Automatic Program Transformation for Program Repair and Improving Analysis

Rijnard van Tonder

Thesis Committee: Dr. Claire Le Goues (ISR), Dr. Christian Kästner (ISR),
Dr. Jan Hoffmann (CSD), Dr. Manuel Fahndrich (Facebook)

Wednesday 19th September, 2018

1 Introduction

Software bugs are not going away. Millions of dollars and thousands of developer-hours are spent finding bugs, debugging the root cause, writing a patch, and reviewing fixes [16, 60]. Automated tools such as linters, gradual type checkers, and static analyzers help drive down costs by catching bugs early and increase developer efficiency [12, 13, 19]. However, truly *fixing* bugs remains a predominantly manual and expensive process. Reliance on manual processes further result in fixable bugs (found by automated tools) remaining unfixed. Automatic Program Repair (APR) [44] promises to cut the costs of fixing bugs manually.

Automatic program repair is hard: the prospect of automatically changing a program's semantics can lead to intractable and undesirable behavior [50, 73]. To become mainstream, APR must produce semantically correct patches that can be trusted for automatic application to improve software.

The vast majority of APR research rely on existing tests to identify bugs, validate fixes, or both [9, 38, 39, 43, 49, 50, 54, 55, 58, 63, 64, 75, 80]. Tests are appealing because they are readily available in practice, and straightforwardly indicate whether a change improves the program in question (i.e., by turning a previously failing test case into a passing one). However, reliance on tests impose several fundamental limitations to approaches that currently dominate the state of the art in APR. First, tests are inherently limited to executing single paths and may consequently miss bugs. Figure 1a illustrates the issue in a real and widely-used open source static analysis tool `error-prone`. The call to `resolveMethod` on line 3 can return `NULL`, leading to a possible null pointer exception on line 6. A developer committed a fix (with a test) that inserts a call to a custom error handler (`checkGuardedBy`, line 5). However, the *very same* mistake was made on lines 10–12 in the same `switch` statement, but remained unfixed for 18 months. Tests usually cannot identify recurring mistakes like this, and incur additional manual overhead (writing additional tests) when fixing semantically similar bugs. Second, developers avoid writing tests for certain classes of common bugs (e.g., resource leaks, Figure 1b), because triggering a resource leak could require resource-intensive actions, like bombarding a server with connections.[1] Finally, tests are inherently limited to testing *known* functionality of programs, and thus cannot be used to fix newly discovered bugs automatically. In contrast, the bugs in Figure 1 can be discovered by a static analysis. To truly propel real-world adoption, APR must expand to bugs discovered by automated tools.

```
 1 case IDENTIFIER: {
 2   MethodSymbol mtd =
 3     r.resolveMtd(node, id.get());
 4   // mtd may be null
 5 + checkGuardedBy(mtd!=null, id);
 6   return bindSelect(computeBase(
       context, mtd), mtd);
 7 }
 8 case MEMBER_SELECT: {
 9   MethodSymbol mtd =
10     r.resolveMtd(node, id.get());
11   // same problem!
12   return bindSelect(computeBase(
       context, mtd), mtd);
13 }
```

(a) Developers fixed the potential null pointer exception on line 6; 18 months later, they addressed the very similar bug on lines 10–12.

```
 1  fp = fopen(rdbfilename,"r");
 2  ...
 3  if (memcmp(buf,"REDIS",5) != 0)
      {
 4      rdbCheckError("...");
 5      // fp leaked!
 6      return 1;
 7  }
 8  if (rdbver < 1 || rdbver >
    RDB_VERSION) {
 9      rdbCheckError("...");
10      // fp leaked!
11      return 1;
12  }
13 ...
```

(b) Example resource leaks detected by a static analyzer. Resource leaks are typically not covered by tests.

Figure 1

Static analysis and verification present opportunity to overcome key limitations above: they can reason over multiple paths without tests, apply to bugs that are difficult to test for, and discover bugs missed by developers and tests. Bugs targeted by existing analyzers therefore motivate the application of APR beyond test-driven validation. The challenge in applying static analysis for repair lies in defining suitable correctness conditions for changing a program, and enabling efficient discovery and validation of candidate changes. This complexity leads to work that may focus on just one bug class at a time [25, 32]. More generally, existing APR research based on analysis tooling demonstrate limited practical applicability to real-world software [35, 40, 78] or lack full automation [47, 48]. Existing works all fail to demonstrate the ability to fix previously undiscovered bugs. Some bug-finding tools provide simple syntax-driven "quick fix" suggestions [4, 36], but do not reason semantically about suggested fixes or their correctness.

Proposal Overview: Semantic-driven Fixes. Recent progress has delivered high quality analysis techniques and tools to cost effectively find real bugs, like the examples in Figure 1. A concrete example is Infer [19], which uses Separation Logic to reason about heap properties like null dereferences and memory leaks. My insight is that the underlying semantic reasoning behind tools, like Infer, enable new APR techniques for fixing common bugs (including unknown ones): they allow (a) efficient semantic-driven search and application of program

[1]https://github.com/cockroachdb/cockroach/pull/22448

transformation and (b) provide high confidence in patch correctness by validating changes with respect to the semantic analysis domain.

As a concrete example, the bugs in Figure 1 are found by a static analysis (namely, Infer) by symbolically executing along paths and summarizing effects on the heap using Separation Logic [14, 15] to detect violations (e.g., null dereferences and leaks). Finding a fix requires finding program transformations that avoid such violations, for example "if the file is open in the precondition of a program, it should be closed in the postcondition." Such repair specifications can be encoded as general *fix effects* (once per bug class) that describe a desirable state transition in Separation Logic (i.e., as pre- and postconditions). A compositional shape analysis [20] underpinning Infer exposes heap effects of individual program terms so that such fixing effects can be discovered efficiently.

The general method defines correctness in terms of the underlying *semantic* property of a bug class (i.e., as fix effects), with respect to an analysis domain (e.g., tracking heap state encoded in Separation Logic); an applied fix is validated by the analysis logic. In this setting, I propose to develop fully automatic, semantic-driven APR processes using recent advances in logic-based analysis reasoning and tooling used for bug finding. The proposed work entails developing new strategies for semantic-driven *search* and *application* of program transformations. My work advances the state of the art by enabling efficient end-to-end bug fixing (i.e., starting with bug discovery through to patch validation) that can fix previously undiscovered bugs. I will focus on a fixed set of common bug classes and identify suitable corresponding analysis domains to demonstrate efficient repair of such bugs in large, real world programs.

Proposal Overview: Improving Analysis with Program Transformation. My insight is that constrained program transformation, (using, e.g., the semantic-driven approach above), can increase effectiveness of analyses. Runtime recovery [51] and error nullification [45, 57] demonstrate benefits of modifying programs and state to avoid program crashes and exploitation. Initial results applying semantic-driven program transformation for APR reveals *static and dynamic analyses themselves* can be improved. Current program transformation techniques, and those afforded by APR research, simply do not explore this potential.

As a concrete example, the two resource leaks in Figure 1b cannot, in fact, both be found by Infer. My work fixing resource leaks reveal that the second leak (Line 10) is only detected once the first leak is fixed (Line 5). Without the fix, Infer short circuits the analysis after the first leak is detected. This early result is the first, to the best of knowledge, showing that program transformation can improve static analysis. Note that test-driven approaches generally fail to recognize this potential, since an external analysis isn't used.

More generally, I observe that programs and analyses may conservatively abort when an error is detected. If analysis continues after detecting an error, the result can be invalid or unsound because the program semantics may be affected by the error in unpredictable ways. However, a conservative change to the program can allow the analysis to make safe assumptions, allowing it to proceed further and still return valid results. I propose to pursue this new direction by applying my program transformation techniques to empirically demonstrate the largely unexplored potential and effectiveness of improving existing analyses.

The rest of the proposal is structured as follows: Section 2 introduces the thesis statement and the thesis hypotheses that motivate the proposed research, and the expected contributions. Sections 3 and 4 explain the two proposed research thrusts in detail. Section 5 describes the expected timeline to complete the research and Section 6 concludes.

2 Thesis Statement

The thesis is that semantic-driven search and application of program transformations, using logic-based validation enable efficient, scalable, and unassisted automated program repair of real-world programs and can improve the effectiveness of existing program analyses.

The thesis is supported by two hypotheses that motivate the proposed research. To affirm the thesis, I will evaluate these two hypotheses and produce the listed contributions to establish their validity. Additional clarity for terms in blue are linked in the Glossary (page 20).

Hypothesis A: Program Transformation Toward Semantic Repair

I hypothesize that

1. the domain of Separation Logic enables efficient, unassisted semantic-driven search and correctness validation (no tests required) of program transformations resulting in fixes that scale to real-world programs, including previously unknown bugs, and that
2. the domain of Separation Logic can be *extended* to fix *additional bug classes* using efficient, unassisted semantic-driven search and correctness validation of program transformations on large real-world programs, including previously unknown bugs.

Hypothesis B: Program Transformation Toward Improving Analysis

I hypothesize that rule-based application of semantic-driven program transformations (using methods in Hypothesis A, or a priori specification) can efficiently improve analyses with unassisted logic-based validation for real-world programs to

1. improve the effectiveness of existing static analyses to (a) find more bugs, (b) fix more bugs, or (c) reduce false positives, and
2. improve the effectiveness of existing dynamic analyses to report fewer duplicate bugs.

Contributions

To affirm the Thesis Statement, I will produce the following concrete contributions. The research will result in the invention, design, and implementation of the first unassisted, end-to-end automated program repair systems for multiple bug classes, describe their effectiveness for fixing real programs, and demonstrate applicability of program transformation to improve existing analyses. The expected contributions of the dissertation are:

1. Development of semantic-driven search, application, and validation using an existing Separation Logic-based bug-finding analysis. The approach will focus on three classes of heap-based defects: null dereferences, resource leaks, and memory leaks. The approach will fix previously undiscovered bugs in real programs, (unlike existing APR techniques) and elides the need for tests.

2. Extension of (1) by developing semantic-driven search, application, and validation using an existing logic-based bug-finding analysis to address at least one additional bug class. Like (1), the approach will fix previously undiscovered bugs in real programs.

3. Development and use of program transformation techniques to demonstrate the improvement of existing analyses through program transformation. I will demonstrate improvement of at least one *static* analysis in terms of analysis effectiveness, where improvement is quantified as one of: the number of true bugs discovered (higher is better), the number of bugs fixed (higher is better), or the number of false positives reported (lower is better). I will also demonstrate improvement of at least one *dynamic* analysis technique in terms of analysis output fidelity, quantified as deduplication of bug reports (less duplicate reports is better).

3 Program Transformation Toward Semantic Repair

Proposed Research: Thrust 1

The vast majority of current APR techniques are test-driven: they rely on tests to identify bugs, perform fault localization, and validate fixes [38, 43, 50, 55, 58, 64]. Relying on tests introduces inherent limitations. Repair can only succeed if tests exist that exercise known buggy functionality [57], and some bugs are simply hard to test for. Demonstrating techniques to fix previously unknown bugs is imperative to advancing program repair in practice, though unprecedented in current literature. Static analyses are used in practice to discover previously unknown bugs [19, 31, 33], including common bugs that are hard to test for (e.g., resource leaks). I propose to develop APR techniques that build on recent advances in static analysis bugs in real-world programs, including previously unknown bugs.

Static analysis techniques and related logic-based reasoning use semantic abstractions to identify buggy behavior [5, 31, 33]. My first insight is that these semantic abstractions can provide precise correctness validation of fixes for cases where tests typically fail. Second, these semantic abstractions present new ways of driving search and application of program fragments to repair bugs.

Thrust 1 comprises two parts. The first is semantic matching and validation for fixing heap properties (**1A**). This first part is existing research that shows the semantic matching and validation works for heap properties. In the second part I propose to show that the semantic technique for repair generalizes to one or more new bug classes. The approach, evaluation strategy, and criteria for success in my existing work (**1A**) informs the planned research extension (**1B**).

3.1 Thrust 1A: Semantic Matching and Validation of Heap Properties for APR

Recent advances have lead to large scale industrial use of static analysis to detect common heap-related defects like null dereferences and resource leaks [19]. I propose to develop bug repair techniques that enable unassisted fixing of such bugs. My approach is to extend an existing analysis based on Separation Logic [66] to drive automated fixing of heap defects. My key insight is that the underlying analysis reasoning enables efficient and correct methods for discovering and applying program transformations.

3.1.1 Preliminaries: Reasoning with Separation Logic

Infer [5] is a successful open source analysis based on Separation Logic and Hoare-style reasoning that finds bugs at scale [19]. The analysis excels particularly at finding heap-related defects in C and Java. Infer translates C and Java programs to the Smallfoot Intermediate Language (SIL) [14, 15] to reason symbolically over a program's manipulation of the heap. A SIL program consists of procedures which comprise sequences of instructions. SIL instructions are annotated with pre- and postconditions assertions expressed in Separation Logic. Assertions describe an instruction's effect over symbolic heaps. Symbolic heaps form the abstract domain to detect faulting conditions like memory leaks. The assertion language encodes heap facts using points-to heap predicates over program and logical variables. Heap predicates form disjoint sub-heaps using the separating conjunction $*$ (read "and separately"). The analysis uses local reasoning [61, 62, 81] with compositional summaries [20] to efficiently infer specifications that summarize the effects of individual commands. These methods are powered by the Frame Rule in Separation Logic, which separates the parts of the heap that an instruction changes (the footprint), and ignores the unchanged part (the frame). The Frame Rule (shown right) allows the analysis of C to continue using just the specification $\{P\}\ C\ \{Q\}$ without considering the frame F. *Frame inference* [15, 20, 30] is a key technique that automatically infers the necessary frame F so that the frame rule can be applied in the analysis proof system. Infer implements this inference procedure and performs symbolic execution of paths to discover violations of heap properties. To illustrate, Infer reports the first resource leak in Figure 1b by identifying the path through line 6 where `fp` is is dead. The symbolic interpreter enters a special `fault` state on this path, denoted formally as $C_\ell, \sigma \rightsquigarrow \texttt{fault}$: the interpretation step $\rightsquigarrow$ for instruction C at location ℓ in symbolic state σ results in a `fault`. The file pointer is still live on the heap (which we denote by a predicate *open*, or $\{\texttt{fp} \Mapsto \mathit{open}\}$) at $\ell = 5$ when it becomes dead: $\texttt{return}_\ell, \{\texttt{fp} \Mapsto \mathit{open}\} \rightsquigarrow \texttt{fault}$. At this point the analysis reports a bug, and does not reason further about `fp`. This is where my work starts, and I invoke the proposed repair procedure.

$$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}$$

The Frame Rule

3.1.2 Existing Research: Semantic Matching and Validation of Heap Properties for Repair

The core idea starts by defining desirable semantic transitions in the abstract domain which I call Repair Specifications. Concretely, a Repair Specification is a Hoare-style triple that defines a precondition and a postcondition. The Repair Specification entails a singleton heap F in the precondition that describes a *fixable* predicate for a placeholder variable *pvar* and a corresponding single heap F' describing a *fixed* predicate over the same placeholder variable *pvar*. Fix effects are generic to entire bug classes. For example, $F = \{pvar \Mapsto open\}$, $F' = \{pvar \Mapsto closed\}$ describes a desirable fixing transition for resource leaks (as in our running example). By reasoning over the abstract domain of what the code *does*, the proposed approach supports multiple languages automatically and is resilient to syntactic customizations.[2] General semantic predicates like *open* correspond to API calls in the source languages (like `fopen` in the C library or open `FileReader` objects in Java). These calls are modeled by Infer, and translated to generic semantic predicates in SIL. While function modeling requires some manual effort, it is typically already added for the analysis (it is for Infer, and our repair technique can use the models without further effort).

Repair Specifications guide repair *search* by matching on the desired semantic effects of instructions and procedures in SIL. We can express this formally by $\{F\}\ ?C_R\ \{F'\}$ which attempts to find an instruction C_R that satisfies the pre- and postconditions. The key idea is that the specification matches precisely the semantic components we care about, but can also extend to match instructions or functions that affect other parts of the heap. For example, we can express $\{F\ *\ P\}\ ?C_R\ \{F'\ *\ Q\}$ where P and Q may be nonempty. Frame inference can be applied to pull out P (resp. Q) to precisely capture any side effects of the candidate repair beyond the desired F, F'. Thus, semantic characteristics can be precisely defined to parameterize search, and makes extra heap effects of matched instructions explicit.

Ultimately, given a set of candidate fixing C_R's returned by repair search, the procedure seeks a program change such that the analysis validates a fixing change. As noted in prior work, validating correctness of a repair (in a static analysis context) depends on the semantic abstraction at play [47]. In our case, we expect symbolic interpretation of the analysis to avoid a `fault` by fixing a fault-inducing heap state. Prior work formalizes correctness in terms of good and bad error traces [47]. Symbolic interpretation in the analysis is conceptually similar: it validates that a change removes a faulting interpretation. In conjunction, fixes are precise because semantic effects on the heap are matched and introduced explicitly, which gives confidence that the rest of the program semantics are unchanged. To complete a repair satisfying these conditions, the program must be changed syntactically at a suitable location. I next describe the proposed approach to achieve this objective.

Approach. The repair procedure starts by searching the abstract program for terms that satisfy a Repair Specification. The approach will rely on manually-supplied Repair Specifications, which is done only once per bug class. I propose to target three heap defects commonly found in practice [5, 18], and the focus of Infer's analysis: null dereferences, resource leaks, and memory leaks. Devising ways to infer Repair Specifications is a promising idea, but based on preliminary work, effort for manual specification per bug class is low. I will consider additive program transformations, because the types of bugs considered are typically caused by lack of certain operations on explicit heap content (e.g., resource release, freeing memory, or checking nullness) and symptomatic of common developer mistakes [7, 79].

Semantically, certain bugs can be fixed in various ways (e.g., null dereferences) [47]. Correct repairs may also admit code change at various locations [28]. As a concrete example,

[2]e.g., functions that wrap the C library call `free` will still match the specification

fixing the first leak in Figure 1b can happen by closing `fp` on line 5, or just before line 4. In general, the bug class bears on the choice of where to apply a fix (e.g., developers might prefer the convention of immediately checking nullness before a dereference). I propose to use the location ℓ reported by the analysis; preliminary results suggest that the convention yields correct patches that are acceptable for merging into upstream repositories.[3] I leave open additional consideration of stylistic conventions or human judgment.

In general, multiple SIL instructions can satisfy a Repair Specification. I propose two ways to rank candidates for patching: variable types and extra semantic effects. For example, between the standard `free` function (which accepts a `void *`) and custom `free` function wrapper (which accepts, say, a more specific type), repair prefers the custom `free` if the variable type in the match specification is the same as the fault-inducing variable type. In terms of semantic effects, the approach prefers candidates semantically "tight" matches, i.e., where extra semantic effects are absent. A fully automatic repair procedure requires making a syntactic change and validating the change against the analysis. To make a syntactic change, repair-satisfying SIL instructions are resolved to their source location, and SIL variables must be renamed to fix the fault-inducing variable.

3.1.3 Evaluation Strategy and Criteria for Success

My goal is to correctly fix bugs for real programs, including previously unknown bugs. The current approach will focus on three common bug classes supported by Infer (and other static analyzers): null dereferences, resource leaks, and memory leaks. I will evaluate my approach by sampling open-source projects (preferring popular, widely used ones) on repository hosting sites like GitHub. I will demonstrate *unassisted* repair (modulo up front Repair Specifications) for multiple languages (C and Java). I will include large programs ($>$100KLOC) to demonstrate that the overall approach *scales*. In terms of *efficiency*, I will measure (a) the speed of the repair procedure, (b) size of the program search space, and (c) size of satisfying candidates to quantify efficiency. I consider search efficient if, for a single real-world program, the repair procedure terminates in a time comparable to the analysis (i.e., in the order of minutes and hours, not days). I will quantify cases where satisfying candidates result from false positives in the underlying analysis to characterize search and repair limitations. As applicable, I will submit pull requests of generated fixes to open source projects as a strong (but not required) validating signal that the APR technique produces correct and human-acceptable patches.

Comparative evaluation to existing APR techniques is difficult due to the unique bugs and challenges I address. However, historically fixed bugs of the classes I consider can inform a ground truth study to strengthen the validity of the approach. Existing bugs and fixes from historic software activity will also be used to develop and test the approach.

The work is successful if it produces a new technique for correctly fixing bugs in large, real-world projects, including previously undiscovered bugs for the three classes considered. Further, the technique can function unassisted, and enables efficient search, application, and validation of program transformations.

[3]For example, the approach generated the following patch, merged into an upstream repository on GitHub: `https://git.io/vpsoV`.

3.2 Thrust 1B: Extend Semantic Matching and Validation for APR

Based on preliminary results, the semantic matching and validation approach introduced in **1A** demonstrates ability to generalize further. I propose to develop additional repair techniques that extend the bug classes considered in **1A**. My insight is that (a) matching over semantic effects of program terms and (b) analysis-based semantic validation enable a general technique that enables correct, unassisted bug fixing.

3.2.1 Preliminaries: Logic-based Reasoning for Repair

Separation Logic is an effective formal foundation to find heap-related defects; my initial work shows that it is also effective for repair. Semantic and logic-based techniques demonstrate similar utility for discovering defects beyond those considered so far [29, 34, 37, 69–72, 74]. My intuition is that these techniques can enable effective repair for further bug classes.

Recent work develops *semantic frameworks* [33, 69] for building bug-finding and verification tools [6, 34, 52, 74]. Frameworks like $\mathbb{K}$ [69] and SeaHorn [33] treat program semantics as a first class concern and express the behavior of program terms with a logic representation. Infer can also be considered a framework: it provides a plugin interface for implementing new analyses that hook into the separation logic reasoning engine. These analyses detect buffer overflows, arithmetic overflows, concurrency errors, information leaks, and further bugs that are actively targeted by APR techniques [25, 45–48]. Like Infer, semantic frameworks (e.g., $\mathbb{K}$ and SeaHorn) detect bugs with respect to a logical abstraction of semantics. Matching Logic, used by $\mathbb{K}$, is a first-order logic variant for specifying and reasoning about structure by means of patterns and pattern matching [70]. Matching Logic is theoretically appealing because it generalizes other logics, including Separation Logic. That is, Separation Logic can be framed as a Matching Logic theory [68]. It provides analogous support for local reasoning [62] using frames [70], but also extends to, e.g., program environments beyond heap. Matching Logic has also been used to check reachability properties [71, 72]; recent work demonstrates that off-the-shelf reachability tools can directly enable repair procedures [59]. Conceptually similar to Infer, $\mathbb{K}$ can perform path directed symbolic execution to discover bugs [10]. My insight is that these methods suggest a natural extension to repair bugs not considered yet (e.g., ones with non-heap properties).

3.2.2 Proposed Research: Extend Semantic Matching and Validation for Repair

The key idea is that semantic matching and validation for repair generalizes to analyses and buggy properties that express semantic effects over state transitions (i.e., as pre- and postconditions). This theoretically enables precise program repair for new classes of bugs targeted by recent analyses (and their underlying semantic reasoning). I propose to develop an unassisted repair procedure for an additional bug class using an existing semantic framework or analysis. Several tools implement logic-based reasoning, most notably the $\mathbb{K}$ framework [69], SeaHorn [33], and Infer [5]. Selecting a semantic framework presents exciting potential for developing new repair techniques, but also presents risk that may arise during the early stages of use, including tool complexity, tool support, or documentation. I will therefore follow a staged process to determine suitability for each of the frameworks in the following order: the

𝕂 framework, SeaHorn [33], and then Infer [5]. Infer presents a safe alternative to 𝕂 and SeaHorn, given my prior experience.

Approach. I will focus specifically on extending the Repair Specification principle in the chosen semantic framework (i.e., matching desirable semantics in pre and postconditions). I will implement prototype tooling for fixing the new class of bug that the framework can support.

My first proposed approach is to develop repair using the 𝕂 framework because it treats program semantics as a first class concern, is mature and actively developed, and has demonstrated ability for finding bugs [34, 74]. I will focus initially on automating repair with Matching Logic, the underlying reasoning of 𝕂. Currently, 𝕂 supports finding through integer overflows, buffer overflows, and more. If I assess that 𝕂 is unsuitable for repair, I will consider SeaHorn as the second option. SeaHorn mainly targets buffer overflow bugs [33]. SeaHorn uses Horn clauses as an intermediate language and performs symbolic execution to perform precise memory analysis. If SeaHorn is also unsuitable I will focus on extending Infer's analysis, which extends to information flow bugs (i.e., dangerous source-sink flows), concurrency bugs, or buffer overflows [5]. Regardless of the specific logic abstraction and tooling, my focus will be to extend the method of semantic matching and validation for repair. As with **1A**, the objective is to target real world bugs.

As in **1A**, repair presents additional unique challenges beyond simply semantic bug detection. In particular, repair requires mapping semantic abstractions back to syntax, entails valid syntactic modification to the program, and presents the choice of patch location. I will develop techniques to allow unassisted repair similar to **1A**, and anticipate that methods and conventions from **1A** will inform design choices to effectively repair further classes of bugs. At least the following bug classes are in scope for extending repair:

- Buffer overflows
- Integer underflow/overflows
- Use-after-free/Double-free bugs
- Division-by-zero bugs
- Information flow bugs
- Concurrency bugs
- Uninitialized variables
- Foreign Function Interface (FFI) bugs (e.g., FFI leaks)

3.2.3 Evaluation Strategy and Criteria for Success

The proposed work extends **1A**, so I will evaluate *efficiency* of repair search and application similarly (cf. Section 3.1.3). The choice of analysis and bug bears on the analysis time; I will similarly consider search efficient if the proposed repair procedure terminates in a time comparable to the original analysis. I will demonstrate *unassisted* repair (modulo up front specification) for large programs to show that the approach *scales*. Comparative evaluation to existing APR techniques will remain difficult and I will consider ground truth historically fixed bugs to compensate. The work is successful if it satisfies the criteria above, as in Section 3.1.3.

3.3 Preliminary Results

My work using Separation Logic as a formal foundation for program repair demonstrates that practical program repair is possible for previously undiscovered bugs in real programs. I prototyped the technique in **1A** with FootPatch, a push-button tool for fixing null dereferences, resource leaks, and memory leaks. Table 1 summarizes the results on a convenience sample of 3 C programs and 2 Java programs. Overall, FootPatch correctly fixes 15 resource leaks, 6 memory leaks, and 2 null dereferences. False positive rates are low, and arise from Infer's analysis missing clean up functions (e.g., that free memory). FootPatch demonstrates the ability to scale to real-world programs, and has produced upstream patches accepted into popular open source software on GitHub. The main take away is that semantic-driven matching and patch validation approach efficiently and correctly fixes a number of real bugs. The results present empirical evidence in support of extending semantic repair to additional logic-based abstract domains and bug classes as proposed for **1B**. Currently, however, there is no evaluation effort yet for supporting new bug classes as proposed in Section 3.2.2.

Project	**Lang**	**kLOC**	**Time (s)**	**Bug Type**	**Bugs**	**Max Cands**	**Fixes**	**FP**
Swoole	C	44.5	20	Res. Leak†	7	1	1	0
				Mem. Leak†	20	3	6	3
dablooms	C	1.2	9	Res. Leak†	7	2	7	0
redis	C	115.0	79	Res. Leak†	8	8	6	0
Apktool	Java	15.0	584	Res. Leak†	19	3	1	0
error-prone	Java	149.0	262	Null Deref	11	66	2	0

Table 1: Bugs repaired with FootPatch. "**Bugs**" is the number of bugs detected by Infer's static analysis. "**Max Cands**" is the maximum number of IL repair candidates for the bug (pre-compatibility check). "**Fixes**" are the number of unique patches fixing unique bugs (post validation). † indicates one or more fixes for previously undiscovered bugs.

4 Program Transformation Toward Improving Analysis

Proposed Research: Thrust 2

Analyses must approximate [42], trading theoretical properties (e.g., precise alias reasoning, soundness) for practical benefits (e.g., speed, accuracy) and vice versa [19, 23, 34]. These tradeoffs manifest as implicit tool assumptions that are difficult to trace and modify [23]. Such assumptions are also generally applied, and can limit confidence in soundness of analysis results [23] or accurate bug detection [22]. My insight is that program transformation can augment static and dynamic analyses to improve analysis results.

4.1 Thrust 2A: Improve Static Analysis Effectiveness

Analyses may abort when an error is detected. This is conservative and generally acceptable behavior: if analysis continues after detecting an error, the result can be invalid or unsound because the program semantics may be affected by the error in unpredictable ways. However, manual controls for customizing analysis reasoning and assumptions can improve the effectiveness of static analyses [21, 34]. My insight is that automated program transformation can change analysis behavior to improve results, so that the original analysis can find more bugs, fix more bugs, or reduce false positives.

4.1.1 Preliminaries: Modifying Analysis Behavior

Consider the Runtime Verification Match tool (RV-MATCH) which detects undefined behavior in C programs using a formal C semantics [34]. Focusing on strict correctness from a semantics perspective, RV-MATCH originally aborted analysis the first time it detects undefined behavior. The authors soon discovered limitations of this (ultimately correct) choice: implementation-defined behavior of various compilers is considered acceptable by developers. RV-MATCH was subsequently changed to continue for "likely expected behavior", and added ways to manually customize formally-defined semantics for such cases [34]. Another example is Infer, which fails to find both resources in the example Figure 1b, and memory leaks that depend on the same data structure.[4] These analysis behaviors are not purely engineering concerns, but conservative compromises in the underlying semantic reasoning of the analysis. Code instrumentation has been used to identify deliberate unsoundness in analysis behavior [23], but does not proactively focus on improving the analysis results.

4.1.2 Proposed Research: Better bug finding and fixing

My insight is that semantic-driven program transformation can improve static analysis for finding bugs, fixing bugs, or reducing false positives (without modifying the underlying analysis). My initial work using FOOTPATCH motivates the idea: Infer can detect strictly more true memory leaks and resource leaks after correct patches are applied. FOOTPATCH can automatically fix bugs correctly, and enables Infer to find more bugs in 3 active C projects; FOOTPATCH can consequently also fix the newly discovered bugs. Tools like RV-Match suggest the idea generalizes to other analyses and bug classes. For example, undefined C behavior can be temporarily changed so that RV-Match continues analyzing.

My key observation is that conservative analysis assumptions causing "abort-on-error" behavior is sensitive to the underlying semantics of the bug class. This means that positively augmenting analysis behavior relies on targeting bug-specific semantic behavior, or program constructs (e.g., loops) that can confound analysis of bug-specific behavior. I propose a principled study of these initial observations to demonstrate analysis improvement. The expected output of this work is to produce the first systematic study that explores, develops, and evaluates semantically-derived program transformations for improving the static analysis.

My work with FOOTPATCH suggests three axes to improve current static analyses: to (a) find more bugs, (b) fix more bugs, or (c) reduce false positives. These directions are related and complementary, e.g., more bugs can be fixed if more bugs are found. In the interest of a conducting a principled study, I will focus primarily one axis: improving analysis

[4]For example, a leaked variable `hmap` can only be detected and freed (Fix: `https://git.io/vxjAO`) after the leaked node `root` is detected and freed inside it (Fix: `https://git.io/vxjAY`).

reasoning and coverage to find more bugs, but leave open the possibility of substituting and supplementing results impacting objectives (b) and (c).

Approach. I will choose two or more popular static analyses to demonstrate improvement using program transformation. Infer is a natural first choice given existing infrastructure to produce automatic fixes (Section 3.1). Additional candidates include analyses covered in the literature: the Clang static analyzer [8], Frama-C [27], Clousot [31], Seahorn [33], DR.CHECKER [53] or another. I will focus on developing bug-fixing program transformations for bug classes that the analyses target, such as null dereferences, leaks, and overflows. There is a precedent for using bug fixing program transformations targeting null dereferences, buffer overflows, and more in Clousot [47]. Analysis customization through program transformation also complements existing work that customizes Clousot's analysis to improve performance and bug finding [24].

Program transformations will be expressed as templates for each particular bug class. I will develop a rule-based approach that applies these templates based on the analysis error reports. Principles of semantic repair in Section 3 are applicable here: inference and use of fixing patterns, correctness validation by the analyzer, and location information to drive patch application. In addition, complex program structures (e.g., loops and conditional predicates) can limit analysis coverage or lead to imprecise analysis results. Such structures offer opportunity for analysis customization. I will investigate fundamental reasoning limitations arising from such structures to inform further program transformation templates.

4.1.3 Evaluation Strategy and Criteria for Success

My goal is to demonstrate practical utility of program transformation for improving analyses. Thus, my priority will be to demonstrate analysis improvement on *real-world programs*. I will sample open source projects (preferring popular, widely used ones) on repository hosting sites like GitHub. I expect the analyzers to support large projects written in common programming languages (e.g., C, Java), demonstrating that the approach works *at scale*. I will focus on improving analysis output for previously unconsidered bug reports (including false positive reports). However, it is difficult to ensure that an analysis can produce new bug reports to inform a systematic evaluation. Most analyzers have a history of fixing bugs (documented in the literature, on the tool website, or in project commit logs). As a risk mitigation strategy, I will use historic bug fixes and project revisions to prototype and evaluate the proposed technique. Initial development will entail specifying repair templates (automatically inferred, manually specified, or both) and implementing conditions for patch application. Beyond this up front cost, the technique should operate *unassisted*, and be able to react to analysis validation, monitor changes in analysis behavior, and apply patches automatically to discover new bugs. I will demonstrate *efficiency* by measuring the time it takes to complete a program transformation cycle for discovering new bugs. This includes measuring the performance impact of transformation on the analysis. If permitting, similar bug classes for different analyses can inform a comparative evaluation of to further evaluate effectiveness and generality of the technique.

The work is successful if applying program transformation can strictly improve the original analysis, and meets the criteria in the evaluation above. Namely, (a) program transformation is automatic (modulo up front specification) (b) analysis overhead and duration is competitive compared to the original analysis, and (c) the technique works for real programs.

I anticipate that my approach using program transformations can reveal shortcomings in the analysis implementation that developers can rectify. Manual improvement to the analysis implementation as a consequence of varying programs supports validation of the claim that program transformation can improve existing analysis, but I will not explicitly evaluate benefits where the underlying analysis itself is improved. The proposed approach will focus on treating an existing static analysis as a "black box" and I will evaluate program transformation for profitably improving the analysis output without modifying the underlying analysis.

4.1.4 Preliminary Results

Using FootPatch allows Infer to discover additional memory and resource leaks in three of the six C programs in Table 1 (Section 3). There is no systematic evaluation yet of FootPatch for improving analyses, which will be informed by considering additional programs and further investigation of the underlying analysis reasoning. Nevertheless, the repair approach shows that automated program transformation can interact positively with the analysis to produce better results. `Redis` is one example of a large program (>100KLOC) that demonstrates potential of improving analysis via program transformation at scale.

4.2 Thrust 2B: Improve Dynamic Analysis Effectiveness

Similar to analyses (but more concretely), programs abort when an error occurs during dynamic execution. Again this is conservative and generally acceptable behavior: if program execution hits a null dereference, we prefer that it crash rather than execute arbitrary code. Analogous to changing static analysis behavior, changing program behavior to tolerate dynamic execution errors can be beneficial to the program operation itself [51, 67]. My insight is that program transformation can also improve *dynamic analysis output*, specifically for high fidelity error reporting and crash deduplication.

4.2.1 Preliminaries: Modifying Program Behavior

Modifying program behavior has been explored for producing fault-tolerant behavior and neutralizing vulnerabilities. Failure-oblivious computing [67] and recovery shepherding [51] show that programs can beneficially continue executing by neutralizing buffer overflows, null dereferences, and division by zero errors. These techniques restrict focus to improving execution of the *program*. My insight is that program modification can improve *analysis output* in terms of bug reporting and triage. For example, symbolic executors and fuzzers generate can generate tens of thousands of crashing inputs for a program [11, 22], thousands of which can be duplicates that correspond to the same bug [22]. Automated crash triage seek to map multiple crashing inputs to the same bug. Root cause analysis of a crash is hard [41, 45, 58] so existing techniques and tools identify unique bugs approximately using callstack information [1, 56], branch coverage [2, 3], path constraints [65], and machine learning [22]. In contrast, *actual fixes* for known bugs can serve as ground truth to precisely map crashing input to bugs [22].

4.2.2 Proposed Research: Semantic Crash Bucketing

My insight is that program transformations for fault tolerance [67] and repair (as in Section 3) can approximate actual fixes for bugs commonly found by dynamic analyses (like null dereferences and buffer overflows) to precisely identify and triage bugs. I propose Semantic Crash Bucketing, which seeks to classify unique bugs by bucketing related crashing inputs as a function of program transformations (i.e., a program delta). The core idea is that changing a program's semantics with approximate fixes can accurately and automatically constrain crashing behavior in a way that mimics real program fixes. My intuition is that program transformation can be more sensitive to the underlying semantic property causing a crash and thus suffer less imprecision than existing approaches. The primary application is to improve the output of dynamic analysis and testing tools, like fuzzers and symbolic executors, that are particularly susceptible to reporting duplicate errors.

Existing approaches typically hard code ad-hoc methods of deduplication (e.g., blacklisting crash patterns [3], callstack traces [1], or branch sequences [2]). In contrast, Semantic Crash Bucketing opens opportunity to customize crash bucketing and operates independent of the diverse, hard coded assumptions across testing tools. The main challenge of the proposed approach is that program transformations can overfit to crashing inputs (similar to how APR techniques can overfit to tests). For example, suppose a program contains more than one unique bug, each with independent fixes. Inserting `exit(0);` at the beginning of a program will satisfy the criterion of neutralizing all crashes, but will associate (and underapproximate) all unique bugs with a single fix. To be effective, program transformations must therefore have constrained semantic effects to precisely identify unique bugs.

Approach. The proposed approach shares similarities to that proposed for improving static analyses. I will focus on developing bug-fixing program transformations for bug classes typically found by fuzzers: null dereferences and buffer overflows. Program transformations will be expressed as templates. In contrast, the conditions for patch application will depend on program behavior, in response to a crash. I will develop a rule-based approach that predicates patch application on semantic cues informed by crashing behavior. The semantic cues are sensitive to a particular bug class. For example, debuggers such as GDB can indicate a null dereference by reporting the last line a program executed, such as `x = p->q;`. This behavior can suggest a change to check whether `p` is null as the reason for the crash (and all related crashing inputs). Such an assumption must then be validated (the program should no longer crash) to be a candidate for identifying a unique bug.

While null dereference crashes can be bucketed based on failure location, the same is not true for other general bug classes. For example, buffer overflows may only manifest in a crash after a corrupted stack value is accessed. However, a bug-fixing transformation should prevent overflow at the point-of-memory-corruption (not point-of-crash). Relying on point-of-crash for buffer overflows may lead to spurious error reports based on the stack contents. These semantic differences make the crash bucketing problem non-trivial, and require multiple rule-based strategies per bug class.

Rules for patch application can profitably incorporate specialist knowledge to remove duplicate reports.[5] These rules enable customizing sensitivity to the semantic property of

[5]https://twitter.com/azonenberg/status/966738179486134272

the bug in conjunction with program transformation (i.e., nullness, or unsafe buffer lengths) to achieve greater precision than, e.g., path execution uniqueness. There is precedent for using repair templates that neutralize exploitation of buffer overflows [45], which informs my template construction for deduplicating error reports.

Note that the intention of the proposed approach is not to identify fully the root cause (which is hard), but to approximate it using a combination of patch application rules, program transformations, and validation of correct program execution.

4.2.3 Evaluation Strategy and Criteria for Success

My goal is to demonstrate that semantic-driven program transformation can improve output fidelity of existing dynamic analysis and testing tools for *real programs*. I will focus on duplicate error reporting as the measure of output fidelity (less duplicates is better). In the interest of generality, I will choose three state of the art fuzzers that use different dynamic analysis techniques to perform path exploration and crash deduplication: American Fuzzy Lop (AFL) [2], CERT BFF [1], and Honggfuzz [3]. I will start by building a corpus of crashing inputs to measure ground truth duplicate error reporting of these tools, for a selection of unique bugs. ground truth duplication is measured by bucketing the crashing inputs using a *correct* fix. I will use these ground truth results to conduct a comparative evaluation: ideal deduplication (determined using correct fixes) will inform the precision of using the technique of approximate fixes proposed above. Semantic Crash Bucketing with approximate fixes applies to newly discovered bugs, but to conduct a ground truth comparison I will need existing bugs with historically correct fixes. I will use historical bug discoveries by the existing fuzzers (such as SQLite bugs discovered by AFL), and extract correct fixes to conduct the study. I will evaluate on at least two classes of bugs commonly detected by these fuzzers: null dereferences and buffer overflows. I leave open the possibility of extending the approach to work for additional memory corruption bugs or arithmetic overflow errors. I will measure standard metrics to evaluate *efficiency*: the time it takes to produce approximate fixes given a crashing input, and the rate of failure (i.e., when the technique fails to produce a patch). I will qualify limitations of the approach by comparing real and approximate fixes.

Similar to Section 3 for static analyses, initial development will entail specifying repair templates (automatically inferred, manually specified, or both) and implementing conditions for patch application. Beyond this up front cost, the technique should operate *unassisted*, and produce output of deduplicated crashes by monitoring changes in program behavior and applying patches automatically to identify unique bugs.

Comparison to existing, alternative deduplication techniques in the literature may complement the evaluation, but lack of available bug corpora and tool implementation from previous studies (e.g., [22, 26, 65]) generally preclude such a comparison. Evaluation will further benefit application to symbolic execution engines (e.g., KLEE [17]), but I leave this extension open.

The work is successful if Semantic Crash Bucketing demonstrates improvement of dynamic analysis output fidelity compared to the original method. Analysis output is quantified as deduplication of bug reports. In addition, the work should (a) demonstrate applicability to real programs, (b) integrate automatic patch application and validation reasoning, and (c) produce tooling that runs in competitive time compared to built-in deduplication methods (i.e., produces patches in the order of seconds to minutes).

4.2.4 Preliminary Results

I have implemented a prototype to bucket crashes for null dereferences. There is work in progress for supporting buffer overflows. The current technique for patching null dereferences starts by identifying pointer dereference patterns at the crash site, for some crashing input. If a program variable is possibly null, a null check for the program variable is inserted at that location. The patch approximates error handling by exiting the program on condition of being null (similar to the common idiom of `return -1;` for C programs). With respect to identifying bug uniqueness, the patch is more accurate than all built-in deduplication methods for the three fuzzers considered. Table 2 illustrates these early results. Each row for the projects `w3m` and `SQLite` correspond to *one* unique bug.

Project	Type	ID	CrashGen	AFL Fuzz	CERT BFF	HFuzz
w3m	Null-deref	1	458	103	25	75
		2	545	23	0	0
		3	507	36	0	6
		4	525	11	0	0
Total			**2,035**	**173**	**25**	**81**
SQLite	Null-deref	1	190	25	*2	*11
		2	481	85	*1	4
		3	152	38	6	*17
		4	325	48	0	1
Total			**1,148**	**196**	***9**	***33**

Table 2: For 8 unique null dereference bugs in SQLite and w3m, Semantic Crash Bucketing detects roughly 517 duplicate error reports that built-in fuzzer methods cannot distinguish. Approximate fixes do precisely as well as the real fix, except for Bugs 1 and 3, marked by (*). (*) indicates a small margin of error: just one crashing input is missed by an approximate fix, compared to the true fix.

CrashGen is the starting corpus of duplicate crashing inputs for each unique bug.[6] Each fuzzer is seeded with these inputs and use their internal methods to deduplicate and minimize crashing inputs. AFL uses branch sequences to determine uniqueness, CERT BFF and HonggFuzz use a callstack techniques. Each value in the fuzzer columns indicate the number of duplicate crashing inputs detected by Semantic Crash Bucketing. For example, AFL considers 103 crashing inputs corresponding to **Bug 1** to be unique bugs. Patching (with either the true fix or the approximate fix) reveal these crashing inputs are the *same* bug. In some cases, the fuzzer's built-in method reports 0 duplicates. In these cases, patching does no worse than built-in methods (e.g., **Bug 2** for HFuzz). The main point is that (a) overall, program transformation beats the diverse best effort techniques in current state of art tools, and (b) illustrates that changes can be semantically precise with respect to a bug (i.e., limited to no overfitting).

[6] Details of how these are generated is elided for brevity.

5 Research Timeline

Figure 2 summarizes the timeline for the proposed work. The proposal initiates an 18 month time period to complete all dissertation work and defend the thesis in October 2019. The first research thrust (**1A**) lays the groundwork for semantic matching and validation using a Separation Logic abstraction. I have recently published work that evaluates the approach on 11 C and Java projects [76].

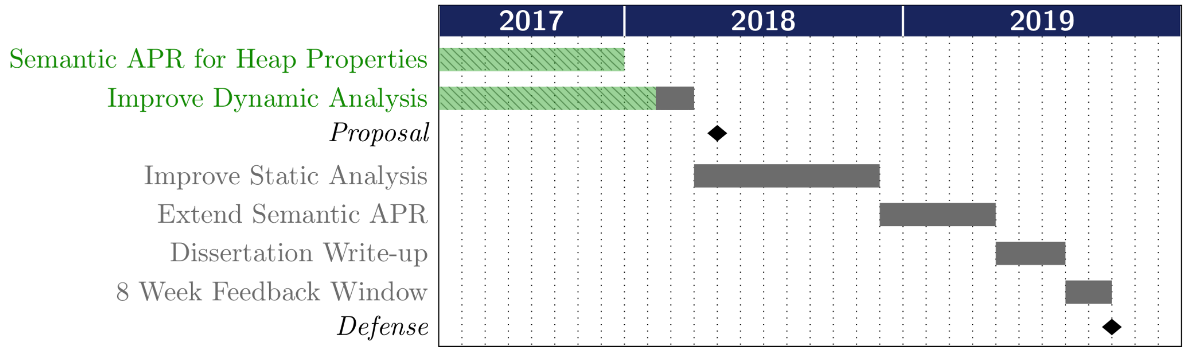

Figure 2: Proposed Research Timeline.

Although not the original intent or focus of the work, it establishes preliminary evidence for improving static analyses (**2A**). I have not begun work on for improving static analysis, which will be my next direction. Some reuse of FootPatch tooling may profit the systematic study proposed by this work. However, an important aspect of improving static analysis is to demonstrate generality for additional tools and techniques. My intent to first work on improving static analysis is partly due to recent progress on improving dynamic analyses (**2B**), wherein I developed tools for flexible syntactic transformation [77]. My work for improving dynamic analyses is currently in preparation and has not been peer reviewed. I have completed a preliminary evaluation of an initial approach for improving dynamic analysis output (deduplication of bug reports). The approach for dynamic analysis improvement may require extension to further classes of bugs, or require extension to other dynamic techniques (e.g., KLEE) if feedback from the first round of peer review (expected July 2018) is unfavorable. I intend to extend the semantic-matching and validation APR work after the "Analysis Improvement" component of the thesis. The two directions are in fact complementary, though no work has been done on extending the Separation Logic APR technique. I am prioritizing static analysis improvement because it presents especially interesting potential with preliminary evidence. Progress in this area may further inform choices for techniques and tooling to adapt for the semantic-based APR extension.

Risks and Mitigations. I identify three main risks. Risk 1: Theoretically promising abstractions proposed for extending semantic-based APR (e.g., Matching Logic) are not fully available, usable, or efficient in current tooling. This poses risk to developing and integrating a semantics-based repair. My mitigation is twofold. First, I will follow a staged process to determine appropriate tooling (cf. Section 3.2); Infer remains in scope for other classes of vulnerabilities and a stable fallback. Second, I will construct bug finding analyses from the ground up with subsets of the theoretical abstractions to support semantic repair. Risk 2: Analysis techniques and tooling for improving static analysis preclude finding previously unknown bugs. To mitigate, I will use historically found bugs and replicate scenarios on older software revisions where no a priori knowledge of a bug was available. Risk 3: the current approach and evaluation of improving dynamic analysis requires additional evidence to support the claims of improving dynamic analysis output. The remedy is straightforward: demonstrate generality to symbolic executors and handle additional bug classes beyond null dereferences and overflows. These risks are ranked in Figure 3.

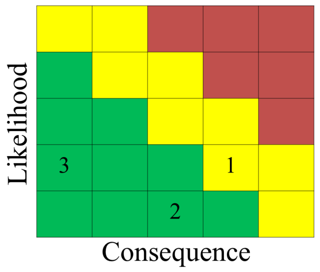

Figure 3

6 Conclusion

The goal of Automated Program Repair is ultimately to cut costs of software quality assurance (QA). However, APR technology currently falls behind other first-class QA tools like linters, gradual type checkers, and static analyzers. To become a first-class concern, APR must present compelling real-world utility. The goal of this proposal is to advance APR technology and demonstrate real-world application to (a) fixing bugs (specifically without tests) and (b) improving existing analyses. The proposed thrusts enable the (largely unprecedented) automated fixing of previously unknown bugs and present the (currently unexplored) ability to improve existing QA tools. To achieve these goals, I propose semantic-driven search and application of program transformations using logic-based domains to validate program changes. I argue that semantic-driven approaches are key to discovering and validating program transformations for correct repairs and positively augmenting analysis behavior. My hope for this work is that demonstrating relatively small scale, real-world application of new program repair techniques urges greater use of program transformation as a first-class approach for software QA.

Glossary

bug class A bug in our context is a software flaw that leads to an error (i.e., undesirable program behavior); an error is a deviation from expected behavior defined by a validation oracle. A *bug class* refers to a sort of bug that shares an undesirable semantic trait. Bug classes occur across multiple programs in multiple locations. "Null dereference bugs" is a concrete example of a bug class. Analyses generally identify the set of bug classes they detect by name, or with an identifier (e.g., CWE-120 for security bugs). 4, 5

logic-based *Logic-based* implies the use of a logical domain during a procedure. In the context of this paper, such procedures include analyses that use a particular logic domain (e.g., Separation Logic). The term *logic-based* can be used in connection with a validation procedure (i.e., validation with respect to the logical domain of an analysis), or a straightforward application of logical implication (e.g., when applying program transformations to categorize program behavior by applying program transformation). 4, 5

semantic-driven A *semantic-driven process* in the context of this work is understood as a *deterministic* process which derives from considering semantic properties of programs. 4, 5

unassisted An *unassisted* process means no human-in-the-loop is required for a process to complete. In the context of automated repair, this means that from the initial point of bug discovery until the end point of validating an applied patch requires no human intervention. Note that "unassisted" does not preclude one-off, up front manual effort (e.g., specifying patch templates) which is done before the automated process begins. 4

References

[1] https://www.cert.org/vulnerability-analysis/tools/bff-download.cfm, 2017. Online; accessed 25 September 2017.

[2] http://lcamtuf.coredump.cx/afl/, 2017. Online; accessed 25 September 2017.

[3] https://github.com/google/honggfuzz, 2017. Online; accessed 25 September 2017.

[4] FindBugs Static Analyzer. https://github.com/findbugsproject/findbugs, 2017. Online; accessed 26 August 2017.

[5] Infer Static Analyzer. http://fbinfer.com/, 2017. Online; accessed 11 May 2017.

[6] Proving Memory Safety with SeaHorn. http://seahorn.github.io/seahorn/memory%20safety/2017/05/27/seahorn-memory-safety.html, 2017. Online; accessed 1 Dec 2017.

[7] Resource Leak in C. http://fbinfer.com/docs/infer-bug-types.html#RESOURCE_LEAK, 2017. Online; accessed 16 January 2017.

[8] The Clang Static Analyzer. https://clang-analyzer.llvm.org/, 2017. Online; accessed 1 Dec 2017.

[9] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 225–236, 2014.

[10] Irina Mariuca Asavoae, Mihail Asavoae, and Dorel Lucanu. Path directed symbolic execution in the K framework. In *SYNASC*, pages 133–141. IEEE Computer Society, 2010.

[11] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing Symbolic Execution with {Veritesting}. In *Proceedings of the International Conference on Software Engineering*, ICSE '14, pages 1083–1094. ACM, 2014.

[12] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Experiences using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008. Special issue on software development tools, September/October (25:5).

[13] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004.

[14] J Berdine, C Calcagno, and Peter W O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*, FMCO '05, pages 115–137, 2005.

[15] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems*, APLAS '05, pages 52–68, 2005.

[16] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[17] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 209–224, 2008.

[18] Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, NFM '11, pages 459–465, 2011.

[19] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods*, NFM '15, pages 3–11, 2015.

[20] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

[21] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for c programs. In *Source Code Analysis and Manipulation*, SCAM '09, pages 123–124. IEEE, 2009.

[22] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Notices*, volume 48, pages 197–208. ACM, 2013.

[23] Maria Christakis, Peter Müller, and Valentin Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, volume 8931 of *Lecture Notes in Computer Science*, pages 336–354. Springer, 2015.

[24] Maria Christakis and Valentin Wüstholz. Bounded abstract interpretation. In *Static Analysis Symposium*, pages 105–125, 2016.

[25] Zack Coker and Munawar Hafiz. Program transformations to fix C integers. In *International Conference on Software Engineering*, ICSE '13, pages 792–801, 2013.

[26] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. Retracer: triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, pages 820–831. ACM, 2016.

[27] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - A software analysis perspective. In *Software Engineering and Formal Methods*, SEFM '12, pages 233–247, 2012.

[28] Andrei Marian Dan, Manu Sridharan, Satish Chandra, Jean-Baptiste Jeannin, and Martin T. Vechev. Finding fix locations for cfl-reachability analyses via minimum cuts. In *Computer Aided Verification*, CAV '17, pages 521–541, 2017.

[29] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 435–445, 2007.

[30] Dino Distefano and Ivana Filipovic. Memory Leaks Detection in Java by Bi-abductive Inference. In *Fundamental Approaches to Software Engineering*, FASE, pages 278–292, 2010.

[31] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software*, pages 10–30, 2010.

[32] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for C programs. In *International Conference on Software Engineering*, ICSE '15, pages 459–470, 2015.

[33] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *Computer Aided Verification*, CAV '15, pages 343–361, 2015.

[34] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Rosu. Rv-match: Practical semantics-based program analysis. In *CAV (1)*, volume 9779 of *CAV '16*, pages 447–453. Springer, 2016.

[35] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program Repair as a Game. In *Computer Aided Verification*, CAV '05, pages 226–238, 2005.

[36] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering*, ICSE '13, pages 672–681, 2013.

[37] Temesghen Kahsai, Jorge A. Navas, Dejan Jovanovic, and Martin Schäf. Finding inconsistencies in programs with loops. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 499–514, 2015.

[38] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing Programs with Semantic Code Search. In *International Conference on Automated Software Engineering*, ASE '15, pages 295–306, 2016.

[39] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[40] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive Program Repair. In *Computer Aided Verification*, CAV '15, pages 217–233, 2015.

[41] Shuvendu K. Lahiri, Rohit Sinha, and Chris Hawblitzel. Automatic rootcausing for program equivalence failures in binaries. In *Computer Aided Verification*, CAV '15, pages 362–379, 2015.

[42] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, dec 1992.

[43] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012.

[44] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software quality journal*, 21(3):421–443, 2013.

[45] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. Autopag: towards automated software patch generation with source code root cause identification and repair. In *ACM Symposium on Information, Computer and Communications Security*, pages 329–340. ACM, 2007.

[46] Peng Liu, Omer Tripp, and Charles Zhang. Grail: context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 318–329, 2014.

[47] Francesco Logozzo and Thomas Ball. Modular and Verified Automatic Program Repair. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '12, pages 133–146, 2012.

[48] Francesco Logozzo and Matthieu Martel. Automatic repair of overflowing expressions with abstract interpretation. In *Festschrift for Dave Schmidt*, volume 129 of *EPTCS*, pages 341–357, 2013.

[49] Fan Long and Martin Rinard. Automatic Patch Generation by Learning Correct Code. In *Principles of Programming Languages*, POPL '16, pages 298–31, 2016.

[50] Fan Long and Martin C. Rinard. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *International Conference on Software Engineering*, ICSE '16, pages 702–713, 2016.

[51] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Conference on Programming Language Design and Implementation*, PLDI '14, pages 227–238, 2014.

[52] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian-Florin Serbanuta, and Grigore Rosu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In *RV*, volume 8734 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2014.

[53] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *USENIX Security Symposium*, pages 1007–1024. USENIX Association, 2017.

[54] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *International Conference on Software Engineering*, ICSE '15, pages 448–458, 2015.

[55] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering*, ICSE '16, pages 691–701, 2016.

[56] D Molnar, XC Li, and DA Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the USENIX Security Symposium*, pages 67–82, 2009.

[57] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):17, 2018.

[58] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. ICSE '13, pages 772–781, 2013.

[59] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *TACAS (1)*, volume 10205 of *Lecture Notes in Computer Science*, pages 301–318, 2017.

[60] National Institute of Standards and Technology. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical Report NIST Planning Report 02-3, NIST, 2002.

[61] Peter O'Hearn. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[62] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, CSL '01, pages 1–19, 2001.

[63] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

[64] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically Patching Errors in Deployed Software. In *Symposium on Operating Systems Principles*, SIGOPS '09, pages 87–102, 2009.

[65] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. Bucketing failing tests via symbolic analysis. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 43–59, 2017.

[66] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[67] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, volume 4, pages 21–21, 2004.

[68] Grigore Rosu. Matching logic-extended abstract (invited talk). In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 36. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[69] Grigore Rosu. $\mathbb{K}$: A semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering*, pages 186–206. 2017.

[70] Grigore Rosu. Matching logic. *Logical Methods in Computer Science*, 13(4), 2017.

[71] Grigore Rosu and Andrei Stefanescu. From hoare logic to matching logic reachability. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 387–402, 2012.

[72] Grigore Rosu, Andrei Stefanescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 358–367, 2013.

[73] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, pages 532–543, 2015.

[74] Andrei Stefanescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Rosu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 74–91, 2016.

[75] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *International Conference on Software Engineering*, ICSE '15, pages 471–482, 2015.

[76] Rijnard van Tonder and Claire Le Goues. Static Automated Repair for Heap Properties. In *International Conference on Software Engineering*, ICSE '18, 2018.

[77] Rijnard van Tonder, Chris Timperley, and Claire Le Goues. Experience Report: Parser Parser Combinators for Structural Matching and Syntactic Rewriting. In *International Conference on Functional Programming, in submission*, ICFP '18, 2018.

[78] Christian von Essen and Barbara Jobstmann. Program Repair without Regret. *Formal Methods in System Design*, 47(1):26–50, 2015.

[79] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2):8:1–8:51, March 2008.

[80] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[81] Hongseok Yang and Peter O'Hearn. A Semantic Basis for Local Reasoning. In *International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS, pages 402–416, 2002.

7 Supplementary: Executive Summary of Proposed Work

I summarize the remaining proposed work to complete the dissertation. The research that remains to be done is expected to produce roughly two publishable units.

7.1 Extension of Semantic Matching and Validation for APR

I will extend semantic-driven matching, application, and validation for at least one bug class (but possibly more). The scope of possible bug classes include at least the following bugs:

- Buffer overflows
- Integer underflow/overflows
- Use-after-free/Double-free bugs
- Division-by-zero bugs
- Information flow bugs
- Concurrency bugs
- Uninitialized variables
- Foreign Function Interface (FFI) bugs (e.g., FFI leaks)

I strongly anticipate that the work will address one of these classes. However, I may branch to a different class not enumerated here if appeal presents itself. I've identified at least the three existing tools to implement the repair approach: the $\mathbb{K}$ framework, SeaHorn, and Infer. These frameworks are not exclusive, and I may branch to a different one if appeal presents itself.

The approach will be **efficient**, **scalable**, and **unassisted**. In the context of this work, these qualifiers mean the following:

- **Efficient.** The repair procedure for fixing a bug terminates in a time comparable to the original analysis. Performing repair requires at least running the original analysis, and a rerun of the analysis to validate one or more program changes. The repair procedure itself also requires time to run. Hence, a "comparable" runtime implies that the repair procedure terminates in the order of 1x to 10x of the original analysis runtime.
- **Scalable.** The procedure works on programs in the order of 100,000 lines of code. I anticipate evaluating on projects containing at least 100KLOC. However if the largest project used in the evaluation is, for example, 50KLOC to 80 KLOC, I consider it sufficient to validate the ability to scale to 100KLOC (i.e., it is in the order of 100KLOC).
- **Unassisted.** Modulo up front specification and configuration, the procedure should automatically produce validated patches without human assistance.

I will evaluate the approach on 5 to 15 open source projects. I strongly anticipate that the extension will be able to fix previously unknown bugs. However, lack of fixing previously unknown bugs will not invalidate the overall claim of the thrust and thesis because (a) I demonstrate application to previous unknown bugs in prior work that forms part of the claim (**1A**) and (b) I foremost aim to demonstrate the claim on real-world programs and can use historic bugs to validate applicability to previously unknown bugs (i.e., by using older project revisions to construct a scenario where there is no prior knowledge of a bug before running an analysis and applying repair).

7.2 Improve Static Analysis Effectiveness

The expected output of this work is to produce the first systematic study that explores, develops, and evaluates program transformations applied to analysis targets to improve the results of real-world static analyses on those targets. Improvement refers to improving analysis output (not the implementation of the analysis itself) in the form of (a) more true positive bug reports, (b) fewer false positive bug reports, (c) more true automated bug fixes. I aim to demonstrate an at minimum quantitative **improvement** as follows: I will show in the order of 5-10% improvement for any of (a)-(c) on a convenience sample of 5 to 15 open source projects. In the unlikely case this metric is unmet, the study will still present a novel contribution documenting the applicability of the proposed approach, and the extent of improvement possible. As an additional **risk mitigation**, I propose to substitute the primary **improvement** metric for improving the analysis *runtime* using program modification and aim to demonstrate at least a 2x analysis speedup for one or more bug classes reported by the original analysis.

I anticipate that autogenerated fixes in Thrust 1 can, in part, realize these benefits. However, I anticipate (due to my existing preliminary work improving dynamic analyses in **1B**) that bug- and analysis-specific template program transformations are needed to fully realize the benefits. Additional example program transformations, which I anticipate can reveal or overcome imprecision in existing analyses, include:

- Releasing a resource twice or removing correct calls that release a resource
- Adding or removing null checks
- Adding or removing loop constructs
- Changing array bounds or bound checks

The proposed work will identify and use these kinds of transformations to improve analysis output. Both the discovery of such program transformations (either automatically inferred or manually specified up front or both) and their application is expected to be novel contributions of the proposed work. I will evaluate on at least two existing static analyses to show that the approach generalizes, but possibly more. At least the following existing analyses are in scope, though I may branch to others if appeal presents itself.

- Clang static analyzer
- NullAway
- Infer
- Frama-C
- SeaHorn
- CodeContracts
- DR.CHECKER

The approach will be **efficient**, **scalable**, and **unassisted**. In the context of this work, these qualifiers mean the following:

- **Efficient.** Similar to the above: producing patches to improve analysis terminates in the order of 1x to 10x of the original analysis time. I consider the approach efficient if the procedure runtime is at least in this range *per* discrete unit of improvement (i.e., a single newly reported bug, fixed bug, or false positive removed). My rationale is that rerunning static analyses is generally cheap: if it takes just one or two minutes to report a new bug or produce and validate a program transformation automatically, it is still competitive (or better than) human effort. I anticipate that efficiency can be improved with caching or by applying multiple patches at once, but leave this exploration open based on need. The failure mode where I substitute analysis runtime as the primary metric will by definition be efficient (at least 2x analysis speedup for one or more bug classes by the original analysis).

- **Scalable.** The same as above.
- **Unassisted.** Modulo up front specification and configuration, the procedure should automatically produce patches that improve the analysis output.

I will focus on demonstrating improvement for previously unknown bugs or false positives. However, it can turn out that this is difficult to demonstrate for the latest revisions of the project selection. As a risk mitigation strategy, I will consider historic bugs (i.e., by using older project revisions) to validate the ability to improve analysis in a scenario where there is no prior knowledge of analysis output.