CAREER: Quality Automation: Push-Button Automatic Program Repair  
PI: Claire Le Goues

A. Project Summary

Overview. Modern software engineers have a wide variety of analyses, tools, and infrastructure available to support their QA activities, from both research and practice. However, actually fixing bugs remains a largely manual and expensive process. There simply are not enough high-quality developer hours to dedicate to fixing all defects, and developers often make mistakes in fixing bugs by hand. Research in automated program repair (APR) offers promise in mitigating these costs. To achieve true practical benefit, such techniques must (1) produce patches of consistently and assuredly high quality, and (2) integrate naturally into the QA processes and analyses that developers already use in maintaining and evolving their code.

The goal of this proposal is to make automated repair technology robust and useful enough that developers run it as naturally as they currently do compilers, linters, and other first-class QA tools. The proposed work develop new APR approaches that explicitly integrate into and then proactively improve real-world program quality.

Intellectual Merit. The proposed research will result in new program analysis and transformation approaches that automatically and assuredly improve code quality. These new approaches are designed to complement and extend existing approaches (manual and automatic) for QA and bug finding, and include:

1. Dynamic strategies that search for and then compose diverse solutions to a given bug repair problem into measurably general patches. These strategies apply to a multiple types of program repair techniques. They will target bugs identifiable by test cases, particularly those identified in continuous integration contexts, and will supplement those test cases with additional assurance evidence drawn from other sources of information about desired program behavior.

2. Static techniques that construct patches for previously undiscovered bugs that are difficult to find via testing. These techniques extend the reasoning that underlie state-of-the-art static analysis tools, increasingly applied at commit time in many organizations.

3. New ways to use automated patching to proactively and automatically improve the output and utility of existing static and dynamic analysis tools.

Broader Impacts. The proposal has the following expected Broader Impact contributions:

1. The proposed approaches will reduce the cost of software defects, an important, wide ranging, and expensive software engineering problem with broad societal implications. The PI regularly releases open-source tools and datasets to the benefit of the research community, promoting both adoption and additional study.

2. The research program is designed to support the inclusion of undergraduate researchers. It will support the PI in her role as co-director of the summer REUSE@CMU program, which introduces approximately 20 students annually to interdisciplinary software research. Many of these students represent groups traditionally underserved in STEM and lack access to traditional research opportunities at their home institutions.

3. The proposed research program will support the mentorship and training of PhD students both in the conduct of research and in the mentorship of more junior students. Lessons from this research will be incorporated into ongoing graduate-level analysis courses, as well as a planned new undergraduate course in DevOps.

4. Results on debugging and software quality will inform the PI’s ongoing mentorship of high school girls in the context of CMU’s Girls of Steel robotics program.
CAREER: Quality Automation: Push-Button Automatic Program Repair  
PI: Claire Le Goues

C. Project Description

1 Introduction

Developers have multiple tools at their disposal to help write, validate, maintain, and evolve code. Test generation [54, 111], minimization, selection, and prioritization [171] help test code to validate functionality and find bugs; static analyses can flag difficult-to-find potential problems, early in development [55]; dynamic approaches like fuzz testing can find dangerous security vulnerabilities [53, 117]; and extensive infrastructures (e.g., source control [15], continuous integration [66], or change review [19]) support tool deployment throughout the development lifecycle.

However, actually fixing bugs remains a manual and time consuming process. Unfortunately, there simply are not enough high quality developer hours available to find and fix all extant defects [16]. Moreover, developers are imperfect, sometimes committing changes that are undesirable, reverted, or simply never accepted [142, 170].

Research in automated program repair (APR) (e.g., [13, 72, 78, 80, 88, 92, 101, 106, 112, 124, 132, 144, 151, 160, 169]) holds promise for reducing software maintenance costs and improving software quality. There are two major challenges such techniques must overcome to reach mainstream applicability. First, APR techniques need to consistently produce high quality, trustworthy patches to the bugs that developers care about. This is difficult, because the search space of possible patches for even a single bug is very large [107]. More critically, reasoning about and ensuring the quality of bug-fixing patches is a challenging and unsolved problem, whether the patches are automatically-generated [92, 107, 146] or otherwise [170]. Second, APR must integrate naturally into engineer workflow for both software development and quality assurance (QA). Developers are especially sensitive the degree to which such tools integrate seamlessly into their existing workflow [75]. Tools can only help improve software quality if they are actually used; tools are only used when developers and organizations perceive that their benefits outweigh their costs [39].

The goal of this proposal is to make automated repair technology robust and useful enough that developers run it as naturally as they currently do compilers, linters, and other first-class QA tools. I propose APR approaches that explicitly integrate with state-of-the-art QA practices to generate measurably high quality bug fixes associated with evidence attesting to patch quality. I further propose to use targeted program transformation to proactively improve tool effectiveness.

Intellectual Merit

This proposal is organized into three high-level research thrusts, each of which independently contributes to scientific understanding of and practical techniques to advance software quality:

1. Develop novel test-guided APR techniques that integrate with the widely-used continuous integration loop. The proposed techniques use behavioral diversity to construct high quality patches, and draw evidence from other engineering artifacts to assure that quality.
2. Extend separation logic-based bug-finding analyses to construct verified fixes for the identified bugs. This complements the first thrust by providing a static, end-to-end process for finding and fixing real bugs that are difficult or impossible to find via traditional testing.
3. Build upon the approaches developed in the first two thrusts to proactively improve existing analysis and testing tools such that they can find and fix more bugs, faster.

Broader Impacts

Industrial and scientific impact. Software bugs cost the economy billions of dollars annually [30] and pose risks to human safety, security, and privacy. The proposed work develops practical bug-
fixing approaches that integrate naturally into the development workflow, supporting industrial uptake [61, 75]. I also release all tools and results from my research, supporting the advancement of the study of software quality generally and APR in particular.

**Research mentorship.** The proposed work will support my ongoing efforts in mentoring undergraduate research students, especially through the REUSE@CMU summer program, which I co-direct. We have a record of mentoring students from diverse and non-research-oriented backgrounds to graduate school. The proposed research will also directly support graduate student training.

**Education and Outreach.** I will develop an undergraduate course on DevOps that will integrate results from this research. I will also highlight novel results in an ongoing graduate course on program analysis, which has previously led several students to publish their first papers in the area. The lessons from this research also fit naturally into my mentorship of high school girls through the Girls of Steel, the country’s largest all-girls First Robotics Team.

## 2 Proposed Research

Sections 2.1–2.3 describe the three thrusts of the proposed research program. I present preliminary results throughout where appropriate, and proposed evaluation at the end of each research thrust.

### 2.1 Thrust 1: Dynamic, test-guided repair

Testing is the most common validation mechanism in QA practice [14]. Tests are well-understood, intuitive proxies for formal correctness specifications and are suitable for finding a wide diversity of bug types. Tests are also the most common mechanism for defining patch acceptability in automatic repair research (e.g., [13, 40, 78, 80, 88, 92, 96, 101, 113, 132, 136, 144, 151, 163, 169]), defining the bug under repair (i.e., failing tests that should pass) and functionality that should be maintained.

The problem is that tests are insufficient to fully assure patch quality, in that patches can cause tests to pass without fully generalizing to the underlying specification [146]. This is exacerbated by the fact that real-world test suites are often inadequate [126]. Techniques that use test cases as a core metric for patch correctness must therefore take particular care to ensure usable, high quality results. I provide additional background in Section 2.1.1.

Fortunately, program repair as a search problem has important properties that can be leveraged to produce higher quality patches by construction (Section 2.1.2). Moreover, tests need not provide the only source of evidence for patch quality; analysis of other development artifacts can help increase confidence in APR output (Section 2.1.3).

#### 2.1.1 Background and preliminaries

The main goal in test-guided program repair is to produce a high quality patch—set of source-level changes—such that the modified program passes a given set of tests, including one or more that initially fails (corresponding to the bug\(^1\) under repair). Automated techniques for solving this problem are typically categorized in one of two ways [94]:

1. **Heuristic- or Syntactic** techniques apply templated transformations typically to the program’s Abstract Syntax Tree (AST) to heuristically generate multiple candidate patches (e.g., [44, 80, 96, 105, 106, 166]). Each candidate is then validated against the tests. These techniques are differentiated primarily by the choices they make in traversing the syntactic search space [163].

2. **Semantics-based** techniques use the test cases as input to a form of semantic reasoning, typically built on symbolic execution, to infer symbolic constraints over buggy code regions. These inferred

\(^{1}\)I use *bug* and *defect* interchangeably in this proposal, with the intended meaning corresponding to colloquial understanding, that is, undesirable functional behavior that can be corrected via patching.
constraints inform a synthesis step that constructs code that satisfies them, often by manipulating expressions used in conditionals or assignments (e.g., [72, 87, 102, 112, 113, 131, 160, 169]). Although these families differ in underlying mechanic, both fundamentally conduct a search over a space defined by either the syntactic program modifications or the program synthesis problem. Tests define the search objective. These may suffice: There is evidence that even imperfect patches can usefully reduce bug triage and repair time [161], especially since patches (regardless of provenance), are usually independently manually validated before deployment (e.g., [1, 58, 120, 121, 168]).

High quality patches are ideal, however, since low-quality patches can increase maintainability costs over time [155]. A major challenge in APR research is how to objectively measure patch quality. To this end, with collaborators and partially in preparation for this proposal, we developed a new metric to measure patch quality in empirical evaluations [146]. Our key idea is that, just as machine learning analysts must take care to ensure that learned models do not overfit to the training data, a generated patch should not overfit to the provided tests, as measured on a held out set of tests.

We used this idea to assess relationships between several key factors (such as test suite coverage and starting program quality) and output patch quality for two representative heuristic techniques with different search traversal approaches: TrpAutoRepair [134] and GenProg [92, 166]. To control for test suite quality, we developed a dataset [95] with two known high quality test suites, based on student programs submitted as homework in an introductory programming class. The assignments came with instructor-provided evaluation tests (which we use for “training”). We used an existing automated test input generator [32] to produce a second set of tests per program, maximizing coverage over the instructor solutions. The tools attempted to repair any student program that failed at least one test in the training suite. The metric for repair quality is the percentage of the held-out tests a patched program passes.

Among other results, we found that the median patch produced by either tool, which passes 100% of the training suite by definition, passes 75.0% of the held-out suite (mean 68.7% for GenProg, 72.1% for TrpAutoRepair); Figure 1 illustrates. Figure 1 also includes student patches, since the students could submit multiple times in an effort to “repair” their assignments against the same instructor-provided tests. The difference between students and tools is not statistically significant.

Overall, tool-generated patches do overfit to the tests used for patch construction (but humans can too, arguing in favor of an objective patch quality metric). This tendency is not unique to the evaluated techniques [106, 113], nor to heuristic approaches [89]. Overfitting is also a useful metric for evaluating patch quality as produced by new techniques [78, 87].

2.1.2 Proposed work: Finding and using diverse patches

One promising avenue for test-guided APR is to take advantage of patch diversity, since APR approaches can often identify multiple patches for one bug [92]. Human-written code typically lacks sufficient diversity [83] to enable true n-version programming [37]. Automated techniques are freer of the human biases [140] that hamper manual efforts [10, 108]. We investigated this in the preliminary work by constructing simple n-version patches that “vote” on program output out of multiple patches generated for a single bug [146]. We found that n-version TrpAutoRepair patches constructed with respect to the instructor tests are slightly better than individual TrpAutoRepair patches (p < 0.001). More interestingly, GenProg patches that used the generated suites for training produced n-version patches that more substantively outperform individual patches on the instructor’s tests (p < 0.001).
This is interesting, because we have preliminary evidence that the generated suites are of generally lower quality than the instructor tests, perhaps better approximating real-world test suites [126, 136].

Especially in light of lower-quality test suites, a group of patches may better represent the desired program behavior than any individual patch. Automatically synthesizing general patches from multiple diverse patches has two components: (1) finding explicitly diverse patches (which the preliminary work did not), and (2) effectively composing diverse patches into a single solution.

**Search for diversity.** Both heuristic and semantics-based searches can be reformulated to explicitly seek a diverse patch set. An adaptive or greedy approach can modify the probabilities that constrain a syntactic search to lead it away from previously-identified solutions, or re-formulate a semantic repair synthesis query to exclude previous solutions [133]. The search space can be sliced by subsampling the tests used for patch evaluation. This has been used to improve repair search efficiency [92, 113, 124], not diversity. However, dividing the search space this way to promote the exploration of different regions is common in distributed evolutionary algorithms [138, 154], suggesting feasibility.

A patch search can also explicitly reward solutions that are different from previously-identified patches. A multi-objective search [43, 149, 175] can optimize simultaneously for patch correctness and functional diversity. Multi-objective search for program modifications has been used to simultaneously optimize for correctness and various quality attributes [29, 139]. Here, the additional objectives will promote diversity instead. It is also possible to search exclusively for behavioral novelty (ignoring tests except to identify actual fixes), a technique known as novelty search [98]. Novelty search is useful in a variety of learning tasks concerning programming [27], though has not previously been used in repair. It is especially applicable to problems with deceptive objective functions, a key issue with tests, which are notably bad at differentiating partial solutions [49].

Both multi-objective and novelty search require a metric characterizing how far a new potential solution is from the rest of the population in the problem’s behavior space so as to reward exploration of new regions of the search space. Measurements of execution diversity (over slices, for example, or low level behavioral signals [77]) are likely to be more effective than behavior on test cases alone. Syntactic difference may serve as useful approximations, measured, e.g., by differencing tools that operate on abstract syntax trees (ASTs) [48, 52], or other lightweight AST similarity metrics [71]. These approaches are efficient on small code snippets, like most bug-fixing patches [174]. Semantics-based APR lends itself to additional diversity metrics based on the underlying synthesis mechanic, like (1) model counting (c.f., [157]) to estimate the degree to which two solutions disagree based on the number of models that satisfy a specially-constructed formula, or (2) output coverage, or how much of the desired output behavior (expressed in the inferred constraints) a solution covers. Collaborators and I recently demonstrated that such metrics can help select between test suite-equivalent patches produced by synthesis [87], suggesting that they importantly capture semantic diversity.

**Composing and using diverse patches.** A diverse pool of candidate repairs for a given bug may be useful on its own: Developers report a desire for tools that suggest multiple fixes to help illustrate a potential problem [74, 75]. The patches can also be composed. I will go beyond the simple patch composition mechanism from the preliminary work to synthesize single general patches from multiple patches. This can work either by identifying behaviors differentiating alternative patches (by comparing execution behaviors [118], or statically predicted execution behavior [31]). Or, it may be more effective to synthesize new patches that abstractly capture behavioral features multiple patches have in common, inspired by techniques that synthesize systematic edits from similar (human-provided, rather than generated) patches [114]. I will experiment accordingly.

2.1.3 **Proposed work: Generating evidence of patch quality**

Better patches by construction are an important step towards practical, usable dynamic APR. However, given the limitations of tests, APR must also augment produced patches with stronger,
orthogonal evidence regarding patch correctness. This can also help rank and select between diverse alternative solutions, especially if patch generalization fails. One promising and understudied way to do this is to draw on evidence from other artifacts associated with software development:

**Documentation.** Developers produce significant prose documentation to support software maintenance. Connecting natural language to concrete code is difficult generally. However, there has been significant recent success in using the structured prose available in javadoc comments for testing [167] by automatically generating unit tests [150], or creating oracles for automatically-generated tests [56]. Integrating these methods into APR can (A) find new bugs to fix, providing initially failing tests (reducing the burden on a developer to provide one, assumed by current approaches), and (B) provide new tests to help validate produced patches.

**History.** Source control and continuous integration histories are rich sources of development information. In both my own and others’ work, source control histories have provided useful information to guide APR [80, 88, 106]. This history can also inform human-understandable evidence of patch quality, drawing on past history to argue for similarity (based on similarity metrics as above), like “patch A is more similar to previous patches to this module than alternative patch B.” This may benefit especially from considering continuous integration history, which contains smaller, more granular commits, and can relate changes directly to test case behavior. Continuous integration is an excellent point at which to APR integrate directly into everyday development, i.e., in response to a regression test failure, when automated assistance may be most profitable [18, 75].

### 2.1.4 Proposed evaluation

I will evaluate the new search strategies for identifying diverse patches by measuring search efficiency, and diversity of identified solutions. I will tune and compare alternative search strategies and diversity metrics in a controlled setting, varying only methods and parameters as appropriate. This works for both heuristic- and semantics-based approaches. For example, my recent work on S3 [87] and JFix [86], explicitly supports integrating new search metrics into semantics-based repair. These experiments may also support principled comparison of metric applicability between heuristic and semantics-based approaches, enhancing understanding about the different repair families. I will evaluate patch generalization mechanism by adapting metrics (like precision and recall) from prior work on systematic refactoring [114]. I will also use the overfitting metric to evaluate generalized patch quality, using individual patches and the simple n-version voting approach as baselines.

I will integrate these innovations into new dynamic APR, evaluating on real programs (following my prior work, e.g., [88, 92, 95]). Two success metrics are expressiveness (variety of bugs and programs repaired) and scalability (size of repaired programs). However, patch quality is the core concern. Measuring overfitting, an important objective quality metric, requires buggy programs with multiple test suites. Small, well-specified programs [95, 152] allow for controlled experimentation. I will support claims of generalizability by producing held-out tests for real programs either by sampling real-world test suites or automatically generating tests for well-understood programs [76].

To evaluate the new evidence proposed in Section 2.1.3, I will measure the agreement of the produced evidence with held-out tests as an objective measure of functional patch quality. This work, as well as the novel APR techniques, may additionally benefit from human studies. Recent work has demonstrated how to engage expert humans in patch generation and assessment [26] (and released materials to support such studies). I also have recent experience conducting human studies involving talk-aloud protocols to study human debugging sessions [41]. I may use a similar protocol to help evaluate tool utility, in addition to simpler, lighter-weight surveys, like those used in prior APR work to access human assessments of patch quality [50, 80, 155, 161].
2.2 Thrust 2: Static repair of previously unknown bugs

Tests are only suitable for finding and guiding the repair of certain kinds of bugs. For example, consider the simplified code snippet from the error-prone project² shown in Figure 2a. The call to `resolveMethod` on line 3 can return NULL, leading (before the addition of the call on line 5) to a null pointer exception on line 6. A developer committed a fix (with a test) inserting a call to `checkGuardedBy`, a custom error handler that throws an `IllegalGuardedBy` exception if the passed boolean condition is false. Unfortunately, tests usually cannot identify recurring bugs: the very same mistake had been made 10 lines later in the same switch statement, but was not fixed for another 18 months. Other bug types are difficult to test for in a finite, deterministic way [123], like concurrency errors or resource or memory leaks. Figure 2b shows an example memory leak from the Swoole³ project (line 7), which may be fixed by adding a call to the project-specific resource allocation wrapper `sw_free` (Figure 2c).

These types of bugs motivate the use of QA techniques beyond testing. Companies like Ebay [70], Google [25], Facebook [34], and others are publicizing their development and use of static analysis tools in engineering practice. Considerable recent progress has been made in expressive, high quality static analyses that can find real bugs like these examples in real programs, cost effectively [5, 25]. Some static tools even provide “quick fix” suggestions for identified bugs, which developers can find useful [75]. However, these suggestions are generic, and provide no correctness guarantees.

I propose to fully integrate automated program repair into the static analysis workflow, extending recently-developed static analyses (Section 2.2.1) that target different bugs than I considered in Section 2.1. This enables program-specific bug repair that integrates with existing static tooling to produce end-to-end identification and repair of previously undiscovered bugs. The patches will be verified, constructed in a sound way, and accompanied by precise guarantees.

I build on a static analysis that abstracts a program to an intermediate language, and then symbolically interprets it to find paths that may lead to particular property violations (like null pointer dereferences). Patching such a bug involves finding a transformation that leads that program to not enter the violating state. This requires (1) A bug class-specific fix effect that specifies how to abstractly avoid the fault state (Section 2.2.2), (2) A specification of an abstract repair to induce that effect, and a way to satisfy it (Section 2.2.3), and (3) A way to convert abstract repairs into validated concrete syntactic patches (Section 2.2.4).

2.2.1 Background and preliminaries

In the interest of concreteness, I propose to build my approach on top of the analysis engine used in Infer [33], an open source [5] framework that uses Separation Logic and Hoare-style reasoning to

²error-prone is a static analysis tool, https://github.com/google/error-prone
³Swoole is a popular event-driven networking engine for PHP, https://github.com/swoole/swoole-src
scally find bugs in real-world programs. Infer works by abstracting a source program in one of several languages (e.g., Java, C, C++) to the Smallfoot Intermediate Language (SIL) [22], an intermediate analysis language. A SIL program $P$ is a set of procedures [23, 24, 35], each of which consists of a series of commands. SIL commands can be decorated with pre- and postcondition assertions expressed in Separation Logic. These assertions describe the commands’ effects, i.e., commands generate assertions over symbolic heaps. Symbolic heaps serve as the abstract domain for detecting faulting conditions, like memory leaks, and follow a fairly standard storage model [35, 130] (though note that my notation will sometimes simplify). The assertion language encodes heap facts using points-to heap predicates over program and logical variables. Heap predicates are “separate”, disjoint sub-heaps, with separation denoted by the separating conjunction $\ast$ (read “and separately”). Pure boolean predicates assert conditional facts over heap predicates, and describe the effects of statements on the heap.

Given this program model, the analysis uses local reasoning [127, 129] to efficiently infer specifications by summarizing the effects of individual commands and composing them into procedure-level specifications [35]. This is possible because many commands only affect a sub-part of the heap, which can be modeled by the separating conjunction. For example, $\text{hmap} = \text{sw_malloc}()$ is modeled as affecting only $\text{hmap}$. The unchanged part of the heap for a command is its frame; the parts of the heap a command changes is its footprint [23, 46, 125].

$$\frac{\{P\} C \{Q\}}{\{P \ast F\} C \{Q \ast F\}} \text{ Frame Rule}$$

Formally, the Frame Rule (left) codifies how to reason on local behavior. It allows analysis of a command $C$ with a specification $\{P\} C \{Q\}$ and a heap state $H$ to proceed, without considering unaffected parts of $H$ (the frame $F$), if it can be separated into parts $\{P \ast F\}$. Frame inference [23, 35, 45, 128]), automatically infers the frame $F$ that allows the frame rule to fire,\footnote{The analysis also infers anti-frames, or missing parts of the heap state, which increases precision.} via a proof system of subtraction and normalization rules [23].

Given this inference procedure, the analysis finds bugs by symbolically executing SIL commands over symbolic heaps, according to a set of operational rules [23], seeking paths with symbolic heaps that violate specified properties. Infer currently detects various heap-related bugs [3]: resource and memory leaks, null dereferences, and, experimentally [4], buffer overflows, thread safety, and taint-style information flow bugs. To illustrate, Infer discovers the memory leak in Figure 2b by identifying the path through line 7 where $\text{hmap}$ becomes dead. The interpreter enters a special fault state when it discovers an error, denoted formally as $C, \sigma \leadsto \text{fault}$: the interpretation step $\leadsto$ for command $C$ at location $\ell$ in symbolic state $\sigma$ results in a fault. $\text{hmap}$ is still allocated in the symbolic heap (denoted by the predicate $\text{allocated}$, or $\{\text{hmap} \Rightarrow \text{allocated}\}$) at $\ell = 7$ when it becomes dead: $\text{return } \ell, \{\text{hmap} \Rightarrow \text{allocated}\} \leadsto \text{fault}$.\footnote{I consider only the predicate on $\text{hmap}$, ignoring other parts of the state, for simplicity.}

At this point, I propose to take over to construct repairs.

\subsection{Proposed work: Fix effect specification and inference}

In this abstract domain, a bug fix corresponds to a program transformation that leads to a fault avoiding interpretation, i.e., where the fault state is not triggered. The first step in statically finding such a transformation is to define precisely what it means for a bug to be fixed, or the desired fix effect. One way to specify this is as two singleton heaps: $F'$, containing a fixable abstract predicate, and a corresponding $F''$, mapping to a fixed predicate. $F, F''$ expresses the desired symbolic transition on the abstract predicates. Note that fix effects are generic to entire bug classes. For example, $F = \{\text{pvar} \Rightarrow \text{allocated}\}, F' = \{\text{pvar} \Rightarrow \text{freed}\}$, specifies a generic fix effect for memory leaks (with $\text{pvar}$ serving as a placeholder symbolic variable). By reasoning over the abstract domain of what the code does, the proposed approach is multi-language and resilient to syntactic customizations (like wrapper functions).
As a starting point, it may be reasonable to manually specify appropriate fix effects corresponding to a given bug class. This can be done once whenever a new analysis is added to the tool, and then applied over multiple code bases. However, semantically, some bugs may be correctly fixed in various ways [104] (e.g., a null dereference can be repaired by initializing the object, wrapping a dereference, or throwing an exception when an object is null). I therefore propose to also formally derive fix effects from reported violations, compared to code paths along which violations are possible (F is observed) but not reported. The fix effects discussed so far correspond to relatively simple specifications and temporal properties, with a rich history of both static and dynamic inference research to build on [9, 97, 99, 164]. More complex fix effects may also be composable from constituent subparts [85]. This generally difficult problem is considerably simplified here, because the faulting condition under repair constrains the desired specification dramatically, and static analysis of a transformed program can validate whether the inferred effect ensures a fault-avoiding interpretation.

2.2.3 Proposed work: Specifying and constructing repairs

The fix effect for a bug class will inform an abstract repair specification that is then used to synthesize or find an appropriate bug-fixing transformation. My main insight is that frame inference can be used in a novel way to efficiently recognize and characterize abstract repairs.

I will first give the intuition in terms of specifying additive repairs, which insert missing statements. Such repairs may be particularly useful for the bugs discussed so far in this exposition (i.e., a memory leak is often caused by the lack of a freeing operation) [7, 165]. A specification for an additive repair should ask, effectively, “Does there exist a program fragment ?C_R that can be inserted at some point such that the fault state is not triggered?” This can be specified more formally as a Hoare triple containing a “hole” for an instruction C that induces the desired fix effect under the standard partial correctness interpretation: \( \{ F \} ?C_R \{ F' \} \). A satisfying C_R would be one whose only effect on the heap is to induce the desired fix effect (like the C library call free for a memory leak). However, it may be impossible to find or construct such “ideal” repairs; or, the best repair might have beneficial side effects (like sw_free, which performs cleanup and logging). This motivates a relaxed repair specification \( \{ F * P \} ?C \{ F' * Q \} \), where P and Q may be be nonempty.

The frame inference procedure FRAME can now be used in a novel way to check whether a piece of code C satisfies this specification. That is, if frame inference succeeds when applied to C’s footprint precondition \( \{ S_P \} \) (inferred by the analysis, Section 2.2.1) with respect to the fix effect precondition F (and, respectively, postconditions \( \{ S_Q \} \) and \( F' \)), C is a candidate repair. This is summarized in the entailments to the right. Importantly, successful frame inference also pulls out P (resp. Q), precisely capturing any side effects of the candidate repair beyond the desired fix effects. This may simplify inspection, which is complicated by tangled changes [64]. The overall approach is sound and decidable by the frame inference procedure [23].

It is also possible to specify repairs that allow for modification beyond insertion. Reasoning about deletion is straightforward, via a specification that asks whether the “opposite” of a command leads to a fault-avoiding interpretation. Inserting control-flow that “wraps” existing commands is more complicated, since a command’s inferred pre- and postconditions change in the presence of conditionals. The analysis can be extended to reason about whether a pre-condition matches a repair specification iff the condition check applies, by carrying additional information throughout inference. Inducing multi-stage fix effects involves identifying sequences of transformations that induce those effects. Fortunately, the reasoning mechanism is naturally compositional, and need reason only over the desired change in behavior, important to APR scalability [92, 93, 113].
There are also multiple ways to satisfy repair specifications. One promising approach is to search over existing program fragments [78, 82, 88, 107], seeking commands that match a repair specification from elsewhere in the same program (which, among other benefits, takes advantage of program-specific protocols and customizations). Other options include synthesis [82] or syntactic mutation [92]. Synthesis may be particularly useful in finding more complex patches, via, e.g., extracting templates from the existing code (which we have demonstrated for dynamic repair [87], and which might be especially applicable to SIL, with its well-specified terms and restricted syntactic forms).

2.2.4 Proposed work: from logic to syntax

A repair in the abstract domain must be translated to a syntactic patch to the source program. Converting to concrete syntax, absent variable names, is straightforward (commands in the IL all map directly to source). However, instantiation involves two additional concerns:

**Location.** Some bug-fixing transformations have the same semantic effect at multiple locations (e.g., `sv_free` may be placed at either lines 6 or 7 in Figure 2b). In general, bug class bears on repair location choice e.g., developers might typically expect a change immediately before a null dereference; for resource leaks, they might expect a change immediately after the resource’s last use. The symbolic interpreter provides a location ℓ where it enters the fault state, a reasonable initial instantiation point. Beyond heuristic convention, I will infer where and how far from defects to instantiate repairs based on other places in the code where fault-avoiding interpretations hold. This may allow patch placement to better conform to stylistic conventions and human judgement.

**Instantiation and selection.** Symbolic program variables in the IL must be renamed to match the enclosing syntactic scope. The fault-inducing symbolic variable can be mapped to the fault-inducing variable straightforwardly based on the repair specification and matching. Patch code may also reference other variables, which must either already exist (a requirement that may help filter a large solution space), or be mapped and renamed to in-scope variables. Second, as in dynamic APR, bugs may have multiple candidate fixes, e.g., memory can be freed by the standard C `free` call, or the `sv_free` wrapper function. One approach to select between them is to prefer smaller patches, syntactically [112] or semantically [87]. The initial prototype (Section 2.2.5) prefers closer matches in terms of extra effects, and between a candidate repair’s variable and the fault-inducing variable. This prefers `sv_free` over `free`, because `hmap` and `map` have the same type (`swHashMap *`). Variable name overlap or other similarity measures may also help choose between patches.

2.2.5 Preliminary results

To demonstrate feasibility in preparation for this proposal, a PhD student and I implemented an initial, constrained prototype of the proposed technique, extending Infer [5]. It works off-the-shelf on multiple languages, using manually-specified fix effects to target resource leaks, memory leaks, and null dereferences. It constructs additive repairs by searching for patch code from elsewhere in the same program. Table 1 summarizes results on a convenience sample of 5 C programs and 3 Java programs, randomly sampled from the top thousand most popular C and Java repositories on GitHub. “Bugs” indicates the number of bugs Infer detected of the given type. “Time” shows total time required to both find and patch all bugs of all considered bug types. We identify “Fixes” according to whether the patch addresses a true positive bug report. Overall, the prototype fixes 15 resource leaks, 8 memory leaks, and 27 null dereferences. Rerunning the analyzer validated that the patches ensure fault-avoiding interpretations. We checked that the patches did not cause test suite regressions in Apk-tool and error-prone.
In repairing the motivating example bugs, the prototype illustrates the importance of a compositional, interprocedural approach: Both fixes involve interprocedural reasoning, and identifying the `checkGuardedBy` repair (Figure 2a) directly requires compositionality. Finally, a particularly exciting aspect of the proposed approach is that it can fix previously-unknown bugs (†) without developer intervention, providing end-to-end, integrated APR. Pull requests based on these preliminary results have been accepted into Swoole, an active, popular GitHub project. Overall, even with a limited scope, the approach efficiently fixes a number of real bugs, while leaving considerable room for improvement through the proposed research.

2.2.6 Proposed evaluation

I will evaluate fix effect inference by checking correctness of inferred effects with respect to a ground truth set developed from bug fixes in real programs. Repair specification and construction can progress independently by beginning with an initial set of manually-specified fix effects. The key metric for that work is whether the produced repairs do in fact produce a fault-avoiding interpretation according to the analyzer, and how efficiently such repairs can be identified. I will evaluate the work on identifying repair locations by comparing proposed locations to those modified by a human, either on a historical dataset of patches or in a human study setting. I will evaluate patch selection and instantiation independently by measuring how well the approaches filter the search space, and produce suitable patches that compile in the context of an overall APR system.

My true goal is to soundly repair bugs that are difficult to test. I will evaluate the overall approach on open-source programs in several languages, measuring time to repair bugs and the quantity and variety of bug types repaired. I will focus especially on fixing difficult-to-test bugs. It may be possible, for instance, to build synthetic datasets of recurring bugs from prior work on systematic refactoring [114]. The defined fix effects constrain the types of bugs that can be repaired. I started with Infer’s default bug types, which are common in practice [33, 34]. I expect the approach to generalize naturally to, e.g., information flow bugs. Separation Logic may also admit repair of non-heap-based bugs (like concurrency errors). I leave this possibility open, but will begin by focusing on the heap. Unfortunately, given the targeted bugs, it will be difficult to compare to other APR techniques. I will mitigate this risk by focusing on real programs, building ground truth on bugs that humans previously repaired, in addition to previously-undiscovered bugs.

Elements of this work will benefit from human studies (similar to those in Section 2.1.4), especially to assess patch acceptability and the utility of the precise guarantees they provide. I will also continue to submit patches for newly discovered defects to open source systems. Regardless of whether this features in a principled evaluation, the feedback will help ground my focus, and ensure practical utility.

2.3 Thrust 3: Better QA tools using automated transformation

Automated analysis tools importantly supplement testing and inspection in QA practice. Any such analysis, dynamic or static, must approximate [103], both theoretically and in the interest of practicality [18]. The problem is that the choices that such tools make to help triage or improve output are, broadly, generic, and the gaps between program reality (i.e., custom protocols or quality

<table>
<thead>
<tr>
<th>Project</th>
<th>Lang</th>
<th>kLOC</th>
<th>Time (s)</th>
<th>Type</th>
<th>Bugs</th>
<th>Fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swoole</td>
<td>C</td>
<td>44.5</td>
<td>56</td>
<td>RL†</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ML†</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>dablooms</td>
<td>C</td>
<td>1.2</td>
<td>9</td>
<td>RL†</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ML†</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>php-cp</td>
<td>C</td>
<td>9.0</td>
<td>20</td>
<td>RL†</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ML</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Apktool</td>
<td>Java</td>
<td>15.0</td>
<td>582</td>
<td>RL†</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ML†</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>sysstat</td>
<td>C</td>
<td>24.9</td>
<td>28</td>
<td>RL†</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rappel</td>
<td>C</td>
<td>2.1</td>
<td>7</td>
<td>ML†</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>error-prone</td>
<td>Java</td>
<td>149.0</td>
<td>262</td>
<td>ND</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>jfreechart</td>
<td>Java</td>
<td>282.7</td>
<td>1,268</td>
<td>ND</td>
<td>53</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 1: Repair results. † denotes fixes for previously unknown bugs. RL denotes resource leaks; ML, memory leaks; ND, null dereferences.
concerns [70]) and tool assumptions can lead to unhelpful, overwhelming, or incorrect output. This can be mitigated by tool customization, but existing customization is often insufficiently granular to be useful [75]. I provide more background in Section 2.3.1. My insight is that program transformation, built partially on the work proposed above, can automatically customize static and dynamic analyses to improve triage and tool accuracy (Section 2.3.2).

## 2.3.1 Background and preliminaries

<table>
<thead>
<tr>
<th>Program</th>
<th>Bug</th>
<th>Total</th>
<th>Reported</th>
</tr>
</thead>
<tbody>
<tr>
<td>libdwarf</td>
<td>1</td>
<td>9.17M</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.75M</td>
<td>314</td>
</tr>
<tr>
<td>w3m</td>
<td>1</td>
<td>10.8M</td>
<td>455</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13.4M</td>
<td>428</td>
</tr>
<tr>
<td>SQLite</td>
<td>1</td>
<td>3.09M</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.04M</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.07M</td>
<td>490</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7.27M</td>
<td>330</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.22M</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>3.44M</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6.71M</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4.15M</td>
<td>236</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>4.64M</td>
<td>389</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2.38M</td>
<td>270</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0.45M</td>
<td>108</td>
</tr>
</tbody>
</table>

Table 2: AFL-identified crashes.

**Triage.** In this context, *tria*ge refers to the process by which developers assess, assign, and respond to reports generated by an automated QA tool [16]. Developers must assess reported bug criticality and likelihood (i.e., the possibility that the report is a false positive), among other concerns. This problem applies to both static and dynamic approaches. Consider fuzz testing [117], a dynamic approach widely used in security practice [57, 67, 119] that generates inputs that crash a tested program. Because many inputs can trigger any given bug fuzzers can produce many inputs for the security analyst to inspect. To illustrate, Table 2 (next page) shows the results of running American Fuzzy Lop, an effective, sophisticated modern fuzzer [172], on three programs with real vulnerabilities. The SQLite bugs are based on a previously-developed corpus [173]; the others are drawn from a manual assessment of vulnerability reports. “Total Crashes” show the raw number of crashes collected over an indicative fuzzing session; these are in the millions. The “Reported crashes” are those that AFL presents to the user. Each row corresponds to a single bug by construction, but no bug has fewer than 31 “unique” crashes reported, and most have many more.

**Bugs masking other bugs.** The presence of defects can prevent analysis from finding or clearly explaining others that appear later in the same path or module. Compilers have this problem, motivating simple but imperfect error correcting strategies [11]. This problem also applies to more complex semantic analyses. Consider the code snippet in Figure 3, where a default switch case leaks two variables (col and col->name, line 9). Infer only reports the leak of col->name, after which its analysis short circuits. This is not merely an implementation detail: is is difficult to continue reasoning under violating conditions. After the static repair prototype (Section 2.2.5) inserts sw_free(col->name) on line 8, Infer reports the second leak. We observed similar behavior in fuzzing, as I explain in Section 2.3.2.

```c
1     swTableColumn *col = sw_malloc(sizeof(swTableCol));
2     sw_malloc(sizeof(swTableCol));
3     col->name = swString_dup(name, len);
4     switch(type) {
5         case SW_TABLE_FLOAT:
6             ...
7             default:
8                 sw_free(col->name);
9                 return SW_ERR;
```

Figure 3: Both col and col->name can be leaked on line 9, but Infer can only report one by default.

## 2.3.2 Proposed work: improving analysis report triage and effectiveness

One simple, intuitive definition of a bug is “a program behavior that should change.” My intuition is that there is much to learn about a behavior by whether and how it can be changed:

**Reduce static false positives.** A potential bug whose behavior cannot easily be changed may indicate a false positive. Manual inspection of Infer’s reports from Section 2.2.5 indicates that its false positives generally arose when it failed to analyze clean up functions due to a time out.
Automated patching may provide important clues about when such a time out is problematic: The rate of successful patch construction for bugs identified by false positive reports is notably lower than for true positive bugs (not shown). The analysis can respond to this indication of potential false positives by adaptively extending its own timeout, without having to do so over the entire program or requiring developer intervention.

**De-duplicate fuzz tests.** Alternatively, if two crashes change in the same way in response to a program modification, they may be caused by the same bug. To substantiate this, a Masters’ student and I implemented a preliminary approach atop AFL that analyzes a crashing input, selects a patch template based on that analysis, transforms the program with that template, and reruns the crashing input. We “bucket” crashing inputs based on which patch leads it to run without crashing. We preliminarily target a narrow class of crashes identified by the Common Weakness Enumeration standard [42]: CWE-476 (null pointer dereferences), and CWE-120 (Buffer copies without checking size of input) attributable to memory-manipulating library functions. The approach correctly clusters all but one of the AFL-reported crashes in Table 2. These results suggest that automated patching is a promising approach for de-duplication in automated fuzzing, which we can extend to additional CWEs and transformation types. Templates from dynamic APR [80] or inferable from source control can augment the set of considered mitigations.

**Find more bugs.** In Section 2.3.1, I showed one example where automated patching helped Infer find a new bug. Automatically patching identified faults may allow for a robust, integrated analysis-repair approach that finds more bugs than the analysis alone. This is also promising for fuzzing: For the single unclustered input discussed above (from SQLite bug 1) the patch allowed the input to progress further, finding a new crash later in the same execution. I will integrate patching into an existing fuzzer to allow patch-guided path exploration, specifically exploring paths enabled by a particular mitigation that would otherwise be ignored.

### 2.3.3 Proposed evaluation

The goal of this thrust is to improve static and dynamic analyses such that they find more bugs and produce more accurate output. I will evaluate these approaches using standard metrics for analysis success: efficiency (time), accuracy, false positives, false negatives, and deduplication success and accuracy. Estimates of developer time saved may also serve as an informative metric, especially for the fuzz testing work. Note that the investigated transformations need not always correspond to actual developer patches for the approaches to succeed, and thus I will not treat quality as a first order concern in these experiments. If time and student interest permits, however, we may engage in a tool or usability study of integrated approaches.

### 3 Closely Related Work

Work in automatic program repair over the past decade predominantly uses tests, rather than other sources of quality evidence, to validate correctness [44, 78, 88, 96, 105, 106, 112, 113, 124, 134, 163, 166, 169]. Considerable progress has been made in achieving scalability and expressivity; the dominant open issue is how to achieve and assure produced patch quality. To date, quality has primarily been promoted by restricting or modifying the APR mutation strategies [105, 112, 153]. Thrust 1 draws inspiration from Search-Based Software Engineering (SBSE) [62], which has also been used for many software problems besides repair [12, 20, 84, 115, 141, 159]. Dynamic techniques targeting particular defect types [13, 28, 36, 40, 100, 101, 143, 145] relate more closely to the proposed work in Thrust 2 (targeting particular bug classes), addressing different, dynamically-detectable bug classes.

Generally, existing static repair techniques target domain-specific or fully specified languages, and are not applied to real-world programs. Previous static repair approaches have reasoned
about formally-specified properties via LTL specifications [73, 158], SAT approaches [59], deductive synthesis [82], contracts [104], and model checking [60, 131, 137]. The most closely related work [104] considers a different abstract domain, and does not instantiate or apply patches automatically.

Considerable research exists to improve the output of static and dynamic analysis, including by reducing false positives, or ranking and deduplicating bug reports or fuzz test crashes [17, 38, 51, 68, 69, 172]. To the best of my knowledge, proactive repair for deduplication has not been explored. Compilers use simple repair strategies to find multiple errors when parsing [11]. Similarly, Quick Fix Scout [122], uses program manipulation to recommend/reorder Eclipse Quick Fix suggestions for compiler errors. The proposed work is novel in its targeting of dynamic and semantic errors.

4 Broader Impacts
This section describes the expected broader impacts of the propose work.

4.1 Industrial and scientific impacts
The goal of this proposal is high quality, developer-oriented automatic program repair, which has critical societal and economic implications [30, 126]. Industry practice is increasingly receptive automated repair approaches [61]. I periodically speak at developer conferences [90, 91], and the proposed work will feature in such talks.

Research interest in the APR subfield continues to grow [81, 162]. I have an established history of publicly releasing and supporting all research tools and data (e.g., [2, 6, 8, 116]). My released materials have been used by others to support analysis of previous results and theoretical foundations of APR (e.g., [109, 110, 136]) and fault localization [135]; replication and extension (e.g., [47, 80]); and comparative evaluations of new techniques (e.g., [80, 105, 106, 113, 124, 134, 136, 151]).

4.2 Research mentorship
Undergraduate research mentorship. I am fundamentally committed to widening the pipeline of students interested in and equipped to pursue a research career. I am co-PI of the REU site, “REU Site: Interdisciplinary Software Engineering” (CCF-1560137), and co-director of the REUSE@CMU summer program. We train students in all elements of research, including understanding literature, formulating research questions, developing novel solutions, and evaluating and communicating results. We specifically seek students representing underserved demographic groups, early in their undergraduate careers, and at schools without traditional research opportunities. The 2016 cohort consisted of 18 students, including 11 female students and 4 from under-represented ethnic minority groups. Four of the eight rising seniors applied to and were accepted at top-tier PhD programs. The 2017 cohort consists of 25 students, including 12 women and 7 from under-represented ethnic minority groups; 16 study at small liberal art or regional state schools. We support a large summer cohort by supplementing the site grant with other funding as available; I plan to work with CMU’s philanthropic outreach to ultimately secure more permanent funding.

The proposed research integrates directly with these activities, especially because debugging is an accessible domain for students with little experience. For example, one of my responsibilities as co-director is to contribute to the program’s weekly summer seminar series. My research features prominently in such lectures, e.g., as an illustrative example on giving conference talks (based on ref [78]) and as the subject of a “keynote-style” talk. I also directly work and publish with undergraduate students, and encourage them to consider graduate school. A representative recent sample includes David Widder (UOregon ’17), joining Carnegie Mellon’s PhD program this Fall;

---

6http://reuse.cs.cmu.edu
Ashley Chen (Simmons ’19) and Liamm Schramm (Bard ’18), who have both already submitted short papers this summer; and Marcos Hernandez (SUNY Potsdam ’19), who has returned to CMU for a second year to continue his work on testing robotics systems.

Undergraduate researchers could contribute to multiple areas of the proposed work. Most of the diversity-search research (Section 2.1) can be decomposed such that an undergraduate can implement and compare various approaches (last summer, Joanna Wang, Dickinson ’17, did conceptually similar work in the context of self-adaptive systems). The work in Thrust 3 (Section 2.3) is also amenable to undergraduate assistance (e.g., a Masters student performed the preliminary fuzz testing work).

**Graduate research mentorship.** The proposed research is intended to be conducted in active collaboration with PhD students (Section 6). For example, the preliminary results in Section 2.2 were produced by one of my PhD students as part of initial work towards his thesis. I supervise six Carnegie Mellon graduate students, including two women. Note that I closely supervise graduate students working with undergraduates to ensure both that the undergraduate is supervised appropriately, and that the graduate student receives guidance and training in how to mentor junior researchers.

### 4.3 Education and outreach

**Undergraduate education.** As a member of the SCS Undergraduate Review Committee and director of the Undergraduate Minor in Software Engineering, I am well-positioned to influence CMU’s undergraduate SE curriculum. I plan to develop an upper-level undergraduate class on DevOps [21], focusing on tools and analysis to perform QA for large, rapidly-evolving systems. That course, along with the upper-level course I teach that is required for the minor (15-313: Foundations of Software Engineering), offer venues for recruiting new undergraduate researchers and disseminating results.

**Graduate education.** CMU’s Masters of Software Engineering (MSE) cohort of experienced industry developers provides a valuable source of practical insight and engaged subjects for human studies. I will continue to integrate salient results (to which the students typically respond positively) into the required course on QA that I regularly teach in that program (17-654: Analysis of Software Artifacts). At the PhD level, I motivate and illustrate the research challenges in developing and evaluating analysis tools (e.g., [63]) by including them in assignments. This includes a homework assignment about GenProg that has been disseminated to colleagues.7 I will update readings and produce new such assignments based on the results from this research, to be assigned in 17-808: Introduction to Software Engineering Research (which I co-teach with other SE faculty), and in 15-819O: Program Analysis. My goal in the latter course is to teach analysis fundamentals to PhD students from a diversity of research backgrounds. I aim to equip them to apply and extend modern analysis techniques in their own research. The course has led several students to publish their first papers in the area (e.g., [79, 147, 156]).

**High school outreach.** A significant factor in recruiting and retaining women and members of underrepresented groups to STEM fields is the existence of relatable mentors and role models [65]. To this end, I serve as a volunteer programming mentor for the Girls of Steel (GoS), the largest all-girls FIRST robotics team in the country. Over 75% of GoS alumnae since 2010 declared STEM majors at the undergraduate level. Although this outreach does not directly concern research, the students, relatively new to programming, relate well to the challenges and potential of automated debugging. Discussion thereof helps me motivate research as an interesting potential career.

### 5 Results from Prior NSF Support

<table>
<thead>
<tr>
<th>Area</th>
<th>Task</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
<th>Year 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Repair&lt;br&gt;(Section 2.1)</td>
<td>Search for diverse patches&lt;br&gt;Synthesize general patches&lt;br&gt;New quality evidence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Repair&lt;br&gt;(Section 2.2)</td>
<td>Fix effect inference&lt;br&gt;Repair spec and satisfaction&lt;br&gt;Patch placement and selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Improved analysis&lt;br&gt;(Section 2.3)</td>
<td>Fuzz test triage&lt;br&gt;Fewer static false positives&lt;br&gt;Finding more bugs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: **Orange** denotes research tasks; **purple**, evaluation/integration (which will also be performed throughout).

07/2016–06/2020. The preliminary work was supported by a completed EAGER project, CCF-1446966, CMU total $95,932): **Intellectual merit**: The proposed work extends and uses SMT-based semantic code search techniques [148] to automatically improve programs by correctly repairing defects and constructing new features, guided primarily by test cases. Contributions to date include new benchmarks and principled quality metrics for APR research [95, 146] and a prototype semantic search-based bug repair approach that produces high quality patches [78]. The present proposal also concerns program repair, but they differ substantially otherwise, as CCF-1446966 focuses primarily on improving semantic code search for this new application. **Broader Impacts**: The project supports three junior co-PIs, two of whom are female, as well as 2 CMU Ph.D. students (one of whom is female). It supported the PIs in presentations at a recent Dagstuhl seminar [81] that I co-organized.

### 6 Project plan

Figure 4 overviews the research plan. The proposed work will form the basis of the dissertations of two graduate students (with undergraduate assistance), one focusing on test case-guided repair, and the other on static repair. The evaluation plan includes extra time to account for the overhead of human studies. Because I already have an interested student, the work on static repair (Section 2.2) will be conducted first. The static work in Thrust 3 depends on Thrust 2; I plan for it to be conducted by an undergraduate mid-grant, providing mentorship experience to the senior student.

I will continue my directorship of the REUSE@CMU program and my work with the Girls of Steel throughout the performance period. I expect to teach the new undergraduate course in the first and third years, the PhD analysis course at least the second and fifth years (depending on enrollment), and the Masters course in the second and fourth years.

### 7 Summary

My goal is to advance the state-of-the-art in automatic program repair (APR) to the point that using it is as natural and push-button as writing a test or running a compiler. The proposed research aims to develop APR techniques that can produce consistently high quality repairs to bugs in real-world programs. This research will develop novel (1) diversity-seeking test-guided repair techniques, that construct general, high quality patches and integrate naturally into SE practices, accompanied by new types of quality assurance evidence, (2) analyses that statically identify and then patch bugs in an end-to-end fashion, producing verified, precisely characterized repairs, and (3) targeted program transformations that customize analyses to find more bugs and improve triage. Each research thrust produces independent advances in our understanding of software quality, while overall addressing one of the most fundamental and exciting problems in Software Engineering. The work also integrates into my educational, mentorship, and outreach activities at all levels, from high school to PhD students, and thus I expect the educational contributions of the research program to also be significant.
References


