

Collaborative Research: SHF: Small: Feedback-Driven Mutation Testing for Any Language

1 Overview and Objectives

1.1 Problem Statement

Mutation testing research dates back to the 70s, and has long aimed to render mutation testing useful for constructing high quality test suites and, by extension, software. Most of this previous research focuses on computing a mutation score, a measure of adequacy for a given test suite. However, this is computationally intensive for realistic projects, because it requires running many tests on many modified versions of a software system. Reducing that computational cost is thus a major thrust of mutation testing research [64].

Moreover, although test suite adequacy is certainly a useful thing to measure, the most important goal of mutation testing — and indeed its original use case [15] — is to help *improve* a test suite. For this purpose, neither a score nor a vast list of all unkillable mutants (which will exist for even very good tests of most software) is useful. An undifferentiated list of unkillable mutants typically contains mostly uninteresting or redundant mutants, and a much smaller number of *actionable mutants* that can improve tests. Examining surviving mutants even in the presence of formal verification and aggressive automated fuzzing is time-consuming and unpleasant, as PI Groce and co-authors have noted [49, 53, 8]. That work proposed a novel approach to formal verification and automated testing combining Karl Popper’s falsification-based notion of scientific discovery [93, 94] with mutation testing [49, 53, 8]. Groce and colleagues used the approach to identify multiple previously unknown faults in the Linux kernel’s RCU [23, 55, 80] module and the pyfakefs Python mock file system [82], despite the existence of very-high-quality automated test generation efforts for these systems [79, 51]. Follow-up preliminary work for this proposal has uncovered significant opportunities to improve the Slither static analysis tool for smart contracts [28, 110]. This prior work proposed a number of algorithms for finding bugs in testing and verification harnesses. At a high level, however, the core concept was simple: users should examine all unkillable mutants, and for each, either understand why it is not a test weakness or actually construct a way to kill it. This harkens back to the earliest ideas about mutation testing, but with automated support for both understanding and killing mutants.

Unfortunately, the methods proposed were limited in applicability, due to an assumption that the number of unkillable mutants was small and manageable. In practice, human attention does not scale to analyzing large numbers of unkillable mutants without considerable assistance in “triaging” them. That process bears a resemblance to the problem of manual confirmation of results from a machine-learning classifier [48, 69], where even highly-motivated scientific users are unwilling to examine more than a few tens of potentially incorrect results [99]. A realistic approach must leverage human effort in such a needle-in-a-haystack setting.

This project aims to make the falsification-driven approach to verification and testing feasible for larger projects, and those with (much) lower mutation scores. Our goal is to enable *Just Enough Mutation Testing*: We propose a mutation testing framework that identifies and interactively presents a few, very different, ranked mutants, and then *works with the user* to effectively improve the program, the test suite, or both.

Figure 1 shows the basic outline of a proposed workflow. Our framework aims to repeatedly compute and present a small, diverse set of unkillable mutants, chiefly characterized by their novelty, to present to the developer or test engineer by working *with* the test engineer to improve the SUT or the underlying tests. A key insight behind our proposed approach is to view mutant triage as analogous to the bug triage (a.k.a. “fuzzer taming”) problem in random testing/fuzzing [19, 58, 115, 116]: a user wants to quickly find mutants that indicate the most important “holes” in a testing or verification effort, and act on those most-critical gaps, possibly revealing faults in the System Under Test (SUT). Analogously, an unkillable mutant *is*, conceptually, a failing test — of the testing effort! Fuzzers tend to produce very large numbers of failing tests for a much smaller number of distinct bugs. Identifying the distinct, important bugs represented by one or two failing

tests in a set of thousands of duplicates, is typically difficult. Users do not (usually) care much about finding the group of all tests failing due to a fault (or, analogously, the set of all mutants killable by the same extension to a test suite or generator), but about seeing *several very different test failures* or *several different unkillable mutants* quickly, to maximize the chance of discovering the most important faults or holes in a testing effort. Previous efforts to identify “critical” mutants tend to focus on subsumption and potential to resolve other mutants, but this may be counterproductive for identifying the mutants most critical to resolve, which are likely to not subsume other mutants, precisely because they correspond to subtle assumptions in testing, not broad gaps in testing (e.g., not testing logging output). Our approach fundamentally seeks to identify a small set of maximally different, *maximally interesting* mutants, based on user feedback and project context.

The use of user feedback is critical to our goal. A test engineer has valuable insight — and the final word — about which unkillable mutants are uninteresting or equivalent, and which require changes to the system itself or the test harness. We propose to explicitly take this feedback into account, refining the analysis accordingly, as the user improves their system. Beyond allowing for more effective mutation analysis, this new paradigm provides a stopping rule other than patience, time available, or “every last mutant”: since mutants are ranked by likely payoff, once a user has examined several mutants in a row without benefit, or mutants that are highly similar in behavior to other mutants, a user may reasonably stop, knowing that the low-hanging fruit have probably all been picked. This approach also helps scale mutation testing to very large projects, since it only has to execute the test suite on mutants currently predicted to be of likely interest to the user.

Overall, our approach requires several fundamental research advances:

- **Efficient, any-language mutation testing.** Our goal of user-interactive mutation testing, integrated directly into the development and test process, requires that mutation generation be *source-level* and *multi-language*. Bytecode-level mutation is highly effective for computing mutation scores [20, 57]. However, developers or test engineers reason more naturally about a mutant’s implications when mutations are presented in the source language in question. Moreover, modern mutation testing frameworks are limited to specific languages, or to IR-defined ecosystems like LLVM or Java bytecode. This leaves out popular languages like Python, Ruby, or Go, not to mention project-specific Domain Specific Languages (DSLs) [29]. Meanwhile, the vast majority of real-world software projects are written in multiple languages [95]. We therefore propose novel mechanisms for *efficient, any language mutation testing* based on our novel recent work on language-agnostic declarative program transformation [113].
- **Mutant prioritization and selection.** Our proposed framework will present a small, maximally-informative ranked list of mutants to the user. Current mutation testing approaches make no real effort, with few exceptions [91, 17] to prioritize mutants. Other than (arguably) some efforts to incorporate dominance results [87], mutation testing approaches currently offer no way to maximize the novelty of presented mutants than stratified sampling [34], which does not aim at (or achieve) significant semantic novelty. Thus, we propose to adapt clustering optimization techniques based on novelty [30] to the problem of mutant selection, informed by a set of novel diversity metrics for the domain.
- **User feedback elicitation and analysis.** A user’s feedback about the most critical-to-test aspects of the code, or hard work examining some mutants, has no influence on the kinds of sampling currently proposed in the mutation testing literature. Even creating simple clusters of mutants that are not killed due to the same underlying omission in tests requires manual effort, with users, e.g., writing a Python script scanning

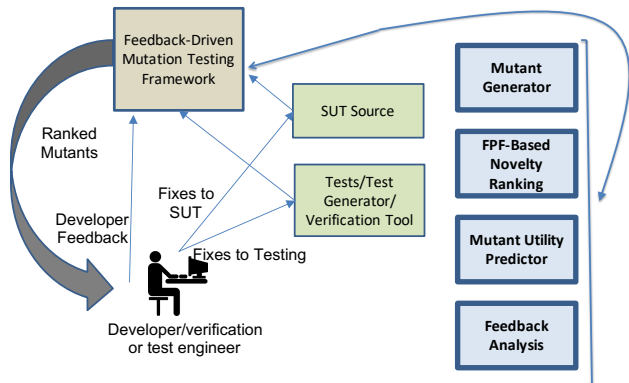


Figure 1: Proposed mutation workflow.

mutants for certain strings and assuming all mutated code with that string is part of the same “equivalence class.” This is a tedious and error-prone process, and only even possible once a “kind” of unkillable mutant is discovered, largely by ad hoc scanning of the list of unkillable, uncategorized, mutants. We propose “feedback-driven” mutation analysis that elicits and incorporates user input on mutants, tests, and the SUT, supporting a concise, updating list of mutants to inspect based on expected utility or payoff for the user. Feedback also opens up new ways to execute fewer mutants, often in parallel with useful user activity.

1.2 Problem Statement

Overall, this project aims to make the use of program mutants practical in non-research settings, in a way that meets developers’ actual needs: to make it possible for someone creating or enhancing a test suite to (1) use “just enough” mutation testing for their needs, maximizing benefit gained in exchange for work performed, and to (2) work in any programming language(s) without worrying about the quality of tool support, using intuitive source-based mutants and easy customization. Beyond traditional manual testing, this proposal additionally targets property-driven testing and formal verification, allowing for application to future-generation safety- and mission-critical systems that require automated test generation. To this end, we envision extending Test-Driven-Development (TDD) to become *Mutation-Driven-Development (MDD)*.

Problem: Develop highly automated methods and tools that allow the practical application of mutation testing to real-world software in a feedback-driven way, where user and mutation testing framework cooperate to improve testing efforts, while minimizing user effort and maximizing payoff.

1.3 PI Qualifications

PI Groce has been a user of, and contributor to, mutation testing tools for many years. He combines a long research track record in software testing, including mutation testing, with actual experience testing critical software systems at NASA’s Jet Propulsion Laboratory. PI Groce’s long-running interest in improving mutation testing arises from frustration in his efforts to apply mutation to the Mars Science Laboratory’s flight software, in particular to the file system [42, 43, 47]. This practical orientation informs his recent work on using mutation testing in a falsification-driven approach to improving verification and automated testing efforts [49, 53, 8]. PI Groce has extensive experience in developing mutation tools for new languages [73, 78, 54], including the first reliable tools for mutation of Haskell, Python, Rust, Swift, and various smart contract languages, as well as in user-facing (vs. researcher-oriented) automated software testing tools [59, 31]. He additionally has expertise in driving testing of machine learning systems through user interaction [69, 48].

PI Le Goues is an expert in applied program analysis, program transformation, and testing, most relevantly through her pioneering work in automated program repair (including heuristic [77] and semantic [76, 68] dynamic approaches, and approaches guided by static analysis [112]). She has significant experience with testing, mutation testing for fault localization and program repair particularly [108], and the challenges of syntactic program modification and transformation. In particular, her recent work has developed novel mechanisms for efficient and expressive language-agnostic syntactic program transformation [113], with applications for, e.g., program repair [112], fuzz test triage [116], and static analysis customization [114].

The PIs provide a detailed work and collaboration plan in the Collaboration Plan supplementary document.

1.4 Intellectual Merit

This project addresses core problems not limited to practical application of mutation testing, but generalizable to fundamental issues in software engineering and program semantics, e.g., how to represent source changes and (mostly statically) predict the similarity of their impact on semantics, and predict which tests are likely to detect these changes. How can novelty of information presented to a user be effectively balanced with a-priori predictions of the utility of that information, where likely-high-utility data points may also be similar to each other? How can user feedback best be incorporated into such efforts? This project also

considers connections raised by preliminary work, concerning new methodologies for testing/verification effort development. Can theoretical ideas about the nature of scientific discovery [93, 94, 71] be applied to such efforts? Is falsification by alternative hypotheses about the power of a testing/verification effort translatable to an actionable, effective approach for building systems [49, 53]? The work on any-language mutation testing looks at syntactic patterns common to almost all programming languages, and relies on categorizing languages into families based on similarity, including of syntactic to semantic mappings.

2 Research Plan

Figure 1 shows the basic outline of a proposed workflow and components needed to support feedback-driven mutation testing. These components serve to organize the research plan.

2.1 Mutant Generator

The most widely-used mutation testing tool in the real world is PIT [20], which targets Java bytecode. There are recent attempts to provide the same kind of support for other languages, especially C, by targeting LLVM IR [57, 18]. This poses several problems for feedback-driven mutation testing. First, bytecode- or IR-level mutation works well to compute a score for a test suite, but is not suitable for presentation to developers or test engineers, who need to reason about a mutant’s implications for their source or test code. Java developers think in terms of Java, not compiled bytecode; C and C++ developers certainly do not generally understand LLVM IR. Even when possible, translation may not help: a bytecode-level mutation may not have a simple source-level equivalent, especially if the bytecode has been optimized. Second, features that help identify semantically similar (or dissimilar) mutants are hard to identify at the bytecode level. Even if the mutant is, for example, a constant replacement in one case and an arithmetic operator replacement in another, the fact that both take place inside an argument to a logging function with an INFO argument may be enough to predict that their effects are redundant. Finally, IR-specific tools are by construction limited to their associated language ecosystems, excluding a large body of code written in other languages, as well as project-specific DSLs. Moreover, the vast majority of software projects are written in multiple languages [95]. Our desire for real-world applicability thus motivates polyglot analysis and testing infrastructure.

Thus, our first research task is to develop a novel mutant generator that (1) operates at the source level, with output that is easy for developers to understand and the framework to analyze; (2) is, as much as possible, *language independent*, and applicable to projects written in a heterogeneity of languages; and finally (3) is efficient. The PIs’ recent advances in universal mutation [54, 40] and language-agnostic declarative program transformation using parser combinators [113] provide key motivation and starting points for this component.

2.1.1 Background and preliminary work

The universalmutator. First, PI Groce and colleagues released a fully functional, regular-expression-based mutant generator [54, 40]. The `universalmutator` does not attempt to parse source code, but simply defines mutation operators by a set of regular-expression-defined text transformations. These are organized into a hierarchy, so that for, e.g., a Swift program, first “universal” operators are applied, then “C-like” operators, and finally Swift-specific rules.

Importantly, PI Groce demonstrated that the `universalmutator` tool generated numbers of mutants and kill ratios for Java code comparable to PIT [20] and Major [66]. For falsification-driven verification, the regular-expression-based approach produced mutants of equal value to those produced by Andrews’ tool [11] and Muupi [78] for C and Python, respectively. This demonstrates that multi-language, syntax-driven, source-level mutation is *feasible* and, indeed, effective, producing results competitive with state-of-the-art single-language mutation tools. It is already being used, for instance as the Ethereum foundation recommended tool for mutation testing of Solidity smart contracts. However, it has significant limitations for long-term utility, motivating this project: Because the source code is not parsed, and applies regular expressions to lines of code, not larger blocks, the technique generates many mutants that are invalid and cannot be compiled,

or that are trivially equivalent because they, e.g., mutate “source code” in a large comment block. Finally, regular expressions are only occasionally a natural notation for expressing source-level mutations.

Comby: Declarative, any-language transformation. Fortunately, recent work by PI Le Goues and collaborators introduced a powerful new representation to declaratively transform richer syntactic structures in programs across multiple languages [113]. The key innovation is to convert declarative transformation templates into parsers that directly match and transform source code of interest. Parser combinators [61] define how to match nested structures (much like traditional AST visitor traversals) but without the need to define or build an intermediate AST. At a high-level, comby performs context-free parsing of syntactic structures that typically correspond to nested expressions and blocks in an underlying AST (without actually building one); regular expressions are not powerful enough to recognize such program terms in general. Our parser combinator mechanism elegantly encapsulates the complexity of heterogeneous, multi-language syntax as a composition of small parsers (some being language-specific, and others being language-general). Conversely, defining and building a generic parse tree data structure (i.e., to encompass the complexity of multi-language syntax, which may be inherently incompatible) *after* parsing is comparably difficult.

This work has already shown its practical utility by performing lightweight refactors in more than 10 languages (including those targeted by `universalmutator`). The associated tool, comby [1], enables a new language-general way of expressing transformations that regular expressions cannot typically recognize (e.g., nested code blocks). We showed that comby has equivalent expressive power for a variety of nontrivial transformations in existing language-specific refactoring tools. Comby’s parser driven approach is highly performant at scale, processing upwards of a quarter billion lines of code in 42 minutes, using 20 cores.

Coupling with declarative syntax manipulation that goes beyond the limits of regular expressions promises to yield more effective mutation testing by (a) targeting more sophisticated properties of programs, and (b) delivering more user-accessible tools for developing mutation transformations. Our joint advances in real-world tooling (i.e., in `universalmutator` and comby) suggest that this goal is imminently feasible. At heart, this line of work represents the conviction of the PIs that mutation testing (like automated program repair) is simply an instance of the general field of automated program transformation [89].

Source mutation as compiler fuzzing method. We have jointly used ideas from `universalmutator` and comby to develop a novel approach to compiler fuzzing, supported by a new tool [41], which has resulted in discovery of more than 100 bugs in production compilers, more than 90 of which have been confirmed and fixed, and a substantial bug bounty from the Ethereum foundation [35, 36]. Work from this project will benefit that effort, and we expect work from that effort to benefit polyglot mutation.

2.1.2 Proposed work: Any language mutation

Our goal is a source-level mutant generator that can apply to any language, maximizing applicability and usability. We propose to enable sufficiently expressive, efficient mutation by using (and extending) comby [1, 113] for specifying transformations and generating mutants. Comby uses declarative templates that describe before/after changes for program fragments. For example, the following transformation swaps the first two arguments of the function `memcpy`:

```
memcpy(:[1], :[2], :[3]) ==> memcpy(:[2], :[1], :[3])
```

Hole syntax `:[1]` binds syntax to a variable. A unique property of comby templates is that variables *only* bind to syntax that occurs inside well-balanced delimiters (like parentheses), whitespace is handled intelligently, and syntax otherwise matches literally. Concretely, this means that the template above seamlessly transforms syntax structure in complex fragments as in the following:

Templates offer a *declarative* description for matching and rewriting these structures in a way that is syntactically close to the source code. In addition, comby can distinguish between code, strings, and comments (common to virtually all languages). Comby is language-aware in the sense that small language

```

memcpy(*stream->main_data + stream->md_len,      memcpy(mad_bit_nextbyte(&stream->ptr),
mad_bit_nextbyte(&stream->ptr),      ==>      *stream->foo_data + stream->md_len,
frame_used = md_len - si.main_data_begin);      frame_used = md_len - si.foo_data_begin);

```

definitions describe whether syntax should be balanced (e.g., parentheses or braces) or delineate strings or comments. These definitions describe a coarse structural decomposition of programs (as typically understood by compilers) rather than just a sequence of characters (as treated by regex). Language definitions already exist for 20+ languages, and comby supports a simple extension mechanism for new languages (e.g., Solidity) or custom DSLs (<https://comby.dev/#faq-language-support>). One ongoing limitation of mutation testing is that tools are often research projects, and eventually become unusable due to lack of support, even in mainstream languages such as Java and C [33]. This is because mutation tools that fully parse a language and guarantee generation of valid programs complex, hard-to-maintain-and-extend systems; language complexity makes such a tool for C++, for example, extremely daunting. From this development and maintenance perspective, comby provides a considerable benefit in its easy adaptability to new languages.

Accessible structural syntax manipulation can achieve a leap in expressivity and effectiveness for mutation testing. For one, targeted structural and contextual changes are more likely to produce well-formed syntactic programs that exercise interesting paths in a test suite. We propose to extend `universalmutator` to generate mutants from comby templates, and to evaluate the efficacy of multi-language mutation testing with structural code changes. We propose to further enhance the expressive power of mutation operators by incorporating static semantics (e.g., type information and variable scope) via a DSL for structural syntax changes. We propose to extend our template language and processing framework to interface with the Language Server Protocol [5], which provides a queryable interface for such static program properties. This new capability will go beyond syntax (and “regexps”) and yield greater, semantic-driven precision.

The generator will also need to be improved to allow users to easily specify novel build environments and plug-ins for checking Trivial Compiler Equivalence [88] to make the entire feedback-driven mutation process workable. Interactive, human-in-the-loop workflows for large-scale automated code changes have proven essential over the last decade, and are now widely adopted in industry [3, 2, 4]. Interactive prompts incorporate human oversight before the code is changed, e.g. as in Facebook’s dart code mod tool <https://pub.dev/packages/codemod>. We propose to adapt these existing interactive workflows to enable a feedback-driven mutation process. For example, the user is presented a source-level mutation, and interacts with a prompt to indicate a yes/no signal to validate and refine desirable mutations generated by new plugins. Because some distance metrics (described below) may require compiling mutants, which is costly, a specialized projection of the distance only requiring textual analysis will need to be developed, to allow generation, compilation, and execution of only high-priority mutants for very large projects.

2.2 FPF-Based Novelty Ranking

Our vision of feedback-driven mutation testing is predicated on selecting and ranking a small set of highly interesting mutants to present to developers. An unkilld mutant is, conceptually, very similar to a failing test. It presents information of possible relevance to a developer, such as the presence of a previously unknown but critical fault (in the SUT or in testing); a previously unknown but unimportant fault; or, even less interesting, an equivalent mutant or inherently flaky test. An unkilld mutant or failing test may also contain information that is uninteresting because *it duplicates information already examined*. Although equivalent mutants may indicate an opportunity for refactoring or improving efficiency [62, 53], examining a mutant that is equivalent or extremely similar to an already-understood mutant is almost never worthwhile. A key research challenge is thus to select and prioritize mutants that provide maximum utility to a user.

As noted above, with a few exceptions [91, 17, 87], current mutation testing approaches offer little in the way of prioritization beyond dominance (which requires executing tests on mutants) or stratified sampling [34, 87]. Stratified sampling does not aim at semantic novelty, and can present many mutants from

the same class, if applied at the method level, even if those mutants are highly similar in impact. Other work [32] proposes random sampling as the most effective way to select mutants. Unfortunately, when an important class of unkillable mutants has only a few members, random sampling is useless. Our preliminary results argue that novelty-based clustering is a promising approach for solving this problem.

2.2.1 Background and preliminary work

Furthest point first and fuzzer taming. Fuzzer taming [19] was a solution PI Groce and colleagues proposed to the problem of triaging test failures in automated test generation [115]. Like mutant generators, fuzzers tend to produce very large numbers of failing tests (mutants) for a much smaller number of distinct bugs (interesting behaviors). Finding the set of distinct bugs, and identifying important bugs that need to be fixed immediately is difficult, because the important bugs may be represented by only one or two failing tests in a set of thousands of failing tests, most of which are duplicates. The fuzzer taming work proposed that rather than highly imprecise clustering, which does not work well in practice, and handles outliers in a way that does not match the “power law” distribution of bugs, an algorithm matching the goal of ranking maximally-different test failures highly was appropriate.

The *furthest-point-first* (FPF) algorithm of Gonzalez [30] does precisely this. FPF, beginning with any randomly chosen test (or mutant, in the present setting), always ranks next the point in a metric-defined space that has the *greatest distance from the previously ranked point to which it is closest*. That is, for each point (test or mutant) not yet presented to the user, FPF finds the closest among all already-ranked points, and associates each unranked point with the distance to that closest point. The unranked point with the largest such distance is then added to the ranking, and the process is repeated. FPF can be computed by a greedy algorithm, and is known to approximate novel-item discovery for an optimal clustering [30]. Preliminary work on the fuzzer taming problem using FPF-based techniques [19, 58] can be directly applied to the different problem of ranking unkillable mutants such that novel mutants are presented first.

Preliminary use of FPF-based mutant ranking: static analysis tool evaluation and improvement. We have used the `universalmutator` and our basic idea to formulate a novel way to evaluate and (most relevant to this proposal) *improve* static analysis tools. This work is in preparation for submission, but some results from it have already appeared in open source tools. The proposed approach is simple in outline:

1. Run each tool on a set of unmutated source code target(s), and determine the *baseline*: the number of (non-informational/stylistic) static analysis findings produced.
2. Generate mutants of the source code targets and run each tool on each mutant of each target. Consider a mutant killed if the number of findings for the mutated code is greater than for the baseline.
3. Compute, for each tool, the *mutant ratio*: the mutation score ($\frac{|killed|}{|mutants|}$) divided by (mean) baseline.
4. Discard all mutants not killed by at least one tool and all mutants killed by all tools. What remains allows *differential* analysis. Examine the remaining mutants in the difference in *prioritized* order.

We used a simple textual, ad-hoc scheme, to examine the most interesting mutants in the “diff” between static analysis tools (rather than a sophisticated prioritization scheme). This was helpful for understanding our general results, showing differences in Solidity, Java, and Python static analysis tools and confirming known rankings of such tools based either on user impressions or more limited experiments. However, it was essential in a way that is highly relevant to this proposal, namely in how to use our results, and the set of mutants in the “diff” of three Solidity smart contract [16, 118] analysis tools, to improve the best of those tools. Using prioritized mutants, we identified three new detector patterns for the Slither [28] tool, based on mutants killed by Securify [111], SmartCheck [97], or both, but not by Slither: Boolean constant misuse, type-based tautologies, and loss of precision due to ordering of arithmetic operations.

All three of these new detectors were submitted as PRs to the Slither project, vetted, and accepted for release in the public version of Slither. All three detectors were added as medium severity detectors, and have detected flaws in real contracts. The first mutants in prioritized rank exhibiting these issues were the 2nd, 9th, and 12th non-statement-deletion mutants ranked for SmartCheck, out of over 800 such mutants.

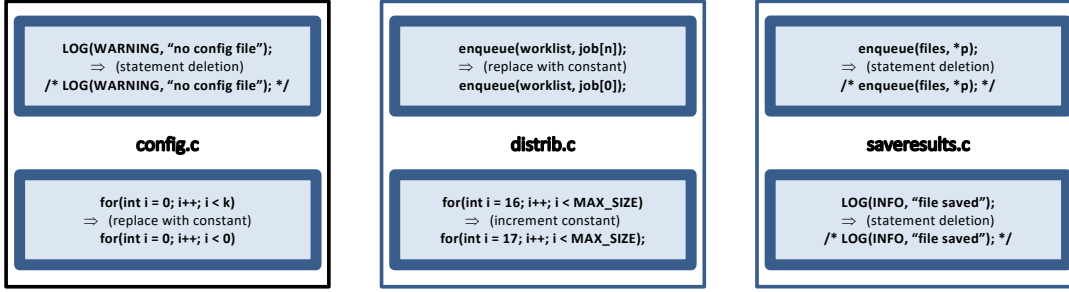


Figure 3: Which mutants are most similar? If the user marked the mutant in the upper left corner as uninteresting and added a test to kill the mutant in the upper middle, which mutant should she examine next?

Using our prioritization, it was possible to identify these issues by examining fewer than 20 unkilld mutants. Without prioritization, on average a developer would have to look at more than 200, 80, and 400 mutants, respectively, to find instances of these problems. Examining the first 100 mutants in the unprioritized lists for SmartCheck and Securify, ordered by contract ID and mutant number we were unable to identify *any* obviously interesting mutants, suggesting that it is indeed hard to use mutation analysis results without prioritization. Our preliminary results are available at <https://github.com/agroce/slithermutate>.

2.2.2 Proposed work: Mutant Novelty Ranking using FPF

Ranking unkilld mutants according to how much “new” information they might provide to users requires more than simply using the FPF algorithm as in fuzzer taming. The key difference is the problem of determining how similar two mutants are. In fuzzer taming, information about similarity of failing tests comes from executing the tests, and, in fact, from executing the tests on program mutants [19, 58]; in contrast, avoiding the expense of even compiling uninteresting mutants is one of our goals.

FPF requires a distance metric, and a distance metric requires a *representation* of mutants. Mutants can be similar because they modify the same line, function, class, or module, but also because, despite being located in very different parts of a program, they are very semantically similar. E.g., a mutant to the parser of a compiler, to an I/O error-handling routine in the code generator, and to a complex optimization pass may all be very “similar” in the only meaningful sense if all three mutants modify logging statements that don’t have any actual effect on the state of the compiler. Figure 3 shows the fundamental problem. It is not, a priori, obvious which mutants here are most (dis-)similar. Every mutant has multiple plausible “nearest neighbors” — another mutant in the same file (likely to impact the same aspects of correctness), another mutant with very similar code (likely to have the same kind of semantic impact on the local context), or another mutant with the same operator (perhaps likely to have some similarity, though probably of a lower importance than the previous two types of similarity). Are all logging statements equivalent, or are only INFO logging calls similar, while every WARNING, ERROR or FATAL is unique? Some of these decisions are likely to be project-independent, or even developer-expertise-driven, and so a good metric may well change during feedback-driven mutation testing, in response to information from users (see Section 2.4 below).

Elements of the distance metric obviously include, at minimum, mutant location, mutation operator, and some representation of the code element modified — language construct, functions called, variables modified, and so forth. These static aspects may also be augmented with user feedback (as noted above), but also with dynamic information obtained during the process, such as frequency with which tests cover the mutated statements/modules, or the way the mutant changes the program path. PI Le Goues, with collaborators, has proposed dynamic metrics for heuristic program repair, including those measured over program state [21, 86] and intermediate predicate behavior [25] (mutation testing has strong conceptual analogies to program repair [117], suggesting natural application of these measures to our new domain).

We likely require *two* distance metrics: one for selecting candidate mutants to execute, that uses only static information and user feedback, and one that uses testing results to refine the notion of similarity among executed mutants. This is therefore a quite complex problem in representation and weighting of elements of a representation, especially for a language- and project- agnostic metric, that is also open to tuning via feedback analysis. One approach to the problem is to exploit metric learning methods [70], which were used in some of PI Groce’s previous work [90]. But, to avoid over-fitting to even a set of good examples, the final metric may have to be largely hand-tuned, and designed to incorporate feedback and dynamically extracted information, which is not easily handled with learned metrics. In part this is due to the difficulty of establishing large amounts of ground truth data, and the reality that cross-project data will be less valuable than project-specific data from users; there are unsupervised approaches to metric learning [98, 109], but most popular approaches require supervision.

2.3 Mutant Utility Predictor

Novelty with respect to previously analyzed mutants is not the only important characteristic of a mutant. Presenting a novel, but likely equivalent mutant is often a waste of time, though some equivalent mutants can be useful for identifying optimization opportunities or refactorings. Furthermore, of two similar mutants next to be presented, it is better to present one that is higher in the mutant dominance hierarchy (the one such that its tests will kill more other mutants). There has been some initial work on predicting mutant quality attributes and utility [87, 17], including estimating how hard mutants will be to kill, statically. In addition to advancing the state-of-the-art in that respect, feedback-driven mutation testing also requires determining how to balance the need for novelty and the predicted utility of a mutant. For example, a utility-driven ranking might suggest avoiding a highly novel mutant because it is likely equivalent; however, it may be that labeling this mutant as equivalent lets the FPF ranking avoid numerous other similar mutants — e.g., postponing labeling a logging statement as confirmed equivalent by the user may be a bad idea.

2.4 Feedback Analysis

The “feedback-driven” aspect of feedback-driven mutation analysis requires that information from the user be given high priority in the process, a process with no clear equivalent in any previously proposed mutation testing work. The most straightforward example is that if a user adds a test to kill a mutant, and marks that as a “high impact” action (the omitted testing was potentially allowing serious faults to pass without detection) or even “fault-revealing” (the new test detected a real fault in the system), then it may be most effective to abandon the search for novelty and instead search for very similar mutants still not killed by any test, in the expectation that these may also result in high impact or fault-revealing tests. If a user marks a mutant as “equivalent, but indicative of a refactoring opportunity”, the same logic may apply: similar mutants in other parts of the code base may show the same problem with code quality, even if they are predicted to be equivalent, and are not highly novel. In addition to informing the system of how useful various analyzed mutants were, a user should also be able to inform the system about correct and incorrect novelty rankings: if the system presents a mutant that is, from the user’s POV, a (near-)duplicate of an already handled mutant, the user should be able to express this fact, and avoid future similar bad novelty estimates.

While large-scale crowdsourcing of user feedback is likely only possible in some unusual industrial settings [91, 62], it may also be possible to apply mini-crowdsourcing techniques developed in the context of testing machine-learning classifiers to mutant ranking and analysis [101]. For high-visibility, high-criticality code such as, e.g., Linux kernel modules, this may be a very powerful tool. The challenge in such a case is to allow communication between feedback-driven mutation efforts, splitting work both so as to minimize duplication and to target appropriate developers, as in automated assignment of bug reports [13, 65].

2.5 Mutation-Driven Development

The primary focus of this project is to develop feedback-driven mutation testing. However, the ideas of Test-Driven Development (TDD) [12, 63], which repeatedly turns requirements into specific test cases, then

implements just enough functionality to pass the current tests, can be generalized into a mutation-driven form. A weakness of TDD is that the code will be narrowly tailored to the requirements, which produce tests most effectively for “shall” type behaviors [60]. But for security and safety, “shall not” requirements that are omitted can be disastrous. Mutation-Driven Development (MDD) in its simplest form would require an application of feedback-driven mutation to the test suite at each development step, to ensure that code not only does what the tests require, but that the tests also sufficiently constrain the code to capture implicit shall-nots. Since such a process implemented by modifying TDD-driven tests would likely break the clean and appealing mapping between tests and requirements, and manual tests are inherently weak, for high-criticality systems, MDD should focus on augmenting TDD-driven tests with falsification-driven formal verification and automated testing. One way to do this would be to “elaborate” TDD-produced unit tests into parameterized unit tests [107, 106], perhaps using a tool like DeepState [31] for C/C++. In such a process, weakness exposed by feedback-driven mutation testing would be addressed by taking an existing unit test and generalizing some parameters and assertions to kill the relevant mutants, letting, e.g., afl [121], libFuzzer [100], or a symbolic execution tool [102, 103, 81] identify specific inputs. The focus of the MDD process would be on producing and enhancing test harnesses [37, 59] that can kill *all* interesting mutants, in a semi-automated fashion.

Our final research goal, to enable cooperation between mutation, developer intuition, static analysis, and fuzzing and other automated testing systems, in a seamless fashion, is most easily expressed in a scenario. We have italicized novel aspects of Laura’s work that would result from this project.

Our Vision: A Day in the Life of a Systems Programmer Laura is developing a data-management system for a NASA/JPL Mars rover. She is responsible both for a catalog process that handles data products and critical configuration files and a low-level file system for custom, radiation-hardened, flash software. Laura has, in the process of developing the code, also developed a set of manually constructed unit tests *and harnesses supporting automated test generation for the system, using IDE-supported MDD*. She employs static analysis tools, with additional custom rules developed by the rover software team and the NASA facility’s software tooling experts. Laura has also *added a few custom rules that eliminate certain file system API mutants not easily detected dynamically*.

Today, Laura is continuing an ongoing effort at changing functionality to the catalog and the file system, rewriting the “rename” call to restrict behavior more than the POSIX standard. After reviewing the risks of arbitrary rename, the software team has agreed this protects the integrity of the file system while retaining all functionality ground operators might need to fix problems. As part of this revision, Laura realized that with these restrictions a change to the core rename atomicity scheme could make mounting and checking the file system both faster and less complex (thus likely less buggy). Laura reaches a point where she believes her change is fully implemented and the tests have been revised to check the changed behavior. *She looks at a panel in her IDE showing Test Effectiveness, and notices that there are four interesting Missed Bugs for her to examine. She clicks on the top one, and the IDE focuses on a line of code in the rename function. The display shows a change to that line, and says that altering the line in the way shown changes the file system binary but is not detected by any existing manual test or static analysis rule, and has not been detected in two minutes of fuzzing, across multiple cores, or by any stored fuzzing-generated test. Because Laura has been tuning the MDD-alert system via feedback, all four mutants are all highly interesting to her, and very different from each other; this one is the most interesting of all.*

Laura thinks for a while about the change made, which involves passing a flag to the call that disables an expensive “sanity check” on header values for a file, to be used in contexts where the header has just been written and checked. She thinks the current context is unsafe, and the check is required. The danger arises from certain hardware failure modes, so she goes to the hardware emulator code that she uses to run tests without access to the rover testbeds, and *requests the system to generate tests that target not just the mutated line of code (the system has already generated a number of these for her to examine), but the additional line*

with the relevant simulated hardware failure injection. The fuzzers run in a targeted mode for a few minutes, and show Laura traces. She runs some of these through a debugger, goes to the blackboard and calls in a colleague to discuss her reasoning. They agree the check is indeed not needed, which will result in a small optimization in the file system. Given the process she uses for developing her code, and the high quality of automated tests associated, Laura feels confident, given the argument, in optimizing the code. The system now informs her that a very different mutant has the highest priority among uncaught bugs. Of course, the system couldn't detect the obvious mutant to the new version of the code, enabling the check, but Laura long ago informed the system that adding the check was never going to break any tests (the check is too cheap to break performance tests in any one instance. It therefore won't trouble her with this problem, now. A small symbol attached to the parameter on the line of code does allow Laura to see that this line has an ignored unkillable mutant, which she can inspect if she is worried about the code.

Because Laura has been using the “MDD” process all along, her additional confidence tests will detect any problems with the optimization is not just based on mutation analysis of her current code. Instead, it is based on mutants of all the versions of the design and functionality she has implemented; if a problem is representable by a mutant of code no longer present (the older rename behavior, for example), then while not forming a basis for mutant alerts now, that behavior, in terms of API calls and hardware simulation choices, will be permanently in place in the stored corpus of fuzzing tests or manually constructed tests to address mutants. So long as Laura allows the tool to guide her to always kill all interesting mutants it advises her about, she will have these “mutant-regression” tests as a “memory” of the varying possibilities of her design, and the assurance tasks associated with design perimeters.

2.6 Core Research Questions

The component-focused sections above provide an overview of the research problems to be addressed by this proposal, but it is also useful to consider the high-level research questions as a whole:

1. What advances are required in order to maximize the efficiency and usability of any-language mutation?
2. How can a domain-specific language enable new, more expressive mutation operators for structural code changes without compromising on the usability of a regular search-and-replace approach?
3. What is a good generalized, language-agnostic mutant representation and distance metric?
4. How can FPF-based selection of mutants for novelty best incorporate predictions of mutant equivalence, outcome, dominance, and productivity? Is novelty or expected utility more important?
5. How can feedback-driven mutation testing most effectively incorporate feedback, including from crowds?
6. How can we perform on-the-fly and parallel mutant evaluation/generation guided by (predicted) FPF?
7. Can we predict the *cause* of mutant unkillability (e.g., oracle, coverage, equivalence, or nondeterminism)?
8. Is distance-based clustering plus timing information useful for quickly eliminating killable mutants similar to already-killed mutants? How does this relate to Predictive Mutation Testing (PMT)?
9. How can we integrate all of these aspects into a complete automated testing approach, and assist users in killing surfaced mutants, by guiding/enhancing automated test harnesses, or targeting fuzzer runs?

2.7 Evaluation Plan

We propose a multi-prong evaluation strategy, including (1) proxy metrics suitable for evaluating program transformation and mutation testing, and the individual components of the overall research program, (2) a series of lab studies on tool usability and mutation-driven development as a paradigm generally, and (3) qualitative experiences using and evaluating the tool, ideally in collaboration with industrial partners.

Proxy metrics for individual research components. Just computing a mutant kill matrix for a test suite can partially evaluate novelty rankings, by ranking killed, rather than unkillable mutants. This experiment realistically represents an early stage of test suite construction by feedback-driven mutation testing, especially if the mutants are only killed by a relatively small set of tests. If mutant X and mutant Y are both highly ranked, but killed by a very similar set of tests, this indicates a possible problem with the measure. In

real-world feedback-driven mutation testing, it is highly desirable not to compute the full kill matrix for all mutants. But, for evaluation purposes, determining the extent to which kill vector similarity agrees with FPF distance metric similarity serves as a basic, if imperfect, sanity check on the novelty ranking. Another automated way to evaluate a novelty ranking is to use automated testing to generate multiple tests to kill each mutant in ranked order. A good ranking will mean that each additional mutant is unlikely to be killed by the killing tests for any previous mutants. A similar, but more robust, measure of mutant similarity is how much adding a test that kills mutant A as a seed in a fuzzer [121, 100] makes it easier to kill mutant B.

Evaluation of the mutant generator can also be partly automated, by comparing the output set of mutants to that of other tools, to ensure no important mutants are omitted; the regular-expression based approach will probably generate valid mutants not generated by other tools, since it aims at a rich operator set, in accord with the suggestions of multiple previous papers on detecting faults via mutation [67, 34]. We will further evaluate and delineate the relative effectiveness of the new modes of expressive power (proposed in Section 2.1.2) compared to regular expressions: We will catalog the efficiency of new mutant operators that rely on structural syntax changes both with and without incorporation of static program properties (like type information). Efficiency gains can be measured by simple, traditional measures.

Lab studies. We will conduct pilot and lab studies to evaluate tool design and usability. These studies will generally involve asking programmers to perform a set of constructed software testing, bug finding/fixing, or maintenance tasks, with or without a tool. The results of such studies can be evaluated using both quantitative measures (like time and success rate on the provided tasks) as well as qualitative and theory-building techniques to surface important challenges or benefits to the tooling or underlying Mutation-Driven Development methodology. We can enhance external validity by basing the programming tasks on common forum postings, as we have done previously for studies of debugging challenges in particular contexts [120]; other researchers have demonstrated this type of methodology useful for tool pilot studies [105]. We may also construct implementation tasks around small, but easy-to-get-wrong code problems like binary search and AVL trees, especially when evaluating the potential benefits and key challenges for MDD. Methods used in our previous studies to control for user expertise and prior knowledge of problems will generally carry over, as we expect not to encounter prior expertise in mutation testing itself in this setting.

In addition to traditional testing tasks, we will also conduct studies on the enhancement of random testing harnesses, with one set of experiments focusing on enhancing specifications/properties, and another set focusing on enhancing the set of generated tests.

We may make use of “dummy” or “Wizard of Oz” versions of elements of the framework to isolate the effects of specific features, like the FPF-based novelty metric, or various choices for feedback elicitation or analysis. We will use think-aloud protocols [27], in which participants are instructed to continuously verbalize what they are thinking/attempting. This method helps the examiners determine why participants behave in particular ways, and to identify and isolate sources of confusion. These barriers can then be used to improve the underlying research technique. Again, to mitigate the risk of highly varying programmer skill, we will use a counterbalanced, within-subjects design, exposing each participant to both experimental and control conditions. The design just outlined has been profitably used in many experiments about programming tools and methodology [26, 104]. PI Le Goues has experience in both lab studies involving think aloud protocols [120] as well as qualitative analysis generally [7], and will lead the design and execution of these lab studies at CMU. Note that, to our knowledge, these experiments will also provide the first well-founded human studies of the baseline value of mutants in improving test suites in a traditional development setting, not just of our proposed methods; in itself this is a major research contribution.

Experiential, qualitative evaluations. Evaluation for this proposal includes both human-performed assessment via trying to use feedback-driven mutation for actual test improvement tasks and automated evaluations with more objective criteria, but a weaker relationship to the actual goal of helping expert developers in real projects quickly find the most important unkillable mutants. For the human portion, informal evaluation will be

performed by the research team itself, using known programs with known testing weaknesses; this will help tune the approach and experiment with new ideas. However, for more advanced assessment, expert users outside the team will be offered the chance to use the system once it is in suitable shape. In the past, PI Groce has worked with IBM Distinguished Engineer Paul E. McKenney on using mutants to improve Linux kernel test suites, and has discussed similar efforts with Richard Hipp, the lead developer of the SQLite database, a famously well-tested program, with some resulting improvements to both test suites. Other potential users with whom PI Groce has a working relationship include NASA/JPL engineers working on upcoming CubeSat missions, colleagues working on the DeepState [31] parameterized unit testing interface to fuzzers and symbolic execution engines, and the developers of pyfakefs. This type of evaluation is, of necessity, somewhat qualitative. A more unbiased retrospective version that retains the core element of human rating of the value of mutants can be performed by examining actual mutants that resulted in improvements to the rcutorture [79] tests for the Linux kernel and the pyfakefs tests [53], and comparing them to highly ranked mutants: could the highly ranked mutants have exposed the same problems?

Mutation-Driven Development. The evaluation of techniques and tools developed in this proposal will also serve a dual purpose with respect to Mutation-Driven-Development. Using an MDD approach to implement various small, but easy-to-get-wrong, code projects, such as binary search and AVL trees, will make it possible both to see how effective the tools for feedback-driven mutation testing are, and to see how effective an MDD approach to development is. In addition, using various versions of actual TDD efforts, with the associated test suites for each step of development, we can identify the “missing tests” MDD would add.

3 Closely-Related Work

Mutation Testing. The vast body of work on mutation testing or analysis dates to the late 70’s [22, 15]. The most closely related work by others is that of Roman and Mnich [96], which studied the effectiveness of introducing mutation testing into the test-driven development (TDD) process in a student setting. They showed that tests written using TDD with the additional aid of mutation analysis had better code coverage on the code from TDD-only groups than the TDD-only tests did, and found more post-release defects in the TDD code. The study is promising in that it suggests that mutants can indeed be useful in real testing tasks. However, the study size is quite small and limited in both the considered contexts and the strength of its conclusions. Critically, the approach suggested simply could not scale to large numbers of mutants without the addition of feedback techniques such as we propose. Similarly, a paper by Petrović et al., to appear at ICSE this year [92], while limited in various ways (e.g., relying on Google’s rule of only looking at one mutant per line, and using mutant exposure and review requests rather than directly measuring use of mutants to derive tests), shows that even a limited use of mutants can positively affect test quality and fault detection power; our approach enables more aggressive and efficient, scalable, approaches.

Numerous approaches seek to reduce the cost [64] of mutation analysis. Offutt and Untch [85] categorize these, as: do *fewer* (e.g., operator selection, mutant sampling or clustering), do *smarter* (i.e., intelligent organization to reduce time taken for the entire analysis), and do *faster* (i.e., in terms of single mutant evaluation time) approaches. We offer a fundamentally different *do-fewer/do-smarter* method that combines FPF-novelty, utility prediction, and user interaction for a more radical reduction in mutants run.

Practical Mutation Analysis. The above work largely focuses on computing or estimating a test suite’s mutation score, using mutation testing as an “evaluation” of a test suite (like a coverage metric). In contrast, this proposal focuses on the problem of presenting unkilld mutations to a developer or test engineer in a way that facilitates the improvement of a test suite and the detection of faults, inspired by PI Groce’s previous work on using mutation to find defects in verification and test generation efforts [49, 53, 8].

Recent work by Papadakis et al. has aimed, unusually, at predicting the “quality” of [87] or even prioritizing [17] mutants, to rank fault-revealing mutants highly so that users can produce tests to find faults, focusing on a single static pass to rank mutants by their fault-revealing potential, informed by (possibly

cross-project) data on fault-revealing tests. There is no feedback loop, or ability to indicate the *importance* of various faults, and only LLVM bitcode mutants are targeted.

Google is well known for applying mutation testing to real-world projects [91]. Their approach uses a notion of feedback, but this is essentially manual, and based on using a classification scheme to heuristically throw out “arid” (likely not to be actionable) mutants. As noted above, the system only permits one mutation per line of code (due to the size of the code base and the nature of the code review process). Our approach in contrast targets the individual(s) developing, testing, or verifying a particular software element (either a small project, or a component of a project), and assumes an iterative process, where developers consider one unkilld mutant at a time. The Google approach does not prioritize unkilld mutants, support custom mutation operators, or learn an individual testing effort’s characteristics; it only even proposes mutants of code in a diff with a previous version of the code. The Google report notes their approach does not truly scale, since it requires manual work for each heuristic and language, a problem this proposal aims to solve.

More broadly, a paper by the authors of the Google report and a group of academic mutation testing researchers [62], uses the Google effort to propose a notion of *productive* versus *unproductive* mutants. Their concepts are highly related to this proposal’s goals, but again centered on a diff-focused, large-scale industrial setting, rather than an approach that, like TDD, may also be applied to smaller coding efforts in a more isolated setting, such as development of embedded software, where crowdsourcing is impractical or irrelevant. Furthermore, we argue mutants are neither “productive” or “unproductive” in an absolute sense, but rather their value of a mutant depends on previous mutants a developer has seen, and other context.

4 Broader Impacts

Improving Software System Reliability: A key element of broader outreach will be to report bugs discovered during testing experiments, and contribute improved test suites to critical open source projects. To that end, this proposal will primarily target real world systems in experiments, in hopes of improving their quality, and the quality of their testing. Infrastructure developed in preliminary work includes automated testing for the Linux kernel RCU module, Google and Mozilla JavaScript engines, a variety of C, smart contract, and Go compilers (including GCC and LLVM), YAFFS2 [119] and other file systems, a large set of Unix utilities, and critical Python libraries (including key scientific, ML, and numeric analysis tools). Discussions with working test engineers at Mozilla, Google, Trail of Bits, and NASA have significantly informed the PIs’ research efforts, and this is likely to continue. PI Groce is discussing plans for incorporating advanced automated test generation into NASA’s open source F Prime flight architecture [14, 83], and F Prime components are a likely target for evaluation efforts in this project. Such efforts will result in a documented process for incorporating feedback-driven mutation testing and MDD into flight software development. Better, cheaper, high-quality testing for small budget CubeSat [84] missions could lead to advances in various fields, especially Earth observation and space-based physics. CubeSat focuses on providing a low-cost way for educational institutions and non-profits to conduct space-based research, and thus is related to education and outreach. In the long term, an MDD paradigm might result in easier development of critical software in tandem with an extremely high-quality, specification-defining automated test suite. The existence of such suites might make modifying critical systems easier, since the in-place test suite would be likely to identify even subtle problems introduced during changes. With the growing impact of embedded, cyberphysical, and Internet-of-Thing systems on the physical world, this has potential safety benefits for the general public.

Education and Outreach: The proposed research yields several opportunities for enhancing CS education, recruiting new CS majors, and retaining CS students, particularly members of underrepresented groups. PI Groce will work with the NAU Student ACM Chapter to present a series of “excursions in testing” that introduce automated testing to students, using feedback-driven mutation testing and Mutation-Driven-Development (MDD) on real code, including code from media player libraries. The work of Guzdial [56] has shown that media computation is a potentially effective way to both recruit and retain female and under-represented minority students in computer science.

Undergraduate research mentorship: The PIs are fundamentally committed to widening the pipeline of students interested in and equipped to pursue a research career. PI Le Goues is co-director of the REUSE@CMU summer program (funded in part by NSF CNS-1852260, a renewal of CNS-1560137, on which she is a Co-PI). The site trains students in all elements of research, and specifically seeks students representing underserved demographic groups, early in their undergraduate careers, and at schools without traditional research opportunities. So far, the REUSE program has resulted in undergraduate research by 89 total students, of whom 48 (54%) identified as women; 24 (27%) were drawn from racial and ethnic groups that are under-represented in computing (some are both). More than 80% of the graduated students are now doing research full-time, either in graduate school or at national or corporate research labs.

Undergraduate researchers could contribute to multiple areas of the proposed work, from design and implementation to evaluation. For example, undergraduates Zhen Yu Ding (University of Pittsburgh '21) and Yiwei Lyu (Carnegie Mellon University '21) designed and evaluated the study of diversity metrics for novelty-promoting search-based software engineering that informs some of the proposed work in novelty measurement; as an REU summer student, David Widder (University of Oregon '17, now a CMU PhD student) designed and conducted half of the talk aloud studies in the qualitative work PI Le Goues conducted with collaborators on the challenges of framework debugging [120]. We will continue to integrate undergraduates both in the summer program and throughout the school year in the proposed project.

5 Results From Prior NSF Support

PI Groce has received support as PI or co-PI from three NSF grants. The most relevant and recent is “Diversity and Feedback in Random Testing for Systems Software” (CCF-1217824, \$491,280, 9/2012–9/2017), a collaborative proposal with John Regehr at the University of Utah. **Intellectual Merit:** The results of CCF-1217824 included a preliminary exploration of how to “tame” fuzzer output, a problem also central to this proposal [19]. In previous work, the goal was to find an algorithm for using hand-chosen distance metrics to identify bugs in tests. Many other key results [9, 122, 44, 45] used mutation testing as an evaluation method. **Broader Impacts:** CCF-1217824 has contributed to the discovery of previously unknown faults in multiple open-source and commercial software systems, including core compilers and system libraries. The development of the central swarm testing techniques has furthered many efforts to improve the quality of compilers, including LLVM and GCC, and to test core language tools in general [74, 72, 24, 75]. **Research Products:** Publications resulting from CCF-1217824 were numerous [45, 19, 122, 46, 44, 9, 52, 59, 50, 10, 59, 38], along with three PhD theses. Source code for software systems developed or enhanced during CCF-1217824 [39, 51] is available on GitHub.

PI Le Goues’ most closely-related prior NSF grant is CAREER Quality Matters: Dynamic, Static and Proactive Analyses for Automated Program Repair (CCF-1750116, \$525,000, 3/2018 – 2/2023). **Intellectual Merit:** The results of this award have so far included novel techniques for static program repair [112]; an initial exploration of diversity-enhancing dynamic repair techniques [25]; and the Comby tool and associated mechanism for declarative, language-agnostic program transformation using parser combinators [113] and its application to improving fuzz testing [116] and static analysis [114]. Neither mutation testing nor language-agnostic program transformation primitives are the core focus of the prior award, however, which instead focuses specifically on developing push-button automatic program repair techniques. This new proposal seeks to extend the work on declarative program transformation, with a particular application to mutation testing. **Broader Impacts:** The award has so far supported five REU students, including several students from underrepresented groups in computing or who lack access to traditional research opportunities at their home institution. Additionally, the tools and techniques developed in the research so far are open source and available on GitHub, and the PPC work has been presented to a mixed audience of academics and developers at StrangeLoop [6], an important form of outreach to the engineering community.

References

- [1] Comby. <https://github.com/comby-tools/comby>, 2019. Online; accessed 22 October 2019.
- [2] Refactoring with codemod.py. <https://www.facebook.com/notes/justin-rosenstein/refactoring-with-codemodpy/44817317582/>, Online. Accessed 16 April 2019.
- [3] Facebook’s Codemod on GitHub. <https://github.com/facebook/codemod>, Online. Accessed 16 April 2019.
- [4] Codemod for Dart. <https://pub.dev/packages/codemod>, Online. Accessed 16 April 2019.
- [5] The Language Server Protocol. <https://microsoft.github.io/language-server-protocol/>, Online. Accessed 16 April 2019.
- [6] Parser Parser Combinators for Program Transformation. <https://www.thestrangeloop.com/2019/parser-parser-combinators-for-program-transformation.html>, Online. Accessed 10 November 2019.
- [7] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 96–107, 2020. doi:10.1109/ICST46399.2020.00020.
- [8] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E. McKenney. Applying mutation analysis on kernel test suites: an experience report. In *International Workshop on Mutation Analysis*, pages 110–115, March 2017.
- [9] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christy. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 70–81, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi:10.1145/2931037.2931056. URL <http://doi.acm.org/10.1145/2931037.2931056>.
- [10] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. Evaluating non-adequate test-case reduction. In *31st IEEE/ACM Conference on Automated Software Engineering*, pages 16–26, September 2016.
- [11] James H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411, 2005.
- [12] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [13] Pamela Bhattacharya, Iulian Neamtiu, and Christian R Shelton. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 85(10): 2275–2292, 2012.
- [14] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. F prime: An open-source framework for small-scale flight software systems. In *Small Satellite Conference*, 2018.
- [15] Timothy Alan Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233. ACM, 1980.
- [16] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. <https://github.com/ethereum/wiki/wiki/White-Paper>, 2013.
- [17] Thierry Titchou Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *CoRR*, abs/1803.07901, 2018. URL <http://arxiv.org/abs/1803.07901>.
- [18] Thierry Titchou Chekam, Mike Papadakis, and Yves Le Traon. Mart: a mutant generation tool for LLVM. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.*, pages 1080–1084, 2019. doi:10.1145/3338906.3341180. URL <https://doi.org/10.1145/3338906.3341180>.
- [19] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John

- Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.
- [20] Henry Coles. Pit mutation testing. <http://pittest.org/>.
 - [21] Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. A novel fitness function for automated program repair based on source code checkpoints. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO 2018, pages 1443–1450. ACM, 2018. doi:10.1145/3205455.3205566. URL <https://doi.org/10.1145/3205455.3205566>.
 - [22] Richard A DeMillo and Richard J LiptonFrederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
 - [23] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012. ISSN 1045-9219. doi:<http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159>.
 - [24] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using clp (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 482–493. IEEE, 2015.
 - [25] Z. Y. Ding, Y. Lyu, C. Timperley, and C. Le Goues. Leveraging program invariants to promote population diversity in search-based automatic program repair. In *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*, pages 2–9, May 2019. doi:10.1109/GI.2019.00011.
 - [26] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefix. How do API documentation and static typing affect API usability? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 632–642, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi:10.1145/2568225.2568299. URL <http://doi.acm.org/10.1145/2568225.2568299>.
 - [27] K. Anders Ericsson and Herbert A. Simon. Verbal reports as data. *Psychological Review*, pages 215–251, 1980.
 - [28] Josselin Feist, Gustavo Greico, and Alex Groce. Slither: A static analysis framework for smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019.
 - [29] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.
 - [30] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.
 - [31] Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.
 - [32] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. How hard does mutation analysis have to be, anyway? In *International Symposium on Software Reliability Engineering*. IEEE, 2015.
 - [33] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. Does choice of mutation tool matter? *Software Quality Journal*, 25(3):871–920, September 2017. ISSN 0963-9314. doi:10.1007/s11219-016-9317-7. URL <https://doi.org/10.1007/s11219-016-9317-7>.
 - [34] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, 66(3):854–874, 2017.
 - [35] Alex Groce. Breaking the Solidity compiler with a fuzzer. <https://blog.trailofbits.com/2020/06/05/breaking-the-solidity-compiler-with-a-fuzzer/>, 2020.
 - [36] Alex Groce. A year in the life of a compiler fuzzing campaign. <https://blog.trailofbits.com/2021/03/23/a-year-in-the-life-of-a-compiler-fuzzing-campaign/>, 2021.
 - [37] Alex Groce and Martin Erwig. Finding common ground: Choose, assert, and assume. In *International Workshop on Dynamic Analysis*, pages 12–17, 2012.
 - [38] Alex Groce and Jervis Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

- [39] Alex Groce, Yang Chen, John Regehr, Chaoqiang Zhang, and Amin Alipour. Swarmed versions of random testing tools. https://github.com/agroce/swarmed_tools, .
- [40] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. Regex based tool for mutating generic source code across numerous languages. <https://github.com/agroce/universalmutator>, .
- [41] Alex Groce, Rijnard van Tonder, Claire Le Goues, and John Regehr. afl-compiler-fuzzer. <https://github.com/agroce/afl-compiler-fuzzer>, .
- [42] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.
- [43] Alex Groce, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *Workshop on Constraints in Formal Verification*, pages 1–15, 2008.
- [44] Alex Groce, Chaoqiang Zhang, Mohammad Amin Alipour, Eric Eide, Yang Chen, and John Regehr. Help, help, I’m being suppressed! the significance of suppressors in software testing. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 390–399, 2013.
- [45] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2014*, pages 255–268, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1. doi:10.1145/2661136.2661157. URL <http://doi.acm.org/10.1145/2661136.2661157>.
- [46] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 243–252. IEEE, 2014.
- [47] Alex Groce, Klaus Havelund, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.
- [48] Alex Groce, Todd Kulesza, Chaoqiang Zhang, Shalini Shamasunder, Margaret Burnett, Weng-Keen Wong, Simone Stumpf, Shubhomoy Das, Amber Shinsell, Forrest Bice, and Kevin McIntosh. You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. *IEEE Transactions on Software Engineering*, 40(3):307–323, March 2014. ISSN 0098-5589. doi:10.1109/TSE.2013.59.
- [49] Alex Groce, Iftekhar Ahmed, Carlos Jensen, and Paul E McKenney. How verified is my code? falsification-driven verification. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 737–748. IEEE, 2015.
- [50] Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.
- [51] Alex Groce, Jervis Pinto, Pooria Azimi, Pranjal Mittal, Josie Holmes, and Kevin Kellar. TSTL: the template scripting testing language. <https://github.com/agroce/tstl>, May 2015.
- [52] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*, 26(1):40–68, 2016.
- [53] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E McKenney, and Josie Holmes. How verified (or tested) is my code? falsification-driven verification and testing. *Automated Software Engineering Journal*, 25(4):917–960, 2018.
- [54] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE ’18*, pages 25–28, New

- York, NY, USA, 2018. ACM. ISBN 978-1-4503-5663-3. doi:10.1145/3183440.3183485. URL <http://doi.acm.org/10.1145/3183440.3183485>.
- [55] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, May 2008.
 - [56] Mark Guzdial. A media computation course for non-majors. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '03, pages 104–108, New York, NY, USA, 2003. ACM. ISBN 1-58113-672-2. doi:10.1145/961511.961542. URL <http://doi.acm.org/10.1145/961511.961542>.
 - [57] Farah Hariri and August Shi. 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 2018, Montpellier, France, September 3-7, 2018*, pages 860–863, 2018. doi:10.1145/3238147.3240482. URL <http://doi.acm.org/10.1145/3238147.3240482>.
 - [58] Josie Holmes and Alex Groce. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *IEEE International Symposium on Software Reliability Engineering*, 2018.
 - [59] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar, and James O’Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 20(1):57–78, 2018.
 - [60] Ivy Hooks. Writing good requirements. In *INCOSE International Symposium*, volume 4, pages 1247–1253. Wiley Online Library, 1994.
 - [61] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.
 - [62] Goran Petrović Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*. IEEE Press, Piscataway, NJ, USA, pages 47–53, 2018.
 - [63] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.
 - [64] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
 - [65] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4):1533–1578, 2016.
 - [66] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE*, 2011.
 - [67] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 654–665, Hong Kong, China, 2014. ACM. ISBN 978-1-4503-3056-5.
 - [68] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306, Lincoln, NE, USA, November 2015. doi:10.1109/ASE.2015.60. DOI: 10.1109/ASE.2015.60.
 - [69] Todd Kulesza, Margaret M. Burnett, Simone Stumpf, Weng-Keen Wong, Shubhomoy Das, Alex Groce, Amber Shinsel, Forrest Bice, and Kevin McIntosh. Where are my intelligent assistant’s mistakes? A systematic testing approach. In *End-User Development - Third International Symposium, IS-EUD 2011, Torre Canne (BR), Italy, June 7-10, 2011. Proceedings*, pages 171–186, 2011. doi:10.1007/978-3-642-21530-8_14. URL https://doi.org/10.1007/978-3-642-21530-8_14.
 - [70] Brian Kulis. Metric learning: A survey. *Foundations & Trends in Machine Learning*, 5(4):287–364, 2012.

- [71] Imre Lakatos. The role of crucial experiments in science. *Studies in History and Philosophy of Science Part A*, 4(4):309–325, 1974.
- [72] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner’s luck: A language for property-based generators. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.
- [73] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. MuCheck: An extensible tool for mutation testing of Haskell programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 429–432. ACM, 2014.
- [74] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2014.
- [75] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 327–337. ACM, 2015.
- [76] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering, ESEC/FSE ’17*, pages 593–604, 2017. doi:10.1145/3106237.3106309.
- [77] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38:54–72, 2012. ISSN 0098-5589. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2011.104.
- [78] Xin Liu. muupi mutation tool. <https://github.com/apepkuss/muupi>.
- [79] Paul E. McKenney. RCU torture test operation. <https://www.kernel.org/doc/Documentation/RCU/torture.txt>.
- [80] Paul E. McKenney. Structured deferral: synchronization via procrastination. *Commun. ACM*, 56(7): 40–49, July 2013. ISSN 0001-0782. doi:10.1145/2483852.2483867. URL <http://doi.acm.org/10.1145/2483852.2483867>.
- [81] Mark Mossberg. <https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/>, April 2017.
- [82] mrbean bremen, jmcgeheeiv, et al. pyfakefs implements a fake file system that mocks the Python file system modules. <https://github.com/jmcgeheeiv/pyfakefs>, June 2011.
- [83] NASA. F prime: A flight-proven, multi-platform, open-source flight software framework. <https://github.com/nasa/fprime>, 2018.
- [84] National Aeronautics and Space Administration. CubeSat launch initiative. <https://www.nasa.gov/content/about-cubesat-launch-initiative>, 2018.
- [85] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.
- [86] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering*, 23(5):2980–3006, 2018. doi:10.1007/s10664-017-9562-9. URL <https://doi.org/10.1007/s10664-017-9562-9>.
- [87] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 32–39, April 2018. doi:10.1109/ICSTW.2018.00025.
- [88] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*, pages 936–946, 2015.
- [89] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236,

- September 1983. ISSN 0360-0300. doi:10.1145/356914.356917. URL <http://doi.acm.org/10.1145/356914.356917>.
- [90] Y. Pei, A. Christi, X. Fern, A. Groce, and W. Wong. Taming a fuzzer using delta debugging trails. In *2014 IEEE International Conference on Data Mining Workshop on Software Mining*, pages 840–843, Dec 2014. doi:10.1109/ICDMW.2014.58.
 - [91] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18*, pages 163–171, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5659-6. doi:10.1145/3183519.3183521. URL <http://doi.acm.org/10.1145/3183519.3183521>.
 - [92] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. Does mutation testing improve testing practices? In *International Conference on Software Engineering*, 2021. To appear.
 - [93] Karl Popper. *The Logic of Scientific Discovery*. Hutchinson, 1959.
 - [94] Karl Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. 1963.
 - [95] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 155–165, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi:10.1145/2635868.2635922. URL <http://doi.acm.org/10.1145/2635868.2635922>.
 - [96] Adam Roman and Michal Mních. Test-driven development with mutation testing—an experimental study. *Software Quality Journal*, pages 1–38, 2020.
 - [97] Tikhomirov S. and et al. Smartcheck: Static analysis of ethereum smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.
 - [98] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.
 - [99] Judith Segal. Some problems of professional end user developers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC '07*, pages 111–118, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2987-9. doi:10.1109/VLHCC.2007.50. URL <http://dx.doi.org/10.1109/VLHCC.2007.50>.
 - [100] Kostya Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity, Development (SecDev), IEEE*, pages 157–157. IEEE, 2016.
 - [101] A. Shinsel, T. Kulesza, M. Burnett, W. Curran, A. Groce, S. Stumpf, and W. Wong. Mini-crowdsourcing end-user assessment of intelligent assistants: A cost-benefit study. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 47–54, Sept 2011. doi:10.1109/VLHCC.2011.6070377.
 - [102] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, 2016.
 - [103] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, 2016.
 - [104] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects’ constructors. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi:10.1109/ICSE.2007.92. URL <https://doi.org/10.1109/ICSE.2007.92>.
 - [105] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. Structuring documentation to support state search: A laboratory experiment about protocol programming. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 157–181, Berlin, Heidelberg, 2014. Springer Berlin

- Heidelberg. ISBN 978-3-662-44202-9.
- [106] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.
 - [107] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests with Unit Meister. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 241–244, 2005.
 - [108] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. An investigation into the use of mutation analysis for automated program repair. In *Proceedings of the 9th International Symposium on Search Based Software Engineering*, SSBSE ’17, pages 99–114, 2017. doi:10.1007/978-3-319-66299-2_7. URL https://doi.org/10.1007/978-3-319-66299-2_7.
 - [109] Michael E Tipping and Christopher M Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.
 - [110] Trail of Bits. Slither: a static analyzer for solidity. <https://github.com/crytic/slither>, 2018.
 - [111] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. ACM Conference on Computer and Communications Security, 2018.
 - [112] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE 2018, pages 151–162. ACM, 2018. doi:10.1145/3180155.3180250. URL <https://doi.org/10.1145/3180155.3180250>.
 - [113] Rijnard van Tonder and Claire Le Goues. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Programming Language Design and Implementation*, PLDI ’19, pages 363–378, 2019.
 - [114] Rijnard van Tonder and Claire Le Goues. Tailoring programs for static analysis via program transformation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE ’20, page 824–834, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi:10.1145/3377811.3380343. URL <https://doi.org/10.1145/3377811.3380343>.
 - [115] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 612–622, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5937-5. doi:10.1145/3238147.3238200. URL <http://doi.acm.org/10.1145/3238147.3238200>.
 - [116] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 612–622. ACM, 2018. doi:10.1145/3238147.3238200. URL <https://doi.org/10.1145/3238147.3238200>.
 - [117] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE’13, pages 356–366, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-0215-6. doi:10.1109/ASE.2013.6693094. URL <https://doi.org/10.1109/ASE.2013.6693094>.
 - [118] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. <http://gavwood.com/paper.pdf>, 2014.
 - [119] Yaffs. Yaffs: A flash file system for embedded use. <http://www.yaffs.net/>.
 - [120] Claire Le Goues Christopher Bogart Zack Coker, David Gray Widder and Joshua Sunshine. A qualitative study on framework debugging. In *IEEE International Conference on Software Maintenance and Evolution*, ICSME ’19, 2019.
 - [121] Michal Zalewski. american fuzzy lop (2.35b). <http://lcamtuf.coredump.cx/afl/>, November

- 2014.
- [122] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 160–170, 2014.