Batching Non-Conflicting Mutations for Efficient, Safe, Parallel Mutation Analysis in Rust

Zalán Lévai
University of Sheffield

Phil McMinn
University of Sheffield

***Abstract*—Rust is a relatively young, memory safe systems programming language which is increasingly being adopted by projects requiring both performance, and safety. While automated testing is built into the language, tool support for mutation analysis is almost non-existent, having not been the subject of past research. This leaves Rust developers without a way to determine test thoroughness. To address this problem, we design a mutation analysis process for Rust that overcomes challenges related to generating viable mutations due to the strictness of the language in terms of its type system, memory restrictions, and the potential to introduce undefined behavior in `unsafe` code blocks. Our technique efficiently evaluates mutations simultaneously through a process we refer to as "batching" — the use of static analysis to determine mutations that are non-conflicting, and therefore are able to be evaluated together. Batching enables our technique to maximize thread usage, executing more tests in parallel, and further reducing the time required to evaluate mutations. We implemented these techniques into a tool, mutest-rs, which we empirically evaluated on a diverse set of common subject libraries and Rust programs, and found that our batching method for increasing parallelism is able to reduce the overall runtime of mutation analysis by up to 66.4%, compared to not applying batching.**

I. Introduction

The lack of mature mutation analysis techniques for Rust represents a serious obstacle in ensuring thorough testing of programs written in this safety-focused programming language [1]. Being a compiled, low-level language, Rust faces a multitude of issues when it comes to applying mutation testing. The language's strictness — both in terms of type safety, and memory restrictions — means that mutation operators have to be constrained using tailored static analysis. Furthermore, the presence of *unsafe* code means that special care has to be taken to avoid the introduction of undefined behavior through mutations. For practical applicability, mutation analysis approaches also have to be efficient in regard to their runtime, scaling well to large numbers of tests, and mutations.

This paper is the first to consider mutation analysis for the Rust language. We devise an efficient mutation analysis pipeline for Rust, define new operators novel to the language, as well as a novel mutation evaluation technique that we refer to as "batching", designed to increase the efficiency of mutation analysis overall, and reduce execution costs.

Batching is the process of grouping individual, *non-conflicting* mutations together into the same mutant, with the purpose of evaluating constituent mutations, and their tests, simultaneously. Rather than using processes, batching allows for efficient in-process parallelism of evaluation, making it possible to use schedulers with significantly less overhead, and more control over the entire evaluation. 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 distinct parts of a program. Our technique determines mutations that are non-conflicting through a conservative static analysis of a program's functions that are potentially reached by individual tests. If two mutations m_1 and m_2 appear in two different functions exclusively reached by two different sets of tests — $\{t_1, t_2\}$ and $\{t_3, t_4\}$, respectively — then m_1 and m_2 are candidates for evaluation in the same batch.

Our mutation analysis pipeline executes the tests relevant to each individual batch in parallel. Since batches of mutations require the execution of larger relevant portions of the test suite, rather than only a few selected tests relevant to a particular mutation, our technique maximizes test evaluation concurrency. These benefits can be illustrated as follows. Supposing two threads, a non-batched, yet non-naïve parallel mutation evaluation process might execute partial test suites to evaluate each mutant; i.e. t_1 and t_2 for m_1, and t_3 and t_4 for m_2, and could utilize each thread to execute each pair of tests in parallel. However, it would need to synchronize at the end of the first pair of tests (to change mutants), and then restart the threads with the second pair. In this example, however, a batched process would be able to continue to evaluate the second pair of tests without the need to synchronize and switch mutant, thereby producing a time saving that increases with larger numbers of tests, and mutants.

Finally, our technique creates a single meta-mutant program [2], which statically embeds all mutations, and builds on the idea of conditional mutation [3], dynamically enabling sets of mutations while evaluating individual batches.

We developed a tool, called mutest-rs [**?**], which implements our technique. We used 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 show that mutest-rs is applicable to a range of Rust subjects, reliably generates mutants, while also demonstrating that batching is effective for reducing the mutation analysis runtimes — with savings of up to 66.4% possible compared to not using batching for the crates we studied.

The contributions of this paper, therefore, are as follows:

1) A set of mutation operators suitable for Rust programs, including adaptations of existing operators, and novel operators specifically designed for the language (Section III-B).

2) An algorithm for batching mutations for simultaneous, parallel, and efficient mutation evaluation (Section III-D).
3) A definition of *mutation safety*, applicable to programming languages with distinct *safe* and *unsafe* scopes, allowing for the mutation of system programs without the fear of introducing undefined behavior (Section III-E).
4) The results of an empirical evaluation (Section IV) of the reduction in testing time possible with mutation batching in practice, revealing a reduction in the overall mutation analysis runtime between 12.9% and 66.4% for a diverse set of commonly-used Rust subject programs (Section V).

Our tool, and empirical data are available online (see [**?**], [4]).

II. Background — Rust and Mutation Analysis

Rust is an emerging programming language that aims to bring proven memory safety to low-level systems programming. Since its 1.0 release in 2015, it has become the language of choice for performant, safety-focused systems in many parts of the industry [5]. 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 [6] — has, to date, not received research dedicated to the language.

Rust is a safety-focused programming language with static analysis, and testing at its core. It has built-in testing facilities, with functions marked with the `test` attribute evaluated by the included test harness. Each such unit of code, known as a "crate", can be tested individually through its unit and integration tests. Rust has a thriving ecosystem of library crates, which become common dependencies of most projects.

Despite this, hitherto, a full-featured mutation analysis technique does not exist for Rust, although two relatively limited solutions exist — mutagen [7], and cargo-mutants [8]. While mutagen is capable of applying simple AST transformations to Rust programs, cargo-mutants only implements extreme mutation [9] — a simplified form of mutation that involves replacing entire function bodies. Both of these tools only operate on a program's syntax. They are not concerned with semantics, resulting in the generation of large numbers of invalid mutants, low coverage of various idiomatic mutations, general inefficiencies in both mutation generation and evaluation, and practical difficulties related to their application.

Meanwhile, the research of Denisov et al. [10] into the mutation analysis of LLVM bytecode, the instruction language Rust compiles to, mentions its potential applicability to Rust. However, a number of problems arise from bytecode-based mutation analysis; mutations are easily introduced in external library code not relevant to the program, and many bytecode mutations do not have source code counterparts. The results seen with generic LLVM bytecode mutation [10], [11], [12] reinforced the need for research based on higher-level analysis.

Finally, Rust statically proves the memory safety of programs, although this requires developers to abide by its *safety* rules. When more flexibility is required, dedicated *unsafe* code sections may be introduced which have to be manually audited to ensure that they uphold safety contracts; i.e., the guarantee that the program will not contain undefined behavior. The interactions between safe and unsafe code have to be considered for mutation analysis to retain these guarantees.

III. A Mutation Analysis Pipeline for Rust, and its Implementation into the Mutest-RS Tool

This section details our mutation analysis pipeline for Rust, which we implemented into a tool named mutest-rs. Our mutest-rs tool is tightly integrated with rustc, the reference implementation of the Rust compiler.

A. Overview

At a high-level, our approach performs two passes on the input Rust source code to perform mutation analysis on a given Rust crate. In the first pass, the original source code is analyzed by an augmented version of the compiler. First, the tool generates a call graph (Section III-C) starting from the crate's test functions. This ensures that the tool only mutates functions that are reachable by the test suite, and allows the tool to build a mapping between mutations, and the tests that may reach them. Second, the tool applies a set of mutation operators (Section III-B) to every possible location in the body of the previously selected functions. Third, the generated mutations are batched (Section III-D). Finally, the conditional code substitutions which make up the mutations are inserted into the original program, alongside the mutation and mutant metadata, an injected global variable `ACTIVE_MUTANT_HANDLE` representing the currently active mutant, and an entry point with a call to the generic mutation test harness, which drives the evaluation of the mutation test. The output of the first pass is the generated source code.

In the second pass, the generated source code of the mutation test program is compiled into a binary. This custom test binary is then executed to perform the evaluation.

B. 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 ensures that we can generate both first-, and higher-order mutations [13] 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 1 shows an example of such a conditional expression. First, substitutions of the various mutations are grouped by the location in the original source code they apply to. Then, for

```
        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. 1. 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 IV-A), illustrating the mutations applied to a memory offset calculation.

each substituted location, the 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. Some of these mutation operators are adaptations of common operators used in other languages (e.g., by Major [14], and PIT [15] for Java) that are applicable to Rust. Others are novel operators that are made possible by a combination of language features provided by Rust, and extensive static analysis. In general, these mutation operators rely heavily on inferred type information. Table I lists these operators, with operators novel to this paper as follows:

ArgDefaultShadow replaces any argument passed to a function parameter with the default value of the type. By inserting a variable binding statement at the beginning of the function, it 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.

CallDelete deletes function calls, replacing the call with the default value of the type. **CallValueDefaultShadow** retains function calls, but replaces the return value at the call site with the default value of the type. The difference between these mutation operators is whether the side effects of the called function are retained with the mutation. These two mutation operators only apply 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. 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.

ContinueBreakSwap replaces `continue` expressions associated with loops with `break` expressions, and vice versa. When considering the added complexity of labels and optional loop return values, the implementation of this rule quickly requires extensive scope analysis.

TABLE I
MUTATION OPERATORS FOR RUST IMPLEMENTED IN MUTEST-RS.
Mutation operators in **bold** are novel and introduced in this paper.

Mutation Operator	Description
ArgDefaultShadow	Replace function argument with default value[1]
BitOpOrAndSwap	Replace bitwise OR with bitwise AND, and vv.
BitOpOrXorSwap	Replace bitwise OR with bitwise XOR, and vv.
BitOpShiftDirSwap	Replace bitwise LSH with bitwise RSH, and vv.
BitOpXorAndSwap	Replace bitwise XOR with bitwise AND, and vv.
BoolExprNegate	Negate boolean expression
CallDelete	Delete function call, replace with default value[1]
CallValueDefaultShadow	Replace function call result with default value[1]
ContinueBreakSwap	Replace `continue` with `break`, and vv.
EqOpInvert	Invert equality operator
LogicalOpAndOrSwap	Replace logical `&&` with logical `||`, and vv.
OpAddMulSwap	Replace addition with multiplication, and vv.
OpAddSubSwap	Replace addition with subtraction, and vv.
OpDivRemSwap	Replace division with modulo, and vv.
OpMulDivSwap	Replace multiplication with division, and vv.
RangeLimitSwap	Change inclusivity of range's upper bound
RelationalOpEqSwap	Change relation operator's bound w.r.t. equality
RelationalOpInvert	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()`.

Finally, **RangeLimitSwap** changes whether the range is inclusive of its upper bound. While not a unique language feature, ranges, along with iterators, make up the majority of loop and array interactions, with manual indexing discouraged.

In addition to the novel operators, many of the common operators have to be adapted to Rust, notably to its trait system. 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.

While in mutest-rs, we did not implement any filtering for equivalent mutants, the mutation operators used are designed to produce only a relatively small amount of collisions. 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.

C. Fully-Resolving Call Graphs as a Pre-Step to Batching

To analyze the functions reachable by individual tests, as a pre-step for batching mutations, our technique builds a call graph. The nodes of the graph represent unique functions defined in the program. The presence of a directed edge between two functions indicates that the function represented by the source node contains a direct call to the function represented by the destination node. We extend the construction of call graphs with propagated type substitutions, which we refer to 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, $C \subset F \times F$.

Require: T
Ensure: L
  $C_1, C_2, \ldots \leftarrow \{\}, \{\}, \ldots$
  for all $t \in T$ **do**
    $C_t \leftarrow$ functions called in body of t
    for all $(f, S) \in C_t$ **do**
      $(f', S') \leftarrow$ resolve call f with types S
      $C_1^{(f',S')} \leftarrow C_1^{(f',S')} \cup \{t\}$
    end for
  end for
  $L \leftarrow \{\}$
  for all $d \in \langle 1, 2, \ldots \rangle$ **do**
    $F_d \leftarrow \{f \mid \forall((f, \_), \_) \in C_d\}$
    $F_\Sigma \leftarrow \{f \mid \forall(f, \_) \in L\}$
    break if $F_d \subseteq F_\Sigma$ ▷ all functions have already been visited
    for all $((f^0, S^0), T') \in C_d$ **do**
      $L_{f^0} \leftarrow L_{f^0} \cup \{(t, d) \mid \forall t \in T'\}$
      $C_{f^0} \leftarrow$ functions called in body of f^0
      for all $(f, S) \in C_{f^0}$ **do**
        $S^+ \leftarrow$ fold type substitutions S^0 into S
        $(f', S') \leftarrow$ resolve call f with types S^+
        $C_{d+1}^{(f',S')} \leftarrow C_{d+1}^{(f',S')} \cup T'$
      end for
    end for
  end for

Fig. 2. Construction of the walks of a fully-resolved call graph.

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

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

Fig. 3. A generic function call with a single level of indirection, which cannot be resolved without propagating type substitutions in the call graph.

The root functions may be any fully-resolved, non-generic functions. These include entry points, like a binary crate's `main` function or test functions. The algorithm in Figure 2 shows the process of building walks of the call graph between test functions T and functions of the program. The output of the algorithm, L, 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 [16].

Figure 3 is an example of a function call which cannot be resolved using the local context of the function definition alone. 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. Our technique instead looks at the invocation `f1::<S1>` (and other invocations of `f1` with differing type arguments). By combining the invocation's types `<T = S1>` with the call's types `<T = ?, V = u8>`, it becomes possible to resolve the same function call to `S1::do_t1::<u8>`.

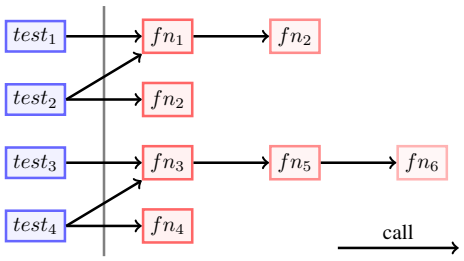

Fig. 4. Reachability-exclusivity, the precondition for mutation batching.

For dynamic polymorphism, the fully-resolved call graph branches off into all possible implementations of the function being called to cover any possible runtime function call. For example, when calling a trait function on a set of trait objects, the dynamic call is represented as a call edge to all type's implementation of the function which implements the trait.

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, namely dynamic invocation through function pointers, which cannot be covered with such a lightweight approach, and requires runtime checks. These are not inherent limitations of mutation batching, and other approaches are left as an item for future work.

D. 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, test authors commonly make their test suites parallel, since that is the built-in test runner's default, presenting an opportunity to further engineer safe ways of parallelizing the evaluation of multiple mutations, making better use of available resources.

Simultaneously enabling multiple mutations requires that changes in behavior remain uniquely identifiable through test results. This invariant can be upheld by ensuring that no two mutations are reachable from any of the same test functions, effectively producing disjunct subprograms within the original program (see Figure 4). We introduce the notion of reachability-exclusivity for such a mutation collision scheme:

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 $T_1 \cap T_2 = \emptyset$.

Our mutation operators cannot add to the set of functions called by other functions in a program, and so its fully-resolved call graph is sound for determining reachability-exclusivity.

Require: M
Ensure: B
  function SORTINGHEURISTIC(m)
    return $-|\{m' \in M \mid m$ and m' are not reachability-exclusive$\}|$
  end function
  function COMPATIBLEBATCH(B, m)
    for all $M' \in B$ **do**
      if $\forall m' \in M' : m$ and m' are reachability-exclusive **then**
        return M'
      end if
    end for
    return None
  end function
  $B \leftarrow \{\}$
  sort M by the element-wise value of SORTINGHEURISTIC
  for all $m \in M$ **do**
    if $M' \leftarrow$ COMPATIBLEBATCH(B, m) **then**
      $M' \leftarrow M' \cup \{m\}$
    else
      $B \leftarrow B \cup \{\{m\}\}$
    end if
  end for

Fig. 5. Static batching of non-conflicting mutations. A greedy, heuristics-based approximation of the optimal batching.

The implementation of our approach generates such non-conflicting sets of mutations upfront, at compile time. The resulting static mutation batches are each evaluated by the test harness. We also encode the necessary metadata to discern the test results corresponding to each mutation at compile time.

The algorithm in Figure 5 shows the process of creating non-conflicting batches B of mutations M. The optimal batching problem — which can be modeled as a graph problem with nodes representing mutations, and edges representing conflicts — is NP-hard. Our algorithm is a greedy, heuristics-based approximation. It works by first sorting mutations into a list, and then working through that list, adding non-conflicting mutations to the batch. This list is pre-sorted according to a heuristic. Through anecdotal initial experimentation, we found the largest batches were possible by sorting the list by the number of conflicts each individual mutation had. We leave the further investigation of this, as well as potentially more optimal algorithms for batching as an item for future work. (In our empirical experiments (Section IV), we apply limits to the number of mutations in a batch to evaluate effectiveness.)

It is important to note that only *safe mutations* — mutations defined in *safe* subtrees of the program — may be safely evaluated in parallel. Mutations that are inside `unsafe` blocks or are invoked by *unsafe* code are not guaranteed to uphold the necessary guarantees. As such, *unsafe mutations* are put into their own singleton set. We discuss mutation safety next.

E. Mutation Safety — Avoiding the Spread of Unsafety

Safety in Rust is the guarantee that *Safe* Rust code cannot cause undefined behavior. *Unsafe* Rust code has no such guarantees, and *unsafe* operations are required to be annotated explicitly, through the use of an `unsafe` block. This strict separation of *safe* and *unsafe* code allows for new considerations to be made when applying mutation testing to Rust.

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

Fig. 6. 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)
};
```

Fig. 7. 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.

There is an intricate relationship between *safe* and *unsafe* Rust code. *Safe* Rust has to trust any *unsafe* Rust where *safety* invariants have been upheld, but *unsafe* Rust must not trust the correctness of *safe* Rust code without care [17]. In practice, however, embeddings of *unsafe* Rust often depend on the correctness of the enclosing *safe* code section — its context (see Figure 6) — and the correctness of the *safe* code it calls (see Figure 7) for their own correctness [18].

In *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 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 in *safe* and *unsafe* scopes respectively.

Definition 3 (Unsafe mutation): We consider mutation m *unsafe* if it fulfills one of the following conditions:

- m has a direct or indirect `unsafe` block parent in the function body it is located in.
- m is in a function body with an `unsafe` block but is not a direct or indirect child of an `unsafe` block. This rule is called *context-tainting*.
- m is in the body of a function which is called directly or indirectly from an `unsafe` block. This rule is called *call-tainting*.
- 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. This rule is called *extended call-tainting*.

Definition 4 (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 8 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

```
fn x { [-]
    fn y { [2.]
        unsafe { [1.]
            fn z { [3.] }
        }
        fn w { [4.] }
        unsafe fn u { [1.]
            fn v { [3.] }
            fn w { [3.] }
        }
    }
}
```

Fig. 8. Illustration of the *mutation safety* rules on a scope-level. Each scope is either a called function or an `unsafe` block. Mutations have the *safety* of their containing scope. Within each scope, an annotation is placed to signify applications of the *safety* rules: `[1.]` unsafety; `[2.]` context-tainting; `[3.]` call-tainting; `[4.]` extended call-tainting; `[-]` none, safety.

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 scopes become *unsafe*.

F. Parallelized Test Evaluation

Our approach generates an instrumented program, which when executed, performs 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, 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. An overall test timeout $t_{timeout}$ is also automatically determined from this test run based on the longest running test's duration t_{max}, as follows: $t_{timeout} = t_{max} + max(0.1 \cdot t_{max}, 1\text{s})$.

After the profiling test run, our technique applies each mutation batch and evaluates 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 (Figure 1). Once the mutation batch is applied, the harness evaluates the tests 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 $t_{timeout}$, 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.

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.

TABLE II
SUBJECTS USED IN OUR STUDY; A COMBINATION OF THE MOST DOWNLOADED RUST LIBRARY CRATES, AND GITHUB PROJECTS.

Crate	Description	SLoC	Unit Tests
`alacritty`	Terminal emulator	15459	65
`bat`	Extended `cat` clone	6795	46
`exa`	Extended `ls` clone	10724	406
`hashbrown`	SwissTable hash map	17651	93
`parking_lot`	Synchronization primitives	5102	86
`rand`	Randomness generator	8616	75
`rand_core`	Randomness generator	1648	9
`regex`	RegEx engine	20982	7832
`regex-syntax`	RegEx parser	51406	324
`rustls`	TLS implementation	25184	174

IV. EVALUATION

We used 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, applying it to commonly-used, and critical Rust code. These have the potential to present obstacles to our approach, for example non-deterministic, flaky tests [19] that are likely to affect mutation scores, and the extent to which tests may be parallelized. Furthermore, we have made improvements to the classic mutation analysis workflow in our approach — in particular, our method of *batching* mutations — which may have wider applicability beyond Rust, and which we also evaluate. Our research questions, therefore, are as follows:

RQ1: How do common Rust projects perform with respect to mutation analysis with our mutation operators?
RQ2: What is the distribution of mutations across batches that can be achieved through reachability-exclusivity for various types and sizes of projects?
RQ3: What are the performance gains provided by reachability-exclusive mutation batching? How does the approach scale with project size and the number of mutations?
RQ4: How does mutation batching perform in the context of flaky, non-deterministic, non-parallelizable test suites?

A. Subjects

We selected Rust crates from the list of the top 100 most downloaded library crates according to crates.io (the primary crate registry), and the list of the largest Rust repositories (according to repository size, and number of stars) on GitHub. We excluded crates that were wrappers of external libraries, primarily comprised of `unsafe` code, contained at most an insignificant amount of executable code, used a custom test runner or had an insignificant amount of unit tests.

Table II lists the crates involved in our experiments. In terms of source lines of code, the subjects ranged from 1648 for `rand_core` to 51406 for `regex-syntax`. In terms of the number of unit tests, the subjects ranged from 9 for `rand_core` to 7832 for `regex`.

B. Methodology

In preparation for the experiments, we forked each subject's source code repository. This was done to pin down the crate versions we were testing against, creating a stable test environment. In addition, for crates with a significant amount of out-of-code standalone tests (programs contained in the `tests/` directory of a Cargo package), we moved the test cases back into the crate. This made it possible for mutest-rs to analyze these test cases as well, allowing for a more extensive evaluation of the crates' test suite. All modifications to the subjects are available in our replication package [4].

We wrote an experiment script to automatically perform the analysis on all selected crates. All information is parsed automatically by the script from the output of mutest-rs. We configured mutest-rs to discard all *unsafe* mutations.

The script first runs mutest-rs to gather information about the generated mutants with batching disabled, with a small batch size limit of 5, and without a batch size limit (unbatched, small batched, and fully batched cases respectively). This includes the total number of mutations generated, and the distribution of mutations across batches for all batching cases.

Then, the script runs mutest-rs to completion to perform the mutation testing. In all cases, the same test ordering, filtering, and multi-threaded scheduling was applied, regardless of the presence of batching. Five evaluations are performed for each batching case to reduce the error in our timing measurements caused by background processes, and non-determinism. For each evaluation, the tool collects the mutation score, the number of mutations that were detected and undetected, the number of mutations that timed out, the number of tests that were 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 tree of the test suite, determining which functions to mutate, and the *mutation safety* of each scope;
- **mutation operator application**, the process of applying each mutation operator to every possible location in the mutable functions, producing mutations;
- **mutation batching**;
- **code generation**, the process of applying the necessary modifications to the program's AST, and printing the resulting code;
- **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 a test timeout;
- **mutation evaluation**, the evaluation of each mutant against the test suite.

The experiments were run on a MacBook Air M1 (2020) with 16 GiB of RAM, running macOS 12.4. We built mutest-rs against the `nightly-2022-06-13` version of rustc, and ran it with a thread pool of size 8 for executing the tests.

C. Threats to Validity

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

TABLE III
MUTATION SCORES.

Crate	Score	Detected (Timed out)	Undetected	Total
`alacritty`	22.8%	329.0(10.0)	1114.0	1443
`bat`	73.7%	221.0	79.0	300
`exa`	73.3%	508.0	185.0	693
`hashbrown`	$76.5 \pm 0.7\%$	205.8 ± 1.8	63.2 ± 1.8	269
`parking_lot`	73.7%	28.0(14.0)	10.0	38
`rand`	59.6%	779.0(88.0)	529.0	1308
`rand_core`	71.4%	145.0	58.0	203
`regex`	62.3%	736.0(7.0)	446.0	1182
`regex-syntax`	65.9%	1520.0(59.0)	786.0	2306
`rustls`	$69.5 \pm 0.1\%$	$960.0 \pm 0.7(15.0)$	422.0 ± 0.7	1382

The choice of the mutation operators applied could be a threat to internal validity. Different or additional 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 choice of subjects could be 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 crates currently available. Therefore, we consider the results to be meaningful.

Defects in the compiler-integrated mutation testing tool, mutest-rs could be a threat to construct validity. We controlled the threat in mutest-rs by testing with several small example programs as well as manually analyzing the generated mutations, mutants, code, and mutation testing results. Moreover, over the course of this study, the tool has generated multiple millions of lines of valid Rust code. We conclude that the implementations of the tools used worked correctly.

Finally, we make our tool, scripts, data, and repository forks of our subjects available in our replication package [4].

V. RESULTS

***RQ1**: How do common Rust projects perform with respect to mutation analysis with our mutation operators?*

Using the results of the unbatched mutation test evaluations over all subjects, we can determine the mean mutation score to be 64.9%, with mutation scores ranging between 22.8% for `alacritty`, and 76.5% for `hashbrown`. Except for `alacritty`'s low mutation score, all other crates had a mutation score above 59%. The low mutation score of `alacritty` can be best attributed to the project having comparably shallow tests. Only `hashbrown`, and `rustls` showed slight variance in its mutation score across multiple evaluations. We investigate this further in RQ4. The tool generated mutations which timed out in half of the crates, with `rand` having the most timeouts at 88, making up 11% of all of its detected mutations. Table III shows the mean results across all 5 unbatched evaluations. Only *safe* mutations were considered for the purposes of batching, however, the number of *unsafe* mutations was not high enough for any crate to

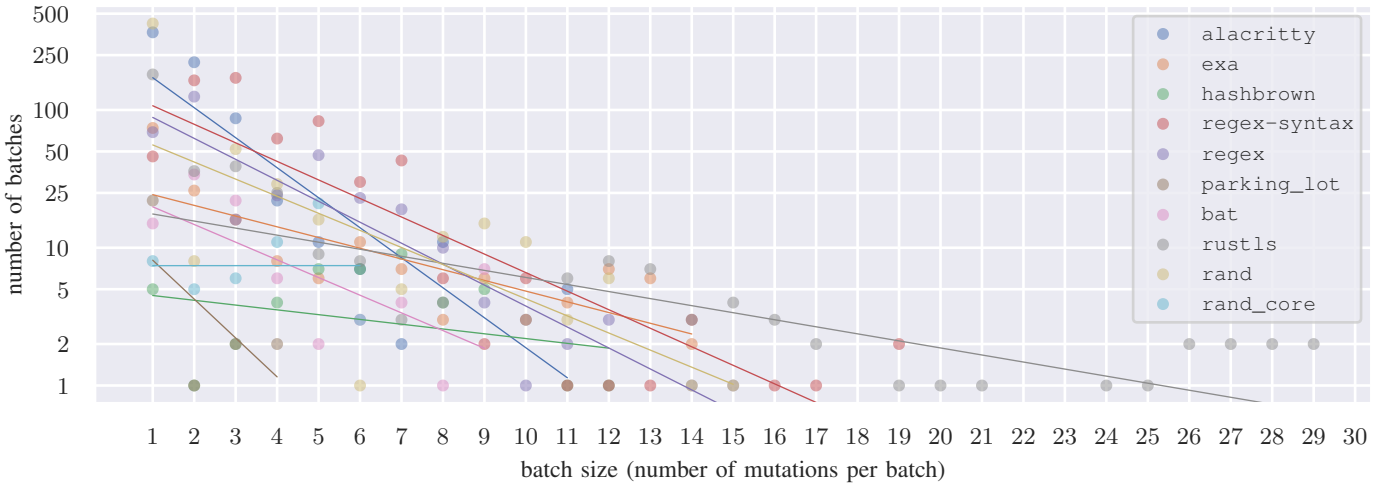

Fig. 9. Distribution of mutations across batches of various sizes on a logarithmic ($\log_{10}$) scale. Logarithmic trendlines fitted over each subject's mutation batching distribution.

have a significant effect on the overall mutation score. While the mutation operators are unlikely to produce a significant amount of collisions, it is important to note that we did not perform filtering for equivalent mutants.

In conclusion for RQ1: Unbatched mutation scores range between 22.8% and 76.5%, hinting that mutation testing may uncover a significant amount of test inadequacies.

RQ2*: What is the distribution of mutations across batches that can be achieved through reachability-exclusivity for various types and sizes of projects?*

To discuss the benefits of mutation batching, it is important to consider the problem of optimal batching, and how the optimal solutions compare to the kinds of batches our algorithm generates. Figure 9 visualizes the distribution of mutations across batches of various sizes. While the data is considerably noisy, it is easy to see that almost all crates follow a logarithmic trend line depicting a high number of unbatched mutations, and a small number of large batches. The optimal batching of these crates would approach a mirrored trend line depicting a small number of unbatched mutations, and a high number of large batches. The batching problem remains the same regardless of the conflict criteria used. The algorithm outlined in this paper is a greedy, heuristics-based approximation of the optimal solution to the batching problem. Regardless, it can still provide a significant improvement over unbatched evaluation techniques.

Table IV shows the batching ratio achieved by reachability-exclusive batching of mutations using this paper's greedy algorithm. The reduction in the number of mutation rounds — the number of mutants for unbatched-, and the number of batches for batched evaluation — ranges between 28.9% for `parking_lot`, and 82.9% for `hashbrown`, with a mean of 65.1%, suggesting a significant improvement in mutation testing runtime for most crates. The number of unbatched mutations ranges between 57.9% for `parking_lot`, and 1.9% for `hashbrown` of the total number of mutations, with a mean of 15.8%. The large number of unbatched mutations

TABLE IV
BATCHING OF MUTATIONS.

Mutations refers to the number of mutations generated for each crate. *Total Batches* refers to the total number of batches the generated mutations were placed into by batching. The figures in brackets denote the ratio of the total number of mutations, and the number of batches; i.e. the batching compression ratio. *Singleton Batches* refers to the number of batches with only a single mutation (i.e. unbatched mutations), a subset of all batches. The figures in brackets denote the ratio of the number of unbatched mutations, and the total number of mutations.

Crate	Mutations	Total Batches	Singleton Batches
`alacritty`	1443	736(49.0%)	366(25.4%)
`bat`	300	94(68.7%)	15(5.0%)
`exa`	693	179(74.2%)	74(10.7%)
`hashbrown`	269	46(82.9%)	5(1.9%)
`parking_lot`	38	27(28.9%)	22(57.9%)
`rand`	1308	584(55.4%)	424(32.4%)
`rand_core`	203	54(73.4%)	8(3.9%)
`regex`	1182	345(70.8%)	69(5.8%)
`regex-syntax`	2306	623(73.0%)	46(2.0%)
`rustls`	1382	351(74.6%)	181(13.1%)

in some cases is an indicator that more optimal batchings exist, suggesting that further performance improvements may be achievable by applying the technique of mutation batching with more optimal batching algorithms.

In conclusion for RQ2: The greedy algorithm presented in this paper is an approximation of the batching problem. The reduction in the number of mutation rounds ranges between 28.9% and 82.9%. Between 57.9% and 1.9% of mutations are unbatched depending on the crate.

RQ3*: What are the performance gains provided by reachability-exclusive mutation batching? How does the approach scale with project size and the number of mutations?*

Mutation batching is a technique for increasing the parallelism of the evaluation of mutations by allowing for mutations with a small number of tests to be included in larger batches, maximizing the number of tests that can be run at once. All timings discussed are the means of the 5 individual evaluations per each batching. We do not report additional metrics or errors, since the timing results exhibit little dispersion.

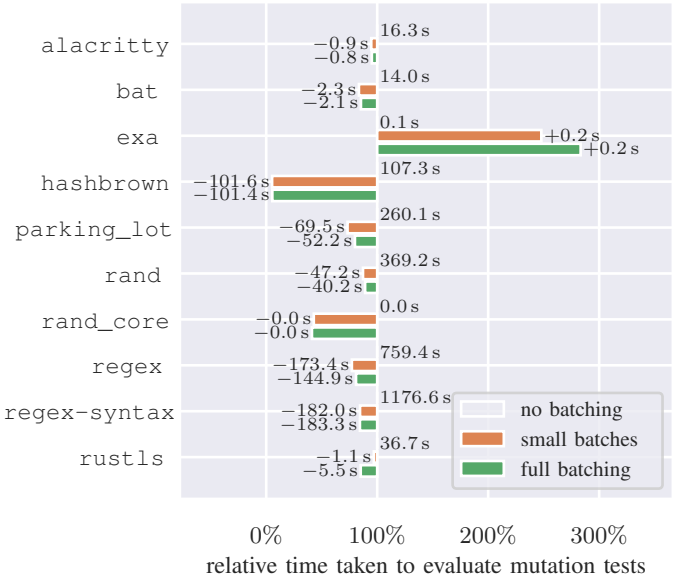

Fig. 10. Decrease in the mean runtime of mutation testing evaluation (with little deviation) with small, and full batching. Figures are scaled according to each crate's original runtime, labeled with the absolute time differences.

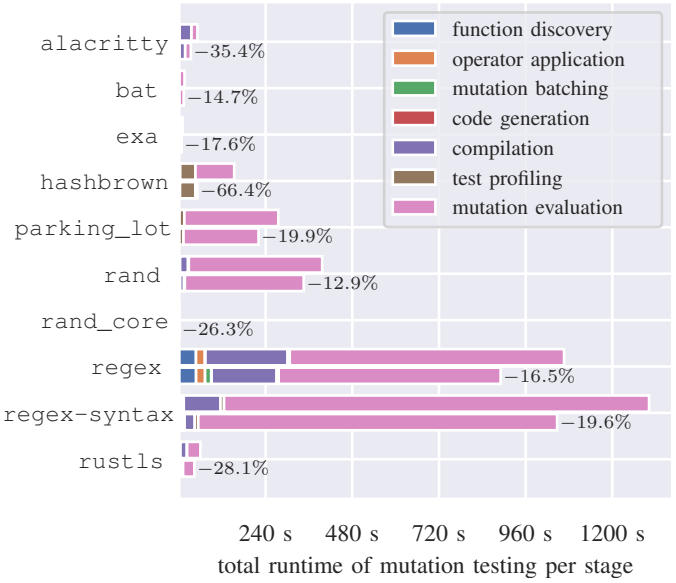

Fig. 11. Decrease in the mean total runtime of mutation testing (with little deviation), including analysis, with full batching. Bars of some stages are not visible for some cases due to their short runtime.

Figure 10 shows the differences in the time taken to evaluate mutation tests between unbatched, small batched, and fully batched mutation tests. All crates except `exa` show a considerable decrease in runtime, ranging between 4.8% for `alacritty`, and 94.5% for `hashbrown` across the tested batchings, with a mean decrease of 28.3% for small batching, and 28.1% for full batching. The outlier improvement seen with `hashbrown` can best be attributed to a combination of an optimal batching — being the most batched crate in our experiment at 17.1% with only 1.9% of its mutations unbatched — and project characteristics — having the smallest number of tests amongst the largest crates. The difference in improvement between small batches, and full batches is negligible in most cases, usually within one to two percent, with the biggest difference being `rustls`'s 11.8%. However, in almost all cases, small batching achieves a higher reduction in runtime compared to full batching. This is most likely the direct result of the greedy algorithm used, which prioritizes creating larger batches over a balanced distribution of mutations across batches. While *unsafe* mutations were discarded, due to the small number of them, we expect to have seen indistinguishable results in most cases, even if we had evaluated them one-by-one in individual processes.

Figure 11 shows the differences in runtime for the entirety of mutation testing, including analysis, with full batching. The decrease in the overall runtime of mutation testing ranges between 12.9% for `rand`, and 66.4% for `hashbrown` with batching, with a mean decrease of 25.7%. We see a drastic decrease in compilation times with batching. This happens because of the way conditional mutations are implemented in mutest-rs. Since a static substitution lookup table is embedded for each mutant into the generated code, reducing the number of them also reduces compile times. This approach is necessary for efficient mutation testing. While it is possible to generate programs which load mutants from a separate data file, such an implementation would require accessing dynamic data structures at every mutated location, drastically increasing evaluation runtime. It is also worth noting that the process of mutation batching takes an insignificant amount of time in most cases, with the median batching time being 1.3 seconds. The batching time of a crate is proportional to the number of tests, and the number of mutations generated. The largest crate in our study, `regex`, with 7832 tests and 1182 mutations, has a batching time of 17.2 seconds.

TABLE V
NUMBER OF TESTS EVALUATED DURING MUTATION TESTING.

Unbatched, and fully batched evaluations listed separately. *Total* refers to the total number of test evaluations if all mutations are undetected, and exhaustive mutation testing has to be performed.

Crate	Unbatched	Batched	Total
`alacritty`	4075.4 ± 156.4	4491.0 ± 160.1	5671
`bat`	771.6 ± 49.0	814.6 ± 47.5	1337
`exa`	6414.2 ± 1086.4	6243.2 ± 715.9	30457
`hashbrown`	495.2 ± 35.3	511.8 ± 52.8	1647
`parking_lot`	228.0	234.0	439
`rand`	1579.0	1848.0	2227
`rand_core`	216.0	242.0	253
`regex`	979154.0	625880.0	1658674
`regex-syntax`	21910.0	14118.0	45660
`rustls`	3881.8 ± 248.2	4336.0 ± 204.3	8093

It is important to point out that mutation batching does not necessarily decrease the number of test evaluations required to produce a mutation score. Table V compares the number of test evaluations performed during unbatched, and batched mutation testing. We see a slight increase in the number of test runs for most crates with a notable decrease of 36.1% for `regex`, and 35.6% for `regex-syntax`. The increase can be explained by the behavior of the parallel test runner, which waits for all already running tests to complete.

TABLE VI
MUTATION SCORES WITH, AND WITHOUT BATCHING.

Crate	Unbatched	Batched
alacritty	329.0(22.8%)	329.0(22.8%)
bat	221.0(73.7%)	235.0(78.3%)
exa	508.0(73.3%)	514.0(74.2%)
hashbrown	205.8 ± 1.8(76.5%)	243.4 ± 0.5(90.5%)
parking_lot	28.0(73.7%)	35.0(92.1%)
rand	779.0(59.6%)	792.0(60.6%)
rand_core	145.0(71.4%)	145.0(71.4%)
regex	736.0(62.3%)	855.0(72.3%)
regex-syntax	1520.0(65.9%)	1865.0(80.9%)
rustls	960.0 ± 0.7(69.5%)	991.4 ± 0.5(71.7%)

In conclusion for RQ3: Our greedy approximation of the batching problem produces significant improvements in runtime. Mutation batching improves evaluation times as well as compile times. The overall runtime of mutation testing is decreased by 12.9% to 66.4% depending on the crate. Mutation batching does not necessarily decrease the number of test evaluations, but we have seen reductions of up to 36.1%.

RQ4*: How does mutation batching perform in the context of flaky, non-deterministic, non-parallelizable test suites?*

While mutation batching does not affect the mutation score of deterministic test suites, it may report slightly different mutation scores on flaky, non-deterministic test suites. These are primarily caused by the presence of shared state, which is regarded as desirable to mock in unit tests, in part because it enables parallelized testing. The degree of variance introduced in these cases is important to consider. Table VI shows the difference in reported mutation scores compared to mutation testing without batching. We see no change in mutation score after batching in half of the studied crates. In the remaining crates — those with flaky test suites — we see deviations ranging between 0.9% for `exa`, and 18.4% for `parking_lot`, with a median deviation of 4.7%. The most affected crates, `parking_lot`, `regex-syntax`, and `hashbrown`, make use of mutable global state, making them difficult to write fully deterministic test suites for.

In conclusion for RQ4: Mutation batching does not affect the mutation score of deterministic test suites. Non-deterministic test suites may be affected to various degrees depending on flakiness. This is caused by the flaky test suite and is not a soundness issue.

VI. RELATED WORK

Mutation testing has been widely applied to many languages [6]. To the best of our knowledge, this is the first research work that targets the Rust programming language. Two recent and popular tools that have been studied include Major [14] and PIT [15] for Java. Major uses a set of syntax-based, compiler-assisted mutation operators, and embeds them into a single meta-mutant. It implements test case prioritization based on a monitored reference run. Our approach improves on Major, using an extended set of mutation operators, along with a parallel test scheduler that leverages simultaneous mutation evaluation that our batching technique affords. Conversely, PIT uses bytecode mutations to avoid compilation, at the expense of mutation operator expressivity.

There has been a lot of attention to reducing the cost of mutation analysis. These techniques are given excellent coverage in the survey by Pizzoleto et al. [20], including works that have sought to reduce the costs of *evaluating* mutations, that, as with this work, fall into Offutt and Untch's categories of "*do faster*" and "*do smarter*" [21]. The approach of Gopinath et al. [22], seeks to optimize the common code paths between the generated mutants, forking the program at each mutant. Sun et al. [23] performed a larger-scale study of a similar approach, grouping mutants in the same block together, compiling the resulting programs separately and forking the program execution for each individual mutant. These efforts primarily focus on reducing repeated work across mutant evaluations rather than increasing parallelism across the entire evaluation. Zhang et al. [24] use a family of techniques to prioritize, and reduce the tests needed to evaluate mutants, applying heuristics relating to coverage and execution history.

Mateo and Usaola [25] apply 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 [26] when developing the Javalanche mutation testing tool for Java. Just et al. [27] take these ideas beyond mutation reachability to derive information about state infection. This work differs in that we focus on the higher level unit of function calls reachable by tests. This is advantageous for most programs, since it can be performed statically, upfront, without any runtime analysis.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a set of novel mutation operators for Rust programs, introduce the notion of mutation *safety*, and explore mutation batching, a promising technique that improves the efficiency of parallelism in executing tests to evaluate mutations, and may be applied in mutation testing at large.

Our experiments show that mutation batching translates into a considerable decrease in the runtime of mutation testing evaluation, ranging between 4.8% and 94.69%, with a mean of 28.1%. The overall runtime of mutation testing is reduced between 12.9% and 66.4%, with a mean decrease of 25.7%.

Due to the promising results of this study, we plan, as part of our future work, to improve the results achievable by mutation batching, by developing better heuristics, and finding other, more effective approximations of the mutation batching problem. While the reachability metrics derived from static call graphs are sufficient for many programs, we intend to investigate the use of additional runtime coverage information, and how it might improve batching quality. Due to the general applicability of mutation batching to other programming languages, we may experiment with its use more widely. Since our technique requires memory safety, it may prove especially useful for languages with managed memory. In addition, we intend to explore the idea of an increased set of novel mutation operators based on common error patterns as well as complex, higher-order mutations' ability to surface more subtle issues.

References

[1] "The Rust programming language." [Online]. Available: https://www.rust-lang.org

[2] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *International Symposium on Software Testing and Analysis (ISSTA)*, 1993.

[3] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using conditional mutation to increase the efficiency of mutation analysis," in *International Workshop on Automation of Software Test (AST)*, 2011, pp. 50–56.

[4] "Replication package." [Online]. Available: https://github.com/rust-mutation-testing/mutation-batching-rep-package

[5] "Production users." [Online]. Available: https://www.rust-lang.org/production/users

[6] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

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

[8] M. Pool, "cargo-mutants," 2022. [Online]. Available: https://github.com/sourcefrog/cargo-mutants

[9] M. Betka and S. Wagner, "Towards practical application of mutation testing in industry — traditional versus extreme mutation testing," *Journal of Software: Evolution and Process*, vol. 34, no. 11, 2022.

[10] A. Denisov and S. Pankevich, "Mull it over: Mutation testing based on llvm," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018, pp. 25–31.

[11] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler intermediate representation," in *International Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 114–124.

[12] T. T. Chekam, M. Papadakis, and Y. Le Traon, "Mart: a mutant generation tool for LLVM," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 1080–1084.

[13] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[14] R. Just, "The MAJOR mutation framework: efficient and scalable mutation analysis for Java," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 433–436.

[15] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for Java (demo)," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 449–452.

[16] "Monomorphization — guide to rustc development." [Online]. Available: https://rustc-dev-guide.rust-lang.org/backend/monomorph.html

[17] "How safe and unsafe interact — the rustonomicon." [Online]. Available: https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html

[18] A. N. Evans, B. Campbell, and M. L. Soffa, "Is Rust used safely by software developers?" in *International Conference on Software Engineering (ICSE)*, 2020, pp. 246–257.

[19] O. Parry, M. Hilton, G. M. Kapfhammer, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, 2022.

[20] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, 2019.

[21] A. Offutt and R. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation Analysis Workshop*, 2000, pp. 34–44.

[22] R. Gopinath, C. Jensen, and A. Groce, "Topsy-Turvy: A smarter and faster parallelization of mutation analysis," in *International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 740–743.

[23] C. Sun, J. Jia, H. Liu, and X. Zhang, "A lightweight program dependence based approach to concurrent mutation analysis," in *Annual Computer Software and Applications Conference (COMPSAC)*, 2018, pp. 116–125.

[24] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2013, pp. 235–245.

[25] P. R. Mateo and M. P. Usaola, "Reducing mutation costs through uncovered mutants," *Software Testing, Verification and Reliability*, vol. 25, pp. 464–489, 2015.

[26] Z. A. Schuler D, "Javalanche: efficient mutation testing for Java," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 209, pp. 297–298.

[27] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 315–326.