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

1 Overview and Objectives

Mutation testing is: ¡FIXME¿. Significant prior work, dating back to the FIXMEs, aims to render mutation testing useful for constructing high quality test suites and, by extension, software. Most of this previous mutation testing 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 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 [62].

Although test suite adequacy is certainly a useful thing to measure, the most important goal of mutation testing — and indeed its original use case [**?**] — is to help *improve* a test suite. For this purpose, either a score or a list of all unkilled mutants generated across an expensive mutation campaign is not useful for practicing engineers. An undifferentiated list of unkilled mutants contains many uninteresting or redundant mutants, and a much smaller number of actionable mutants that are maximally useful in guiding test improvement. Thus, examining all unkilled mutants is only practical for formal verification efforts or critical software systems with high-powered test suites.

But even in these settings, examining surviving mutants produced by modern mutation testing campaigns is time-consuming and unpleasant, as PI Groce and co-authors noted in prior work on using mutants to drive formal verification and automated testing [47, 51, 3]. This work proposes a novel approach to formal verification and automated testing combining Karl Popper's falsification-based notion of scientific discovery [97, 98] with mutation testing [47, 51, 3]. The heart of the idea is (1) a surviving non-equivalent mutant *falsifies* the claim that a given formal verification or test effort captures a full notion of correctness and (2) refining a verification or testing effort by repeated efforts at falsification is an effective method for ensuring the quality of verification and testing efforts. This work resulted in the identification of multiple previously unknown faults in the Linux kernel's RCU [21, 53, 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, 49]. Out preliminary work on falsification-driven methods proposed a number of algorithms and methods for finding bugs in testing and verification harnesses, rather than the code under test. At a high level, however, the core concept was simple: users should examine all unkilled mutants of a program, and for each mutant either understand why it is equivalent or uninteresting, or actually construct a way to kill it. In a sense, this actually harkens back to the earliest ideas about mutation testing, but with much more automated support.

Unfortunately, the methods proposed were, while useful, limited in applicability. They simply assumed that the number of unkilled mutants was small, and focused on solving the problem of helping a developer or test engineer move from an unkilled mutant to killing it. In practice, however, human attention does not scale to analyzing large numbers of unkilled mutants without further assistance in "triaging" the mutants. The process of manually examining mutants bears a resemblance to the problem of manual confirmation of results from a machine-learning classifier [46, 67], where even highly-motivated scientific users are typically unwilling to examine more than a few tens of potentially problematic results [104]. The manual mutant-examination process was simply not feasible unless the number of unkilled mutants was relatively small, because the testing was already very high quality. For sufficiently large software systems, even a very high quality testing effort may fail to kill a large absolute number of mutants. The original approach simply provided no way to scale human efforts to 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 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 use those mutants to effectively improve the program, the test suite,

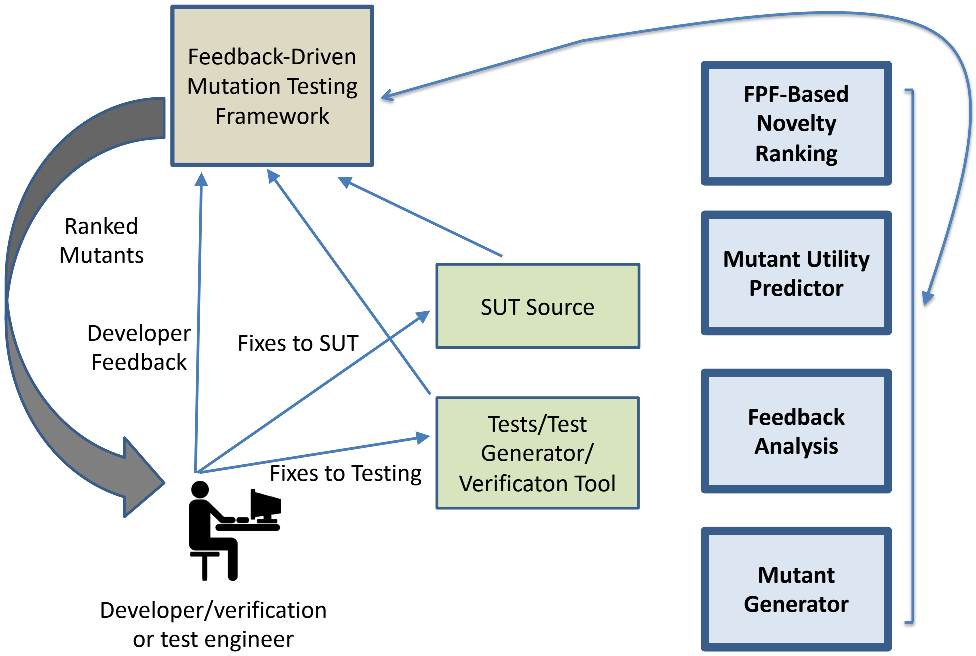

Figure 1: Basic flow of feedback-driven mutation testing. FIXME: unify caption phrasing with "just enough mutation testing" while still talking about feedback-driven mutation. FIXME: possibly reorder the figure to match the fact that I moved mutant generation up.

or both.

Figure 1 shows the basic outline of a proposed workflow. An unkilled mutant is, conceptually, very similar to a failing test. With a larger number of unkilled mutants, the problem becomes one very much like the bug triage or "fuzzer taming" problem in random testing/fuzzing [16, 121, 122]: 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). Fuzzers tend to produce very large numbers of failing tests for a much smaller number of distinct bugs. 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. Users do not (usually) care much about finding the group of all tests failing due to a fault, or the set of all mutants killable by the same extension to a test suite or generator, but about seeing *many very different test failures* or *many different unkilled mutants* quickly, to maximize the chance of discovering the most important faults or holes in a testing effort. Thus, our proposed mutation framework aims to generate a small, diverse set of mutants that are characterized by their novelty to present to the developer or test engineer.

The proposed framework then works *with* the test engineer to improve the SUT or the underlying tests, and incorporate user feedback into which mutants are interesting or useful (or not). This results in a constantly-updated ranking of mutants, based on actions and feedback from the engineer. This 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 are highly similar in behavior to other mutants, a user may reasonably stop, knowing that the low-hanging fruit have probably all been picked. It also enables a new way to improve the efficiency of mutation testing: even for a very large project, it only has to build and execute a small set of mutants, because it only runs the test suite on mutants currently predicted to be of likely interest to the user.

Overall, our approach requires several fundamental research novelties:

- **Efficient, any-language mutation testing.** The most widely used mutation testing tool in the real world is PIT [17], which targets Java bytecode. There are recent attempts to provide the same kind of support for other languages, especially C, by targeting LLVM IR [55]. This poses several problems for real-world mutation testing. First, bytecode- or IR-level mutation works well to compute a score for a test suite, but is not not suitable for presentation to developers or test engineers, who need to reason about a mutant's implications for their source or test code (i.e., Java developers think in terms of Java, not compiled bytecode). 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 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, bytecode-level mutation is highly language-specific, leaving out popular languages like Python, Ruby, or Go, not to mention project-specific Domain Specific Languages (DSLs) [27]. This is a problem for tool uptake and applicability: the vast majority of real-world software projects are written in multiple languages [99]. We propose novel mechanisms for *efficient, any language mutation testing* based on our novel recent work on language-agnostic declarative program transformation using parser combinators [120]. This technique operate at the source level and produces mutants and mutations that are easy for developers to understand.
- **Mutant prioritization and selection by predicted payoff.** Current mutation testing approaches make no real effort, with few exceptions [96, 15] to prioritize mutants, and none are based on a user-centered feedback loop, where the user and mutation testing framework interact to improve a test suite, automated test generator, or verification harness — and the SUT. Other than (arguably) some efforts to incorporate dominance results [91], no mutation testing approaches currently suggest any more sophisticated way to maximize the novelty of presented mutants than stratified sampling [34]; 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 [31] proposes random sampling as the most effective way to select mutants. Unfortunately, when an important class of unkilled mutants has only a few members, random sampling is almost guaranteed to fail to present any of them. We propose to adapt clustering optimization techniques based on the idea of novelty [29] to the problem of mutant selection, informed by a set of novel diversity metrics selected for this 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 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 unkilled mutant is discovered, largely by ad hoc scanning of the list of unkilled, 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.

Because our proposed feedback-driven approach no longer requires small absolute numbers of unkilled mutants, it extends the applicability of the approach to using manual tests to kill mutants, which was not in scope when extremely high kill-rates were required.

1.1 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 without worrying about the quality of tool support, and while providing intuitive source-based mutants and easy customization. This project also aims to make use of the insights of Test-Driven-Development (TDD), and proposes using mutation testing to move beyond a paradigm where developers build a series of tests narrowly tailored to steps in development, and use Mutation-Driven-Development (MDD) to build automated test generators or verification harnesses that handle not only anticipated problems imagined during development, but problems not anticipated by developers. In addition to traditional manual testing, this proposal targets property-driven testing and full formal verification of software components, in order to be practical in the future, when software systems will often be so safety- or mission- critical that even "good" manual testing is simply not an acceptable approach.

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 the ability to quickly find the most important weaknesses of testing or verification.

1.2 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 [40, 41, 45]. This practical orientation informs his recent work on using mutation testing in a falsification-driven approach to improving verification and automated testing efforts [47, 51, 3]. PI Groce has extensive experience in developing mutation tools for new languages [71, 77, 52], including the first reliable tools for mutation of Haskell, Python, and Swift, as well as in user-facing (vs. researcher-oriented) automated software testing tools [57, 30]. He additionally has expertise in driving testing of machine learning systems through user interaction [67, 46].

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 [**? ?**] and semantic [74, 66] dynamic approaches, and approaches guided by static analysis [119]). She has significant experience with testing, mutation testing for fault localization and program repair particularly [115**?**], 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 [120], with applications for, e.g., program repair [119], fuzz test triage [122], and static analysis customization [**?**].

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

1.3 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

```
\+ ==> -            == ==> !=         (\D )(\d +)(\D ) ==> \1(\2+1)\3
\+ ==> *            == ==> <          (\D )(\d +)(\D ) ==> \1(\2-1)\3
\+ ==> /            == ==> >          (\D )(\d +)(\D ) ==> \g <1>0\3
".+" ==> ""         while ==> if      (^\s *)(\S +.*)\n ==> \1\2\n \1break;\n
```

Figure 2: Some universal mutation rules

effort development. Can theoretical ideas about the nature of scientific discovery [97, 98, 69] 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 [47, 51]? 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, and how they share common meaningful syntactic changes that translate to interesting semantic changes.

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 `mutant generator` is the source of all mutants to be presented to the user. Our goal of user-interactive mutation testing, integrated directly into the development and test process, imposes several requirements on this component.

First, to be practically effective, mutations must be presented at the source level, rather than at the level of an intermediate representation like Java bytecode or LLVM. Bytecode-level mutation is highly effective for computing mutation scores [17, 55]. However, developers or test engineers can reason more naturally about a mutant's implications when mutations are presented in the source language in question. That is, Java developers think in terms of Java, not compiled bytecode; C and C+++ developers certainly do not generally understand LLVM IR; test engineers are even less likely to appreciate such low-level descriptions. Bytecode-level modifications can be difficult to translate to source level, especially when optimized. Moreover, many mutations that are natural or conceptually at teh source level (such as statement deletion) are known to be difficult to implement in bytecode. Bytecode-level modifications can also be more difficult to reason about in terms of similar semantic effects: two very different operator replacements may be very related, if they are both applied to the arguments to the same method.

This motivates both mutation generation and presentation at the program source level. Note, however, that the vast majority of software projects are written in multiple languages [99], motivating polyglot analysis and testing infrastructure. This further motivates that the `mutant generator` be language agnostic, and applicable to languages beyond those supported by a particular IR, or even customizable project-specific DSLs.

2.1.1 Background and preliminary work

The PI's combined prior work provides two important pieces of evidence for the feasibility of Any-Language mutation.

The universalmutator. First, PI Groce and colleagues released a functional, regular-expression-based mutant generator [52, 39]. This `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 if a program is, e.g., written in Swift, the "universal" mutation operators that apply to all programming languages are first applied, then operators for "C-like" languages, and finally a set of Swift-specific rules are applied. Figure 2 shows some of the current set of "universal" rules applied to all languages. Adding a new language, even a custom DSL, or a new set of project-specific rules for an existing language, in this approach, simply requires writing a new rule file and defining where it lies in the language hierarchy. In our feedback-driven setting, the problem of generating "too many" mutants is irrelevant: only a small set of highly diverse and likely-actionable mutants is ever presented to the user.

Critically, PI Groce demonstrated that the `universalmutator` tool generated numbers of mutants and kill ratios for Java code comparable to PIT [17] and Major [64]. For falsification-driven verification, the regular-expression-based approach produced mutants of equal value to those produced by Andrews' tool [6] and Muupi [77] for C and Python, respectively. As it stands, the `universalmutator` tool is usable for real-world mutation (in fact, it is being considered for use by NASA/JPL engineers in testing the C code for upcoming CubeSat [88] missions) for languages including: C, C++, Java, Python, Swift, Rust, Go, and the Solidity smart-contract language. The `universalmutator` allows easy definition of new rules, and supports automated analysis of mutants, coverage-based pruning of mutants, and (in some languages) trivial compiler equivalence [92] checks.

This preliminary work demonstrates that multi-language source-level mutation is feasible and, indeed, effective, producing results competitive with state-of-the-art single-language mutation tools. It therefore forms a suitable platform for generating mutants to be used in this project's early phases. However, it has significant limitations for long-term utility, motivating the propsoed research below: Because the source code is not parsed, and applies the regular expressions to lines of code, not larger blocks, the technique generates many mutants that are not valid programs, and cannot be compiled, or that are trivially equivalent because they, e.g., mutate "source code" in a large comment block. Integrating mutation generation with execution is currently supported, but extending it to new languages or build systems is hard for users, requiring writing considerable Python code or complex shell scripts. It is currently impossible to define mutation operators that apply to blocks of code rather than text within a single line, and standard regular expressions are not really suited to describing code constructs such as blocks, functions, classes, or structs. Natural formatting of, e.g., an s-expression in a LISP-family language can hide opportunities for mutation, such as switching argument orders. Finally, regular expressions are only occasionally a natural notation for expressing source-level mutation; source in all languages differs from arbitrary unstructured strings.

Comby: Declarative, any-language transformation. Recent work by PI Le Goues and collaborators introduced a powerful new representation to declaratively transform richer syntactic structures in programs across multiple languages [120]. **FIXME: 2–3 sentences on how it works: what's the key magic? Words like "parser combinators" would be useful; the idea is to talk about how we get parsing without needing a parse tree.** 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). **FIXME: 2–3 sentences summarizing results (esp efficient, expressive power)** Coupling declarative syntax manipulation that goes beyond the limits of regular expressions with mutation testing promises to yield greater effectiveness by (a) targeting more sophisticated properties of programs, and (b) delivering more user-accessible tools for developing mutation transformations. Our already achieved 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 [94].

2.1.2 Proposed work: Any language mutation

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 parse a language and guarantee generation of valid programs in the source language are complex, hard-to-maintain-and-extend systems; language complexity makes such a tool for C++, for example, an extremely daunting task.

The source of all mutants to be presented to the user is the mutant generator, and in order to maximize the effectiveness of the approach, this proposal aims to allow effective generation of mutants for any programming language or DSL, with minimal additional effort. The `universalmutator` provides an initial source of mutants that satisfies this requirement for initial experimentation, but for long-term effectiveness is both inefficient and inexpressive. The current implementation avoids parsing to such an extent that it generates numerous useless mutants embedded in code comments, or that are obviously syntactically invalid. While avoiding a parser-based approach, simple additional constraints could avoid this, without adding burden on users, such as allowing the definition of a language's comment mechanisms, and not producing mutants inside comments. More generally, a mechanism for disabling mutation in contexts defined in the same way as mutation operators would handle other, even project-specific, constraints (e.g., never mutate inline assembly in C/C++). A problem with the current representation of mutation operators and such contexts is that regular expressions are currently applied only at the line level, and in any case are not effective for defining such fundamentally non-regular aspects of code as blocks and nested delimiters. A key goal in this project is to greatly enhance the expressity of transformations (e.g., operators for multi-line, context-dependent program fragments) while retaining the simplicity of specifying usual operators.

We propose to use `comby` [1, 120] 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:

```
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);
```

At a high-level, `comby` performs context-free parsing of syntactic structures that typically correspond to nested expressions and blocks in the underlying AST.[1] 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. Templates contextually matches or ignores these dimensions of a program. `Comby` is language-aware in the sense that small language 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 supports a simple extension mechanism for new languages (e.g., Solidity) or custom DSLs.[2]

We believe that 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 a 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. **FIXME: propose the LSP work to pull in type/static information. A reviewer who knows our work might say "what's the new research, here?" (indeed, a reviewer did say that about universalmutator). Sometimes it's fine to just be like, there is no new research, but we'll use this thing for this part...but given that we HAVE new research to do,**

[1] Regular expressions are not powerful enough to recognize such program terms in the general case.

[2] `https://comby.dev/#faq-language-support`

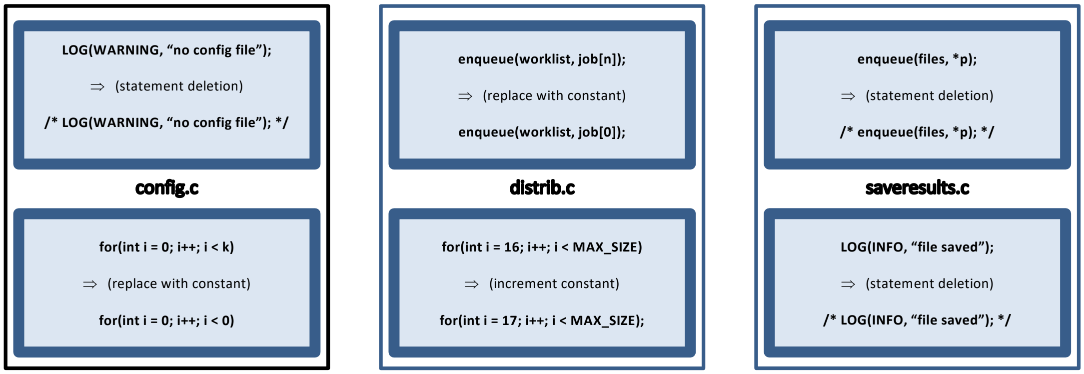

Figure 4: Which mutants are most similar? If the user marked the mutant in the upper left corner as uninteresting and added a test to kill the mutant in the upper middle, which mutant should she examine next?

it's worth proposing it.

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 [92] to make the entire feedback-driven mutation process workable. Because some distance metrics 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.

2.1.3 FPF-Based Novelty Ranking

Feedback-driven mutation testing is predicated on selecting and ranking a small set of highly interesting mutants to present to the developers. An unkilled mutant is, conceptually, very similar to a failing test. It presents information of possible relevance to a developer. The mutant or test *may* indicate the presence of a previously unknown fault that needs to be fixed, either in the SUT or in testing. It may indicate the presence of a previously unknown fault of less importance. It may also indicate an even less interesting result: an equivalent mutant or an inherently flaky test. Additionally, an unkilled mutant or failing test may contain information that is uninteresting because *it duplicates information already examined.* While examining an equivalent mutant is not always useless (e.g., it may indicate an opportunity for refactoring or improving the efficiency of code [60, 51]), examining a mutant that is equivalent to, 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. **Possible FIXME: rehash of the fact that previous work doesn't do this?**

2.1.4 Background and preliminary work

The mutant selection and prioritization problem is analogous to a similar problem in fuzz testing and triage. That is, instead of contending with a large list of undifferentiated inputs (or mutants), a user ideally preferes a few, most important, or most-critical inputs, that maximize the amount of payoff in terms of bugs found or information gained.

Furthest point first and fuzzer taming Fuzzer taming [16] was a solution PI Groce and colleagues proposed to the problem of triaging test failures in automated test generation [121]. 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 [29] 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 [29]. Preliminary work on the fuzzer taming problem using FPF-based techniques [16, 56] can be directly applied to the different problem of ranking unkilled mutants such that novel mutants are presented first.

Preliminary use of FPF-based mutant ranking. In collaboration with security analysts at Trail of Bits, PI Groce implemented a prototype version of mutant priorization, without feedback, using a manually constructed distance metric tailored to Solidity smart contracts. This was an essential step in an effort to use mutation analysis to compare three static analysis tools for smart contracts [26, 102, 117]. The comparison of three such tools over 100 random contracts from the Ethereum blockchain [13, 125] required analysis of 46,769 mutants, with 46,752 of these not killed by at least one static analysis tool. The aim of the analyis was to 1) find cases where mutation caused each tool to flag a *new* issue with the source code (thus differentially identifying kills), then 2) find cases where at least one tool flags a mutant, while others do not, and finally 3) baseline this with respect to general warning rates for the tools over the same contracts. Sorting through the surviving mutants to understand weaknesses of the tools was simply not possible without using a simple version of the FPF ranking algorithm, combined with an *ad hoc* distance metric. With even this rudimentary, untuned version of our approach, PI Groce and Trail of Bits engineers identified three significant new detectors, which were implemented and added to the Slither static analysis tool [26]. These are currently in use by Trail of Bits for security audits, and will be released to the public after tuning. Without the FPF ranking, finding these three opportunities for improving Slither would not have been feasible. To our knowledge, the use of mutation to compare and improve static analysis tools, rather than test suites, in this *differential* (both within tools and across tools) sense, is novel, and we believe that feedback-driven approaches are essential in this setting, since kill rates will obviously always be relatively low for purely static analysis without a specification. The preliminary results from the Solidity mutation analysis are available at `https://github.com/agroce/slithermutate`, with a publication to follow once the same technique has been applied to other languages.

2.1.5 Proposed work: FPF-Based Novelty Ranking FIXME

Ranking unkilled 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, it is possible to extract a large amount of information about similarity of failing tests from executing the tests, and, in fact, from executing the tests on program mutants [16, 56]. There is no obvious equivalent to "just run the test" for mutants, and in fact avoiding the expense of running the test suite on uninteresting mutants is one of the goals of feedback-driven mutation testing in the first place.

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 4 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 unlikely to be project-independent, and so a good metric may well change during feedback-driven mutation testing, in response to information from users (see Section 2.1.7 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 from the unmutated code in tests. In fact, there may need to be two distance metrics: one for selecting likely candidate mutants to execute, that uses only static information and user feedback, and one that uses dynamic results from compiling and testing mutants to refine the notion of similarity for likely-novel 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 [68], which was used in some of PI Groce's previous work [95], but in order 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 expectation that cross-project data will be less valuable than project-specific data extracted during the process itself; although there are unsupervised approaches to metric learning [103, 116], the most popular approaches require supervision.

2.1.6 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 [91, 15], 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 not be a good idea. Also, if the mutant is *not* equivalent (perhaps the user decides this kind of logging needs to be tested), then the information obtained may be high-value.

2.1.7 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 [96, 60], it may also be possible to apply mini-crowdsourcing techniques developed in the context of testing machine-learning classifiers to mutant ranking and analysis [107]. 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 developers most familiar with different aspects of the code base, as done in automated assignment of bug reports [10, 63].

2.2 Mutation-Driven-Development

Rather than a separate research focus, the idea of mutation-driven-development will inform the other research topics. In particular, once tools reach sufficient maturity, they will be used to conduct preliminary experiments in MDD as a methodology.

Fixme: this was from the introduction, and I moved it here, since it sort of ties everything together nicely. We may need one more sentence on the idea to preface it in the introduction, though. Maybe in intellectual merit? The primary focus of this project is to develop feedback-driven mutation testing. However, the ideas of Test-Driven Development (TDD) [9, 61], 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 potential weakness (and, of course, an actual goal) of TDD is that the code will be narrowly tailored to the requirements, which produce the tests, which means that missing requirements will almost always be omitted both from the tests and the code. For "shall" type behaviors [58], this is not a key problem. 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 many 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 [114, 113], perhaps using a tool like DeepState [30] 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 AFL [128], libFuzzer [105], or a symbolic execution tool [108, 109, 81] identify specific inputs. The focus of the MDD process would be on producing a test harness [36, 57] that allows an automated tool to kill interesting mutants. More radically, MDD could be implemented via a radical departure from normal TDD, with a single testing or model checking harness iteratively enhanced with assertions and checks drawn from more requirements, always requiring the ability to kill (most) mutants of the current implementation. This would not produce the usual large set of TDD tests, but instead produce a single high-powered test generator and formal specification.

2.3 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 to be addressed, some of which are cross-cutting concerns independent of any single component.

2.3.1 Feedback-Driven Mutation Testing Research Questions

1. What is a good generalized, language-agnostic mutant representation and distance metric?
2. 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?
3. How should feedback-driven mutation testing actually incorporate feedback from users into the ranking of mutants? What feedback should users be able to express, and how strongly should it be weighted?
4. Can mini-crowds be effectively leveraged to enhance the utility of user feedback?
5. Is it possible to identify outliers in otherwise similar groups of mutants, (e.g. one killed mutant in a cluser of unkilled mutants) and is such identification useful to users?
6. How can we quickly estimate whether a mutant is killable?
7. Is it possible to predict whether a mutant's unkillability is due to poor test generation (hard to reach error states), oracle weakness (unidentified error states), or actual semantic equivalence?
8. How can we most effectively use already generated killing tests and counterexamples to prune mutants?
9. 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)?

2.3.2 Any-Language Mutation Research Questions

1. What advances are required in order to maximize the efficiency and usability of a fundamentally language-agnostic approach to mutant generation?
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. Is it possible to perform on-the-fly mutant generation for very large projects, and reconcile this approach with FPF (e.g., generate new mutants with, possibly approximate, desired distances from already evaluated mutants)?

2.4 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 good test suite can also partially evaluate novelty rankings, by ranking killed, rather than unkilled mutants; the experiment even 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 bitvector similarity agrees with FPF distance metric similarity serves as a basic, if quite 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 one mutant as the seed in a fuzzer [128, 105] improves time required to kill the other mutant, on average. Unfortunately, these measures cannot effectively measure "real distance" if one of the mutants is equivalent.

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 [65, 34]. Efficiency gains (e.g., removal of invalid or equivalent-by-construction mutants) can be measured by simple, traditional measures. **FIXME: RVT, please add metrics/experiments for the type stuff?**

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 coding [**?**] 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 [127]; other researchers have demonstrated this type of methodology useful for tool pilot studies [112]. 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 or challenges to an MDD approach to programming.

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 [25], 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 undelrying research technique. 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 [24, 111]. PI Le Goues has experience in both lab studies involving think aloud protocols [127] as well as qualitative analysis generally [**?**], and will lead the design and execution of these lab studies at CMU.

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 users quickly find the most important unkilled 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, a prominent Linux kernel developer, on using mutants to improve kernel test suites, and has discussed similar efforts with Richard Hipp, the lead developer of the SQLite database, a rather 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 [30] 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 in previous work [51], and comparing the useful mutants to the highly ranked mutants: could the highly ranked mutants have motivated the key improvements to the test suites?

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, it should be possible to evaluate how much additional testing power MDD would have required at each step of development.

3 Closely-Related Work

Mutation Testing. There is a vast body of work on mutation testing or analysis, an area whose foundations date to the late 70's [18, 12]. Mutation analysis has been shown to subsume multiple coverage measures, including all the basic coverage measures [84, 90] and data flow subsumption measures [78]. Andrews et al. [7, 8] found that ease of detection of mutants was similar to that of real faults. The relationship between mutation score and test case effectiveness is sometimes empirically stronger than coverage [65]. However, Papadakis et al. [93] recently showed in a large scale study that this relationship is sometimes weak: "mutants provide good guidance for improving the fault detection of test suites, but their correlation with fault detection [is] weak." Ahmed et al. reached similar conclusions [2]. This is a foundational assumption of this proposal.

Numerous approaches seek to reduce the cost [62] of mutation analysis. Offutt and Untch [89] 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. Determining relative merits of selective mutation strategies such as operator selection and random sampling has long been an active field of research [124, 83, 130] Namin et al. [85, 86] formulated the concept of *sufficient mutation operators*, and others [118, 19] even proposed simply using statement deletion as "the" mutation operator. The subsumption of individual mutants and mutation operators is also an active area of research [32, 106, 75]. Mutation clustering[20, 110, 59] is another *do-fewer* approach where similar mutants are identified based on various properties. **FIXME: maybe 1-2 sentences on how we're different, given that we have a totally different focus in this proposal, in lieu of the paragraph that I cut (still in comments).**

Practical Mutation Analysis The above work largely focuses on computing or at least estimating the total mutation score of a test suite. The assumption is that mutation testing is meant to be, like a coverage metric, a kind of "evaluation" of a test suite, a number used to say "this is a good test suite" or at least "this is a better test suite than that test suite." In contrast, this proposal considers the problem of presenting unkilled mutations to a developer or test engineer in a way that facilitates the improvement of a test suite and the detection of faults. This is inspired by PI Groce's previous work on using mutation to find defects in formal verification and automated test generation efforts [47, 51, 3].

Very recent work by Papadakis et al. (some unpublished in a conference of journal at the time this proposal was written) has aimed, unusually, at predicting the "quality" of [91] or even prioritizing [15] mutants, to rank fault-revealing mutants highly so that users can produce tests to find faults. This work is highly relevant to this proposal's aims, but focuses 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 in the program under test, and the language support issue is dodged by targeting LLVM bitcode. Nonetheless, the goals, methods, and results of this work will inform this proposal's approach, in particular in the utility estimation aspect that balances FPF-determined novelty.

The single most relevant work by others is to be found in a report on actual techniques in use at Google for applying mutation testing to real-world projects [96]. Their approach also uses a notion of feedback, but this is manually handled and based on using a classification scheme to heuristically throw out "arid" (likely not to be actionable) mutants; due to the size of Google's code base and the integration of their approach in Google's code review process, there is also a decision to only allow one mutation (initially randomly decided) per line of code. While the underlying motivation, of giving developers actionable information rather than computing a mutation score, is similar, and the idea of using some form of "feedback" is common, this approach considered in this proposal targets the individual 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 unkilled mutant at a time. The Google approach does not attempt to prioritize the unkilled mutants it surfaces, or support custom mutation operators, or learn an individual testing effort's characteristics. It is an attempt to select mutants in an industrial scale code review setting, not an effort to propose a new way to construct or enhance test suites; e.g., it only even proposes mutants of code in a diff with a previous version of the code, which makes it completely unusable in after-the-fact testing and verification efforts. Further, it uses Google-wide coding conventions and developer suggestions to fix heuristics such as "avoid mutation of logging statements" rather than trying to learn (with human assistance) such heuristics for projects that may vary widely in language and coding style. The techniques used in this proposal may be useful in generalizing or enhancing an effort such as Google's, however, and share the focus on mutants as tools for focusing developer/tester attention and producing action on the part of humans, not computing a mutation score. Indeed, the report itself allows that the current approach does not scale, since it requires extensive manual support for each heuristic and language, a problem this proposal aims to directly address.

More broadly, a paper by the authors of the Google report and a group of academic mutation testing researchers [60], uses the Google effort to propose a notion of productive and 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. Another key difference is that, while their work used EvoSuite to enhance a test suite to kill additional mutants, the assumption was that most suite enhancement would be due to developers adding manual tests. Furthermore, we argue mutants are neither "productive" or "unproductive" in an absolute sense, but rather the value of a mutant also depends on previous mutants a developer has manually examined, and the results of that examination; this project embodies this concept in the FPF-centric workflow.

A second thrust of the efforts in this proposal is to simplify the development, maintenance, and (especially) extension of mutation testing tools by expanding the expressive power for mutation operators through domain-specific templates, and separating the generation of mutants from language or build-environment specific techniques for pruning invalid mutants. This aspect of the proposal primarily builds on PI Groce's initial work on the topic of regular-expression-based mutant generation [52] and PI Le Goues's work on declarative transformation for multiple languages [120].

Bug Triage and Distance Metrics in Software Engineering This proposal's approach is centered on the idea of computing distances between mutants. The use of distance metrics in software engineering for a variety of purposes is long-standing. Almost all such uses are essentially spectrum-based [101] (that is, using counts of coverage of code entities), except for some work in model-checking [35, 14] and some of the metrics in PI Groce's recent fuzzer taming work [16]. Reneiris and Reiss initially proposed using distance between executions to localize faults [100], and Liu and Han [76], Vangala [123], and others have followed this line with various metrics. Methods for clustering to identify bugs all rely on a distance [28]. PI Groce's previous work has used distance metrics for a variety of purposes [16, 44, 129]. Recently, PI Groce proposed a novel, causal metric for fuzzer taming and fault localization, itself based on mutation testing results [56]. Work on presenting a few well-chosen mutants is also related to PI Groce's own work on end-user testing of machine learning systems [67, 46], where the limits of human patience are a key factor. **FIXME: distance metrics in repair.**

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 compilers (including

GCC and LLVM), YAFFS2 [126] and other file systems, Google's Go compiler, a large set of Unix utilities, and a large number of Python libraries (including some of the most widely used libraries, and key scientific and numeric analysis packages). In previous work, discussions with working test engineers at Mozilla, Google, and NASA have significantly informed the PI's research efforts, and this is likely to continue. PI Groce is currently discussing plans for incorporating more advanced automated test generation into NASA's open source F Prime flight architecture [11, 87], and F Prime components, such as the auto-coder, are a likely target for evaluation efforts in this project. Such efforts are planned to result in a documented process for incorporating feedback-driven mutation testing and MDD into flight software component development. Better, cheaper, high-quality testing for small budget CubeSat [88] missions could lead to advances in various fields, especially Earth observation and space-based physics. The CubeSat initiative focuses on providing a low-cost way for educational institutions and non-profits to conduct space-based research, and thus is also related to education and outreach activities. In the long term, a mutation-driven development paradigm might result in easier development of critical software components 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 very subtle problems introduced during changes, or at least force revision of outdated specifications. With the growing impact of embedded, cyberphysical, and Internet-of-Thing systems on the physical world, this has potentially large benefits in terms of the safety and security of 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 [54] has shown that media computation is a potentially effective way to both recruit and retain female and under-represented minority students in computer science.

FIXME Clairen eeds to talk about retention/REU stuff.

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 [16]. 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 [4, 129, 42, 43] 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 [72, 70, 22, 73]. **Research Products:** Publications resulting from CCF-1217824 were numerous [43, 16, 129, 44, 42, 4, 50, 57, 48, 5, 57, 37], along with three PhD theses. Source code for software systems developed or enhanced during CCF-1217824 [38, 49] is available on GitHub.

Le Goue's most closely-related prior NSF grant is **FIXME: number**, CAREER Quality Matters: Dynamic, Static and Proactive Analyses for Automated Program Repair. **Intellectual Merit:** The results of this award have so far included novel techniques for static program repair [119], an initial exploration of diversity-enhancing dynamic repair techniques [23], and the Comby tool and associated mechanism for declarative, language-agnostic program transformation using parser parser combinators [120]. 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 two REU students, including one member of an underrepresented group in computing and another student without access to traditional research opportunities at their home institution. Sophia Kolak recently gave a well-received talk at ROSCon 2019 on her summer research; Zhen Yu Ding is the first author on a published paper on his work [23]. Both are continuing their research with the PI and have expressed plans to attend graduate school in Computer Science. 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 [**?**], 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] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. Can testedness be effectively measured? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 547–558, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi:10.1145/2950290.2950324. URL `http://doi.acm.org/10.1145/2950290.2950324`.

[3] 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.

[4] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 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`.

[5] 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.

[6] 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.

[7] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411. IEEE, 2005.

[8] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32 (8):608–624, 2006.

[9] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] 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.

[11] 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.

[12] 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.

[13] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2013.

[14] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering*, pages 73–82, 2004.

[15] Thierry Titcheu 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`.

[16] 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.

[17] Henry Coles. Pit mutation testing. `http://pittest.org/`.

[18] 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.

[19] Lin Deng, A Jefferson Offutt, and Nan Li. Empirical evaluation of the statement deletion mutation

operator. In *IEEE 6th ICST*, Luxembourg, 2013.

[20] Anna Derezińska. Toward generalization of mutant clustering results in mutation testing. In *Soft Computing in Computer and Information Science*, pages 395–407. Springer, 2015.

[21] 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.

[22] 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.

[23] 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.

[24] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 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`.

[25] K. Anders Ericsson and Herbert A. Simon. Verbal reports as data. *Psychological Review*, pages 215–251, 1980.

[26] 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.

[27] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[28] Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. Tree-based methods for classifying software failures. In *International Symposium on Software Reliability Engineering*, pages 451–462, 2004.

[29] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[30] Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.

[31] 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.

[32] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. Measuring effectiveness of mutant sets. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2016.

[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. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.

[36] Alex Groce and Martin Erwig. Finding common ground: Choose, assert, and assume. In *International Workshop on Dynamic Analysis*, pages 12–17, 2012.

[37] Alex Groce and Jervis Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

[38] Alex Groce, Yang Chen, John Regehr, Chaoqiang Zhang, and Amin Alipour. Swarmed versions of random testing tools. https://github.com/agroce/swarmed_tools, .

[39] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. Regexp based tool for mutating generic source code across numerous languages. `https://github.com/agroce/universalmutator`, .

[40] 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.

[41] 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.

[42] 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.

[43] 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`.

[44] 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.

[45] 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.

[46] Alex Groce, Todd Kulesza, Chaoqiang Zhang, Shalini Shamasunder, Margaret Burnett, Weng-Keen Wong, Simone Stumpf, Shubhomoy Das, Amber Shinsel, 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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 Proceeedings*, 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`.

[53] 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.

[54] 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`.

[55] 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`.

[56] Josie Holmes and Alex Groce. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *IEEE International Symposium on Software Reliability Engineering*, 2018.

[57] 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.

[58] Ivy Hooks. Writing good requirements. In *INCOSE International Symposium*, volume 4, pages 1247–1253. Wiley Online Library, 1994.

[59] Shamaila Hussain. Mutation clustering. Master's thesis, King's College London, Strand, London, 2008.

[60] 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.

[61] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[62] 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.

[63] 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.

[64] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE*, 2011.

[65] 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.

[66] 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.

[67] 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`.

[68] Brian Kulis. Metric learning: A survey. *Foundations & Trends in Machine Learning*, 5(4):287–364, 2012.

[69] Imre Lakatos. The role of crucial experiments in science. *Studies in History and Philosophy of Science Part A*, 4(4):309–325, 1974.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Birgitta Lindström and András Márki. On redundant mutants and strong mutation. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2015.

[76] Chao Liu and Jiawei Han. Failure proximity: a fault localization-based approach. In *Foundations of Software Engineering*, pages 46–56, 2006.

[77] Xin Liu. muupi mutation tool. https://github.com/apepkuss/muupi.

[78] Aditya P Mathur and Weichen Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[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] Elfurjani S Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.

[84] Glenford J Myers. The art of software testing. *A Willy-Interscience Pub*, 1979.

[85] Akbar Siami Namin and James H Andrews. Finding sufficient mutation operators via variable reduction. In *Workshop on Mutation Analysis*, page 5, 2006.

[86] Akbar Siami Namin, James H Andrews, and Duncan J Murdoch. Sufficient mutation operators for measuring test effectiveness. In *International Conference on Software Engineering*, pages 351–360. ACM, 2008.

[87] NASA. F prime: A flight-proven, multi-platform, open-source flight software framework. `https://github.com/nasa/fprime`, 2018.

[88] National Aeronautics and Space Administration. CubeSat launch initiative. `https://www.nasa.gov/content/about-cubesat-launch-initiative`, 2018.

[89] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[90] A Jefferson Offutt and Jeffrey M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, 1996.

[91] 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.

[92] 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.

[93] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden*, pages 537–548, 2018.

[94] 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`.

[95] 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.

[96] 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`.

[97] Karl Popper. *The Logic of Scientific Discovery*. Hutchinson, 1959.

[98] Karl Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. 1963.

[99] 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`.

[100] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, 2003.

[101] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference Held Jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 432–449, 1997.

[102] Tikhomirov S. and et al. Smartcheck: Static analysis of ethereum smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.

[103] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[104] 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`.

[105] Kostya Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity, Development (SecDev), IEEE*, pages 157–157. IEEE, 2016.

[106] Donghwan Shin and Doo-Hwan Bae. A theoretical framework for understanding mutation-based testing methods. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2016.

[107] 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.

[108] 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.

[109] 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.

[110] Joanna Strug and Barbara Strug. Machine learning approach in mutation testing. In *Testing Software and Systems*, pages 200–214. Springer, 2012.

[111] 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`.

[112] 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.

[113] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.

[114] 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.

[115] 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`.

[116] 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.

[117] 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.

[118] Roland H Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-421-8.

[119] 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`.

[120] 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.

[121] 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`.

[122] 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.

[123] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. Test case comparison and clustering using program profiles and static execution. In *ESEC/FSE*, pages 293–294, 2009.

[124] Weichen Eric Wong and Aditya P Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185 – 196, 1995. ISSN 0164-1212.

[125] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, 2014.

[126] Yaffs. Yaffs: A flash file system for embedded use. http://www.yaffs.net/.

[127] 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.

[128] Michal Zalewski. american fuzzy lop (2.35b). http://lcamtuf.coredump.cx/afl/, November 2014.

[129] 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.

[130] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *International Conference on Software Engineering*, pages 435–444, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6.