and vice versa
OpDivRemSwap [32]	Replace division with modulo, and vice versa
OpMulDivSwap [32]	Replace multiplication with division, and vice versa
RangeLimitSwap	Change inclusivity of range's upper bound
RelationalOpEqSwap [1]	Change relation operator's bound with regard to equality
RelationalOpInvert [1]	Invert relational operator

[1]The default value for the type as defined by the implementation of the `Default` trait in Rust; i.e., the value of `Default::default()`.

The remaining five **BoolExprNegate**, **EqOpInvert**, **LogicalOpAndOrSwap**, **RelationalOpEqSwap**, and **RelationalOpInvert** traditional mutation operators target various Boolean expressions. These mutation operators are more similar to their original counterparts, as they are only applicable to Boolean-typed expressions. These mutations are applicable in all cases, and are unlikely to produce truly equivalent mutants by themselves.

The **ArgDefaultShadow** additional mutation operator targets function arguments, replacing any argument passed to a function parameter with the default value of the type — defined by the author of the type through the implementation of the `Default` trait. By inserting a variable binding statement at the beginning of the function, mutest-rs can ensure that the original function body does not have access to the passed argument. This is possible since Rust allows for rebinding variable names, and has a common way of representing the intended default value of each type, by means of the `Default` trait. The mutation operator only applies to types with this trait implemented. The mutations generated by this operator cannot be equivalent mutants, unless the argument value is unused. In the case when the test suite only ever invokes the function with the default argument value, the mutation will not be detected. However, this is an indicator of either a lack of testing — if the test directly or indirectly influences the value of the function argument, or dead program code (i.e., code that is never exercised within the program through any accessible means) rather than an equivalent mutant. In the case of the default values originating from test cases, the mutation could be detected by a sufficiently extensive test suite. In the case of the default values originating from fixed points in program code with no other accessible code paths, it might be advisable for such non-exercised, untested code paths to be removed. This mutation operator can be seen as a generalization of the existing AOC (Argument Order Change) and AND (Argument Number Decrease) mutation operators, which are not directly applicable to Rust as these mutation operators target method overloading not supported by the language.

The **CallDelete** additional mutation operator targets function calls by deleting them, replacing the call with the default value of the type — as defined by the type's author through the `Default` trait. This mutation operator can be seen as a restricted, Rust-specific version of statement deletion, as Rust's strict rules make ad-hoc statement deletion difficult. **CallValueDefaultShadow** additional mutation operator targets function calls in a more subtle way, by retaining the function call and replacing the return value at the call site with the default value of the type, similarly to CallDelete. This mutation operator is analogous to existing mutation operators targeting the return-sites of functions, but this mutation operator achieves a similar effect by mutating the call-site instead. The difference between these mutation operators is whether the side effects of the called function are retained with the mutation. These two mutation operators filter out equivalent mutants; only applying to function calls which do not resemble a default constructor (i.e., take at least one argument), and whose return type implements the `Default` trait and is not the `()` unit type. In addition, these mutation operators filter out any mutations, which would cause infinite recursion through the insertion of the call to the `Default::default` trait function implementation. When the function call is retained, the return type position is explicitly annotated with the semantic type information from the original function call to ensure that the resolution of the call remains unchanged after mutation.

The **ContinueBreakSwap** additional mutation operator targets loop control flow statements by replacing `continue` expressions associated with loops with `break` expressions, and vice versa. This mutation operator only applies to loop control flow statements in loops with no return values. Since `continue` and `break` statements represent jumps to different parts of the code, they are unlikely to produce equivalent mutants. When considering the added complexity of labels and optional loop return values, the implementation of this rule quickly requires extensive scope analysis.

Finally, the **RangeLimitSwap** additional mutation operator targets Rust's range expression syntax by changing whether the range is inclusive of its upper bound. This mutation operator is primarily intended to replicate traditional mutations of for loop conditions (`for (int i = 0; i <= 10; i++)`), adapted to the range-based syntax of Rust (`for i in 0..=10`). Since this mutation operator only targets bounded ranges, which are likely to be fully consumed (by iterating over them), it is unlikely that these mutations would be equivalent mutants. While not a unique language feature, ranges, along with iterators, make up the majority of loop and array interactions in Rust programs (with manual indexing discouraged), making this an important operator addition for the language.

As mentioned previously, mutest-rs builds rule-based equivalent mutant filtering directly into the mutation operators, which can catch the large majority of equivalent mutants. In addition, the mutation operators used are designed to produce only a relatively small amount of overlapping behavior, and thus are intended to not be equivalent to each other. Most of the mutation operators match distinct code patterns, and those that match similar patterns produce mutations that exhibit slight differences in behavior that might otherwise be hard to notice.

3.3 Analyzing Function Calls as a Pre-Step to Batching

To be able to safely combine mutations, we must first analyze the functions — in which our mutations will be placed — reachable by individual test cases, and build a call graph for the program. This call graph will be used to conservatively determine dependencies between parts of the program, and ultimately give us the ability to check mutations for compatibility with regards to mutation batching.

The nodes of a call graph represent unique functions defined in the program. The presence of a directed edge between two functions indicates that that function represented by the source node contains a direct call to the function represented by the destination node. To help us with resolving intricate generic function calls that can be commonly found in Rust programs, we extend the construction of call graphs with the propagation of generic type arguments. This

```
Require: Set of Test Functions $T$
Ensure: Mapping $L$ : Function $F \longrightarrow$ (Test Function $T$, Distance $\mathbb{N}$)
  Mappings: Function Definition $\longrightarrow$ Test Function $C_1, C_2, \ldots \leftarrow \{\}, \{\}, \ldots$
  for all Function Definition $t \in T$ do
      Set of Function Calls $C_t \leftarrow$ functions called in body of $t$
      for all (Function Definition $f$, Generic Type Arguments $S$) $\in C_t$ do
          (Function Definition $f'$, Generic Type Arguments $S'$) $\leftarrow$ resolve call to $f$ with types $S$
          $C_1^{(f',S')} \leftarrow C_1^{(f',S')} \cup \{t\}$
      end for
  end for
  $L \leftarrow \{\}$
  for all Distance $d \in \langle 1, 2, \ldots \rangle$ do
      Set of Function Definitions $F_d \leftarrow \{f \mid \forall((f, \_), \_) \in C_d\}$
      Set of Function Definitions $F_\Sigma \leftarrow \{f \mid \forall(f, \_) \in L\}$
      break if $F_d \subseteq F_\Sigma$                          ▷ all functions have already been visited
      for all $((f^0, S^0), T') \in C_d$ do
          $L_{f^0} \leftarrow L_{f^0} \cup \{(t, d) \mid \forall t \in T'\}$
          Set of Function Calls $C_{f^0} \leftarrow$ functions called in body of $f^0$
          for all $(f, S) \in C_{f^0}$ do
              ▷ combine generic type arguments from the call to the containing function with those of the local function call
              Generic Type Arguments $S^+ \leftarrow$ fold type substitutions $S^0$ into $S$

              (Function Definition $f'$, Generic Type Arguments $S'$) $\leftarrow$ resolve call to $f$ with types $S^+$
              $C_{d+1}^{(f',S')} \leftarrow C_{d+1}^{(f',S')} \cup T'$
          end for
      end for
  end for
```

Fig. 3. Algorithm for constructing the walks of a fully-resolved call graph. Function calls are represented as tuples (f, S), where f is the function being called, and S is the set of local generic type arguments of the call. The output of the algorithm is $L : F \longrightarrow (T, \mathbb{N})$, a mapping between called functions and the tests they are reachability from, with a distance associated with each mapping. First, function calls in the bodies of each of the test function entry points in set T is used to populate C_1. Then, for each consecutive depth d, C_d is first used to populate the output mapping L from the previous depth, and then function calls in the bodies of the functions in C_d is used to populate C_{d+1}, with the added step of first combining the generic type arguments from the previous depth, S_0, with those of the local function call, S, thus propagating generic type arguments. The analogous code sections between the iteration of the entry point functions, and consecutive iterations of called functions is highlighted.

helps us find the exact function definitions called by generic calls. We refer to the resulting call graph as a fully-resolved call graph:

Definition 1 (Fully-resolved call graph). *A fully-resolved call graph is a directed graph $G_C = (F, C)$ over a set of root functions R, where:*

- *F (function nodes) is a set of (f, S) tuples, where f is a function definition, and S is a set of type substitutions applicable to f; and*
- *C (call edges) is a set of directed edges between two function nodes, $C \subset F \times F$.*
- *The set of root functions R may be any fully-resolved, non-generic functions. (These include entry points, like a binary crate's* `main` *function or test functions.)*

```
fn f1<T: T1>() -> T {
    T::do_t1::<u8>(1)
}

let _ = f1::<S1>();
```

Fig. 4. A generic function call with a single level of indirection, which cannot be resolved without propagating type substitutions in the call graph.

Based on the idea of fully-resolved call graphs, the algorithm in Figure 3 shows the process of building walks of the call graph between test functions T, and functions of the program. Function calls are represented as (f, S) tuples, where f is the function definition, and S is the set of call-site type substitutions passed to the function. The output of the algorithm, $L : F \longrightarrow (T, \mathbb{N})$, is a mapping between functions and tests, with a distance associated with each mapping. For each mapping, distance is the length of the shortest call path between the two functions. $C_1, C_2, \ldots, C_d$ represent the callees at their respective levels d of the call tree. These get populated as the depth-wise iteration of the tree progresses.

The difference from the construction of a partially-resolved call graph is that the concrete types each generic function is called with are taken into account. Instead of looking at just a function's definition, each individual, uniquely type-parameterized invocation of the function is resolved independently. This is analogous to the monomorphization of generic functions performed during code generation in compiled languages [54].

Figure 4 is an example of a function call which cannot be resolved using the local context of the function definition alone, but can be when using our approach for building a fully-resolved call graph. Inside the generic `f1` function, a call is made to the `T1::do_t1::<V>` generic trait function, with the known type parameter `<V = u8>`. Since the `T` type parameter of the function is not known, the call can only be partially resolved to `<T as T1>::do_t1::<u8>`, which does not identify the actual function body, as the type on which the `do_t1` function is defined, is unknown. Our technique instead looks at the function invocation `f1::<S1>` (and other invocations of the function `f1` with differing type arguments). By combining the types function `f1` was invoked with (`<T = S1>`), with the call's types (`<T = ?, V = u8>`), it becomes possible to correctly resolve the same function call to `S1::do_t1::<u8>`.

When constructing a Rust program's call graph, it is important to consider every language feature with semantics that can result in the introduction of a function call, not just explicit function calls. Most notably, this includes almost all unary (`*` for dereferencing, `-` for negation, `!` for logical negation), binary (+, -, *, /, %, |, ^, &, <<, >>, ==, !=, <, <=, >, >=, and `[]` for indexing), and corresponding binary assignment operators, which have corresponding traits (i.e., interfaces) that can be used to implement the operators for user-defined types. It also includes implicit calls to types' `Drop::drop` user-defined cleanup functions at the end of scopes, and implicit calls to iterators' `Iterator::next` implementations when the iterator itself is the "subject" of a for loop. It is also important to consider not just the function calls that are made to definitions strictly within the program, but function calls to library code as well, as these library definitions can then make further function calls back to program code. For example, a library function might be generic over a trait (i.e., interface) and the library function is defined in such a way that it calls a user-defined trait function. If a type with a program-defined trait implementation is passed into such a library function, then the library function will end up calling back to program code via the trait implementation. It is worth noting however that even in such cases, only calls to program code will ultimately be considered for mutation analysis, and the library call paths between program code sections are only needed to establish full reachability analysis of all of the program code sections.

In the case of dynamic polymorphism, the fully-resolved call graph must branch off into all possible implementations of the function being called to cover any possible runtime function call. For Rust programs, we can distinguish between three different kinds of runtime dynamic polymorphism, which we refer to as virtual calls, dynamic calls, and foreign calls. *Virtual calls* refer to trait (i.e., interface) methods that can only be resolved dynamically, at runtime. Virtual calls are represented as multiple call edges to all of the possible implementations of the trait in the program. For example, when calling a trait function on a set of trait objects (e.g., iterating over a collection of orderable elements), the virtual call is represented as a call edge to all type's implementation of the function which implement the trait. *Dynamic calls* refer to calls through opaque function pointer types, which do not represent any exact function definition. Dynamic calls are represented as multiple call edges to all of the possible function definitions in the program with matching function signatures. *Foreign calls* refer to calls made to foreign definitions (e.g. C library interfaces), and are the only function call kind that cannot be represented fully, as it would require additional call graph analysis of the external code in question. This is not an inherent limitation of our call graph technique however, and it is possible, as an item of future work, to expand our graph analysis to extend to external code as well.

By employing a fully-resolved call graph, our approach is able to discover the exact functions reached by individual test functions. However, compile-time call graph analysis has its limitations. Dynamic invocation through function pointers cannot be covered exactly with a lightweight approach, and would require either extensive data flow analysis to track what functions each function pointer may refer to, or the significantly more conservative approach of marking such function invocations as being able to call any function with the same signature as the pointer's type, which is what our approach utilizes. In our approach, our tool simply uses information available from the compiler about the function definition each function call refers to, and we acknowledge, that programs making heavy use of dynamic function pointers may produce slightly overly conservative call graphs, which may, as a result, have an impact on the effectiveness of mutation batching. However, the impact of dynamic function pointers on the accuracy of our call graph and resulting mutation batching is drastically limited by the prevalence of these function calls. As Astrauskas et al. [3] found in their extensive empirical evaluation covering 31,867 Rust crates, only 0.7% of all function calls were made to either closures or raw function pointers. We discuss the prevalence of such function calls in our subjects in Section 4.3, as part of our threats to validity. It is also important to note, that the limitations of static call graph construction are not inherent limitations of mutation batching, and approaches based on runtime instrumentation are left as an item for future work.

3.4 Mutation Batching

The evaluation of mutations takes up the majority of time spent on mutation testing. For larger projects, this limitation may disqualify the use of automated mutation testing entirely, since it takes a prohibitively long time to perform. Therefore, it is important to look for optimized evaluation techniques to reduce the time needed to get results. In Rust, developers commonly make their test suites parallelizable, since the built-in test runner, by default, executes test cases in parallel, through the use of multiple in-process threads. This presents an opportunity to further engineer safe ways of improving the runtime of mutation analysis by parallelizing the evaluation of not just test cases, but multiple, independent mutations, making even better use of available resources. The ability to evaluate multiple mutations simultaneously makes significantly more efficient mutation analysis possible, but it cannot be performed without caution.

Simultaneously enabling multiple mutations requires that changes in behavior remain uniquely identifiable through test results, and that the mutations do not influence each others' changes in behavior (i.e., they do not combine into a

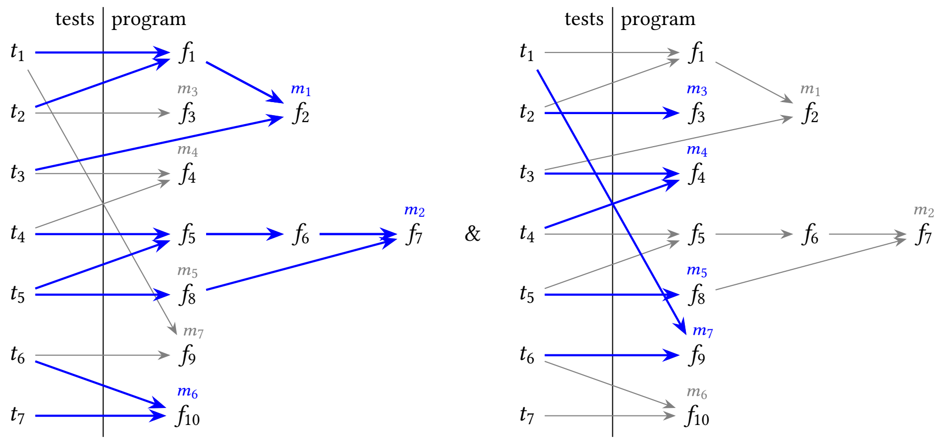


Fig. 5. Example of mutations in a call graph, separated by reachability-exclusivity. The two graphs represent two separate sets of mutations which are each internally reachability-exclusive, but where no mutation from one set could be added to the other. Functions (f_i) are annotated with the mutations placed into them (m_i). Highlighted edges represent the call paths that reach mutations from their corresponding test cases. Inactive call paths and mutations are dimmed. Mutations m_1, m_2, and m_6 are reachability-exclusive with regards to each other, and thus are compatible with each other. Mutation m_3 is not reachability-exclusive with m_1 because they are both reachable from test t_2. Mutation m_4 is not reachability-exclusive with m_1 and m_2, because they are both reachable from tests t_3 and t_4 respectively. Mutation m_5 is not reachability-exclusive with m_2 because they are both reachable from test t_5. Mutation m_7 is not reachability-exclusive with m_1 and m_6, because they are both reachable from tests t_1 and t_6 respectively.

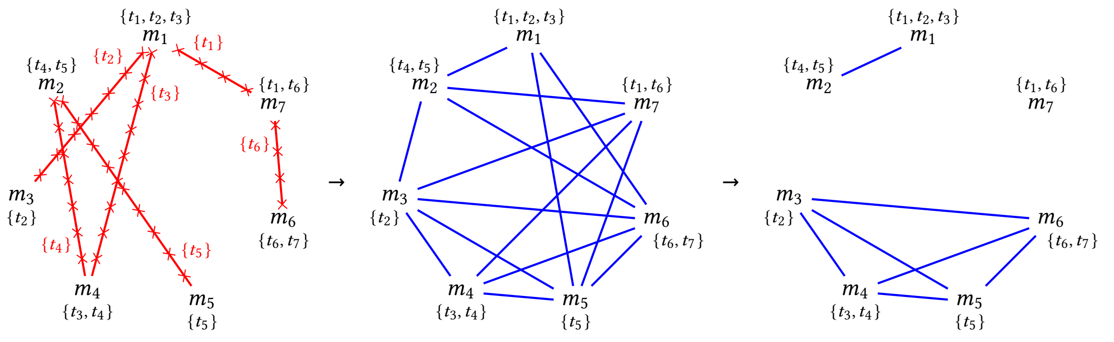
higher-order mutant). The first invariant can be upheld by ensuring that no two mutations are reachable from any of the same test functions. This results in a one-to-one mapping between test case results, and mutation detection. The second invariant then is automatically met by the above constraint: as the code executed by each test case only ever has one mutation applied to it, no other mutation can influence its behavior. This constraint effectively produces disjunct subprograms within the original program. We introduce the notion of reachability-exclusivity to formally represent this constraint:

Definition 2 (Mutation reachability-exclusivity). *Two mutations m_1 and m_2, reachable from the sets of tests T_1 and T_2, respectively, are said to be* mutation reachability-exclusive *iff $T_1 \cap T_2 = \emptyset$.*

The property of reachability-exclusivity is the precondition for mutation batching, our novel method for grouping mutations together, while ensuring that no change in behavior occurs compared to the mutations being applied individually. Mutation batching is a combination of this grouping strategy, and the corresponding test–mutation mapping, that can be used to recover causality between test case failures and mutation detection. Figure 5 shows an example of reachability-exclusivity, and mutation batching in action, on a call graph with various mutations applied. When looking at the call subtree of any given test function, we can only ever reach a single enabled mutation from all possible call paths combined.

Definition 3 (Compatible Mutations). *Two mutations m_1 and m_2 are compatible with regards to mutation batching, and thus can be batched together, iff they are reachability-exclusive.*

Definition 4 (Conflicting Mutations). *Two mutations m_1 and m_2 are conflicting with regards to mutation batching, and thus cannot be batched together, iff they are not reachability-exclusive.*



(a) Mutation conflict graph, where crossed-out edges (in red) are drawn between conflicting pairs of mutations.

(b) Mutation compatibility graph, where edges (in blue) are drawn between compatible pairs of mutations.

(c) Graph of batched mutations, constructed by partitioning the mutation compatibility graph into cliques.

Fig. 6. The steps taken to batch mutations based on conflicting test cases. In all the above graphs, nodes represent mutations, red edges represent conflicts between mutations, and blue edges represent that two mutations are compatible. Mutations may be labeled with the set of test cases it is reachable from. The crossed-out conflict edges (in red) are labeled with the set of test cases that cause the conflict, those which are common between the two mutations.

Since our mutation operators are guaranteed to not add new call edges to the call graph of the program, the fully-resolved call graph we constructed earlier is (bar dynamic function calls through function pointers discussed in Section 3.3) sound for determining reachability-exclusivity for the final meta-mutant program as well. The implementation of our approach, mutest-rs, generates such non-conflicting sets of mutations upfront, at compile time, and encodes the necessary metadata to discern the test results corresponding to each mutation. The resulting static mutation batches are then each evaluated by the test harness.

Definition 5 (Mutation Batch). *A set of mutations B, where $\forall m_1, m_2 \in B : [m_1$ and m_2 are compatible$]$.*

3.4.1 Theory behind Mutation Compatibility and Batching. When analyzed through the lens of graph theory, it can be shown that mutation batching is equivalent to the problem of clique cover, i.e., partitioning a graph into cliques, over the mutation compatibility graph.

Garey and Johnson [17] give the following definition to the clique cover problem: "Given graph $G = (V, E)$, and positive integer $K \leq |V|$, can the vertices of G be partitioned into $k \leq K$ disjoint sets $V_1, V_2, \ldots, V_k$ such that, for $1 \leq i \leq k$, the subgraph induced by V_i is a complete graph?". Following from this definition, mutation batching can be defined as the following instance of clique cover: $G = (V, E)$, where V is the set of mutations, and E is the set of edges between compatible mutations. This works, because every batch, i.e., subgraphs of G, needs to have mutations in it that are compatible with every other mutation in the same batch, meaning every subgraph of G is a complete graph, i.e., a clique. Importantly, every mutation is in exactly one batch, making batches disjoint sets of vertices, i.e., partitions.

Since mutation batching is equivalent to clique cover on the mutation compatibility graph, it follows that it is also equivalent to graph coloring on the mutation conflict graph, given that the two problems can be transformed into each other [31].

Definition 6 (Mutation Conflict Graph). *An undirected graph $G = (M, C)$ over a set of mutations M, where undirected conflict edges $C \subset M \times M$ are placed between every pair of conflicting mutations in M:*
$C = \{(m_1, m_2) \mid \forall m_1, m_2 \in M, [m_1$ *and* m_2 *are conflicting*$]\}$.

Definition 7 (Mutation Compatibility Graph). *An undirected graph $G = (M, C)$ over a set of mutations M, where undirected conflict edges $C \subset M \times M$ are placed between every pair of compatible mutations in M:*
$C = \{(m_1, m_2) \mid \forall m_1, m_2 \in M, [m_1 \text{ and } m_2 \text{ are compatible}]\}$.

Figure 6 shows the steps taken to perfectly batch mutations based on conflicting test cases, explained through graphs. First, a mutation conflict graph (Figure 6a) is constructed by placing edges between every pair of conflicting mutations. Then, a mutation compatibility graph (Figure 6b) is constructed as the complement graph of the mutation conflict graph. Finally, compatible mutations are batched (Figure 6c) by partitioning the mutation compatibility graph into cliques, each clique representing a mutation batch. This ensures that for each batch, every mutation is compatible with every other mutation in the batch.

Garey and Johnson [17] write that clique cover is solvable in polynomial time in the following cases:

- for $K \leq 2$,
- for graphs containing no complete subgraphs on 3 vertices,
- for circular arc graphs,
- for chordal graphs, and
- for comparability graphs.

From these, comparability graphs stand out as potentially applicable. However, this requires that the edges of the graph be defined by a strict partial order.

Mutation Compatibility Relation. Let ΣM be the set of all possible mutations. Let $E_t(m), \forall m \in \Sigma M$ be the set of test case entry points that mutation m is reachable from based on the call subgraph of the test case entry points. Let c_m be the mutation compatibility relation on set ΣM of all mutations, where

$$\forall m_1, m_2 \in \Sigma M : m_1 \neq m_2 \implies [m_1 \; c_m \; m_2 \equiv E_t(m_1) \cap E_t(m_2) = \emptyset]$$

The mutation compatibility relation c_m holds the following properties:

(1) **Reflexivity**: The relation c_m is reflexive: $\forall m \in \Sigma M : m \; c_m \; m$, because every mutation may be batched with itself, producing an equivalent batch with the same mutations.
(2) **Symmetry**: The relation c_m is trivially symmetric: $\forall m_1, m_2 \in \Sigma M : m_1 \; c_m \; m_2 \implies m_2 \; c_m \; m_1$, and can be represented for any set of mutations M using an undirected graph.
(3) **Intransitivity**: Let $E_t(m_1) = \{t_1, t_2, t_3\}$, $E_t(m_2) = \{t_4, t_5\}$, $E_t(m_3) = \{t_2\}$. Then, $m_3 \; c_m \; m_2 \land m_2 \; c_m \; m_1$, but $m_3 \; \cancel{c_m} \; m_1$. Therefore, the relation c_m is not transitive.

From this, it follows that mutation compatibility is not a partial order and as such does not form a poset. Mutation batching remains an NP-hard instance of clique cover (i.e., partition graph into cliques). Most importantly, the lack of transitivity of mutation compatibility makes any exact search method unfeasible in general. As Gramm et al. [19] wrote, “Sensible inputs for clustering problems are expected to exhibit transitivity in the sense that if $\{a, b\}$ and $\{b, c\}$ are edges, then probably also $\{a, c\}$ is an edge (that is, its clustering coefficient is high)”. Search methods benefit greatly if they can rely on the relation being (mostly) transitive as it can greatly reduce the effective search depth.

Through experimentation with existing implementations of established algorithms [8, 19, 24], we were able to confirm that perfectly batching mutations through graph algorithms is indeed not computationally feasible, even on our smallest subject, given the number of graph edges (i.e., conflicts and compatibilities) and the resulting enormous search space.

```
Require: Set of Mutations $M$
         Sorting Heuristic Function $f_{sort} : M \to \mathbb{R}$
         Probability $\epsilon \in [0, 1]$ ▷ only in $\epsilon$-greedy algorithm
Ensure: Set of Mutation Batches $\mathbb{B}$
  Set of Mutation Batches $\mathbb{B} \leftarrow \{\}$
  sort $M$ by the element-wise value of $f_{sort}$
  for all Mutation $m \in M$ do
      if $m$ is unsafe then          ▷ (see Section 3.5 for the definition of mutation unsafety)
          Mutation Batch $B' \leftarrow$ None
      else if RANDOM$(0, 1) \leq \epsilon$ then          ▷ only in $\epsilon$-greedy algorithm
          Mutation Batch $B' \leftarrow$ RANDOMCOMPATIBLEBATCH$(B, m)$
      else
          Mutation Batch $B' \leftarrow$ FIRSTCOMPATIBLEBATCH$(B, m)$
      end if
      if $M' \neq$ None then
          $B' \leftarrow B' \cup \{m\}$
      else
          $\mathbb{B} \leftarrow \mathbb{B} \cup \{\{m\}\}$
      end if
  end for
```

Fig. 7. Greedy, and ϵ-greedy algorithm for creating a static batching $\mathbb{B}$ of non-conflicting mutations M, based on the mutation sorting heuristic function f_{sort}, which determines the order in which mutations are considered for batching. The additions in the ϵ-greedy algorithm compared to the original greedy algorithm are highlighted. See Figure 8 for the selection functions FIRSTCOMPATIBLEBATCH and RANDOMCOMPATIBLEBATCH.

With all of these issues in mind, we need to look at how we can approximate these solutions instead.

3.4.2 Algorithms for Approximating Mutation Batching. We developed two fast approximation algorithms for mutation batching: a fully-deterministic greedy algorithm (originally published in our ICST conference paper [36]), and a new, partially-deterministic algorithm that we call epsilon-greedy (ϵ-greedy) batching. Figure 7 shows both of these algorithms, with the additions of the new epsilon-greedy algorithm highlighted. Both the greedy and epsilon-greedy algorithms create non-conflicting batches $\mathbb{B}$ of mutations M by first sorting mutations based on the mutation sorting heuristic function $f_{sort} : M \to \mathbb{R}$, and then working through that list, in that order, adding mutations to the first mutation batch they do not conflict with (see Figure 8a), and creating new mutation batches as necessary. For each mutation m, the choice of a compatible batch is stored in B', and if a compatible mutation batch is found, then mutation m is added to mutation batch B', and if a compatible batch is not found, then a new mutation batch is created with only mutation m as its initial member. This means that the first mutation is always stored in a new, "first" mutation batch, and future iterations either add mutations to the mutation batches created up to that point, or create a new mutation batch which may get extended by later iterations.

The epsilon-greedy algorithm is an extension of the greedy algorithm that takes an additional ϵ parameter, where each mutation has an ϵ probability of being placed into a random compatible mutation batch (see Figure 8b), rather than the first compatible mutation batch. These occasional probabilistic choices help the epsilon-greedy algorithm escape local minima that the greedy algorithm would be "stuck in", which happens if mutations in the existing set of mutation batches prevent further mutations from being batched due to conflicts between them. It is worth noting that the epsilon-greedy algorithm is equivalent to the greedy algorithm when $\epsilon = 0$.

```
function FirstCompatibleBatch(Set of Mutation Batches $\mathbb{B}$, Mutation $m$)
    for all Mutation Batch $B' \in \mathbb{B}$ do
        if $\forall m' \in B'$ : [$m$ and $m'$ are reachability-exclusive] then
            return $B'$
        end if
    end for
    return None
end function
```

(a) Selection function FirstCompatibleBatch chooses the first compatible mutation batch — which only consists of mutations compatible with the input mutation m — from the current working set of mutation batches $\mathbb{B}$.

```
function RandomCompatibleBatch(Set of Mutation Batches $\mathbb{B}$, Mutation $m$)        ▷ only in $\epsilon$-greedy algorithm
    Set of Mutation Batches $\mathbb{B}_{compatible} \longleftarrow \{B' \in \mathbb{B} \mid \forall m' \in B' : [m \text{ and } m' \text{ are reachability-exclusive}]\}$
    return choose random element from $\mathbb{B}_{compatible}$, if any
end function
```

(b) Selection function RandomCompatibleBatch chooses a compatible mutation batch — which only consists of mutations compatible with the input mutation m — through random sampling from the current working set of mutation batches $\mathbb{B}$.

Fig. 8. Methods of selecting compatible mutation batches used in each iteration of the greedy, and ϵ-greedy algorithms. The RandomCompatibleBatch method in (b) is only used by the ϵ-greedy algorithm. Both selection functions return the empty value *None* if there are no mutation batches compatible with the input mutation m in the current working set.

To support multiple mutation sorting heuristics, the greedy, and epsilon-greedy algorithms take an $f_{sort} : M \longrightarrow \mathbb{R}$ comparison function, which is used to sort the list of mutations before they are considered for batching, in that order. This ordering greatly influences the outcome of these algorithms. The following ordering heuristics were considered for the greedy, and epsilon-greedy algorithms:

(1) **Ascending ordering by number of conflicts**: $m \mapsto |\{m' \in M \mid m \text{ and } m' \text{ are not reachability-exclusive}\}|$.
This ordering ensures that the least "conflicting" mutations are batched first, with the goal of creating as large mutation batches as possible, by first combining mutations that are less conflicting to make batches that are less conflicting overall, which can later incorporate more mutations.
(2) **Descending ordering by number of conflicts**: $m \mapsto -|\{m' \in M \mid m \text{ and } m' \text{ are not reachability-exclusive}\}|$.
This ordering ensures that the most "conflicting" mutations are batched first, with the goal of creating fewer really large, and more, evenly sized mutation batches.
(3) **Random ordering**, i.e., shuffling (baseline).
This ordering, which shuffles the mutations, is used as a baseline with which to compare the effects of heuristic orderings in our evaluation (Section 4).

Originally, we ordered the list of mutations by the number of conflicts each individual mutation had, in descending order. This was done with the goal of creating fewer very large batches and more, evenly sized ones from the set of mutations, as the algorithm had to batch the most "conflicting" mutations first. This choice was based on findings from anecdotal initial experimentation that we did not explore further in the original ICST conference paper [36].

In addition to the greedy, and epsilon-greedy algorithms outlined above, we also implemented a baseline random batching algorithm, which for every mutation m in mutations M, places the mutation into a random compatible mutation batch according to RandomCompatibleBatch in Figure 7, if any. If no compatible mutation batch exists for the given mutation, the mutation is placed into a new mutation batch instead. This effectively mimics the behavior of the original

```
let xs = [0; 3];
let i = 100;
let el = unsafe { xs.get_unchecked(i) };
```

(a) Example of unsafe code depending on its safe, but incorrect enclosing scope.

```
fn size() -> usize {
    100
}

let xs = [0; 3];
let el = unsafe {
    let i = size() - 1;
    xs.get_unchecked(i)
};
```

(b) Example of unsafe code depending on a call to a safe, but incorrect function. The issue becomes clear if the body of the called function is inlined.

Fig. 9. Unsafe code has hidden dependencies on the correctness of both its enclosing scope, and its called scopes.

greedy algorithm, but each choice of mutation placement is entirely random, rather than the greedy algorithm's method of placing mutations into the first compatible mutation batch. The random algorithm also performs no sorting or shuffling before iterating over all mutations.

It is trivial that the epsilon-greedy algorithm is equivalent to the greedy algorithm when $\epsilon = 0$. However, similarly, it is also equivalent to the random algorithm when $\epsilon = 1$. This means that the epsilon-greedy algorithm behaves very differently depending on the value of ϵ. In our approach, with low values of ϵ (i.e., $\epsilon \approx 0.1$), we use the epsilon-greedy algorithm as a way of augmenting the greedy algorithm with a small number of random choices.

It is important to note that only safe mutations — mutations defined in safe subtrees of the program — may be safely evaluated in parallel, without any undefined behavior. Mutations that are inside `unsafe` blocks of code or are invoked by unsafe code are not guaranteed to uphold the necessary guarantees. As such, unsafe mutations are put into their own singleton mutation batch, i.e., a mutation batch which is a singleton set containing one and only one mutation. We discuss these safety characteristics, and mutation safety next.

3.5 Mutation Safety — Avoiding the Spread of Unsafety

In Rust, *safety* is defined by the Rust project developers [56] as the guarantee that code cannot cause undefined behavior: no dangling pointers, no use-after-frees, no out-of-bounds memory accesses, etc. Because of this, Safe Rust code, by itself, cannot cause undefined behavior. Unsafe Rust code has no such guarantees, and allows for operations that may introduce undefined behavior. These operations however are required to be annotated explicitly, by wrapping the code in an `unsafe` block. This strict separation of safe and unsafe code allows for new considerations to be made when applying mutation testing to Rust.

There is an intricate, asymmetric trust relationship between the way safe and unsafe Rust code interacts. Safe Rust has to trust that any Unsafe Rust it interacts with was written correctly, and that the safety invariants assumed by the compiler have been upheld. Unsafe Rust on the other hand cannot trust any Safe Rust it interacts with, without care, and must be resilient to incorrect (but not undefined) behavior exhibited by Safe Rust code, as stated by the Rust project developers [56]. In practice, however, due to the difficulty of writing resilient unsafe code, embeddings of Unsafe Rust often depend on the correctness of the enclosing safe code section — its context (Figure 9a), and the correctness of the safe code it calls (Figure 9b) for their own correctness [15]. Therefore, the relationship between safe and unsafe code must be carefully considered, when introducing code changes into the program through generated mutations.

```
fn x { [-]
    fn y { [context-tainting]
        unsafe { [unsafety]
            fn z { [call-tainting] }
        }
        fn w { [extended call-tainting] }
        unsafe fn u { [unsafety]
            fn v { [call-tainting] }
            fn r { [call-tainting, context-tainting]
                unsafe { }
            }
        }
    }
}
```

Fig. 10. Illustration of the mutation safety rules on a scope-level. Each scope is either an inlined call to a function or an `unsafe` block of Rust code. Mutations have the safety of their containing scope. Within each scope, an annotation is placed in `[]` brackets to signify applications of the safety rules, where `[-]` means no rule, i.e., safety. The body of function `y` is tainted because it contains `unsafe` blocks. Function `z` is tainted because it is called from an `unsafe` block in function `y`. Function `w` is tainted because it is defined in a body that contains `unsafe` blocks. Function `u` is defined as `unsafe`. Function `v` is tainted because it is called from `unsafe` function `u`. Function `r` is tainted because it is called from `unsafe` function `u`, and it also contains an `unsafe` block.

In entirely Safe Rust, according to the safety rules mentioned above, mutations may cause undesired behavior but they will never introduce undefined behavior. Mutations introduced into unsafe code however — including otherwise safe code that unsafe code might (incorrectly) rely on — are likely to lead to the introduction of undefined behavior which was not present in the original code. While this may be desirable to test for, such undefined behavior-inducing mutations still have to be differentiated. For example, they have to be tested in a separate process to guard against the newly-introduced undefined behavior causing the mutation test evaluation to crash or otherwise behave incorrectly. We introduce the notion of safe and unsafe mutations to differentiate between mutations that may or may not cause undefined behavior to be introduced, based on their location in safe and unsafe scopes respectively.

Definition 8 (Unsafe mutation). *We consider mutation m* unsafe *if it fulfills at least one of the following conditions:*

(1) m has a direct or indirect `unsafe` *block parent in the function body it is located in.*
(2) m is in a function body with an `unsafe` *block but is not a direct or indirect child of an* `unsafe` *block. We refer to this rule as* context-tainting.
(3) m is in the body of a function which is called directly or indirectly from an `unsafe` *block. We refer to this rule as* call-tainting.
(4) m is in the body of a function which is called directly or indirectly from a function body with an `unsafe` *block, but the call is not a direct or indirect child of an* `unsafe` *block. We refer to this rule as* extended call-tainting.

Definition 9 (Safe mutation). *We consider mutation m* safe *if it is not* unsafe *according to the definition above.*

A safe mutation is guaranteed to not introduce undefined behavior into the program when applied. Figure 10 shows an outline of how safe code becomes tainted by unsafe code as the call tree is traversed from an entry point, according to the rules defined above. Tainted scopes — scopes which are matched by one of the tainting rules — are part of the program's extended unsafe scope. Mutations in the same scope have the same safety, and mutations in tainted or unsafe scopes (i.e., the extended unsafe scope) become unsafe.

Mutation safety extends and complements the program safety rules of the Rust language, reflecting on the asymmetric trust relationship between safe and unsafe Rust code. This similarity also extends to the expectations the Rust language — and by extension our mutation safety rules — have towards program code, specifically unsafe code sections. The Rust language makes no guarantees about any program behavior if the program contains incorrectly implemented unsafe code sections that might lead to undefined behavior. Because of this, our mutation safety rules also cannot make any guarantees about such incorrectly implemented unsafe code in the existing program code. For example, if a safe function is declared with some "interior" unsafe code inside, then it is the responsibility of its authors to ensure that the unsafe code follows the rules required from it by the Rust language. If those requirements are not met, then it can no longer be considered a valid Rust program, according to the Rust language rules, and as such the rules of mutation safety will also not apply correctly. However, the rules of mutation safety ensure that, given valid Rust code, which must by definition have implemented all of its unsafe code correctly according to the language contract, we will not introduce any undefined behavior using safe mutations, and only unsafe mutations may do so.

The rules of mutation safety must also be considered when designing mutation operators. Because mutation safety is based on the call graph analysis of the original, unmutated program, any mutation introducing new function calls must carefully consider mutation safety rules. In unsafe and tainted contexts, all mutations introduced will be considered unsafe by default. In safe contexts, mutations which introduce calls to only safe functions can be considered safe mutations according to the rules of mutation safety. However, any mutation which introduces a call to an unsafe function will not just result in an unsafe mutation, but will also affect the safety of its containing function scope. Of our eighteen mutation operators, only three of them can introduce new function calls into the program: **ArgDefaultShadow**, **CallDelete**, and **CallValueDefaultShadow**. All three of these mutation operators introduce a call to the type-dependent `Default::default` trait (i.e., interface) method, which is declared safe. Thus, they will generate safe mutations in safe contexts.

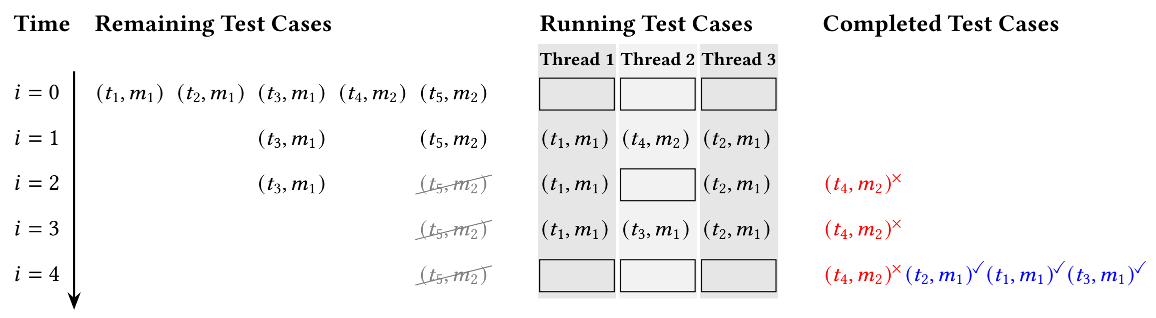
3.6 Parallelized Test Evaluation

Our approach generates an instrumented program, which can be executed to perform mutation analysis. This program includes conditionally branching code for all mutations, metadata representing the mutations and mutation batches, and a generic mutation test harness. The harness acts as the main control loop of the program, iterating over mutation batches, enabling and disabling mutations in the program, and evaluating the tests corresponding to mutations.

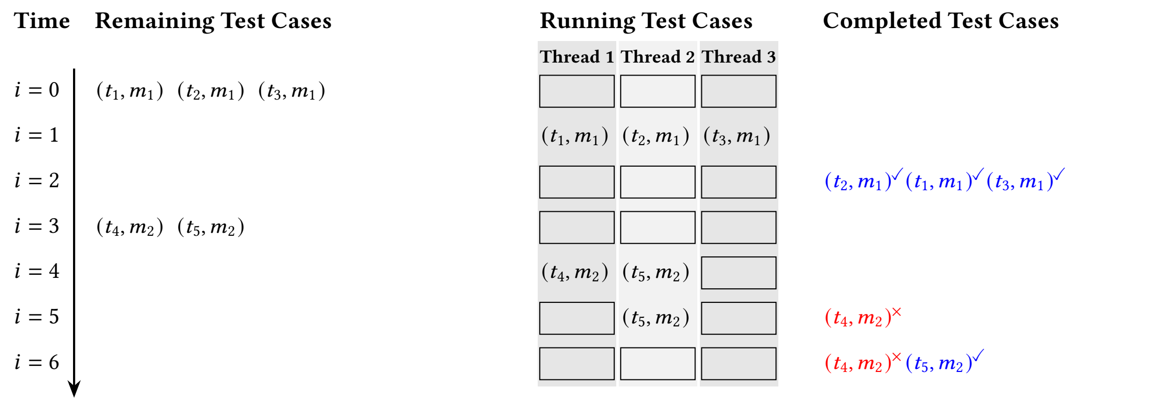
First, all tests are evaluated without any mutations applied. The information from this profiling test run is used to sort tests by execution time. This ordering is later used for further test runs, in anticipation that the majority of the time, mutations will not change the execution time of any test significantly. The results of the profiling test run is also used to establish the timeout duration for tests, based on their execution time. This is important, since mutations may change the code paths of the program in ways that can cause infinite loops, or just increase execution time significantly. Compared to the overall test timeout we used in the original ICST conference paper [36], which was based on the longest running test's duration, we now determine test timeouts $T_{timeout}$ for each test case i individually, as follows:

$$T^{i}_{timeout} = t_i + max(0.1 \cdot t_i, 1\text{s})$$

After the profiling test run, our technique finally performs the mutation analysis by applying each mutation batch one-by-one, and evaluating it. A mutation batch is applied by changing the reference stored in the injected global variable `ACTIVE_MUTANT_HANDLE`, which is referenced in all of the conditionally branching code generated by mutest-rs


Time
Remaining Test Cases
Running Test Cases
Completed Test Cases
Thread 1
Thread 2
Thread 3
$i = 0$
(t_1, m_1) (t_2, m_1) (t_3, m_1) (t_4, m_2) (t_5, m_2)
$i = 1$
(t_3, m_1) (t_5, m_2)
(t_1, m_1) (t_4, m_2) (t_2, m_1)
$i = 2$
(t_3, m_1) ~~(t_5, m_2)~~
(t_1, m_1) (t_2, m_1)
$(t_4, m_2)^{\times}$
$i = 3$
~~(t_5, m_2)~~
(t_1, m_1) (t_3, m_1) (t_2, m_1)
$(t_4, m_2)^{\times}$
$i = 4$
~~(t_5, m_2)~~
$(t_4, m_2)^{\times}$ $(t_2, m_1)^{\checkmark}$ $(t_1, m_1)^{\checkmark}$ $(t_3, m_1)^{\checkmark}$

(a) Test evaluation of a single batch of mutations, timestep-by-timestep. At $i = 1$, the test runner starts running (t_1, m_1), the fastest test case for mutation m_1; (t_4, m_2), the fastest test case for mutation m_2; and (t_2, m_1), the second fastest test case for mutation m_1. At $i = 2$, test case (t_4, m_2) fails, which means that mutation m_2 is detected. Thus, the test runner can remove all remaining tests for mutation m_2 from the queue: $\{(t_5, m_2)\}$. At $i = 3$, the test runner schedules the last remaining test case (t_3, m_1) in place of the now completed test case (t_4, m_2).


Time
Remaining Test Cases
Running Test Cases
Completed Test Cases
Thread 1
Thread 2
Thread 3
$i = 0$
(t_1, m_1) (t_2, m_1) (t_3, m_1)
$i = 1$
(t_1, m_1) (t_2, m_1) (t_3, m_1)
$i = 2$
$(t_2, m_1)^{\checkmark}$ $(t_1, m_1)^{\checkmark}$ $(t_3, m_1)^{\checkmark}$
$i = 3$
(t_4, m_2) (t_5, m_2)
$i = 4$
(t_4, m_2) (t_5, m_2)
$i = 5$
(t_5, m_2)
$(t_4, m_2)^{\times}$
$i = 6$
$(t_4, m_2)^{\times}$ $(t_5, m_2)^{\checkmark}$

(b) Test evaluation of mutations without batching, timestep-by-timestep. At $i = 3$, evaluation of the second mutation starts, independent from the first mutation. At $i = 5$, test case (t_4, m_2) fails, which means that mutation m_2 is detected, however, the test runner cannot stop execution there, as (t_5, m_2) is already running, despite its result not influencing the already known final outcome.

Fig. 11. Test evaluation of a single batch of mutations (showing the first batch of Figure 6c), and the same mutations without batching, timestep-by-timestep. The empty box represents an empty thread with no running test case. Passed test cases are shown in blue, failed test cases are shown in red.

(Figure 2). Once the mutation batch is applied, the harness evaluates the test cases corresponding to the mutations in the batch, determining the detection of each mutation separately based on the results of the corresponding test completions.

If a test runs for longer than the automatically determined test timeout $T^i_{timeout}$, then its corresponding thread is abandoned. The thread is kept running, but may terminate later, avoiding the potential for undesired state that would be caused by forcibly terminating the thread. The parallel test runner used by the harness works using a fixed number of threads. By modifying the queue of unscheduled tests during the test run, removing tests corresponding to mutations which have already been detected, the number of evaluated test cases can be reduced. However, it is worth noting that mutation batching can be used to evaluate the full test–mutation detection matrix, if desired, by disabling this test culling behavior. This is useful for performing additional testing techniques, for example, mutation subsumption analysis [33], test generation [16, 20], test suite prioritization [13, 38, 49], and fault localization [40, 43]. In mutest-rs, an exhaustive mutation evaluation can be performed, with or without the use of mutation batching, by adding the `--exhaustive` flag.

In addition, before each mutation batch evaluation, the tests are reordered again using a stable sort, which bubbles up a single test for each mutation in cycles, keeping the relative order — based on execution time — the same. This ensures that the evaluation of all mutations starts as soon as possible, increasing the overall likelihood of a shorter overall test run.

Figure 11 shows an example of how test cases $t_1, t_2, \ldots, t_5$ corresponding to two compatible mutations m_1, m_2 — $E_t(m_1) = \{t_1, t_2, t_3\}$, $E_t(m_2) = \{t_4, t_5\}$ — are evaluated with and without mutation batching. (These mutations, and their corresponding test cases can be batched, since they are only reachable from a non-overlapping set of tests, see the first batch of Figure 6c.) With batching (Figure 11a), the test runner can start the evaluation of test cases for both mutation m_1, and mutation m_2 at the same time, with the help of the cyclic test ordering method described above. Due to multiple mutations being evaluated at the same time, if a mutation's test case is found to be failing, indicating that the mutation was detected, then there is a higher likelihood of fewer unscheduled test cases remaining for the mutation which can all be unqueued, leading to more effective pruning of unnecessary test case evaluations. Most importantly, the available threads are fully utilized throughout the evaluation of the mutation batch. In comparison, without batching (Figure 11b), the test runner has to evaluate mutation m_1, and mutation m_2 separately, one after another. In this scenario, overall thread utilization is much lower; mutations which have few corresponding test cases never fill the available threads, and there is more time spent overall on switching between mutations, when nothing is being evaluated. This also makes it less likely that remaining test cases can be unqueued, as more test cases for the same mutation will already have been started on a thread by the time a failing test case is reached, leading to less effective pruning of unnecessary test case evaluations.

4 Evaluation

We used our tool implementing our technique, mutest-rs, to evaluate a series of research questions. Since Rust is a new language in the field of mutation testing research, we need to evaluate the effectiveness of applying mutation analysis in the first instance, by applying it to commonly-used and critical Rust code. We must also evaluate the effectiveness of our five additional Rust mutation operators. Additionally, regarding mutations, we need to evaluate the potential impact of undefined behavior causing mutations, and our technique for distinguishing them based on Rust's safety. Furthermore, since we have made improvements to the classic mutation analysis workflow in our approach — in particular, our method of batching mutations — we also evaluate our mutation pipeline's improved efficiency. Finally, we must address the potential impact of non-deterministic, flaky tests [44], and evaluate the extent to which they may affect mutation scores. Our research questions, therefore, are as follows:

RQ1: Mutations. How many mutations do traditional and our additional mutation operators produce, and how effective are Rust test suites at detecting them? At what rate are our additional mutations subsumed at, compared to traditional mutations?
We use this RQ to show that our new, additional mutation operators (Section 3.2) — tailored to the unique characteristics of Rust code — are valid, and are comparable to traditional mutation operators in terms of the number of mutations they produce, their detection rate, and the rate at which they are subsumed by traditional mutations.

Table 2. Subjects used in our empirical study in descending order of criticality score. The subjects used are a combination of some of the most downloaded Rust libraries and GitHub projects selected to represent a wide variety of use cases, project sizes, and testing disciplines. Subjects are ordered according to their OpenSSF criticality score, shown in the last column.

Subject	**Description**	**SLoC**	**Unsafe SLoC**[1]	**Unit Tests**	**Crit. Score**
`clap/clap_builder`	Command line argument parser	27941	0 (0.0%)	83	0.79
`rand/rand_distr`	Library for sampling random distributions	6433	0 (0.0%)	128	0.76
`rand/rand`	Library for random number generation	9178	30 (0.3%)	79	0.76
`rand/rand_core`	Interfaces for random number generation	1651	0 (0.0%)	6	0.76
`json/serde_json`	JSON serialization/deserialization library	22360	10 (0.0%)	135	0.73
`regex/regex-syntax`	Regular expression parser	56934	0 (0.0%)	147	0.72
`regex/regex-automata`	Regular expression automatas	64283	540 (0.8%)	129	0.72
`regex/regex`	Regular expression library	11737	0 (0.0%)	51	0.72
`chrono/chrono`	Date and time library	27741	1 (0.0%)	214	0.69
`ripgrep/ripgrep`	Line-oriented search tool, similar to `grep`	12947	1 (0.0%)	114	0.69
`ripgrep/grep-printer`	Printer for grep search results	8949	0 (0.0%)	105	0.69
`ripgrep/grep-searcher`	Regular expression searcher	6361	4 (0.1%)	77	0.69
`alacritty/alacritty`	OpenGL terminal emulator	19948	2609 (13.1%)	78	0.68
`hashbrown/hashbrown`	Port of Google's SwissTable hash map	23516	1803 (7.7%)	103	0.67
`itertools/itertools`	Library to extend iterators	17680	15 (0.1%)	327	0.66
`gleam/gleam-core`	Programming language for the Erlang VM	69970	0 (0.0%)	1719	0.66
`image/image`	Image encoding/decoding/manipulation	35121	17 (0.0%)	212	0.66
`bytes/bytes`	Library for working with bytes	9840	792 (8.0%)	125	0.65
`rustls/rustls`	Modern TLS implementation	30737	0 (0.0%)	202	0.64
`parking_lot/parking_lot`	Synchronization primitives	5136	367 (7.1%)	87	0.62
`exa/exa`	Directory listing tool, similar to `ls`	10986	27 (0.2%)	406	0.61
`bat/bat`	File printing tool, similar to `cat`	8850	0 (0.0%)	74	0.54

[1]Number of SLoC that are considered to be in an unsafe safety context, according to the Rust language rules, including unsafe definitions and blocks (counting signature lines and unsafe block delimiter lines), and excluding unsafe declarations without bodies.

RQ2: Safety. How many generated mutations are unsafe? At what rate are unsafe mutations detected? What is the impact of evaluating these unsafe mutations?
We use this RQ to show that our approach distinguishes unsafe mutations (Section 3.5), and thus mitigates their impact: the adverse effects of crashing mutations and mutations introducing undefined behavior.

RQ3: Reduced runtimes. What are the performance gains produced by reachability-exclusive mutation batching with our old and new algorithms? How does the approach scale with project size and the number of mutations?
We use this RQ to show that our initial findings in our original conference paper [36] hold for a much wider set of commonly-used Rust subject programs.

RQ4: Variance. What is the variance in mutation scores with our subjects?
We use this RQ to show that certain test behaviors, such as flaky tests and test timeouts, may affect mutation scores with and without mutation batching.

4.1 Subjects

We started by listing the top 500 most downloaded Rust library crates — crates are Rust's notion of packages, i.e., a library or a binary — by "Recent Downloads" according to crates.io (Rust's primary crate registry). In addition, we also collected the top 500 most starred Rust repositories — repositories which contain majority Rust code — on GitHub. From the two datasets, we manually excluded crates that were thin wrappers around external libraries, primarily comprised of `unsafe` code, contained at most an insignificant amount of executable code, used non-standard tests or a custom test runner, or had an insignificant amount of unit tests. We combined the two datasets, resolved overlaps manually, and grouped crates by project, i.e., GitHub repository. From this combined dataset, we selected projects with at least 100 test cases in any constituent crate. Within each selected project, we selected constituent crates with at least 75 test cases. The final list of subjects were considered in descending order of the criticality score of the overall project given by the `2022-06-07` OpenSSF criticality score dataset [2, 52]. The full dataset is available in our replication package [35]. In addition to this list of subjects, we included four crates that do not meet the above criteria from the original ICST paper's [36] subjects, namely: `bat/bat`, `parking_lot/parking_lot`, `rand/rand_core`, and `regex/regex`. We included these four, smaller subjects to present comparative findings, and for the the sake of consistency with our original paper.

Table 2 lists the 22 subject crates involved in our experiments. These subjects vary widely in terms of intended purpose, code size, testing methodology, and the resulting number of test cases. The largest subject, in terms of source lines of code, is `gleam/gleam-core` with 69,956 lines, and the smallest is `rand/rand` with 1,651 lines. `gleam/gleam-core` has the most test cases at 1,719, while `rand/rand` has the fewest test cases at 6. Compared to the 10 subjects we used in the original ICST paper [36], these 22 subjects are significantly larger, with the largest subject in terms of the number of test cases going from 406, in the case of `exa/exa`, to 1,719, in the case of `gleam/gleam-core`. The new set of subjects are also a lot more varied in terms of their intended purpose and include, for example, parsers, automatas, interpreters, date and time libraries, data encodings, data structures, synchronization primitives, graphical applications, and command-line tools.

Table 2 also lists the number and percentage of unsafe source lines of code; lines of source code that correspond to unsafe safety contexts, according to the Rust language rules, for each subject. This total includes unsafe definitions and blocks (counting signature lines and unsafe block delimiter lines), and excludes unsafe declarations without executable bodies (e.g., declarations of external C library interfaces). It is worth noting that the exclusion of unsafe declarations without executable bodies is not a concern, since calls to these declared functions will still have to be made from unsafe calling contexts, which mutest-rs counted. From this, we can see that 9 out of the 22 subjects contain no unsafe lines of code, with an additional two subjects containing only 1 unsafe line of code. 18 out of the 22 subjects had at most 0.8% of their source code be unsafe contexts. The subject with the most unsafe source lines of code, `alacritty/alacritty`, had 13.1% of its source lines of code be unsafe contexts, with the majority of these being calls to OpenGL functions. The other three subjects with significant unsafe source lines of code, `bytes/bytes`, `hashbrown/hashbrown`, and `parking_lot/parking_lot`, all make use of unsafe operations to optimize memory accesses in their low-level primitives.

4.2 Methodology

In preparation for the experiments, we forked each subject's source code repository. This was done to pin down the versions of the projects we were testing against, creating a stable test environment.

Rust projects primarily distinguish between two kinds of tests: unit tests, which are written as part of the program, and integration tests, which are individual Rust files placed in a separate `tests/` directory. Additional test kinds include so-called "doctests" that evaluate Rust code blocks in documentation comments for validity, and experimental facilities for writing benchmarks. However these were not considered in our work due to their specialized and experimental natures respectively. This distinction extends to how these tests are compiled; with unit tests being compiled as part of the main program, when testing is performed (test code is omitted otherwise), and with integration test files each analyzed and compiled as individual test programs, separately from the main program or library they correspond to. Because of this, and because Rust compilation works by analyzing each compilation unit separately, mutest-rs cannot currently analyze integration tests by default. While the primary focus of our research is unit tests, some of our subjects' authors made the stylistic choice to write their unit tests in the form of integration tests. Because of this, we converted these standalone "integration" tests into unit tests by moving them into the crate's `src/` directory. This made it possible for mutest-rs to analyze these test cases as well, allowing for a more extensive evaluation of the subject's test suite. All necessary modifications to the subjects required for this evaluation are available in our replication package [35].

We wrote an experiment runner script to automatically perform the analysis on all selected subject crates. The experiment runner invokes mutest-rs multiple times, with different sets of configuration options set each time. All information is parsed automatically by the experiment runner from the verbose output of mutest-rs. In all of our experiments, we configured mutest-rs to discard all unsafe mutations, since unsafe mutations are not guaranteed to be free from undefined behavior and thus cannot be safely parallelized, as we explained in Section 3.4.2 and Section 3.5.

The experiment runner first runs mutest-rs to gather information about the generated mutant batches — including the total number of mutations generated, and the distribution of mutations across batches — with the following configurations presented in the original ICST conference paper [36]:

$\cancel{\mathbf{B}}$	batching disabled; sequential evaluation of mutations with parallelized evaluation of reachable test cases (baseline, corresponds to the example in Figure 11b);
$\mathbf{g}_{\downarrow}$	greedy batching with reverse conflicts ordering, limited to 5 mutations in a batch;
$\mathbf{G}_{\downarrow}$	greedy batching with reverse conflicts ordering, unlimited;

and the following additional configurations new to this paper:

R	random batching (baseline);
$\mathbf{G}_{\uparrow}$	greedy batching with conflicts ordering;
$\mathbf{G}_{\sim}$	greedy batching with random ordering;
$\mathbf{G}_{\epsilon\uparrow}$	epsilon-greedy batching with $\epsilon = 0.1$, and conflicts ordering;
$\mathbf{G}_{\epsilon\downarrow}$	epsilon-greedy batching with $\epsilon = 0.1$, and reverse conflicts ordering;
$\mathbf{G}_{\epsilon\sim}$	epsilon-greedy batching with $\epsilon = 0.1$, and random ordering.

After the initial runs, the experiment runner runs mutest-rs to completion for each previously listed configuration to perform the mutation analysis. In all configurations, the same test ordering, filtering, and multi-threaded scheduling is applied, regardless of the presence of batching. The experiment runner script performed ten evaluations for each configuration to reduce the error in our timing measurements caused by background processes, and non-determinism. The experiment runner script evaluated all configurations with some level of probabilistic behavior — i.e., random batching, greedy batching with random ordering, and epsilon-greedy batching — with 30 different seeds each, with one run performed for each seed. For each evaluation, the experiment runner collects the mutation score, the number of mutations that the test suite detected and did not detect, the number of mutations that timed out, the number of tests that mutest-rs evaluated to determine the detection of mutations in each mutant, and the time each stage of mutest-rs took:

- **function discovery**, which consists of building the call graph of the test suite, determining which functions to mutate, and the mutation safety of each scope (Section 3.3 and Section 3.5);

- **mutation operator application**, the process of applying each mutation operator to every possible location in the mutable functions, producing mutations (Section 3.2);
- **mutation batching** (Section 3.4);
- **code generation**, the process of applying the necessary modifications to the program's AST, and printing the resulting code (Section 3.2);
- **compilation** of the generated program;
- **test profiling**, the evaluation of the unmodified test suite to gather execution time metrics used for test ordering, and determining test timeouts (Section 3.6);
- **mutation evaluation**, the evaluation of every mutant against the test suite (Section 3.6).

The experiment runner also recorded the total runtime of `mutest-rs`, which is effectively the sum of the runtime of each of the relevant stages. This is the end-to-end time that it takes for a user of `mutest-rs` to get the results of mutation analysis from first invoking the tool. In configurations with mutation batching enabled, it includes both the overheads of the mutation batching process, and the possible time gains in the compilation and mutation evaluation stages. We refer to this as the total mutation analysis runtime, and this is the primary subject of RQ3.

Not all stages of `mutest-rs` are run, depending on whether batching is enabled or not, and while most stages are identical across batching configurations, some perform more or less work depending on the presence of mutation batching. While mutation batching requires that all stages of `mutest-rs` are run, the batching disabled ($\not{\mathbf{B}}$) configuration omits the mutation batching step entirely. However, all configurations require that the call graph is built, as it is always used for determining which test cases can reach which mutations. While the runtime of almost all stages should be identical with or without batching, both the main mutation evaluation stage and the compilation stage are expected to take a different amount of time depending on whether batching is used. As batching allows for a more compact representation of the mutant metadata, this can speed up compilation significantly.

To provide more insight into the mutations produced by our five additional mutation operators, we perform a dynamic mutation subsumption analysis on our subject programs. Mutation subsumption analysis is used to identify redundancy in sets of mutations [33], which we use as a metric to compare our five additional mutation operators' mutations to traditional mutations in **RQ1**. Specifically, we use dynamic mutation subsumption analysis, based on the test case–mutation detection pairs resulting from exhaustive mutation analysis. We run these experiments separately from the timed mutation analysis runs, as dynamic mutation subsumption analysis requires an exhaustive evaluation of mutations, running every relevant test case for each mutation regardless of any previous detections.

In addition to these experiments, the experiment runner ran a separate, unique configuration of `mutest-rs`, which performed no mutation batching, but instead generated unsafe mutations in addition to the safe mutations used in every other experiment outlined above. This dataset including both safe and unsafe mutations was used to answer **RQ2**.

We ran the experiments concurrently on 10 core (20 thread) "slices" of a 64 core AMD Ryzen Threadripper 7980X with 32 GiB of RAM each, running Fedora Server 39. We isolated each 10 core "slice" of the host computer using Linux cgroups with each job getting exclusive access to its respective CPU cores. This effectively emulates multiple, identical 10 core computers — representative of common developer systems — at the same time. We used the Slurm workload manager to manage these concurrent "slices" as individual jobs, each running the experiments for a particular subject. We built `mutest-rs` against the `nightly-2024-05-16` version of rustc, and ran it with a thread pool of size 16 for executing the tests. Effectively, each concurrent computer "slice" allocated threads as follows:

- 16 exclusive threads to test execution (part of `mutest-rs`),

Table 3. Number of function calls in each subject crate not directly resolvable to a singular function definition within Rust program code. *Total Calls* refers to the total number of function calls encountered during call graph construction. *Virtual Calls* refers to the number function calls that refer to trait (i.e., interface) methods that can only be precisely resolved dynamically, at runtime. *Dynamic Calls* refers to the number of function calls through opaque function pointer types, which do not represent any exact function definition. *Foreign Calls* refers to the number of function calls made to foreign definitions (e.g. C library interfaces), whose call graph is not analyzed by mutest-rs.

Subject	**Total Calls**	**Virtual Calls**	**Dynamic Calls**	**Foreign Calls**
`alacritty/alacritty`	25848	9 (0.0%)	57 (0.2%)	0 (0.0%)
`bat/bat`	9871	42 (0.4%)	9 (0.1%)	0 (0.0%)
`bytes/bytes`	4697	20 (0.4%)	12 (0.3%)	0 (0.0%)
`chrono/chrono`	15710	5 (0.0%)	16 (0.1%)	0 (0.0%)
`clap/clap_builder`	148752	296 (0.2%)	106 (0.1%)	0 (0.0%)
`exa/exa`	10847	8 (0.1%)	0 (0.0%)	0 (0.0%)
`gleam/gleam-core`	725319	283 (0.0%)	674 (0.1%)	0 (0.0%)
`hashbrown/hashbrown`	17985	40 (0.2%)	25 (0.1%)	0 (0.0%)
`image/image`	130113	347 (0.3%)	200 (0.2%)	0 (0.0%)
`itertools/itertools`	66948	705 (1.1%)	9 (0.0%)	0 (0.0%)
`json/serde_json`	45297	2 (0.0%)	0 (0.0%)	0 (0.0%)
`parking_lot/parking_lot`	7772	1 (0.0%)	23 (0.3%)	24 (0.3%)
`rand/rand`	10841	18 (0.2%)	6 (0.1%)	1 (0.0%)
`rand/rand_core`	221	0 (0.0%)	0 (0.0%)	0 (0.0%)
`rand/rand_distr`	6542	0 (0.0%)	0 (0.0%)	3 (0.0%)
`regex/regex`	3962	20 (0.5%)	9 (0.2%)	0 (0.0%)
`regex/regex-automata`	73166	193 (0.3%)	37 (0.1%)	0 (0.0%)
`regex/regex-syntax`	65202	4 (0.0%)	38 (0.1%)	0 (0.0%)
`ripgrep/grep-printer`	6468	1 (0.0%)	0 (0.0%)	0 (0.0%)
`ripgrep/grep-searcher`	12065	16 (0.1%)	22 (0.2%)	0 (0.0%)
`ripgrep/ripgrep`	3662	43 (1.2%)	3 (0.1%)	0 (0.0%)
`rustls/rustls`	18232	84 (0.5%)	25 (0.1%)	0 (0.0%)

- 1 exclusive thread to test orchestration (part of mutest-rs),
- 1 exclusive thread to invoking experiments, and writing and processing mutest-rs logs (part of the experiment runner), and
- 2 threads not used explicitly by any of our processes (however individual test cases of subjects may spawn additional threads besides the test cases' main thread).

4.3 Threats to Validity

It is important to consider the threats to the validity, and representativeness of the empirical results of this study.

The choice of the mutation operators applied are a threat to internal validity. Using a different set of mutation operators may affect the number of mutations, the resulting batching of mutations, and the runtime of mutation analysis and mutation testing. However, since the majority of the chosen mutation operators are frequently used in mutation testing literature, we consider the reported results to be meaningful.

The potential presence of equivalent mutants is a threat to internal validity, as they can skew observed mutation scores. To control this threat, our mutation operators implement extensive rule-based filtering of equivalent mutants, which we describe in detail in Section 3.2. We also ensured that we used the exact same set of mutants for each tested mutant evaluation configuration, including the baseline non-batched evaluation, inclusive of any potentially remaining equivalent mutants. While we acknowledge that Trivial Compiler Equivalence [42] — a technique for detecting some cases of mutation equivalence based on whether the mutant code gets optimized to the same operations as the original program — is a valid technique for estimating mutation equivalence of individual mutant programs, it is not directly applicable to our meta-mutant based approach. Trivial Compiler Equivalence requires individual mutant programs for the compiler-based optimization, while our approach generates a singular meta-mutant program containing all mutated code branches. We consider the application of a meta-mutant compatible Trivial Compiler Equivalence approach, which considers the equivalence of optimized meta-mutant code branches instead, an item of future work.

The choice of subjects are a threat to external validity. The reported results may be different for other crates. Nevertheless, the analyzed crates vary in terms of program size, number of test cases, complexity and implemented functionality, and cover many of the most common and critical projects currently available [52]. Therefore, we consider the results to be valid, and representative.

Our mutation analysis approach is based on an extensive, conservative static call graph analysis technique. This is used both for test–mutation reachability discovery, as well as for our mutation batching technique. As described in detail in Section 3.3, the implementation of our call graph construction technique handles calls requiring runtime dynamic dispatch — such as virtual calls to trait (i.e., interface) methods and dynamic calls through opaque function pointers — by representing them as multiple call edges to all of the function definitions in the program that could be called from that function call. This conservative approach ensures correctness, at the expense of capturing more than the necessary amount of call relations. However, foreign calls to foreign definitions (e.g. C library interfaces) are not represented fully in our current implementation. The presence of such foreign function calls is a potential threat to internal validity. Table 3 lists the number of total function calls, and the number and percentage of virtual calls, dynamic calls, and foreign calls for each of our subject crates. From this, we can see that virtual calls do not exceed 0.5% of all calls in the large majority of cases and never exceed 1.2% of all calls; dynamic calls never exceed 0.3% of all calls; and foreign calls are only present in 3 of the 22 subjects, with `rand/rand` only having 1 foreign call (0.0% of all calls), `rand/rand_distr` only having 3 foreign calls (0.0% of all calls), and `parking_lot/parking_lot` only having 0.3% of all of its calls being foreign calls. This shows that our call graphs — through our representation of virtual and dynamic calls — are only marginally more expansive than necessary, and that only a small number of unresolved foreign calls are present in 3 out of our 22 subjects.

The dynamic mutation subsumption analysis we perform for **RQ1** is an approximation of "true" mutation subsumption. Since it is based on the test case–mutation detection pairs resulting from mutation evaluation, the results are dependent on the "granularity" of the test suite's ability in distinguishing distinct program behaviors [33]. The use of dynamic mutation subsumption analysis may be a threat to construct validity, however "true" mutation subsumption is "undecidable to compute" [33], and not feasible to compute for our subject programs and their corresponding mutations.

When observing the effects of unsafe mutations in **RQ2**, we only consider those that crash any of their corresponding test cases' respective processes. While we focus on this effect of evaluating unsafe mutations as the most influential in our discussion, we must note that crashes are only one possible side effect of undefined behavior, and many others — like some memory corruption or aliasing violations — may not have obvious, observable side effects. The lack of observing these alternative side effects is a potential threat to construct validity. We consider the analysis of other potential side effects of unsafe mutations as an item of future work.

Defects in our compiler-integrated mutation testing tool, `mutest-rs`, are a threat to construct validity. We controlled the threat in `mutest-rs` by maintaining and running an automated test suite based on several small example programs,

Table 4. Numbers of mutations generated by traditional and additional mutation operators (Table 1), and the number of mutations detected by each subject's test suite with no mutation batching applied, and unsafe mutations excluded. The percentage of detected mutations relative to the total number of mutations in the corresponding category is denoted in parentheses. Amongst traditional and additional mutations, highlighted cells of total mutation counts (in blue) correspond to a higher number of mutations, and highlighted cells of mutation detection rates (in blue) correspond to a lower detection rate in the corresponding subject.

	Safe Mutations		Traditional Mutations		Additional Mutations	
Subject	**Total**	**Detected**	**Total**	**Detected**	**Total**	**Detected**
`alacritty/alacritty`	1809	469 (25.9%)	844	212 (25.1%)	965	257 (26.6%)
`bat/bat`	506	345 (68.2%)	88	53 (60.2%)	418	292 (69.9%)
`bytes/bytes`	422	340 (80.6%)	121	102 (84.3%)	301	238 (79.1%)
`chrono/chrono`	2882	2532 (87.9%)	1338	1142 (85.4%)	1544	1390 (90.0%)
`clap/clap_builder`	1447	537 (37.1%)	468	169 (36.1%)	979	368 (37.6%)
`exa/exa`	888	758 (85.4%)	293	257 (87.7%)	595	501 (84.2%)
`gleam/gleam-core`	4173	4083 (97.8%)	522	512 (98.1%)	3651	3571 (97.8%)
`hashbrown/hashbrown`	273	214 (78.4%)	43	31 (72.1%)	230	183 (79.6%)
`image/image`	4885	3345 (68.5%)	2087	1434 (68.7%)	2798	1911 (68.3%)
`itertools/itertools`	1246	1107 (88.8%)	335	293 (87.5%)	911	814 (89.4%)
`json/serde_json`	1133	955 (84.3%)	489	421 (86.1%)	644	534 (82.9%)
`parking_lot/parking_lot`	163	161 (98.8%)	98	97 (99.0%)	65	64 (98.5%)
`rand/rand`	747	568 (76.0%)	329	257 (78.1%)	418	311 (74.4%)
`rand/rand_core`	179	139 (77.7%)	70	58 (82.9%)	109	81 (74.3%)
`rand/rand_distr`	2126	1442 (67.8%)	1679	1073 (63.9%)	447	369 (82.6%)
`regex/regex`	209	133 (63.6%)	28	23 (82.1%)	181	110 (60.8%)
`regex/regex-automata`	2698	2050 (76.0%)	1029	821 (79.8%)	1669	1229 (73.6%)
`regex/regex-syntax`	2709	2244 (82.8%)	1113	935 (84.0%)	1596	1309 (82.0%)
`ripgrep/grep-printer`	376	318 (84.6%)	165	141 (85.5%)	211	177 (83.9%)
`ripgrep/grep-searcher`	669	557 (83.3%)	301	246 (81.7%)	368	311 (84.5%)
`ripgrep/ripgrep`	97	78 (80.4%)	14	13 (92.9%)	83	65 (78.3%)
`rustls/rustls`	1127	713 (63.3%)	353	229 (64.9%)	774	484 (62.5%)

as well as manually analyzing the generated mutations, mutants, code, and mutation testing results. Moreover, over the course of this study, mutest-rs has generated multiple millions of lines of valid Rust code (disregarding the potential unchecked misuses of Unsafe Rust by explicitly generated unsafe mutations). We therefore conclude that the implementations of the tools used in our experiments worked correctly.

Finally, we make our tool, scripts, data, detailed execution logs, and the repository forks of our subjects available in our replication package [34, 35]. All versions of our tool, mutest-rs, are available at https://mutest.rs.

5 Results

RQ1: Mutations. How many mutations do traditional and our additional mutation operators produce, and how effective are Rust test suites at detecting them? At what rate are our additional mutations subsumed at, compared to traditional mutations?

In this paper and the original conference paper [36], we introduce five new, additional mutation operators in addition to a selection of thirteen traditional mutation operators widely applied in mutation analysis literature (Table 1). To evaluate these additional mutation operators, we look at the number of mutations they produce, and the number of these mutations which are then detected by the respective program's test suite, and compare them to those of our comprehensive set of traditional mutation operators. Table 4 lists the number of mutations produced by all mutation operators, and additionally lists the number of these mutations grouped by whether they were produced by a traditional mutation operator, or an additional mutation operator. In addition, the number of mutations detected by the respective program's test suite — with no mutation batching applied ($\not{B}$) — is listed for each category of mutations, with a percentage of detected mutations relative to the total number of mutations in the corresponding category.

From this, we can determine that for 20 out of 22 subjects, the additional mutation operators collectively generated significantly more mutations than the traditional mutation operators, with only one subject, `rand/rand_distr`, having significantly fewer mutations produced by additional mutation operators than by traditional mutation operators. The largest difference was in the case of `gleam/gleam-core`, where the additional mutation operators produced 3,129 more mutations than the traditional mutation operators. This can be attributed to the subject program making heavy use of function calls, while having fewer instances of complex mathematical or Boolean expressions.

When it comes to the detectability of these additional mutations, less than half of the subjects had a similar rate of detection (within 3%) to the mutations produced by traditional mutation operators. On average, across all subjects, our additional mutations were detected at a very similar rate to the traditional mutations, with a mean rate of 75.5% of additional mutations (median of 79.3%) and 76.6% of traditional mutations (median of 82.5%) being detected by our subjects' test suites. The largest deviation in detection rate was in the case of `regex/regex`, whose additional mutations were detected at a 21.4% lower rate than traditional mutations. In the case of `regex/regex`, traditional mutation operators only produced 28 mutations, of which 82.1% were detected, while the additional mutation operators produced four times more mutations at 110, of which only 60.8% were detected. Similarly, in the case of `ripgrep/ripgrep`, traditional mutation operators only produced 14 mutations, of which 92.9% were detected, while the additional mutation operators produced almost six times more mutations at 81, of which only 78.3% were detected; a difference of 14.5%. The largest difference in detection rate where traditional mutations were detected at a lower rate than our additional mutations was in the case of `rand/rand_distr`, where additional mutation operators produced 73.4% fewer mutations than traditional mutation operators, and the mutation detection rate increased by 18.6%, meaning that more of the additional mutations were detected.

In addition to the mutation detection-based analysis above, we also performed dynamic mutation subsumption analysis on our generated mutations, grouped by their mutation operators. It is worth noting that, as opposed to static mutation subsumption — which gives concrete subsumption relations between either pairs of mutations or all potential mutations that can be generated by pairs of mutation operators applied to the same locations — dynamic mutation subsumption is derived only from test case–mutation detection pairs, and is thus limited by the "granularity" of the test suite in distinguishing distinct program behaviors. Therefore, these numbers can only be seen as an approximation of static mutation subsumption rates. Table 5 lists the rate mutations generated by our additional mutation operators are subsumed by mutations generated by traditional mutation operators, as well as the rate mutations generated by traditional mutation operators are subsumed by other mutations generated by traditional mutation operators, with the latter acting as a baseline for our additional mutation operators. In addition, the table again lists the number of mutations produced by all mutation operators.

From this, we see that the rates of mutation subsumptions between mutations generated by our additional mutation operators and mutations generated by traditional mutation operators are largely comparable, and in some cases even significantly lower. 18 out of the 22 subjects show high rates ($\geq$ 60%) of mutation subsumption amongst traditional mutations. The baseline corresponding mutation subsumption rate of traditional mutations is matched or lower for 19 out of the 22 subjects with CallDelete and CallValueDefaultShadow mutations, 12 out of the 22 subjects by

Table 5. Rate mutations are dynamically subsumed by mutations generated by traditional mutation operators, for each subject crate, based on test–mutation detections. Each value refers to the percentage of mutations of the corresponding kind that are dynamically subsumed by (other) traditional mutations. A crossed-out cell (—) means that the corresponding mutation operator did not generate any mutations for the subject. Highlighted cells, in blue, correspond to mutation subsumption rates of additional mutation operators that match or are lower than the mutation subsumption rate of traditional mutations of the same subject.

		…mutations subsumed by other traditional mutations					
Subject	**Safe Mut's**	**Traditional**	**Arg-Default-Shadow**	**Call-Delete**	**CallValue-Default-Shadow**	**Continue-Break-Swap**	**Range-Limit-Swap**
`alacritty/alacritty`	1809	24.9%	28.0%	22.1%	21.8%	41.7%	23.1%
`bat/bat`	506	53.4%	55.5%	55.4%	53.8%	50.0%	75.0%
`bytes/bytes`	422	81.2%	40.5%	44.5%	44.5%	—	62.5%
`chrono/chrono`	2882	84.2%	74.9%	77.7%	77.7%	100.0%	52.9%
`clap/clap_builder`	1447	35.7%	42.9%	32.5%	32.0%	0.0%	50.0%
`exa/exa`	888	84.3%	53.9%	56.7%	56.7%	100.0%	—
`gleam/gleam-core`	4173	93.5%	69.6%	89.9%	90.9%	81.3%	40.0%
`hashbrown/hashbrown`	273	62.8%	42.1%	43.8%	46.7%	—	0.0%
`image/image`	4885	68.3%	71.6%	60.0%	60.0%	71.4%	77.1%
`itertools/itertools`	1246	86.3%	78.7%	64.4%	63.7%	42.9%	76.5%
`json/serde_json`	1133	84.1%	74.2%	72.1%	68.9%	—	100.0%
`parking_lot/parking_lot`	163	41.8%	14.3%	17.2%	34.5%	—	—
`rand/rand`	747	77.2%	78.6%	66.5%	66.0%	—	75.0%
`rand/rand_core`	179	82.9%	85.7%	73.3%	73.3%	—	60.0%
`rand/rand_distr`	2126	63.4%	95.2%	73.5%	73.5%	66.7%	60.0%
`regex/regex`	209	71.4%	36.4%	34.2%	35.5%	100.0%	80.0%
`regex/regex-automata`	2698	74.7%	82.4%	63.3%	61.9%	41.7%	92.3%
`regex/regex-syntax`	2709	82.7%	75.3%	79.8%	75.9%	65.2%	78.1%
`ripgrep/grep-printer`	376	79.4%	89.7%	76.5%	76.5%	100.0%	100.0%
`ripgrep/grep-searcher`	669	81.4%	87.2%	81.5%	100.0%	83.3%	81.4%
`ripgrep/ripgrep`	97	85.7%	83.3%	77.8%	77.8%	80.0%	—
`rustls/rustls`	1127	62.6%	43.3%	56.8%	55.2%	100.0%	47.8%

ArgDefaultShadow mutations, 12 out of the 19 applicable subjects by RangeLimitSwap mutations, and 7 out of the 16 applicable subjects by ContinueBreakSwap mutations. Overall, we see an improvement in mutation subsumption rates in 68.3% of the subject-additional mutation pairs (where the corresponding mutation was applicable to the subject) compared to the corresponding rate of traditional mutation subsumption.

For all 22 subjects, at least one of the additional mutation operators produced mutations which had lower mutation subsumption rates than traditional mutations. For 17 out of the 22 subjects, at least three of the additional mutation operators produced mutations which had lower mutation subsumption rates than traditional mutations. Across all five additional mutation operators, the mean rate of mutation subsumption was matched or lower for 18 out of 22 subjects compared to traditional mutations, with the maximum increase of 10.4% in the case of `rand/rand_distr`, and the maximum decrease of 33.2% in the case of `bytes/bytes`.

Across all 22 subjects, all five additional mutation operators had lower mean mutation subsumption rates than traditional mutations, with CallDelete and CallValueDefaultShadow having 11.0% and 9.8% lower mean mutation subsumption rates respectively, ArgDefaultShadow and RangeLimitSwap improving on traditional mutations overall with their 7.2% and 6.2% respective lower mean mutation subsumption rates, and ContinueBreakSwap showing similar rates with a 0.7% lower mean mutation subsumption rate compared to traditional mutations.

There is no correlation between the number of mutations generated and the prevalence of mutation subsumptions, with higher numbers of mutations not correlating with higher rates of mutation subsumptions amongst traditional mutations ($\rho = 0.10$), or otherwise. However, rates of mutation subsumptions of our additional mutations by traditional mutations are correlated with rates of mutation subsumptions of traditional mutations by other traditional mutations, with CallDelete and CallValueDefaultShadow being the two most correlated ($\rho = 0.80$ and $\rho = 0.77$ respectively), followed by ArgDefaultShadow ($\rho = 0.66$), ContinueBreakSwap ($\rho = 0.59$) and RangeLimitSwap ($\rho = 0.44$). This is expected, and is most likely to be caused by the respective subject test suites' differing "ability" to distinguish distinct program behaviors.

In conclusion for RQ1, the five additional mutation operators produce, in general, more mutations than the thirteen traditional mutation operators widely applied in mutation analysis literature. These additional mutations are, in the majority of cases, detected at a similar rate to traditional mutations, with a mean detection rate of 75.5% for additional mutations and 76.6% for traditional mutations across our subjects. However, there are big variances in detection rates across the individual subjects, with some subjects detecting additional mutations at a significantly lower rate than traditional mutations, by up to 21.4%, and with other subjects detecting traditional mutations at a significantly lower rate than additional mutations, by up to 18.6%. Our additional mutations are dynamically subsumed, on average, at a lower rate by traditional mutations than traditional mutations are subsumed by other traditional mutations, according to the subjects' test suites, with 68.3% of the subject-additional mutation pairs having matched or lower subsumption rates than the corresponding traditional mutations.

RQ2: Safety. How many generated mutations are unsafe? At what rate are unsafe mutations detected? What is the impact of evaluating these unsafe mutations?

We introduced the notion of mutation safety in Section 3.5 to be able to distinguish between mutations that may introduce undefined behavior into a program, from those that will not. To quantify the impact that these potentially unsafe mutations might have on mutation analysis, we compare the number of unsafe mutations generated, to the number of safe mutations generated. Table 6 lists the number of safe mutations, the number of unsafe mutations, and the number of unsafe mutations which caused a crash for each subject crate, alongside the percentage of unsafe mutations compared to the total number of mutations generated, and the percentage of crashed mutations compared to the number of unsafe mutations generated.

From this, we see that mutest-rs generates unsafe mutations for 6 out of the 13 subjects with any unsafe SLoC (Table 2), as the majority of subjects do not contain enough unsafe code for mutest-rs to generate unsafe mutations. The subjects mutest-rs was able to generate unsafe mutations for are the ones with the most significant amount of unsafe SLoC (Table 2), with the exception of `alacritty/alacritty`, which has no unsafe mutations despite its significant unsafe SLoC count. This can be attributed to the fact that the majority of unsafe SLoC in `alacritty/alacritty`'s code are simple calls to the foreign OpenGL C API functions. However, out of the 6 subjects that mutest-rs was able to generate unsafe mutations for, 2 of the subjects contained crashing mutations, which were all successfully distinguished by our technique, thus avoiding all potentially crashing mutations. Evaluating these crashing unsafe mutations would

Table 6. Number of unsafe mutations compared to the number of safe mutations generated for each subject crate. The percentage of crashed mutations relative to the number of unsafe mutations is denoted in parentheses.

	Safe Mutations		**Unsafe Mutations**		**Crashed Mutations**[1]
Subject	**Total**	**Detected**	**Total**	**Detected**	**Total / Detected**
`alacritty/alacritty`	1809	469 (25.9%)	0	—	—
`bat/bat`	506	345 (68.2%)	0	—	—
`bytes/bytes`	422	340 (80.6%)	303	227 (74.9%)	8 (2.6%)
`chrono/chrono`	2882	2532 (87.9%)	0	—	—
`clap/clap_builder`	1447	537 (37.1%)	0	—	—
`exa/exa`	888	758 (85.4%)	0	—	—
`gleam/gleam-core`	4173	4083 (97.8%)	0	—	—
`hashbrown/hashbrown`	273	214 (78.4%)	86	79 (91.9%)	1 (1.2%)
`image/image`	4885	3343 (68.4%)	0	—	—
`itertools/itertools`	1246	1107 (88.8%)	0	—	—
`json/serde_json`	1133	955 (84.3%)	4	0 (0.0%)	0 (0.0%)
`parking_lot/parking_lot`	163	163 (100.0%)	171	156 (91.2%)	0 (0.0%)
`rand/rand`	747	568 (76.0%)	14	12 (85.7%)	0 (0.0%)
`rand/rand_core`	179	139 (77.7%)	0	—	—
`rand/rand_distr`	2126	1442 (67.8%)	0	—	—
`regex/regex`	209	133 (63.6%)	0	—	—
`regex/regex-automata`	2698	2050 (76.0%)	398	90 (22.6%)	0 (0.0%)
`regex/regex-syntax`	2709	2243 (82.8%)	0	—	—
`ripgrep/grep-printer`	376	318 (84.6%)	0	—	—
`ripgrep/grep-searcher`	669	557 (83.3%)	0	—	—
`ripgrep/ripgrep`	97	78 (80.4%)	0	—	—
`rustls/rustls`	1127	713 (63.3%)	0	—	—

[1]Crashes are only one possible side effect of undefined behavior, and many others – like some memory corruption or aliasing violations — may not have obvious side effects.

have prohibited the efficient evaluation of mutations by crashing the analysis process either directly, or through subtle uncontrollable corruptions introduced into the running process.

For 5 out of the 6 subjects, mutest-rs was able to generate fewer unsafe than safe mutations, with the only exception being the case of `parking_lot/parking_lot`, where mutest-rs generated 163 safe mutations and 8 more unsafe mutations at 171. This can be primarily attributed to `parking_lot/parking_lot` being the smallest subject, in terms of SLoC, with one of the highest proportions of unsafe SLoC (Table 2). 2 subjects had a similar number of safe and unsafe mutations (including `parking_lot/parking_lot`), and 4 subjects had considerably more safe than unsafe mutations.

Among the subjects that we were able to generate unsafe mutations for, the detection rate of unsafe mutations is significantly different from safe mutations in all cases, with the most significant being `regex/regex-automata`, where only 22.6% of unsafe mutations are detected compared to 76.0% of safe mutations, a difference of 53.4%. While there was a larger difference in detection rate in the case of `json/serde_json`, this was only because none of its four unsafe mutations are detected, while 84.3% of its safe mutations are detected. Overall, from the 6 subjects with unsafe mutations available, we can see that the detection rate of unsafe mutations is generally lower than the detection rate of safe mutations, which may suggest that unsafe code is less thoroughly tested.

While the data shows that in the majority of cases no unsafe mutations are generated due to the lack of unsafe code, it also highlights that a portion of unsafe mutations are crashing mutations, which when not properly discerned, can result

Table 7. Mean mutation analysis runtimes in seconds of safe mutations across the various subject crates without mutation batching ($\not{\mathrm{B}}$), and with mutation batching using the various mutation batching configurations (R, $\mathrm{G}_{\uparrow}$, $\mathrm{G}_{\downarrow}$, $\mathrm{g}_{\downarrow}$, $\mathrm{G}_{\sim}$, $\mathrm{G}_{\epsilon\uparrow}$, $\mathrm{G}_{\epsilon\downarrow}$, $\mathrm{G}_{\epsilon\sim}$, as defined in Section 4.2). Highlighted cells, in blue, correspond to the lowest mean mutation analysis runtime for each subject.

Subject	$\not{\mathrm{B}}$	R	$\mathrm{G}_{\uparrow}$	$\mathrm{G}_{\downarrow}$	$\mathrm{g}_{\downarrow}$	$\mathrm{G}_{\sim}$	$\mathrm{G}_{\epsilon\uparrow}$	$\mathrm{G}_{\epsilon\downarrow}$	$\mathrm{G}_{\epsilon\sim}$
`alacritty/alacritty`	30.5	36.8	27.8	27.9	28.0	36.7	36.4	36.7	36.8
`bat/bat`	11.2	11.7	10.6	10.4	10.4	11.6	11.4	11.6	11.5
`bytes/bytes`	22.6	28.7	24.2	26.0	31.3	23.2	25.1	25.8	23.5
`chrono/chrono`	76.1	59.5	49.7	50.8	49.6	60.6	60.9	59.9	59.4
`clap/clap_builder`	14.5	23.4	14.2	13.3	12.1	23.2	21.4	22.7	23.0
`exa/exa`	7.6	9.4	6.8	6.8	6.9	9.6	9.6	9.6	9.6
`gleam/gleam-core`	461.0	403.4	397.8	401.2	387.2	406.4	402.2	402.7	401.5
`hashbrown/hashbrown`	20.0	19.5	18.8	18.8	18.6	19.7	19.4	19.4	19.6
`image/image`	545.1	517.1	483.2	535.1	538.6	516.1	494.5	521.4	482.5
`itertools/itertools`	21.5	17.9	13.8	14.8	13.8	17.5	17.5	18.3	18.1
`json/serde_json`	13.8	12.8	9.5	9.0	9.8	12.8	12.4	12.4	12.7
`parking_lot/parking_lot`	161.1	118.9	121.5	119.8	120.0	119.7	120.6	120.0	118.0
`rand/rand`	17.0	18.3	15.7	18.6	18.3	17.7	17.7	19.4	17.7
`rand/rand_core`	0.6	0.8	0.5	0.5	0.5	0.8	0.8	0.8	0.8
`rand/rand_distr`	119.7	125.4	104.7	519.9	482.6	124.4	115.4	307.2	125.6
`regex/regex`	2.8	3.3	2.7	2.8	2.8	3.3	3.2	3.1	3.3
`regex/regex-automata`	1385.6	1295.4	1600.0	1453.7	1454.5	1596.6	1628.2	1493.9	1593.7
`regex/regex-syntax`	117.1	151.2	126.8	134.7	134.1	157.8	161.1	171.7	157.8
`ripgrep/grep-printer`	4.1	4.6	4.0	3.9	3.9	4.6	4.6	4.5	4.6
`ripgrep/grep-searcher`	143.0	147.1	143.8	142.3	142.7	141.4	147.2	145.9	144.4
`ripgrep/ripgrep`	2.8	2.8	2.9	2.9	2.9	3.1	2.8	2.8	3.2
`rustls/rustls`	22.4	20.7	17.3	18.0	10.7	19.1	19.6	20.0	19.3

in the inability to run an efficient mutation analysis evaluation process to completion. Crashing is only one possibly side effect of unsafe mutations, and the remaining unsafe mutations may have each caused silent memory corruption, aliasing violations, or other silent undefined behavior that are difficult to detect, and can all non-deterministically influence the results of mutation analysis if they are not isolated into separate processes.

In conclusion for RQ2, while only 6 out of the 22 subjects resulted in mutations which were unsafe, 2 out of these 6 subjects contained crashing mutations. Our technique is able to successfully distinguish unsafe mutations from safe mutations, and thus avoid all crashing mutations in all cases. From the 6 subjects with unsafe mutations available, we can determine that unsafe mutations are generally detected at a lower rate than safe mutations, by up to 53.4% less, which may suggest that unsafe code is less thoroughly tested.

RQ3: Reduced runtimes. What are the performance gains produced by reachability-exclusive mutation batching with our old and new algorithms? How does the approach scale with project size and the number of mutations?

The primary goal of mutation batching is to reduce mutation analysis runtimes. This is done by increasing the number of test cases that can be evaluated in parallel at any given time, leading to a higher utilization of the available resources, and thus reducing the overall time required for mutation analysis to complete.

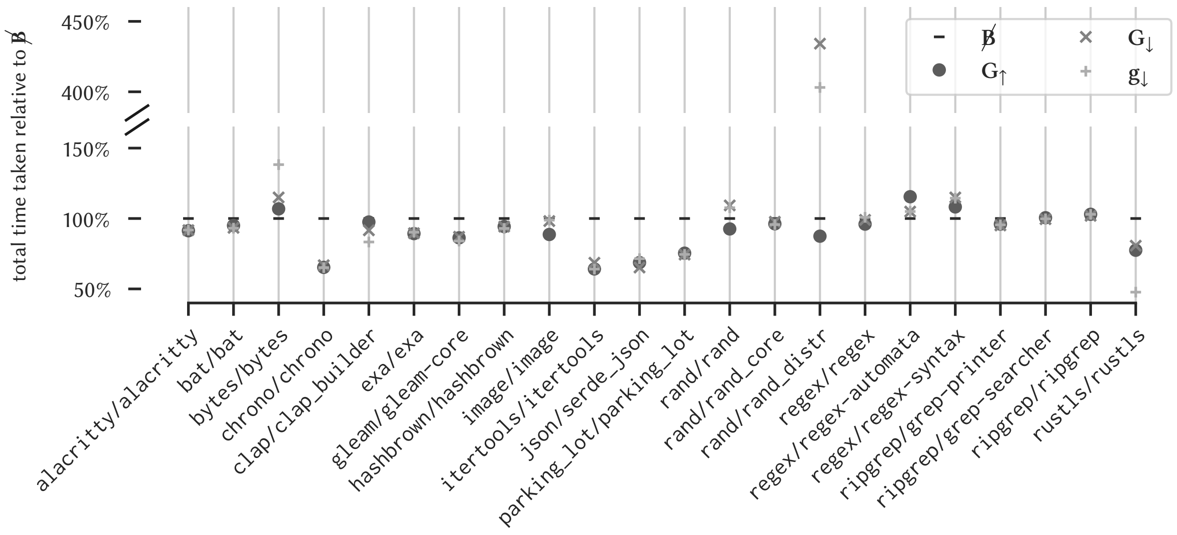


Fig. 12. Mean mutation analysis runtimes of safe mutations across the various subject crates with the various deterministic greedy mutation batching configurations ($G_{\uparrow}$, $G_{\downarrow}$, $g_{\downarrow}$), relative to mutation analysis runtimes without mutation batching ($\not{B}$), in percentages.

Table 7 shows the mean total runtimes of mutation analysis in seconds of safe mutations across the subject crates of various sizes without mutation batching ($\not{B}$), and with mutation batching using the various algorithms and configurations ($\mathbf{R}$, $G_{\uparrow}$, $G_{\downarrow}$, $g_{\downarrow}$, $G_{\sim}$, $G_{\epsilon\uparrow}$, $G_{\epsilon\downarrow}$, $G_{\epsilon\sim}$). These runtimes are the total time taken for mutest-rs to process the subject programs, generate and batch mutations (where applicable), and perform mutation analysis by evaluating the test cases against the mutants, referred to as the total mutation analysis runtime (Section 4.2).

Reduction of Mutation Analysis Runtimes with Deterministic Greedy Mutation Batching. When comparing total mutation analysis runtimes between the non-batched ($\not{B}$), and the greedily, deterministically batched ($G_{\uparrow}$, $G_{\downarrow}$, $g_{\downarrow}$) runs (Figure 12), we see an improvement in 18 out of the 22 subjects in at least one of the batched configurations Among the four non-improved subjects, `ripgrep/ripgrep` shows marginally longer runtimes, by 0.1 seconds, with greedy mutation batching compared to the baseline non-batched runs. The other three non-improved subjects are all significantly impacted by large variances in test case timeouts. In the case of `bytes/bytes` and `regex/regex-syntax`, we see an increase of around 1% in the rate of mutation timeouts (i.e., the proportion of all mutations of the subject which timed out) over the baseline non-batched runs, while in the case of `regex/regex-automata`, all mutation analysis runs are significantly impacted by test case timeouts, with a baseline mean rate of mutation timeouts of 50.0%, rising to between 88.2% and 91.3% during the bathed runs. Among the subjects with improved runtimes, 6 subjects exhibit marginal improvement of less than a second. These are primarily small subjects with unbatched ($\not{B}$) runtimes under fifteen seconds. The other 12 subjects see much more significant improvements. The largest relative improvement in runtime is in the case of `rustls/rustls`, where our approach reduced the original, unbatched ($\not{B}$) runtime by 52.3% using the $g_{\downarrow}$ mutation batching configuration. The largest absolute improvement in runtime is in the case of `gleam/gleam-core`, where our approach reduced the original, unbatched ($\not{B}$) runtime of 461.0 seconds (~ 7.7 minutes) down to 387.2 seconds (~ 6.4 minutes), by 16.0%, using the $g_{\downarrow}$ mutation batching configuration, an improvement of 73.8 seconds, or about 1.2 minutes.

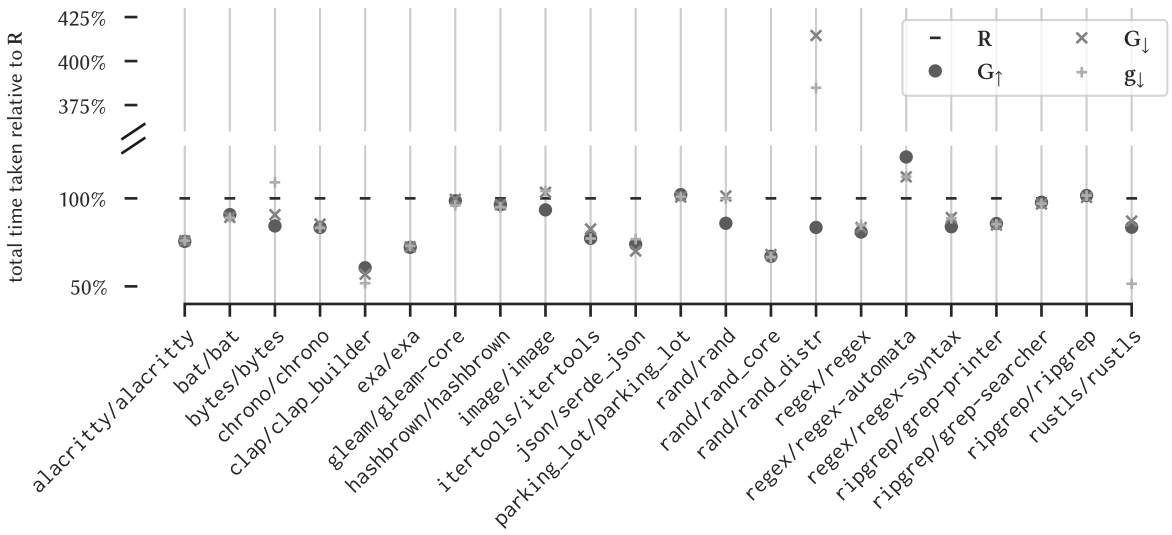


Fig. 13. Mean mutation analysis runtimes of safe mutations across the various subject crates with the various deterministic greedy mutation batching configurations ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$), relative to mutation analysis runtimes with random batching (**R**), in percentages.

Comparing Deterministic Greedy Mutation Batching to Random Mutation Batching. Comparing non-batched ($\cancel{B}$) mutation analysis runtimes to those of the random baseline batches (**R**), we see that 9 subjects have improved mean mutation analysis runtimes with random batching (**R**), and 7 subjects have regressed mean mutation analysis runtimes with random batching (**R**), with the remaining 6 subjects having similar mean mutation analysis runtimes to their unbatched ($\cancel{B}$) runs. Among the 9 subjects with significantly improved mutation analysis runtimes, 4 of the subjects are large — with mutation analysis runtimes over 150 seconds, and 5 of the subjects are medium-sized — with mutation analysis runtimes around 10 to 80 seconds.

In comparison to the baseline random batching (**R**) runtimes, the deterministically batched ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$) runs (Figure 13) see further significant improvements with the subjects where random batching (**R**) already had a significant effect, and an additional 9 subjects where the deterministically batched ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$) runs improve on runtimes, where random batching (**R**) did not. For all but three subjects, at least one of the deterministically batched ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$) runs have significantly lower mutation analysis runtimes than with the baseline random batching (**R**), with the exception of `ripgrep/ripgrep`, `parking_lot/parking_lot`, and `regex/regex-automata`. In the case of `ripgrep/ripgrep` and `parking_lot/parking_lot`, the difference is marginal, with deterministically batched ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$) runs taking 0.1 and 0.9 seconds longer respectively. In the case of `regex/regex-automata`, the deterministically batched ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$) runs have longer runtimes by 10.9%, largely due to a difference in the rate of mutation timeouts.

Comparing The Various Ordering Heuristics Used with Deterministic Greedy Mutation Batching. To look at the effects of the greedy mutation batching algorithm's ordering heuristic, we compare total mutation analysis runtimes between the various greedily, deterministically batched ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$) runs, and the greedily batched runs using random ordering ($G_\sim$), which we use as a baseline (Figure 14).

Compared to the greedily batched runs using random ordering ($G_\sim$), the greedily batched runs using the various conflict orderings ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$) have lower mutation analysis runtimes in the case of 20 subjects, while random ordering

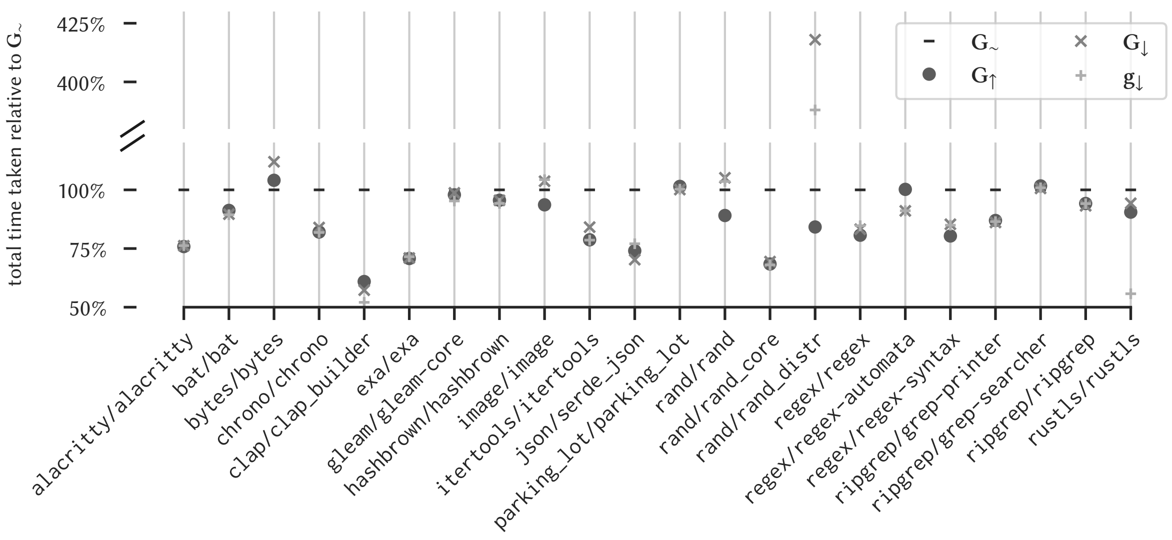


Fig. 14. Mean mutation analysis runtimes of safe mutations across the various subject crates with the various deterministic greedy mutation batching configurations ($G_\uparrow$, $G_\downarrow$, $g_\downarrow$), relative to mutation analysis runtimes with deterministic greedy mutation batching using random ordering ($G_\sim$), in percentages.

has lower mutation analysis runtimes in the case of only 2 subjects. The various conflict orderings thus show a consistent improvement over the baseline random ordering.

Comparing the two conflict orderings with each other ($G_\uparrow$ vs. $G_\downarrow$, $g_\downarrow$), we see that ascending ordering by number of conflicts ($G_\uparrow$) produces lower mutation analysis runtimes in the case of 7 subjects, while descending ordering by number of conflicts ($G_\downarrow$, $g_\downarrow$) produces lower mutation analysis runtimes in the case of 11 subjects. While the difference is marginal in the case of most subjects, it is important to point out the case of `rand/rand_distr`, which has outlier mutation analysis runtimes when applying greedy mutation batching with descending ordering by number of conflicts ($G_\downarrow$, $g_\downarrow$), as mutation analysis runtimes are increased well beyond the original, non-batched runtimes, which is not the case when using ascending ordering by number of conflicts. This is because while only 2.7% of mutations time out when using greedy mutation batching with ascending ordering by number of conflicts ($G_\uparrow$), same rate as with baseline, unbatched runs, using descending ordering by number of conflicts ($G_\downarrow$, $g_\downarrow$) results in 45.9% and 53.1% of mutations timing out respectively.

Overall, we can determine that using either ascending ordering by number of conflicts ($G_\uparrow$) or descending ordering by number of conflicts ($G_\downarrow$) results in lower mutation analysis runtimes at about the same rate, with the difference between the two orderings being marginal in the majority of cases.

The Effects of Limiting the Maximum Size of Mutation Batches on Deterministic Greedy Mutation Batching. To look at the effects of limiting the maximum size of mutation batches on the greedy mutation batching algorithm, we compare total mutation analysis runtimes between two versions of the greedily batched mutation analysis runs, one with no limit on the maximum size of mutation batches ($G_\downarrow$), and one with a limit of 5 on the maximum size of mutation batches ($g_\downarrow$), referred to as "small batching" in the original conference paper [36] (Figure 15). We see that, in the case of 18 subjects, the two configurations have very similar total mutation analysis runtimes, within three seconds of each other,

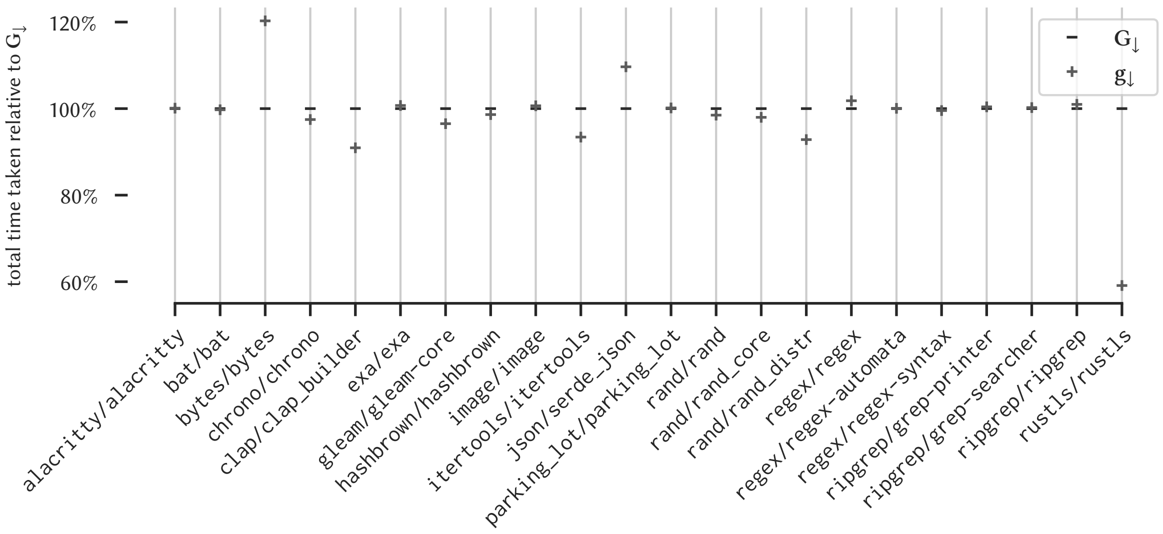


Fig. 15. Mean mutation analysis runtimes of safe mutations across the various subject crates with deterministic greedy mutation batching with a limit of 5 on the maximum size of mutation batches ($g_\downarrow$), relative to mutation analysis runtimes with no limit on the maximum size of mutation batches ($G_\downarrow$), in percentages.

with the remaining 4 subjects showing more significant differences. Amongst the 4 subjects with more significant differences, only one subject, `bytes/bytes`, had worse runtimes by limiting the size of the mutation batches, with the "smaller batches" ($g_\downarrow$) taking 31.3 seconds, 5.3 seconds longer than without limiting the maximum size of mutation batches ($G_\downarrow$), a 20.3% difference. The remaining 3 subjects benefitted from limiting the size of the mutation batches, with `gleam/gleam-core` taking 14.0 seconds less using the "smaller batches" ($g_\downarrow$), a 3.5% difference, `rand/rand_distr` taking 37.2 seconds less, a 7.2% difference, and most significantly `rustls/rustls`, taking 7.4 seconds less with the batch size limits enabled, a difference of 40.9%.

Overall, while limiting the maximum size of mutation batches ($g_\downarrow$) resulted in faster mutation analysis runtimes in the case of 12 subjects, and not limiting the maximum size of mutation batches ($G_\downarrow$) resulted in faster mutation analysis runtimes in the case of 10 subjects, it is still difficult to determine which of them provide a higher reduction in runtimes overall, as the two tested configurations ($G_\downarrow$, $g_\downarrow$) perform similarly across the majority subjects, with the differences being largely marginal.

Comparing Epsilon-Greedy Mutation Batching to Deterministic Greedy Mutation Batching. To look at the effects of the additional ϵ parameter in our new epsilon-greedy mutation batching algorithm, we compare total mutation analysis runtimes between the deterministic non-ϵ ($G_\uparrow$, $G_\downarrow$, $G_\sim$), and the ϵ ($G_{\epsilon\uparrow}$, $G_{\epsilon\downarrow}$, $G_{\epsilon\sim}$) variants of the corresponding greedily batched runs (Figure 16).

First, we compare the baseline batching cases with random ordering ($G_\sim$ vs. $G_{\epsilon\sim}$), where we see the two perform very similarly, with runtimes within five percent of each other in all except one cases, and with the ϵ variant of the greedy mutation batching algorithm having lower mean runtimes in the case of 12 subjects, and the non-ϵ variant having lower mean runtimes in the case of 10 subjects.

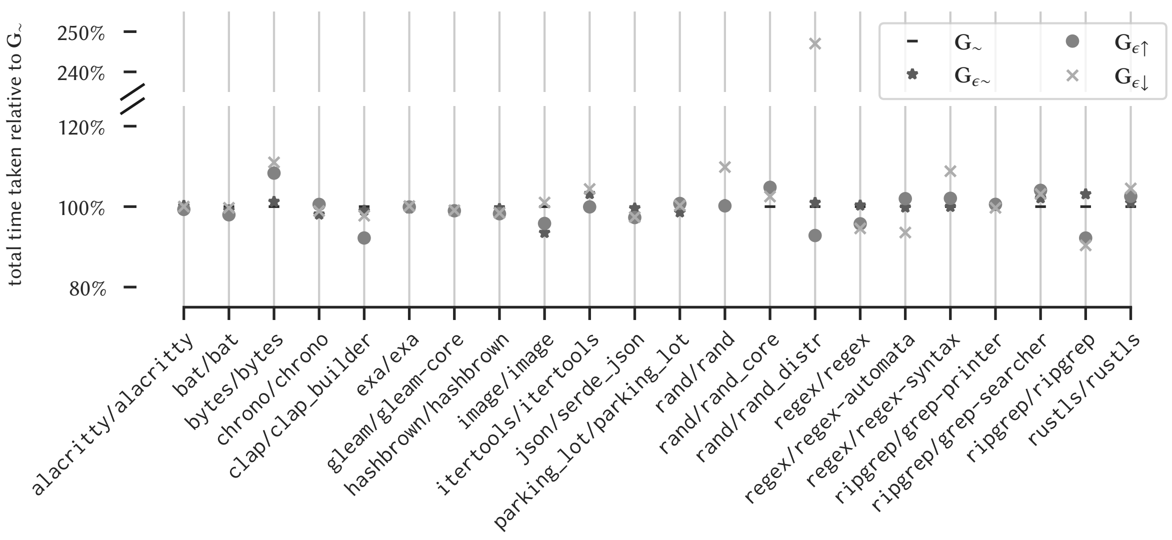


Fig. 16. Mean mutation analysis runtimes of safe mutations across the various subject crates with epsilon-greedy mutation batching ($G_{\epsilon\uparrow}$, $G_{\epsilon\downarrow}$, $G_{\epsilon\sim}$), relative to mutation analysis runtimes with deterministic greedy mutation batching using random ordering ($G_{\sim}$), in percentages.

Comparing the resulting mean mutation analysis runtimes between ϵ and non-ϵ greedy mutation batching with ascending ordering by number of conflicts ($G_{\uparrow}$ vs. $G_{\epsilon\uparrow}$), we see a much bigger difference, with the ϵ variant ($G_{\epsilon\uparrow}$) having lower runtimes in the case of only 2 subjects, and the non-ϵ variant ($G_{\uparrow}$) having lower runtimes in the case of the remaining 20 subjects, with significant differences across almost all subjects. Notably, the ϵ variant ($G_{\epsilon\uparrow}$) performs significantly worse in the cases of `alacritty/alacritty`, `chrono/chrono`, `clap/clap_builder`, `exa/exa`, `json/serde_json`, `itertools/itertools`, and `regex/regex-syntax`.

Similarly, comparing the resulting mean mutation analysis runtimes between ϵ and non-ϵ greedy mutation batching with descending ordering by number of conflicts ($G_{\downarrow}$ vs. $G_{\epsilon\downarrow}$), we see the ϵ variant ($G_{\epsilon\downarrow}$) having lower runtimes in the case of only 4 subjects, and the non-ϵ variant ($G_{\downarrow}$) having lower runtimes in the case of the remaining 18 subjects, with similarly significant differences across almost all subjects. Notably, the ϵ variant ($G_{\epsilon\downarrow}$) performs significantly worse in the cases of `alacritty/alacritty`, `chrono/chrono`, `clap/clap_builder`, `exa/exa`, `itertools/itertools`, `json/serde_json`, and `regex/regex-syntax`. The ϵ variant ($G_{\epsilon\downarrow}$) however performs significantly better, in comparison, in the case of `rand/rand_distr`, with 26.7% fewer mutation timeouts.

Overall, we can determine that using epsilon-greedy mutation batching with $\epsilon = 0.1$ introduces a lot of noise into the algorithm, resulting in generally worse mutation analysis runtimes across the majority of subjects. It is worth noting that the results may differ with different values of ϵ.

Improvements in Compilation Times. Because mutation batching — in addition to improving the time it takes to evaluate mutations through increased parallelism — also reduces the amount of meta-mutant metadata required, it can have a positive effect on compilation times, which is a significant part of the total mutation analysis runtime for compiled languages like Rust. This is especially important for Rust, as it performs significant static analysis on the program during each compilation. Table 8 shows the mean time it took, in seconds, to compile the meta-mutant of safe mutations

Table 8. Mean compilation times in seconds of safe mutations across the various subject crates without mutation batching ($\not{\mathbf{B}}$), and with mutation batching using the various mutation batching configurations ($\mathbf{R}$, $G_{\uparrow}$, $G_{\downarrow}$, $g_{\downarrow}$, $G_{\sim}$, $G_{\epsilon\uparrow}$, $G_{\epsilon\downarrow}$, $G_{\epsilon\sim}$, as defined in Section 4.2). Highlighted cells, in blue, correspond to the lowest mean compilation times for each subject.

Subject	$\not{\mathbf{B}}$	**R**	$G_{\uparrow}$	$G_{\downarrow}$	$g_{\downarrow}$	$G_{\sim}$	$G_{\epsilon\uparrow}$	$G_{\epsilon\downarrow}$	$G_{\epsilon\sim}$
`alacritty/alacritty`	12.8	17.8	8.9	8.9	8.9	17.9	17.4	17.6	17.8
`bat/bat`	2.1	2.6	1.7	1.7	1.7	2.6	2.4	2.6	2.5
`bytes/bytes`	1.3	1.3	0.7	0.8	0.7	1.3	1.2	1.3	1.2
`chrono/chrono`	56.2	41.3	31.4	33.1	31.3	42.4	42.6	41.9	41.3
`clap/clap_builder`	11.2	20.0	10.8	9.9	8.9	19.9	18.1	19.4	19.7
`exa/exa`	4.9	6.8	4.2	4.2	4.2	6.9	7.0	6.9	6.9
`gleam/gleam-core`	208.7	192.8	168.2	167.4	167.7	193.4	190.4	189.1	189.1
`hashbrown/hashbrown`	1.2	1.4	0.9	0.8	0.9	1.4	1.3	1.3	1.4
`image/image`	165.5	103.1	88.6	77.6	78.6	104.1	114.9	97.8	98.0
`itertools/itertools`	11.5	8.0	4.6	4.5	4.6	8.0	8.1	8.0	8.1
`json/serde_json`	10.4	9.8	6.5	6.0	6.9	9.8	9.4	9.5	9.7
`parking_lot/parking_lot`	0.6	0.8	0.5	0.5	0.5	0.8	0.6	0.7	0.9
`rand/rand`	2.9	2.7	1.5	1.5	1.8	2.7	2.7	2.7	2.7
`rand/rand_core`	0.3	0.6	0.3	0.3	0.3	0.6	0.6	0.6	0.6
`rand/rand_distr`	27.2	12.5	9.0	8.5	7.7	12.6	12.7	12.7	12.5
`regex/regex`	1.2	1.7	1.1	1.2	1.2	1.7	1.6	1.6	1.7
`regex/regex-automata`	28.5	41.0	17.2	14.2	14.8	42.6	42.5	40.8	41.4
`regex/regex-syntax`	28.0	66.6	29.7	28.8	29.0	66.9	65.5	66.8	66.3
`ripgrep/grep-printer`	1.6	2.0	1.4	1.4	1.4	2.0	2.0	2.0	2.0
`ripgrep/grep-searcher`	2.9	5.5	3.2	2.9	3.2	6.2	3.6	4.6	6.3
`ripgrep/ripgrep`	1.4	1.4	1.4	1.4	1.4	1.7	1.4	1.4	1.7
`rustls/rustls`	8.5	6.4	4.2	4.2	3.5	6.4	6.5	6.3	6.4

generated by mutest-rs, across the subject crates of various sizes without mutation batching ($\not{\mathbf{B}}$), and with mutation batching using the various algorithms and configurations ($\mathbf{R}$, $G_{\uparrow}$, $G_{\downarrow}$, $g_{\downarrow}$, $G_{\sim}$, $G_{\epsilon\uparrow}$, $G_{\epsilon\downarrow}$, $G_{\epsilon\sim}$).

From this, we see that greedy mutation batching ($G_{\uparrow}$, $G_{\downarrow}$, $g_{\downarrow}$) improves meta-mutant compilation times in all cases, with the exception of `regex/regex-syntax`, where compilation times remain comparable. In some cases, such as when non-batched meta-mutant compilation times are significant, mutation batching improves mutation analysis runtimes significantly through a reduction in compile times. In other cases, such as `parking_lot/parking_lot`, the difference in compilation times is minimal, however mutation analysis runtimes are significantly reduced through the improved parallelism of test evaluation during mutation evaluation with mutation batching. We can also determine that mutation batching with random ordering ($G_{\sim}$), and epsilon-greedy mutation batching configurations ($G_{\epsilon\uparrow}$, $G_{\epsilon\downarrow}$, $G_{\epsilon\sim}$) improve compile times in fewer cases, primarily amongst subjects with longer non-batched compilation times.

Overall Trends Across Mutation Batching Configurations. Overall, across all of the mutation batching configurations, the subjects `gleam/gleam-core`, `image/image`, `parking_lot/parking_lot`, `chrono/chrono`, and `rand/rand_distr` have the most improved mutation analysis runtimes, while `rustls/rustls` sees the most significant improvement in runtimes proportionally, with an improvement of up to 52.3%. Only three subjects, `bytes/bytes`, `regex/regex-automata` and `regex/regex-syntax`, have worse mean mutation analysis runtimes across all of the mutation batching configurations compared to the original, non-batched ($\not{\mathbf{B}}$) runs, primarily due to an increase in the number of timed out mutations.

Among the 22 subjects, only one, namely `rand/rand_distr`, has outlier mutation analysis runtimes with certain mutation batching configurations applied, which is also caused by an increase in the number of timed out mutations.

Large subjects — those with high non-batched mutation analysis runtimes — reliably see improvement across the board, while smaller subjects have more varied results, with the original greedy algorithms performing well. This suggests that we can expect to see similarly large improvements with other large projects as well, and potentially further improvements as project sizes, numbers of tests cases, and mutation analysis runtimes grow further. Looking at the variance in the rate of improvement, we can determine that subjects with shorter non-batched mutation analysis runtimes have more variance in the rate of improvement across the various mutation batching configurations.

Some of the reduction in runtime in general can be attributed to the reduction in the amount of embedded meta-mutant metadata with mutation batching, as the same number of mutations can be represented with significantly fewer mutant program descriptors. In a compiled language, like Rust, the amount of such metadata has a significant impact on compilation times, and as such should be minimized.

In conclusion for RQ3, we see an improvement in 19 out of the 22 subjects in mean mutation analysis runtimes using greedy mutation batching, with improvements of up to 52.3% in the case of `rustls/rustls`, and 73.8 seconds in the case of `gleam/gleam-core`. We can determine that overall, using either ascending ordering by number of conflicts or descending ordering by number of conflicts as an ordering heuristic for greedy mutation batching results in lower mutation analysis runtimes at around the same rate, and that it is difficult to determine whether limiting the maximum size of mutation batches improves mutation analysis runtimes. We see that epsilon-greedy mutation batching with $\epsilon = 0.1$ results in marginally worse mutation analysis runtimes across the majority of subjects.

RQ4: Variance. What is the variance in mutation scores with our subjects?

Mutation batching changes how mutations — and their corresponding test cases — are evaluated, without affecting the detection of individual mutations, and thus the overall mutation score. However, this can only be guaranteed in cases where the test suite shows no behavioral variance across repeated, and potentially slightly differing runs. For example, the presence of flakiness and non-determinism in test suites, or the presence of test timeouts in the test suite can cause the behavior of the test suite to change across repeated runs of identical or slightly different execution environments and configurations. In such cases, certain mutations' detection with mutation batching may be affected by the variance in the test evaluation process, which might result in slightly altered mutation scores. The variance in mutation scores with and without mutation batching — when present — can be primarily attributed to flaky test effects, like order dependence or the presence of shared state, which is commonly regarded as undesirable for robust testing, in part because it prohibits efficient, parallel test execution. It may also stem from other forms of test run variance, like test case timeouts, which may cause a particular test case to either run to completion or time out, depending on the exact timeout and the test environment. It is important to note that mutation batching does not *introduce* these variances, as they are already present across multiple, repeated mutation analysis runs without mutation batching. Mutation batching may only be affected by existing variance in test suite runs. Table 9 shows the mean mutation score of the various subject crates from safe mutations without mutation batching ($\not{\mathbf{B}}$), and with mutation batching using the various algorithms and configurations ($\mathbf{R}$, $\mathbf{G}_{\uparrow}$, $\mathbf{G}_{\downarrow}$, $\mathbf{g}_{\downarrow}$, $\mathbf{G}_{\sim}$, $\mathbf{G}_{\epsilon\uparrow}$, $\mathbf{G}_{\epsilon\downarrow}$, $\mathbf{G}_{\epsilon\sim}$).

Table 9. Mean mutation scores from safe mutations across the various subject crates without mutation batching ($\cancel{B}$), and with mutation batching using the various mutation batching configurations (**R**, $G_\uparrow$, $G_\downarrow$, $g_\downarrow$, $G_\sim$, $G_{\epsilon\uparrow}$, $G_{\epsilon\downarrow}$, $G_{\epsilon\sim}$, as defined in Section 4.2). Highlighted cells, in blue, correspond to mutation scores that equal the corresponding unbatched mutation score for each subject.

Subject	$\cancel{B}$	**R**	$G_\uparrow$	$G_\downarrow$	$g_\downarrow$	$G_\sim$	$G_{\epsilon\uparrow}$	$G_{\epsilon\downarrow}$	$G_{\epsilon\sim}$
`alacritty/alacritty`	25.9%	25.9%	25.9%	25.9%	25.9%	25.9%	25.9%	25.9%	25.9%
`bat/bat`	68.2%	68.2%	68.2%	68.2%	68.2%	68.2%	68.2%	68.2%	68.2%
`bytes/bytes`	80.6%	80.9%	80.6%	80.8%	81.1%	80.6%	80.6%	80.7%	80.6%
`chrono/chrono`	87.9%	87.9%	87.9%	88.1%	87.9%	88.0%	87.9%	88.0%	88.0%
`clap/clap_builder`	37.1%	37.1%	37.1%	37.1%	37.1%	37.1%	37.1%	37.1%	37.1%
`exa/exa`	85.4%	85.4%	85.4%	85.4%	85.4%	85.4%	85.4%	85.4%	85.4%
`gleam/gleam-core`	97.8%	98.0%	98.3%	98.1%	98.1%	98.1%	98.3%	98.1%	98.0%
`hashbrown/hashbrown`	78.4%	78.4%	78.4%	78.4%	78.4%	78.4%	78.4%	78.4%	78.4%
`image/image`	68.5%	68.4%	68.5%	68.5%	68.5%	68.4%	68.5%	68.5%	68.4%
`itertools/itertools`	88.8%	88.8%	88.8%	88.8%	88.8%	88.9%	88.8%	88.8%	88.9%
`json/serde_json`	84.3%	84.7%	84.2%	84.5%	84.6%	84.6%	84.2%	84.5%	84.6%
`parking_lot/parking_lot`	98.5%	97.3%	95.1%	99.5%	99.5%	96.9%	94.6%	98.9%	95.4%
`rand/rand`	76.0%	76.1%	76.0%	76.3%	76.0%	76.0%	76.0%	76.3%	76.0%
`rand/rand_core`	77.7%	77.6%	77.7%	77.7%	77.7%	77.6%	77.6%	77.6%	77.6%
`rand/rand_distr`	67.8%	67.8%	67.8%	81.7%	79.1%	67.8%	67.8%	76.5%	67.9%
`regex/regex`	63.6%	63.6%	63.6%	63.6%	63.6%	63.6%	63.6%	63.6%	63.6%
`regex/regex-automata`	76.0%	88.2%	94.2%	96.9%	96.8%	98.1%	94.6%	97.5%	98.1%
`regex/regex-syntax`	82.8%	82.8%	83.0%	82.9%	82.9%	82.8%	83.2%	82.9%	82.8%
`ripgrep/grep-printer`	84.6%	84.6%	84.6%	84.6%	84.6%	84.6%	84.6%	84.6%	84.6%
`ripgrep/grep-searcher`	83.3%	83.3%	83.3%	83.3%	83.3%	83.3%	83.3%	83.3%	83.3%
`ripgrep/ripgrep`	80.4%	80.4%	80.4%	80.4%	80.4%	80.4%	80.4%	80.4%	80.4%
`rustls/rustls`	63.3%	63.3%	63.7%	63.8%	63.3%	63.4%	63.6%	63.8%	63.5%

We can determine that, out of the 22 subjects, 13 subjects have the exact same mutation score with and without mutation batching, and across all of the various mutation batching configurations. These subjects do not exhibit flaky test and mutation outcomes on repeated invocations, and have either no mutation timeouts at all (7 subjects), or have a very low rate of timed out mutations at less than 0.6% of all evaluated mutations. A further 6 subjects only show differences in their mutation scores within 0.5% with mutation batching compared to their original, non-batched mutation scores, with a median difference of less than 0.3% across all of the various batching configurations. Among the remaining 3 subjects, namely `parking_lot/parking_lot`, `rand/rand_distr` and `regex/regex-automata`, we can determine that their mutation scores are partly determined (or at least largely impacted) by the number of timed out test cases in individual runs, with both the rate of mutation timeouts and mutation scores varying across repeated invocations of the same mutation analysis configuration. In the case of `rand/rand_distr`, only three batching configurations ($G_\downarrow$, $g_\downarrow$, $G_{\epsilon\downarrow}$) show a variance in mutation scores, inline with when the baseline, unbatched mean mutation timeout rate of 2.7% is exceeded, at 45.9%, 53.1%, and 26.4% respectively. In the case of `parking_lot/parking_lot`, mutation timeout rates are consistently high across unbatched and all batched runs, between 85.1% and 97.1%, with variances in the rate of mutation timeouts even across repeated runs. Mean mutation scores closely follow the trends in these mean mutation timeout rates across unbatched and all batched runs, showing that mutation scores are largely dependent on exactly which test cases time out during each mutation analysis run. In the case of `regex/regex-automata`, variances in mutation scores again largely follow variances in the rates of mutation timeouts, with the baseline, unbatched mutation timeout rate of 50.0% exceeded in all batched configurations, between 75.6% and 94.6%. Variances in the rates of mutation timeouts during repeated invocations of the same configuration are also very high, which is also reflected in the variance of mutation scores during these repeated invocations. The median mutation score variance compared to the baseline, unbatched mean mutation score across all subjects is 0.0% in the case of all batched configurations.

The observed variances in mutation scores may be due to a number of reasons, including potential shared state, and environmental factors (e.g., network or filesystem calls). In the case of our subjects, these factors contribute to the variances in the number of test timeouts across unbatched and all batched configurations, which result in the observed variances in mutation scores. However, we theorize that behavior arising from these factors is more likely to cause additional test failures, and as such, additional, false positive mutation detections, explaining the occasional slight, overall increase in mutation scores. It is worth noting that the observed variances in test timeouts occurred despite generous timeouts calculated for each test by mutest-rs (Section 3.6). Amongst our subjects, without mutations applied, the affected tests ran for up to around 1.5 milliseconds only, for which our automatically determined timeouts were just over a second. As such, these non-deterministic test timeouts happened even though our mutation evaluation process gave these test cases orders of magnitude more time to terminate.

In conclusion for RQ4, Mutation batching does not affect the mutation score of deterministic test suites, but may be affected by existing variances in the test evaluation process, such as test timeouts. Out of the 22 subjects, 13 subjects have the exact same mutation score with and without mutation batching, and a further 6 subjects only show differences in their mutation scores within 0.5% of their non-batched mutation scores. Amongst our subjects, the occasional variances in mutation scores can be attributed to variances in the number of test case timeouts.

6 Discussion

6.1 Examining and Analyzing the Mutation Compatibility Graphs

As discussed in Section 3.4.1, mutation compatibility graphs provide the basis for mutation batching, as the problem of mutation batching is equivalent to finding cliques in a mutation compatibility graph. We noted however that, in practice, mutation compatibility graphs are large, complex, and not transitive by definition, making computation of a perfect solution infeasible.

To quantify the complexity of mutation compatibility graphs, and to investigate its clustering properties, we analyze the invariants of the mutation compatibility graph of mutations of each subject; instances of the abstract mutation compatibility graph outlined above. Table 10 shows the number of safe mutations, which correspond with the nodes of the mutation compatibility graph, the number of mutation conflicts (i.e., the number of edges in the mutation conflict graph), the number of mutation compatibilities (i.e., the number of edges in the mutation compatibility graph), the 3-cycle transitivity of the mutation compatibility graph, and the diameter of the mutation compatibility graph corresponding to the safe mutations generated for each subject. Percentages are also noted for mutation conflicts and compatibilities, which are the percentage of the respective edges compared to the complete graph of the corresponding mutations. Since the two edge sets are inverses of each other and together they make a complete graph, these percentages add up to one hundred percent.

Firstly, we see that the ratio of conflicts to compatibilities — i.e., which mutation relation edges correspond to a conflict and to a compatibility respectively, varies highly across the subjects. In the case of 15 out of the 22 subjects, there are more mutation compatibilities than conflicts, while 2 out of the 7 remaining subjects have remarkably low percentages of

Table 10. Properties of the safe mutations — and their corresponding mutation compatibility graphs — produced for each subject. *"Transitivity"* and *"Diameter"* are properties of each subject's mutations' respective mutation compatibility graph. Diameter is not listed for individual subjects separately in a standalone column, as the value is 2 in all cases. Amongst mutation conflicts and compatibilities, highlighted cells, in blue, correspond to the higher of the two values for each subject. For the Transitivity column, highlighted cells, in blue, correspond to values of 0.70 or higher for each subject.

Subject	**Safe Mutations**	**Mutation Conflicts**	**Mutation Compatibilities**	**Transitivity (3-cycles)**
`alacritty/alacritty`	1809	562653 (34.4%)	1072683 (65.6%)	0.56
`bat/bat`	506	24780 (19.4%)	102985 (80.6%)	0.78
`bytes/bytes`	422	13352 (15.0%)	75479 (85.0%)	0.87
`chrono/chrono`	2882	1775029 (42.8%)	2376492 (57.2%)	0.63
`clap/clap_builder`	1447	923064 (88.2%)	123117 (11.8%)	0.16
`exa/exa`	888	119195 (30.3%)	274633 (69.7%)	0.70
`gleam/gleam-core`	4173	5968202 (68.6%)	2736676 (31.4%)	0.40
`hashbrown/hashbrown`	273	8577 (23.1%)	28551 (76.9%)	0.83
`image/image`	4885	3630168 (30.4%)	8299002 (69.6%)	0.75
`itertools/itertools`	1246	76653 (9.9%)	698982 (90.1%)	0.91
`json/serde_json`	1133	248981 (38.8%)	392297 (61.2%)	0.57
`parking_lot/parking_lot`	163	9278 (70.3%)	3925 (29.7%)	0.45
`rand/rand`	747	52713 (18.9%)	225918 (81.1%)	0.81
`rand/rand_core`	179	3679 (23.1%)	12252 (76.9%)	0.71
`rand/rand_distr`	2126	349079 (15.5%)	1909796 (84.5%)	0.84
`regex/regex`	209	10669 (49.1%)	11067 (50.9%)	0.55
`regex/regex-automata`	2698	2220787 (61.0%)	1417466 (39.0%)	0.47
`regex/regex-syntax`	2709	2955912 (80.6%)	712074 (19.4%)	0.21
`ripgrep/grep-printer`	376	17811 (25.3%)	52689 (74.7%)	0.71
`ripgrep/grep-searcher`	669	221538 (99.1%)	1908 (0.9%)	0.00
`ripgrep/ripgrep`	97	4494 (96.5%)	162 (3.5%)	0.33
`rustls/rustls`	1127	101244 (16.0%)	533257 (84.0%)	0.86

mutation compatibilities, with `ripgrep/ripgrep`'s 97 mutations at 3.5%, and `ripgrep/grep-searcher`'s 669 mutations at only 0.9%. It is important to point out that the ratio of mutation compatibilities to mutation conflicts appears to have no strong correlation with the improvements in runtimes measured in **RQ3**, and some of the subjects with the most-improved runtimes have a low mutation compatibility to mutation conflict ratio, such as `gleam/gleam-core`, and `parking_lot/parking_lot`. We do however see a strong correlation between the percentage of mutation compatibilities, and the transitivity of the corresponding mutation compatibility graph.

Secondly, and as a result of the correlation noted earlier, we see that the transitivity of the mutation compatibility graphs varies highly across subjects too, and ranges between 0.16 for `clap/clap_builder`, and 0.91 for `itertools/itertools`, with 11 out of the 22 subjects having mutation compatibility graphs with transitivity at least 0.70. The median transitivity of the mutation compatibility graphs is 0.66. These results indicate that improved, search-based optimization algorithms for mutation batching might be feasible in some cases, and would be able to make further improvements to runtimes with mutation batching; a potential topic of future research. We see that the diameter — the maximum distance between any pair of mutations — of all subject's mutation compatibility graphs is 2, which shows that these graphs are highly connected. This is common for large, complex graphs in general. The one outlier mutation compatibility graph is the one associated with `ripgrep/grep-searcher` with a transitivity of 0.00, however this can be attributed to the subject's remarkably low 0.9% rate of mutation compatibilities.

The ratio of mutation compatibilities to mutation conflicts is highly dependent on how the code is organized. Dividing up the program into more, distinct, individually tested modules gives mutation batching a greater ability to batch mutations further, giving rise to additional reduction in mutation analysis runtimes.

7 Related Work

7.1 Applying Mutation Analysis to Rust Programs

There are two notable pieces of research works on software testing for Rust, however both of these works are focused on test case generation, rather than applying alternative software testing techniques. Takashima et al. [51] used complex program synthesis techniques to generate test suites for libraries. They encoded type, ownership, and lifetime constraints, and were able to generate valid Rust code that utilized complex patterns to interface with library APIs. Their work does an excellent job of highlighting the difficulties of generating valid Rust programs, in particular, for test case generation. In this work, we specifically focused on the challenges of generating Rust code based on existing, valid Rust code through mutations, rather than test case generation. Sharma et al. [48] generated random Rust programs that respected the ownership, borrowing, and lifetime rules of the language in order to test alternative Rust compiler implementations. They did this by generating random, context-aware ASTs which conformed to the grammar of the language. Due to their lack of data flow analysis, they had to place slightly stricter restrictions on their generated code than the actual borrowing rules of Rust, and could not generate code which exercised the lifetime coercion rules of the language. Their technique can be best thought of as an advanced, compiler-specific fuzzing technique, and showcases the limitations of generating Rust programs based on syntactical approaches.

Research into applying testing techniques to Rust is currently sparse. While mutation analysis has been widely applied to many languages [27], this work constitutes the first piece of research on the testing technique that specifically targets the Rust programming language. While, as previously discussed in Section 2, we found two existing, relatively limited hobbyist mutation tools for the language — Bogus's mutagen [7] and Pool's cargo-mutants [46], and noted that LLVM bytecode mutations, like Denisov and Pankevich's [12] work may be adaptable to Rust programs, these approaches both have significant limitations associated with them. Both mutagen, and cargo-mutants only support a very basic set of mutation operators, generate large numbers of invalid mutants due to their limited, syntax-based approaches, and do not optimize mutation evaluation in any way; mutants are compiled, and run one-by-one, sequentially. While there are multiple works discussing generic mutations on LLVM bytecode [9, 12, 22, 23], they all share the same fundamental limitations; mutations are easily introduced in external library code that is not part of the written program code, and many bytecode mutations do not have source code counterparts. This work specifically focuses on source code mutations, which alleviates all of these problems, and performs efficient mutation analysis through a meta-mutant model, rather than by editing bytecode directly.

7.2 Existing Approaches and Tools for Mutation Analysis

It is important to compare our approach to well-established mutation analysis tools targeting other languages. Two mutation analysis tools stand out in terms of their recent research coverage and popularity: Just's Major [28], and Coles et al.'s PIT [10], both for Java. Major uses a set of syntax-based, compiler-assisted mutation operators, and embeds the resulting mutations into a single meta-mutant. It implements test case prioritization based on a monitored reference run, but is unable to evaluate test cases in parallel. This in many ways mirrors some of the fundamental characteristics of our approach, however, we improve on Major using an extended set of mutation operators, along with a highly parallel test scheduler which is able to parallelize execution of test cases pertaining to multiple mutants, resulting in the simultaneous evaluation of multiple mutants. Meanwhile, PIT primarily uses bytecode mutations to avoid compilation overhead. This has many of the same drawbacks as previously discussed with LLVM bytecode mutation tools, primarily the ability to generate "junk" mutations unrepresentable by a developer in source code. In addition, PIT also trades mutation operator expressivity with its bytecode approach compared to Major, as it cannot analyze higher-level language constructs.

7.3 Reducing the Cost of Mutation Analysis through Test Case Prioritization

There has also been a lot of attention towards alternative techniques for reducing the cost of mutation analysis. These techniques are given excellent coverage in the survey by Pizzoleto et al. [45], including works that, as with this work, have sought to reduce the cost of *evaluating* mutations, that correspondingly fall into Offutt and Untch's "do faster", and "do smarter" [41] categories. Among these existing cost reduction techniques, we must highlight the differences between approach, and related work in test case prioritization and process-based parallelism approaches.

In order to prioritize test cases during mutant evaluation, Zhang et al. [62] used a family of techniques to prioritize, and reduce the number of tests cases needed to determine the set of killed, and surviving mutants. The did so by applying heuristics relating to code coverage, and execution history. Unlike our approach, which prioritizes fast test cases based on their original, unmutated execution times, they prioritized test cases that executed more statements before reaching the active mutation. Mateo and Usaola [39] applied statement coverage analysis to determine which tests reach mutations, so as to remove other tests from consideration. This builds on a similar approach taken by Schuler and Zeller [47] when developing the Javalanche mutation testing tool for Java. Just et al. [29] took these ideas beyond mutation reachability to derive information about state infection. Compared to the analysis performed by these techniques, our approach only relies on the possibility of functions being reachable by individual tests, which can be determined statically, upfront, based on the construction of a call graph, without the need to perform any expensive, instrumented runtime analysis.

7.4 Reducing the Cost of Mutation Analysis through Parallelism

The fundamental idea for modern process-based parallel mutation evaluation techniques was first proposed by King and Offutt [32], as the "split-stream" execution method in their interpreter-based mutation analysis tool. They took advantage of the unique property of program mutations; that mutant programs are identical in execution to the original program up until the point of mutation. By tracking the state of the program at these diverging points, they suggested that duplicate work could be reduced. However, they noted that the overhead of keeping track of the large numbers of program states necessary for this approach was not feasible in most cases. More recently, this approach has been tested through various process-based approaches. Tokumoto et al. [57] implemented split-stream execution of mutants for C a bytecode interpreter to instrument program execution. However, their approach's reliance on an instrumented interpreter limited the performance of each mutant compared to an equivalent compiled program. Gopinath et al. [18] combined mutations close to each other for split-stream execution by forking the meta-mutant process at each point of divergence, and running the forked processes simultaneously. Sun et al. [50] performed a larger-scale study of a similar approach, grouping mutations in the same block together, and forking the program execution for each individual mutant, again, running the forked processes simultaneously. More recently, Vercammen et al. [61] implemented split-stream execution of mutants for C and C++ programs using the Clang and LLVM compiler infrastructure, instrumenting the mutant program and forking it for each individual mutant during execution. These approaches seek to optimize the common code paths between the generated mutants, reducing repeated work across mutant evaluations. However, there are various problems, and costs associated with split-stream, and process-based parallelization of mutant execution. Methods that fork the running program at points of divergence are limited by the amount of shared code up to the point of the first execution of a mutation. Keeping track of the large number of program states possible in most programs also comes with a significant memory overhead, even when implemented though forking. Forked processes also generally execute slower than regular processes, or threads in a process, due to their copy-on-write memory semantics, and context switching overhead respectively. Our approach addresses these shortcomings by having a fundamentally different approach to parallelism. Rather than processes, our approach uses lightweight threads to schedule test cases, and to increase the amount of parallelizable work, schedules test cases for multiple non-conflicting mutations at the same time.

7.5 Mathematical Theory behind Mutation Batching

Bampis et al.'s [4] theoretical work into bounded max-coloring of graphs, a theoretically more computationally feasible approach to graph coloring (and clique cover) in cases where the maximum clique size can be bounded, may be applicable to mutation batching. They wrote that bounded and unbounded max-coloring of graphs "are also equivalent with scheduling problems where jobs correspond to the vertices of a graph, which describes incompatibilities between them". Unfortunately, hitherto, no exact algorithm, and implementation has been made available for the problem of bounded max-coloring of graphs to the authors' knowledge.

8 Conclusions and Future Work

Mutation analysis is a valuable testing technique that is often regarded as computationally expensive, preventing its use on large programs and test suites. While solutions to better utilize parallel computation to evaluate mutations in less time overall have been proposed and evaluated in existing research, they all make use of individual, expensive processes, with significant overhead.

In this paper, and its ICST predecessor [36], we propose a novel technique for speeding up mutation analysis through better utilization of parallel computers without the overhead cost of processes — mutation batching. We implement our technique in a tool called mutest-rs, a novel, robust mutation analysis tool for the safety-focused Rust programming language, alongside a set of mutation operators with Rust programs in mind. To accommodate mutation analysis to systems-level Rust programs, we introduce the notion of mutation safety, which allows mutest-rs to distinguish between mutations which may have the ability to introduce undefined behavior into the program, and those which do not.

Mutation batching is a novel technique that increases the utilization of parallel processors for test evaluation during mutation analysis by applying multiple, compatible mutations at the same time. This increases the number of test cases that can be parallelized at any one point in the mutation evaluation process, leading to better use of available resources, and a shorter overall mutation analysis runtime. To ensure that mutations do not influence each other's behavior, only mutations that are in distinct parts of the program are considered compatible with regards to mutation batching. This is determined ahead of time, based on a fully-resolved call graph used to prove which test cases the mutations are reachable from. This property ensures that we can determine individual mutation detection through the test case that signaled the change in behavior. To complete mutation batching, mutations are partitioned into groups of intercompatible mutations, forming mutation batches. For this, we use greedy, heuristics-based algorithms in mutest-rs to approximate optimal mutation batching.

We use mutest-rs on 22 subject Rust programs of various sizes, testing methodologies, and test suite sizes to evaluate mutation batching, and our mutation analysis approach. To ensure that our additional mutation operators are comparable to existing, traditional mutation operators used throughout mutation analysis research, we compare them side-by-side, and find that our additional mutations are detected at a similar or lower rate than traditional mutations. We find that the prevalence of unsafe mutations across our subjects is low, but significant in the cases we encounter it. We use versions of our greedy mutation batching algorithm to determine the effect of mutation batching as a whole, as well as the effect of various variations of the greedy algorithm on mutation analysis runtimes. Our experiments show that mutation batching translates into a significant decrease in the runtime of mutation testing evaluation, with an improvement in mean mutation analysis runtimes in 18 out of 22 subjects, with improvements of up to 52.3% in the case of `rustls/rustls`, and 73.8 seconds in the case of `gleam/gleam-core`. We also conclude that mutation batching does not affect the mutation score of deterministic test suites, but may cause deviations in the case of flaky tests, with a median deviation of 1.6%.

In addition, we discuss the thread utilization characteristics of test case evaluation with batched mutation analysis, and find that mutation batching significantly reduces the amount of test case “waterfalls”, in which the test threads are not utilized in parallel as a result of evaluating small mutations with few test cases. We also explore and discuss the properties of the graphs of mutation compatibilities resulting from mutation batching to explore the feasibility of using search-based optimization algorithms for mutation batching.

As part of our future work, we plan to explore the use of search-based optimization algorithms to find improved approximations for mutation batching with a focus on reducing runtimes, and additional techniques for reducing the unused parallel capacity of modern computers further and reducing “waterfalls” throughout test case execution. While the reachability metrics derived from static call graphs are sufficient for the majority of programs, we intend to investigate the use of additional runtime coverage information, and how it might improve mutation batching. Since mutation batching is language-agnostic and may have wider applicability beyond Rust, we encourage future work to implement the technique for use in other programming languages.

With regard to the threat of potentially remaining equivalent mutants, we intend on exploring novel techniques for applying the fundamental ideas behind Trivial Compiler Equivalence to meta-mutant programs, such as those generated by mutest-rs. Such a tailored Trivial Compiler Equivalence approach could be used to further reduce the number of potentially remaining equivalent mutants through checking for equivalence in the already compiled program code, while keeping the efficiency characteristics of meta-mutants. By inspecting only the branching parts of the compiled meta-mutant specifically, we can avoid inspecting separate compilations of individual mutants.

With regard to unsafe mutations, and the specific challenges of applying mutation analysis to Rust programs, we intend on investigating other potential side effects of unsafe mutations in more detail, and providing a more detailed, dedicated analysis of our separation of safe and unsafe mutations. One possible route for more thoroughly evaluating the safety properties of our distinction between safe and unsafe mutations would be to evaluate unsafe mutations using an undefined behavior detection tool, such as Rust’s Miri [55], evaluating mutest-rs meta-mutants of existing verified Rust programs. This analysis could be used to more thoroughly reveal the lack undefined behavior during the tested executions.

Acknowledgements

Zalán Lévai is funded by UKRI EPSRC Grant No. EP/W524360/1.

Phil McMinn is supported, in part, by the UKRI EPSRC Grant "Test FLARE" (EP/X024539/1).

References

[1] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1979. *Mutation Analysis.* Technical Report. Defense Technical Information Center, Fort Belvoir, VA, USA. doi:10.21236/ADA076575

[2] Abhishek Arya, Caleb Brown, Rob Pike, and The Open Source Security Foundation. 2023. *Open Source Project Criticality Score.* The Open Source Security Foundation. https://github.com/ossf/criticality_score

[3] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 136:1–136:27. doi:10.1145/3428204

[4] E. Bampis, A. Kononov, G. Lucarelli, and I. Milis. 2014. Bounded Max-Colorings of Graphs. *Journal of Discrete Algorithms* 26 (May 2014), 56–68. doi:10.1016/j.jda.2013.11.003

[5] Maik Betka and Stefan Wagner. 2021. Extreme Mutation Testing in Practice: An Industrial Case Study. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST).* IEEE, Madrid, Spain, 113–116. doi:10.1109/AST52587.2021.00021

[6] Maik Betka and Stefan Wagner. 2022. Towards Practical Application of Mutation Testing in Industry — Traditional versus Extreme Mutation Testing. *Journal of Software: Evolution and Process* 34, 11 (2022), e2450. doi:10.1002/smr.2450

[7] Andre Bogus. 2022. mutagen. https://github.com/llogiq/mutagen

[8] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM* 16, 9 (Sept. 1973), 575–577. doi:10.1145/362342.362367

[9] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: A Mutant Generation Tool for LLVM. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* Association for Computing Machinery, New York, NY, USA, 1080–1084. doi:10.1145/3338906.3341180

[10] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016).* Association for Computing Machinery, New York, NY, USA, 449–452. doi:10.1145/2931037.2948707

[11] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. doi:10.1109/C-M.1978.218136

[12] Alex Denisov and Stanislav Pankevich. 2018. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, Västerås, Sweden, 25–31. doi:10.1109/ICSTW.2018.00024

[13] Hyunsook Do and G. Rothermel. 2006. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Transactions on Software Engineering* 32, 9 (Sept. 2006), 733–752. doi:10.1109/TSE.2006.92

[14] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* Association for Computing Machinery, New York, NY, USA, 830–840. doi:10.1145/3338906.3338945

[15] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20).* Association for Computing Machinery, New York, NY, USA, 246–257. doi:10.1145/3377811.3380413

[16] Gordon Fraser and Andreas Zeller. 2010. Mutation-Driven Generation of Unit Tests and Oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10).* Association for Computing Machinery, New York, NY, USA, 147–158. doi:10.1145/1831708.1831728

[17] Michael Garey and David Johnson. 1979. *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, NY, USA.

[18] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2016. Topsy-Turvy: A Smarter and Faster Parallelization of Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16).* Association for Computing Machinery, New York, NY, USA, 740–743. doi:10.1145/2889160.2892655

[19] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. 2009. Data Reduction and Exact Algorithms for Clique Cover. *ACM Journal of Experimental Algorithmics* 13 (Feb. 2009), 2:2.2–2:2.15. doi:10.1145/1412228.1412236

[20] Hirohide Haga and Akihisa Suehiro. 2012. Automatic Test Case Generation Based on Genetic Algorithm and Mutation Analysis. In *2012 IEEE International Conference on Control System, Computing and Engineering.* IEEE, Penang, Malaysia, 119–123. doi:10.1109/ICCSCE.2012.6487127

[21] R.G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering* SE-3, 4 (July 1977), 279–290. doi:10.1109/TSE.1977.231145

[22] Farah Hariri and August Shi. 2018. SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18).* Association for Computing Machinery, New York, NY, USA, 860–863. doi:10.1145/3238147.3240482

[23] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. 2019. Comparing Mutation Testing at the Levels of Source Code and Compiler Intermediate Representation. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, China, 114–124. doi:10.1109/ICST.2019.00021

[24] Falk Hüffner. 2021. Falk-Hueffner/Clique-Cover. https://github.com/falk-hueffner/clique-cover

[25] Yue Jia and Mark Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, Beijing, China, 249–258. doi:10.1109/SCAM.2008.36

[26] Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Information and Software Technology* 51, 10 (Oct. 2009), 1379–1393. doi:10.1016/j.infsof.2009.04.016

[27] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678. doi:10.1109/TSE.2010.62

[28] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 433–436. doi:10.1145/2610384.2628053

[29] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 315–326. doi:10.1145/2610384.2610388

[30] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2011. Using Conditional Mutation to Increase the Efficiency of Mutation Analysis. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. Association for Computing Machinery, New York, NY, USA, 50–56. doi:10.1145/1982595.1982606

[31] Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations, Held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and Sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger (Eds.). Springer US, Boston, MA, 85–103. doi:10.1007/978-1-4684-2001-2_9

[32] K. N. King and A. Jefferson Offutt. 1991. A Fortran Language System for Mutation-based Software Testing. *Software: Practice and Experience* 21, 7 (1991), 685–718. doi:10.1002/spe.4380210704

[33] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutant Subsumption Graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, Cleveland, OH, USA, 176–185. doi:10.1109/ICSTW.2014.20

[34] Zalán Lévai. 2025. mutest-rs. https://github.com/zalanlevai/mutest-rs

[35] Zalán Lévai. 2025. Replication package. https://github.com/rust-mutation-testing/publications#mutation-batching-article-rep-pak

[36] Zalán Lévai and Phil McMinn. 2023. Batching Non-Conflicting Mutations for Efficient, Safe, Parallel Mutation Analysis in Rust. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, New York, NY, USA, 49–59. doi:10.1109/ICST57152.2023.00014

[37] Richard J. Lipton. 1971. *Fault Diagnosis of Computer Programs*. Student. Carnegie Mellon Univ., Tech. Rep.

[38] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-Based Test-Case Prioritization in Software Evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Gaithersbury, MD, USA, 46–57. doi:10.1109/ISSRE.2015.7381798

[39] Pedro Reales Mateo and Macario Polo Usaola. 2015. Reducing Mutation Costs through Uncovered Mutants. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 464–489. doi:10.1002/stvr.1534

[40] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*. IEEE, Cleveland, OH, USA, 153–162. doi:10.1109/ICST.2014.28

[41] A. Jefferson Offutt and Roland H. Untch. 2001. Mutation 2000: Uniting the Orthogonal. In *Mutation Testing for the New Century*, W. Eric Wong (Ed.). Springer US, Boston, MA, USA, 34–44. doi:10.1007/978-1-4757-5939-6_7

[42] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, Florence, Italy, 936–946. doi:10.1109/ICSE.2015.103

[43] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-Based Fault Localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628. doi:10.1002/stvr.1509

[44] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Transactions on Software Engineering and Methodology* 31, 1 (Oct. 2021), 17:1–17:74. doi:10.1145/3476105

[45] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing. *Journal of Systems and Software* 157 (Nov. 2019), 110388. doi:10.1016/j.jss.2019.07.100

[46] Martin Pool. 2025. cargo-mutants. https://github.com/sourcefrog/cargo-mutants

[47] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 297–298. doi:10.1145/1595696.1595750

[48] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. doi:10.1145/3597926.3604919

[49] Donghwan Shin, Shin Yoo, Mike Papadakis, and Doo-Hwan Bae. 2019. Empirical Evaluation of Mutation-Based Test Case Prioritization Techniques. *Software Testing, Verification and Reliability* 29, 1-2 (2019), e1695. doi:10.1002/stvr.1695

[50] Chang-ai Sun, Jingting Jia, Huai Liu, and Xiangyu Zhang. 2018. A Lightweight Program Dependence Based Approach to Concurrent Mutation Analysis. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. IEEE, Tokyo, Japan, 116–125. doi:10.1109/COMPSAC.2018.00023

[51] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 899–913. doi:10.1145/3453483.3454084

[52] The Open Source Security Foundation. 2022. Criticality Score. https://commondatastorage.googleapis.com/ossf-criticality-score/index.html

[53] The Rust Project Developers. 2024. Rust: Production users. https://www.rust-lang.org/production/users

[54] The Rust Project Developers. 2025. Guide to rustc Development: Monomorphization. https://rustc-dev-guide.rust-lang.org/backend/monomorph.html

[55] The Rust Project Developers. 2025. Miri. https://github.com/rust-lang/miri

[56] The Rust Project Developers. 2025. The Rustonomicon: How Safe and Unsafe Interact. https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html

[57] Susumu Tokumoto, Hiroaki Yoshida, Kazunori Sakamoto, and Shinichi Honiden. 2016. MuVM: Higher Order Mutation Analysis Virtual Machine for C. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Chicago, IL, USA, 320–329. doi:10.1109/ICST.2016.18

[58] Roland H. Untch. 1992. Mutation-Based Software Testing Using Program Schemata. In *Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE 30)*. Association for Computing Machinery, New York, NY, USA, 285–291. doi:10.1145/503720.503749

[59] Roland H. Untch. 1995. *Schema-Based Mutation Analysis: A New Test Data Adequacy Assessment Method.* Ph. D. Dissertation. Clemson University, Clemson, SC, USA. https://www.proquest.com/docview/304173339/abstract/F975B52F62924781PQ/1

[60] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation Analysis Using Mutant Schemata. In *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '93)*. Association for Computing Machinery, New York, NY, USA, 139–148. doi:10.1145/154183.154265

[61] Sten Vercammen, Serge Demeyer, Markus Borg, Niklas Pettersson, and Görel Hedin. 2024. Mutation Testing Optimisations Using the Clang Front-End. *Software Testing, Verification and Reliability* 34, 1 (2024), e1865. doi:10.1002/stvr.1865

[62] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster Mutation Testing Inspired by Test Prioritization and Reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 235–245. doi:10.1145/2483760.2483782