Static Automated Program Repair for Heap Properties

Rijnard van Tonder
School of Computer Science
Carnegie Mellon University
rvt@cs.cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
clegoues@cs.cmu.edu

ABSTRACT

Static analysis tools have demonstrated effectiveness at finding bugs in real world code. Such tools are increasingly widely adopted to improve software quality in practice. Automated Program Repair (APR) has the potential to further cut down on the cost of improving software quality. However, there is a disconnect between these effective bug-finding tools and APR. Recent advances in APR rely on test cases, making them inapplicable to newly discovered bugs or bugs difficult to test for deterministically (like memory leaks). Additionally, the quality of patches generated to satisfy a test suite is a key challenge. We address these challenges by adapting advances in practical static analysis and verification techniques to enable a new technique that finds and then accurately fixes real bugs without test cases. We present a new automated program repair technique using Separation Logic. At a high-level, our technique reasons over semantic effects of existing program fragments to fix faults related to general pointer safety properties: resource leaks, memory leaks, and null dereferences. The procedure automatically translates identified fragments into source-level patches, and verifies patch correctness with respect to reported faults. In this work we conduct the largest study of automatically fixing undiscovered bugs in real-world code to date. We demonstrate our approach by correctly fixing 55 bugs, including 11 previously undiscovered bugs, in 11 real-world projects.

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**; **Maintaining software**; **Software defect analysis**;

KEYWORDS

Automated Program Repair, Separation Logic

ACM Reference Format:
Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180250

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180250

1 INTRODUCTION

Software bugs are expensive and time-consuming [15, 43], motivating research to find and fix them automatically. Research in automated program repair (APR) holds promise for reducing software maintenance costs due to buggy code. Considered broadly, a *program repair* is simply a transformation that improves a program with respect to some abstract domain that describes correct versus incorrect program behavior. The vast majority of modern repair techniques (e.g., [29, 33, 37, 39, 41, 49]) use test cases to construct this domain. Tests are appealing because they are intuitive and widely-used in practice (more so than, e.g., formal specifications) and can straightforwardly indicate whether a given change improves the program in question (i.e., by turning a previously failing test case into a passing one).

However, tests are limiting in several ways, especially (though not exclusively) for APR. Writing good tests is nontrivial [43], rendering some real-world suites a weak proxy for patch correctness [51]. APR techniques and humans alike can overfit to even high quality tests, producing patches that do not generalize to the true desired functionality change [53]. Developers must write a deterministic, reproducible test case corresponding to a bug under repair to use test-driven APR. This use case is particularly applicable to, e.g., regressions, but is limiting for previously-unknown defects.

More fundamentally, tests are only suitable for finding and guiding the repair of certain kinds of bugs. Some bug types are simply difficult to test for in a finite, deterministic way [40]. Consider concurrency errors or resource or memory leaks: Figure 1a shows an example memory leak from `Swoole`[1] (line 11), which may be fixed by adding a call to the project-specific resource allocation wrapper `sw_free` (Figure 1b). Alternatively, consider the code in Figure 2a, from `error-prone`.[2] 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* had been made on lines 10–12, in the same `switch` statement, but was not fixed for another 18 months. Even when bugs are deterministically reproducible, tests usually cannot identify recurring mistakes like this.

Finding and fixing these types of bugs motivate the use of QA techniques beyond testing. Considerable recent progress has been made in expressive, high quality static analyses that can cost effectively find real bugs like these examples in real programs [5, 14]. Companies like Ebay [25], Microsoft [14], Facebook [17], and others are publicizing their development and use of static analysis

[1] `Swoole` is a popular event-driven networking engine for PHP, https://github.com/swoole/swoole-src
[2] `error-prone` is an open-source static analysis tool, https://github.com/google/error-prone

```
    swHashMap *hmap =
      sw_malloc(sizeof(swHashMap));
    if (!hmap) {
      swWarn("malloc[1] failed.");
      return NULL;
    }
    swHashMap_node *root =
      sw_malloc(sizeof(swHashMap_node));
    if (!root) {
      swWarn("malloc[2] failed.");
      return NULL; // returns, hmap not freed
    }
```

(a) Memory leak: forgetting to free memory before return in `Swoole`.

```
#define sw_free(ptr)
if(ptr) {
  free(ptr);
  ptr=NULL;
  swWarn("free");
}
```

(b) `sw_free` wraps a call to `free`.

Figure 1: Fixing a memory leak in the `Swoole` project

tools in engineering practice. Some bug-finding tools, like **`error-prone`** or **`Findbugs`** [8] even provide "quick fix" suggestions for certain bug classes, simplifying the process of repairing them. Developers find such suggestions very useful [9], and indeed their absence has been identified as a barrier to uptake and utility of static analysis tools [27]. However, the suggestions present in current tools are simple, generic, and usually syntactic (e.g., recommending the addition of **`static`** or **`final`** modifiers to variable or function declarations). Moreover, they provide no semantic correctness guarantees.

We propose a new technique that automatically generates patches for bugs in large projects without a need for either tests or developer annotation, for a semantically rich class of bugs that can be identified by recent, sophisticated static bug-finding techniques based on Separation Logic [10, 11, 13, 21]. Our key insight lies in the novel way we adapt *local reasoning* [44, 46, 62], "the workhorse of a number of automated reasoning tools" [45], to pull out relevant parts of program state, and then search for repairing code from elsewhere in the same program. At a high level, our technique searches for and adapts program fragments satisfying a generic, pre-specified semantic effect that address a given bug class (such as "if the file is open in the precondition of a program fragment, it should be closed in the postcondition" to address resource leaks, like the one shown in Figure 1a). These fix effects are generic, and need only be specified once per bug class. They are also language- and API-agnostic, which means our approach applies off-the-shelf to multiple source languages, and its patches automatically conform to the programming conventions in a given project (e.g., constructing a patch using a project-specific custom resource handler like **`sw_free`** for memory leaks, if available, or **`free`**, if not), without requiring any additional customization.

We instantiate our approach in a tool called FootPatch, an extension to the Infer [17][3] static analysis tool. Infer finds bugs by automatically inferring Separation Logic assertions over program

[3]https://github.com/facebook/infer

```
case IDENTIFIER: {
  MethodSymbol mtd =
    resolver.resolveMethod(node, id.getName());
  // method may be null
+ checkGuardedBy(mtd != null, id.toString());
  return bindSelect(computeBase(context, mtd), mtd);
}
case MEMBER_SELECT: {
...
  MethodSymbol mtd =
    resolver.resolveMethod(node, id.getName());
  // same problem!
  return bindSelect(computeBase(context, mtd), mtd);
}
```

(a) Developers fixed the potential null pointer exception on line 6; 18 months later, they addressed the very similar bug on lines 12–13.

```
public static void checkGuardedBy(boolean cnd,
 String fmtStr, Object... fmtArgs) {
  if (!cnd) {
    throw new IllegalGuardedBy(String.format(fmtStr,
        fmtArgs));
  }
}
```

(b) error-prone's custom guard handler.

Figure 2: Fixing a null dereference in Google's error-prone tool.

statements. Infer reasons over a semantic, analysis-oriented Intermediate Language (IL), and applies to large, real-world programs written in multiple source languages. Separation Logic can be used to encode a variety of desirable correctness properties [17, 22]. We situate our approach by extending analyses that find bugs related to the violation of pointer safety properties, the focus of Infer. In this paper, we restrict our focus to constructing additive patches for resource leaks, memory leaks, and null dereferences. We discuss directions for generalizing our technique in Section 4.5.

Our approach provides several important benefits over previous techniques for automatic patch generation or fix suggestion. By integrating directly into the static analysis workflow, our approach addresses different types of bugs than are handled by most dynamic APR techniques, and can encourage the adoption of these robust static bug finding tools in practice [27]. Because of the way our approach uses compositional specifications, it can produce fixes that are significantly semantically richer than existing static "quick fix" suggestions. FootPatch can construct repairs that cross procedure boundaries, entail multiline fixes, and are robust to program-specific customization like wrapper APIs. These benefits are evident in the above examples, both of which FootPatch can repair automatically. Note, for example, a call to **`checkGuardedBy`** does not on its own constitute a repair, as it simply checks the results of a boolean expression. FootPatch can determine that the function implements the desired behavior because it searches over compositional function call results. Additionally, both bug fixes use custom resource wrappers, which are often desirable as fixes because they are consistent with the convention in other parts of the program. For example, **`sw_free`** wraps the **`free`** function, and performs additional, non-interfering operations by setting the pointer to null

and logging a debug message. Finally, unlike previous repair techniques that build on more formal abstract domains [19, 30, 35, 48], our approach scales to real-world programs, automatically instantiates and applies its patches, and relies on a principled semantic treatment to argue patch correctness and prevent patch overfitting. That is: FootPatch demonstrates a promising and previously underdeveloped application of *end-to-end* identification and repair of previously undiscovered bugs in real programs.

Our contributions are the following:

- **Program Repair with Separation Logic.** We present a repair technique using Separation Logic to ensure a desired correctness property based on pointer safety. The abstract domain provides a basis for reasoning about explicit semantic effects introduced by patch fragments, and enables a formal argument for semantic patch correctness.
- **Repair Extraction and Application Formalism.** We formalize the search and extraction of program fragments with respect to a repair specification, and define the conditions for patch generation and automatic patch application with respect to a detected bug.
- **Evaluation.** We present an evaluation on popular software projects. Our approach fixes 24 resource leaks, 7 memory leaks, and 24 null dereferences in popular Java and C programs, including 11 previously undiscovered bugs. We are unaware of any prior repair tool that supports multiple languages under a single analysis. Our implementation runs on big projects (> 200 kLOCs). Run time ranges from 7 seconds to 21 minutes per project, to perform both finding and fixing bugs within the project. For applicable projects, our experiments show that the majority of correct patches (53) are found by searching for repair candidates that are callgraph-local to the bug, and expanding repair search to the project globally fixes 2 additional bugs. We observe a false positive rate of only 5% for fixes. Moreover, we demonstrate anecdotal evidence that our technique can lead the original static analysis to discover more bugs after performing repair.
- **Open Source Repair Tool.** We implement our technique in a tool called FootPatch, built on top of the open source Infer static analyzer.[4]

Section 2 provides background theory underpinning our approach. Section 3 details our repair approach using Separation Logic. Section 4 evaluates FootPatch, a tool that implements our approach. Section 5 discusses related work; Section 6 concludes.

2 PRELIMINARIES

We build our approach on top of the analysis engine used in Infer [16], an open source framework that uses Separation Logic and Hoare-style reasoning to scalably find bugs, particularly those related to heap or pointer errors. This analysis 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). This section outlines background concepts underpinning our approach and the analysis it extends: the abstract program model and Separation Logic assertions (Section 2.1),

[4]https://github.com/squaresLab/footpatch

$$E ::= x \mid nil \mid c$$
$$B ::= E = E \mid E \neq E$$
$$S ::= x := E \mid x := [E] \mid [E] := E \mid x := \textbf{new}() \mid \textbf{dispose}(E)$$
$$C ::= S \mid C; C \mid \text{if } (B)\ \{C\} \text{ else } \{C\} \mid \text{while}(B)\ [I]\ \{C\} \mid x := f(\overrightarrow{E})$$
$$\mathbb{P} ::= \cdot \mid f(\overrightarrow{E})\{\textbf{local}\overrightarrow{E}; C; \text{return } E\}; \mathbb{P}$$

(a) A simplified Smallfoot grammar, for illustration.

$$H ::= E \mapsto E$$
$$\Sigma ::= H_1 * \cdots * H_n \mid \text{emp}$$
$$\Pi ::= B_1 \wedge \cdots \wedge B_n \mid \text{true} \mid \text{false}$$
$$P, Q ::= \Pi \wedge \Sigma \mid \text{if } B \text{ then } P \text{ else } P$$

(b) The assertion language grammar.

Figure 3

the frame inference procedure for discovering specifications (Section 2.2), and an overview of how the concepts fit together to find bugs statically in real-world programs (Section 2.3).

2.1 Program Model and Assertion Language

Infer and our analysis both begin by abstracting a source program in one of several languages (e.g., Java, C, C++) to the Smallfoot Intermediate Language (SIL) [10]. SIL is an intermediate analysis language that represents source programs in a simpler instruction set describing the program's effect on symbolic heaps. This is particularly suitable for static analyses that find bugs related to pointer safety properties.

Program Model. Figure 3a shows a simplifed Smallfoot grammar to illustrate the overall program model. A SIL program $\mathbb{P}$ is a set of procedures [11, 12, 18]. SIL procedures have single return values and do not access global variables. Procedures consist of a series of *commands* (C, in Figure 3a), which model actions that generate assertions over symbolic heaps (described next). The storage model is fairly standard [18, 47]: `Heap` is a partial function from locations `Loc` to values `Val` (for simplicity, locations are positive integers and values are integers): $\texttt{Heap} \stackrel{def}{=} \texttt{Loc} \rightharpoonup \texttt{Val}$. `Stack` is a function from variables to values $\texttt{Stack} \stackrel{def}{=} (\texttt{Var} \cup \texttt{LVar}) \rightharpoonup \texttt{Val}$. Variables are two disjoint sets: a set of program variables `Var` and a set of logical variables `LVar`. A program `State` is simply the combination of its heap and stack: $\texttt{State} \stackrel{def}{=} \texttt{Stack} \times \texttt{Heap}$.

Assertions. SIL commands primarily capture effects over symbolic heaps, which comprise the abstract domain for detecting faulting conditions (e.g., memory leaks, resource leaks, null dereferences).[5] These effects can be described via pre- and postconditions expressed in Separation Logic, which decorate the SIL commands accordingly. Figure 3b shows the assertion language grammar. The grammar encodes heap facts using points-to heap predicates over program and logical variables (i.e., $E \mapsto E$). Heap predicates are considered "separate" sub-heaps (or heaplets), whose separation is denoted by

[5]Symbolic heaps enable a decidable proof system for entailments under a prescribed semantics [10]; we elide details for brevity.

the *separating conjunction* $*$ (read "and separately"). The separating conjunction implies that the two sub-heaps are disjoint. Pure boolean predicates of the form $B_1 \wedge \cdots \wedge B_n$ assert conditional facts over heap predicates (e.g., $E = nil$).

Given our focus on repair, assertions are relevant insofar as they describe semantic effects of statements (as predicates) on the heap. We denote by $E \Rightarrow alloced$ a predicate *alloced* on E (e.g., we may represent it in the grammar by the assertion $E \mapsto x \wedge true$ for x fresh). For simplicity, this notation may assume a program variable evaluates to a heap location (such as `hmap`); we do this with the understanding that stores and heaps are typically treated separately in the storage model [18].

2.2 Frame Inference

A key novelty in our work is the way we extend *frame inference* [11] to find bug repairs; we thus briefly overview frames and frame inference in this context. Infer's Separation Logic-based analysis uses Hoare-style reasoning to find specification violations. It does this at scale by summarizing the effects of individual terms in a procedure, and then composing them into procedure-level specifications. *Local reasoning* [44, 46] is used to summarize the effects of those individual terms. Local reasoning is enabled by the fact that a program command can often affect only a sub-part of the heap. For example, the statement `hmap = sw_malloc(sizeof(swHashMap))` is modeled as affecting only `hmap`; the rest of the heap is unaffected by the allocation. The fact that sub-states can change in isolation is modeled by the separating conjunction. The unchanged part of the heap for a command is its *frame*; the parts of the heap a command changes is known as its *footprint*.

Thus, *frame inference*, which automatically infers command frames such that they can be composed efficiently into procedure summaries [18][6] is key to a number of automated reasoning tools [45]. By discovering the unchanged heap portion of an operation, frame inference discovers footprints, expressed as small specifications of program terms [11, 22, 42].

More formally, the Frame Rule codifies the notion of reasoning over local behavior:

$$\frac{\{P\}\ C\ \{Q\}}{\{P\ *\ F\}\ C\ \{Q\ *\ F\}}\ \textsc{Frame Rule}$$

The Frame Rule 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 $\{P * F\}$. Frame inference [11, 18, 20] discovers a frame F that allows the Frame Rule to fire, enabling local reasoning over the footprint. We can summarize frame inference as follows. Let $\textsc{Frame}(H, S)$ be the frame inference procedure that returns a frame F (if it succeeds) for a given heap state H and a specification S of a command (expressed in the grammar of Figure 3b). The procedure consists of a proof system using subtraction and normalization rules to partition the heap H into $S * F$. We refer to previous work for the complete algorithm and proof system [11].

[6] For completeness, this compositional analysis also infers anti-frames, or the missing parts of the heaps state. Anti-frame inference allows the analysis to deal with unknown calling contexts and increases precision by propagating intermediate results. Anti-frame inference is not critical to our approach.

2.3 Finding bugs using Separation Logic

The previously described reasoning enables scalable, compositional static bug finding with minimal developer effort (in the form of, e.g., annotations or customization) over real-world code bases. Infer implements these by ideas by converting source programs into SIL, and then infers specifications (described as symbolic heaps) for SIL program fragments. It discovers bugs by symbolically executing SIL commands over symbolic heaps, according to a set of operational rules that update symbolic heap assertions [11]. The general goal of this static analysis is to discover program paths with symbolic heaps that violate heap-based properties.

Infer currently supports detecting a wide number of heap-related bugs using Separation Logic: resource leaks, memory leaks, null dereferences, as well as experimental support for buffer overflow detection, thread safety, and taint-style information flow bugs (e.g., detecting SQL injections for unsanitized values) [3, 4]. While we believe our approach generalizes to these bug types, in this work focus on pointer-safety properties of heaps for repair, specifically resource leaks, memory leaks, and null dereferences, as the support for them in Infer is most mature. We leave extension of our approach to further classes of bugs for future work.

To illustrate, consider the memory leak described in Figure 1a. Infer disovers this error by identifying the path through line 11 where the variable `hmap` is allocated but not freed before becoming dead. When it discovers such an error, the symbolic interpreter enters a special state, `fault`. We denote this formally by $C_\ell, \sigma \rightsquigarrow$ `fault`, meaning that the interpretation step $\rightsquigarrow$ for instruction C at location ℓ in symbolic state σ results in a `fault`. For this example, `hmap` is still allocated in the symbolic heap (denoted by the predicate *alloced*, i.e., $\{\texttt{hmap} \Rightarrow alloced\}$) at the location $\ell = 11$ when it becomes dead: $\texttt{return}_\ell, \{\texttt{hmap} \Rightarrow alloced\} \rightsquigarrow \texttt{fault}$.[7] At this point, our approach takes over from the bug-finding analysis, to seek a potential fix.

3 REPAIR WITH SEPARATION LOGIC

This section presents our program repair technique using Separation Logic. Section 3.1 formalizes our notion of repair with respect to heap-based property violations. In Section 3.2 we formalize the search procedure to discover candidate patch code, drawn from existing program fragments (i.e., from elsewhere in the program under repair). In Section 3.3 we describe the application of candidate fragments in source code, in terms of where to introduce a change and how we filter out invalid candidates. We illustrate throughout by referring to the motivating example in Figure 1.

3.1 Formulating Repair

Fundamentally, any program transformation (for repair or otherwise) is composed of either one or a combination of two primitive operations: addition and removal of program terms. Taking Separation Logic as the abstract domain, a bug fix corresponds to a program transformation that leads to a *fault-avoiding interpretation* in the analysis with respect to the property in question. We presently consider only *additive* program transformations, and do not perform removal operations, because the types of bugs we consider are typically caused by the *lack* of certain operations on

[7] For illustration, we consider only the predicate on `hmap`, ignoring the rest of the state.

explicit heap content (e.g., resource release, freeing memory, or checking nullness of return values). For instance, developers often forget to insert missing checks [6, 60]. Bug-fixing changes for these types of bugs thus correspond to inserting missing statements (e.g., checks, initializations, or cleanup handlers). Other types of repairs are certainly valuable, in this domain and others, but we leave discovery of them to future work.

The central repair question is thus: Does there exist a fragment $?C$ that can be used to transform the program such that the `fault` state is no longer triggered? We express this formally as follows:

Definition 3.1. **Program Repair.** Let $\mathbb{H}_{Bad}$ be the heap configuration that results in a fault under interpretation of a command C at location ℓ. A repair is an additive transformation T on a program $\mathbb{P}$ that transitions the fault-inducing predicate in the heap state $\mathbb{H}_{Bad}$ to a heap state $\mathbb{H}_{Good}$ that preserves a fault-avoiding interpretation for C at ℓ. A repair satisfies:

$$C_\ell, \mathbb{H}_{Bad} \rightsquigarrow \texttt{fault} \implies T_{\ell'}, \mathbb{H} \overset{*}{\rightsquigarrow} C_\ell, \mathbb{H}_{Good} \not\rightsquigarrow \texttt{fault}$$

The additive transformation T is a program fragment (operating on the program heap $\mathbb{H}$ at some point ℓ') that induces a desired *fix effect* on the heap, producing $\mathbb{H}_{Good}$ at ℓ. The fix effect precisely defines what it means to avoid the `fault` state.

We encode the transformation T (i.e., the fix effect) as a Hoare-style triple that we call a Repair Specification. We specify T as two singleton heaps (i.e., a single points-to relation as in Figure 3b): F mapping to a *fixable* predicate over a placeholder variable *pvar* in the precondition, and a corresponding F' mapping to a *fixed* predicate over that placeholder *pvar* variable in the postcondition. F and F' express the desired symbolic transition on the abstract predicates. For example, $F = \{pvar \Mapsto alloced\}, F' = \{pvar \Mapsto freed\}$, specifies a generic fix effect for memory leaks over placeholder variable *pvar*. Note that such fix effects are generic to entire bug classes. By expressing repair over the abstract domain describing what the code *does*, this fix effect specification approach is multi-language and resilient to syntactic customizations (like APIs or wrapper functions).

Definition 3.2. **Repair Specification**. A Repair Specification R is a specification containing a program term **repair fragment** C_R (a command C in the Smallfoot grammar) that effects a state transition from an error heap configuration F to a fixed heap configuration F', denoted $\{F\}\ C_R\ \{F'\}$, via an atomic update of an abstract predicate.

In this paper, we manually specify appropriate F and F' corresponding to the general bug classes in question. Defining a mechanism to automatically determine F and F' (e.g., by formally deriving it from a violation reported by an analysis) is an interesting and plausible research direction that we leave for future work. Note however, that fix effects are generic to an entire bug class, and thus need only be specified once per analysis type to be applied to a given program. Moreover, the static analysis provides a degree of confidence in the choice of F and F': a poor choice will not ensure a fault-avoiding interpretation for a particular fault, and will be detected by analysis of the transformed program.

Our implementation provides default fix effects for the bug classes we consider that suffice for many off-the-shelf projects, requiring customization only when a project uses a particular or unique paradigm for handling, e.g., custom exceptions. Extending our technique to new static checkers (employing automated semantic reasoning as found in Infer) could similarly involve specifying default fix effects for patching them, eliminating the need for developer-provided specifications for many real-world projects.

3.2 Searching for Repairs

Repair Queries. Different automated techniques can discover repairs, including program synthesis [30] or syntactic program mutation [33]. Our technique does so by searching over existing program fragments, which can often exhibit the desired semantic effects to fix faults [30, 31, 37]. Using existing program fragments also preserves program-specific syntactic structure and semantic shape that accompany a fix, and may decrease the risk of overfitting repairs [29]. We thus search for program fragments C_R in the Smallfoot IL from across the rest of the program with a *Repair Query*. A Repair Query encodes the desired semantic transition and returns satisfying Repair Specifications.

We illustrate this overall framing with respect to our running example in Figure 4: The computation in (1) shows the semantic change that must be induced by some $?C_\ell$ to preserve a fault-avoiding interpretation, fixing the memory leak bug in question. The computation in (1) informs the repair question labeled (2) in Figure 4. The specification in (2) describes a desired program fragment $?C$ that induces the desired fix effect on symbolic variable *pvar*, which is allocated on precondition to $?C$, and freed on post-condition. Note that this fix effect is flexible, and could describe fix code such as **`free`** modeled in generic C or the custom **`sw_free`** function. A Repair Query seeking to repair a file resource leak, on the other hand, could find fragments such as **`close`** or **`fclose`** in C, or **`f.close()`** for a file **`f`** modeled in Java. Although the fix effects must be either inferred or specified, their portability across multiple programs and languages amortizes the manual burden and represents an important improvement over the labor required to use test-based APR techniques, which require a triggering test case per bug under repair.

$$?C_{\ell'},\ \{\texttt{map} \Mapsto alloced\} \rightsquigarrow \texttt{return}_\ell,\ \{\texttt{map} \Mapsto freed\} \not\rightsquigarrow \texttt{fault} \quad (1)$$

$$\{pvar \Mapsto alloced\}\ ?C\ \ \{pvar \Mapsto freed\} \quad (2)$$

Figure 4: Modeling repair search.

We call (2) the Repair Query, a Hoare triple containing a "hole" for an instruction C that induces the desired semantic change under the standard partial correctness interpretation. For example, the fixing fragment **`sw_free(map);`** corresponds in the IL to a command C of the form $\{\texttt{map} \Mapsto alloced\}\ call(\cdot)\ \{\texttt{map} \Mapsto freed\}$, with the concrete program variable `map` bound to *pvar*.

Semantic Search Constraints. A Repair Query expresses a *symbolic* transition on the abstract predicates, providing a basic structure that expresses fixes in terms of desired semantic heap properties. So far, our example Repair Query specifies C_R program fragments that perform strictly the desired symbolic transition,

disallowing any C_R that may introduce extra semantic effects. However, as a practical consideration, it may be desirable to search for program fragments that introduce extra semantic effects in addition to the fixing effect. For example, `sw_free` performs cleanup and logging in addition to resource freeing; in other cases, such side effects may be benign, or undesirable.

Fortunately, Separation Logic allows us to elegantly relax strict Repair Queries by explicitly allowing for and then capturing extra semantic effects beyond the desired fix effect. This produces a repair search method that can be parameterized by extra semantic effects, explicitly partitioning such effects from the fixing effect (absent a priori knowledge to the fixing effect). Such a search returns relaxed Repair Specifications of the form $\{F * P\}\ C_R\ \{F' * Q\}$, where (potential) extra semantic effects are bound in the pre- (P) or postcondition (Q) respectively. Finding satisfying Repair Specifications comes with an analogous extension of a Repair Query, i.e., $\{F * P\}\ ?C\ \{F' * Q\}$.

Pertinently, we can use *frame inference* in a novel way to solve two important (and distinct) purposes in the program repair search problem: (1) Check whether a given specification satisfies a repair query, and (2) discover the extra semantic effects not part of the fix. That is, suppose some command C has a footprint, expressed as $\{S_P\}\ C\ \{S_Q\}$. Our first goal is to check whether the footprint satisfies the Repair Query,which we do with respect to corresponding pre- and postconditions F and F'. Our second goal is to partition the footprint into the fixing transitions and extra semantic effects for pre- and postconditions, respectively.

Our key insight is to perform the frame inference procedure Frame (Section 2) on the *footprint* precondition (resp. postcondition) with respect to the *Repair Query* precondition (resp. postcondition). Formally, C is a candidate repair fragment when the following entailments hold:

In the precondition:

$$\textsc{Frame}(S_P, F) = P \implies S_P \vdash F * P \tag{3}$$

Respectively, in the postcondition:

$$\textsc{Frame}(S_Q, F') = Q \implies S_Q \vdash F' * Q \tag{4}$$

We achieve (1) because the entailment does not hold (frame inference fails) when F is not satisfied by the query. If frame inference succeeds, it pulls out P (resp. Q), discovering the extra semantic effects in the footprint of C, achieving (2). Our approach is sound and decidable by the frame inference procedure [11].

Assume $\mathfrak{F}$ contains the specifications inferred over all individual commands in a program. Algorithm 1: FindCandidates(R,$\mathfrak{F}$) soundly returns all candidate Repair Specifications for query R. In Line 4 and 5 of Algorithm 1, we use the frame inference procedure Frame, to match candidate fragments with a repair query as the conjunction of matching pre- and postconditions. Note that the procedure returns the entire Repair Specification because we use assertions in the precondition for repair application (Section 3.3).

Algorithm 1: Find Candidate Repair Fragments

```
1  Match(S,R);
2      let {R_F} ?C_R {R_F'} = R in
3      let {S_P} C {S_Q} = S in
4      if Frame(S_P,R_F) = R_P ≠ fail ⇒ S_P ⊢ R_F * R_P
5      ∧ Frame(S_Q,R_F') = R_Q ≠ fail ⇒ S_Q ⊢ R_F' * R_Q then
6          return {R_F * R_P} ?C_R {R_F' * R_Q}[C/?C_R]
7      else
8          return fail
9      end
10
11 FindCandidates(R,𝔉);
12     𝔈 := ∅ // Candidates
13     foreach S ∈ 𝔉 do
14         if Match(S,R) = C ≠ fail then
15             𝔈 := 𝔈 ∪ C
16         end
17     end
18     return 𝔈
```

3.3 Applying repairs: from logic to programs

A Repair Specification in the abstract domain must be translated to a syntactic program fragment in the source program. Every intermediate language (IL) instruction corresponds to a line in the original program source. When a candidate program fragment is translated from IL to source, we rename the program variable bound to *pvar* in the fixing fragment to that of the fault-inducing variable, if necessary. This substitution is the only syntactic transformation that we apply on source fragments. Beyond this renaming, we must decide where to insert repairs, and check translated code for compatibility given associated heap assumptions and available variables and type information (available in the IL).

Determining Repair Location. Repair techniques typically rely on dynamic fault localization techniques to determine placement or manipulation of code [7]. By contrast, we rely on the static analysis to localize faults: When the symbolic interpreter enters a `fault` state, it provides a location ℓ where the fault occurs.

In general, the fault class bears on the choice of where to apply a repair fragment. For null dereferences, we speculate that a developer might typically expect a change before the null dereference (e.g., a check), whereas for resource leaks we expect a change after the point at which the resource is last used (e.g., closing a file). Using Infer, a null dereference reports the location ℓ immediately preceding the dereference. For resource leaks and memory leaks, the location ℓ is the point where the resource (resp. memory) goes out of scope without being closed (resp. freed). We make the design choice to insert a repair fragment C_R directly preceding the program term at location ℓ, satisfying

$$C_{R_{\ell'}}, \mathbb{H}_{Bad} \rightsquigarrow C_\ell, \mathbb{H}_{Good} \not\rightsquigarrow \texttt{fault}$$

We refer to the example in Section 1 to motivate our choice. Our approach places the `sw_free` call at line 11 in Figure 1a. The first motivation is that repair location is typically restricted to few alternatives; the only other correct choice of placement is line 10. The location ℓ thus offers the most immediate (and sometimes strictly) correct choice. The second motivation is that among multiple placement locations, the semantic effects of repair remain the same. For example, placing `sw_free` at line 10 in Figure 1a. produces the same effect since the `swWarn(...)` statement does not affect the heap. Thus, although there is no universally correct choice for repair placement, our domain of repair benefits from a general strategy that preserves semantic correctness. Determining the ideal placement of patches

with respect to non-semantic attributes may vary by context, and may be subject to stylistic conventions and human judgment that go beyond the scope of this work.

Determining Patch Compatibility. Patch compatibility determines whether we *can* insert a syntactic program fragment at a particular program point (ℓ, in our case). FootPatch performs two compatibility checks. The first addresses bugs that can have multiple candidates. For example, memory may be freed by the standard C library `free` call, or a wrapper function such as `sw_free` as in Listing 1b. When multiple candidate fragments in SIL matches the desired semantic effect (based on predicates that do not consider types), FootPatch prioritizes patch generation by matching type information of a candidate fragment's variable and the fault-inducing variable. This means that FootPatch prefers `sw_free` over `free` in our example, because we can infer the type of `hmap` to be `swHashMap *`, which matches the same type of `map` in the candidate fixing fragment `sw_free(map)`. FootPatch falls through to matching candidates with generic types (i.e., `void *` for C) if it cannot match specific types. Although typing information could be used to refine patches for null dereferences, we ignore typing information in this case, since any object can be compared to null.

Recompilation serves as our second compatibility check, ensuring (a) that syntactically malformed patches fail (due to poor IL-to-source translation or fault locations), and (b) that program fragments with unbound free variables are invalidated. For instance, the fix for the bugs in Figure 2 binds to the variable `id` that is in scope. If `id` were *not* in scope, patch generation fails. Relatedly, FootPatch allows capture of program variables (beyond the fault-inducing variable) in fixing fragments if they are available (such as `id` which is in scope).

4 EVALUATION

This section describes our the results of using FootPatch to fix bugs in real-world programs. Section 4.1 describes our experimental setup. Section 4.2 describes overall repair results, where we use FootPatch to fix 55 bugs in 11 projects. In Section 4.3 we discuss patch quality, and particularly its relationship to Infer's underlying static analysis and our other design decisions. Section 4.4 analyzes FootPatch's repair discovery for resource leaks, memory leaks, and null dereferences in the context of our general formalism (Section 3). Section 4.5 provides limitations and further discussion.

4.1 Setup

Implementation. Based on the techniques described above, we implemented an automatic bug repair tool called FootPatch extending the Infer static analyzer. FootPatch works on multiple source languages (we evaluate on programs in C and Java; Infer also supports C++ and ObjectiveC) because, like Infer, it reasons over programs translated into SIL. FootPatch uses the same predicates in the IL irrespective of source language, meaning it can apply directly to new languages as support for them is added to Infer.

FootPatch uses three general specifications for repairing null dereferences, resource leaks, and memory leaks; Figure 5 provides a simplified representation. The specifications are implemented as OCaml functions that match pre/post conditions on heap state and

$$\{\ pvar \Mapsto Null\ \}\ ?C\ \{\ \_\ \Mapsto Exn\ e\ \} \quad (5)$$

$$\{\ pvar \Mapsto \langle File, Acquired\rangle\ \}\ ?C\ \{\ pvar\ \Mapsto \langle File, Release\rangle\ \} \quad (6)$$

$$\{\ pvar \Mapsto \langle Memory, Acquired\rangle\ \}\ ?C\ \{\ pvar\ \Mapsto \langle Memory, Release\rangle\ \} \quad (7)$$

Figure 5: Repair Specifications.

predicates (referencing Infer's datatypes). Note that this specification mechanism is not intrinsic to the technique (i.e., it is possible write a DSL for expressing such specifications). FootPatch performs type matching for determining patch compatibility, (Section 3.3) by extending specifications (6) and (7) in Figure 5 with an optional guard clause "`when` $pvar.type = t$" if we can determine the type t of a faulting variable. In general, since simple expressions do not model semantic effects of interest to the bug types in question, we restrict repair queries to calls and branch statements in the IL.

Data and experimental parameters We ran our experiments on an Ubuntu 16.04 LTS server, with 20 Xeon E5-2699 CPUs and 20GB of RAM. Table 1 includes projects which (a) successfully built on our system, (b) could be analyzed by Infer, and (c) generated patches. Our project selection represents a convenience sample, intended to substantiate our claims about FootPatch applicability to real bugs in real and actively developed open-source systems; we include discussion of sources of failures and other technique limitations in Section 4.5. We evaluate on 8 C programs and 3 Java programs averaging 64 kLOC. We initially developed FootPatch based on existing bugs in `error-prone` and `jfreechart` and new bugs found in `swoole`. The rest of the projects are C and Java repositories on GitHub that are either (a) randomly sampled from the top one thousand most popular repositories (by user favorites/stars) for C and Java respectively, or (b) contain any combination of the terms "leak", "resource", "memory", "file", or "fix" in their commit messages. Projects in our sample are excluded if they fail to compile in our environment, if they cannot be analyzed by Infer, or if FootPatch did not find patches (either because no bugs were found or due to some other failure).

We evaluated FootPatch in two modes to characterize its search behavior. In *callgraph-local* mode, FootPatch searches over candidates from the callgraph of functions where Infer reports a bug. This mode tracks whether candidates are found local to the function containing the bug, local to the file containing the bug, or from an external file. In *global mode*, FootPatch searches over all (disjoint) callgraphs. Global mode subsumes callgraph-local mode and, in our experiments, only discovers additional repair candidates in external files not included in the local callgraph. Global mode is naturally more time-intensive, but may identify additional patches; comparing the two modes allows a more precise understanding of the importance of locality in searching for bug fixes within a given program. In all results discussion, ΔGL indicates the increment searching globally has over searching locally.

4.2 Repair results

Table 1 show results for each project. "Bugs" indicates the number of bugs detected by Infer of the given type. It is possible for multiple semantic fragments to repair each type of bug, found at different locations in the SIL callgraph. For each bug type per program, "Max Cands" shows maximum number of IL repair candidates before the

Project	Lang	kLOC	Time (s)	ΔGL	Bug Type	Bugs	Max Cands	ΔGL	Fixes	ΔGL	FP	ΔGL
Swoole	C	44.5	20	+83	Res. Leak†	7	1	+6	1	+2	0	+0
					Mem. Leak†	20	3	+0	6	+0	3	+0
lxc	C	63.0	51	-	Res. Leak	3	5	-	1	-	0	-
					Mem. Leak	8	13	-	0	-	1	-
Apktool	Java	15.0	584	+92	Res. Leak†	19	3	+2	1	+0	0	+0
dablooms	C	1.2	9	+0	Res. Leak†	7	2	+0	7	+0	0	+0
php-cp	C	9.0	20	+5	Res. Leak†	4	3	+1	1	+0	0	+0
armake	C	16.0	10	+13	Res. Leak†	5	7	+4	4	+0	0	+0
sysstat	C	24.9	28	+10	Res. Leak	1	10	+0	1	+0	0	+0
redis	C	115.0	79	+121	Res. Leak†	8	8	+10	6	+0	0	+0
rappel	C	2.1	7	+3	Mem. Leak†	1	6	+0	1	+0	0	+0
error-prone	Java	149.0	262	+602	Null Deref	11	66	+0	2	+0	0	+0
jfreechart	Java	282.7	1,268	-	Null Deref	53	221	-	22	-	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 check). † indicates one or more fixes for previously undiscovered bugs. "ΔGL" is the change in associated column when using the global search space.

compatibility check which determines whether a patch candidate is suitable syntactically. FootPatch emits the first compatible patch produced from the candidates. "Fixes" shows the number of unique patches that fix true semantic errors. Conversly, "FP" shows the number of unique patches that fix false positive bug reports; we discuss patch quality and correctness in Section 4.3.

In total, we discover 24 fixes for resource leaks, 7 fixes for memory leaks, and 24 fixes for null dereferences. The "Time" columns of Table 1 shows *total* time required to both find and patch all bugs of all types considered in that program. Performance ranges from 7 seconds to 22 minutes over all projects (note that FootPatch performance is intertwined with Infer's analysis time). The `jfreechart` experiment failed to terminate in global mode because the Infer analysis phase ran out of memory; `lxc` failed to build in the global configuration.

Global mode (ΔGL) discovers only 2 additional fixing patches. This patch fixes a resource leak due to discovering a `close`-like function that is not present in the local search. Our results suggest that localizing search for fixing fragments is an effective strategy for repair. This is consistent with empirical results that suggest that code is redundant locally, especially within a module [55].

```
1       fp = fopen(rdbfilename,"r");
2       ...
3       if (memcmp(buf,"REDIS",5) != 0) {
4           rdbCheckError("Wrong signature trying to load
                DB from file");
5   +       fclose(fp);
6           return 1;
7       }
8       rdbver = atoi(buf+5);
9       if (rdbver < 1 || rdbver > RDB_VERSION) {
10          rdbCheckError("Can't handle RDB format
                version %d",rdbver);
11  +       fclose(fp);
12          return 1;
13      }
14      ...
```

Figure 6: `fp` can be leaked on at least two paths (lines 6 and 12), but Infer short circuits the analysis and only reports the leak on line 6 by default. With FootPatch, the leak is fixed at line 6, allows Infer to find the another resource leak, which is then also fixed at line 11.

4.3 Patch quality

Patch correctness and success. All produced patches ensure a fault-avoiding interpretation; in practice, we apply each patch generated by FootPatch and rerun the static analyzer to see if the patch removes the bug (all did). Where possible, we ran a project's test suite after applying our patches to validate that our changes do not break tests. We successfully ran the test suites for `Apk-tool`, `armake`, and `error-prone`, which pass. Two projects contained no tests, and the remaining six test suites could not be successfully configured/built.

A "fix" in Table 1 produces a patch that addresses a *true positive* bug report from the static analysis. To be useful in practice, analyses approximate [34]. Infer is no exception, and it sometimes skips inferring specifications for a function due to an analysis timeout, continuing with partial results. This can lead to false positives. FootPatch uses Infer's results to perform patching, and cannot distinguish between true and false positives (if it could, it would be a better analysis than Infer; this task is outside its scope). A manual inspection of Infer's reports indicate that its false positives generally arose when it failed to analyze loops or clean up functions due to time out. Due to the complexity of `jfreechart` it is difficult to precisely determine how many of the Infer-reported null dereferences are false positives. However, our manual inspection of did not reveal obvious errors in reasoning behind the produced patches.

False positives that produced patched bugs are listed in column "FP" of Table 1. In general, the false positive rate is low, in the order of 5%, where fixes are produced for false positive error reports ("FP" column) compared to fixes for true positives ("Fixes" column).

```
1    int fd = open(filename, O_RDONLY);
2    ...
3    swString *content = swString_new(filesize);
4    if (!content) {
5  +   close(fd); // FootPatch repair
6      return NULL; // function returns with fd not closed
7    }
8    ... // continues with normal operation
```

Figure 7: Resource Leak: forgetting to close a file.

On the other hand, Infer may find bugs that do not result in a fix (there are typically more "Bugs" than "Fixes" in Table 1). FootPatch finds 54 fixes out of 145 bugs (excluding false positives). Patch generation fails when no repair candidate can be found for the bug. Generally, this happens when (a) Infer's analysis times out (e.g., due to loops), leading to incomplete function specifications or short circuited analysis that miss fixing fragments, (b) Infer does not resolve program variables associated with a bug report, (c) no type compatible fragments are discovered, or (d) memory time outs occur for parallel analysis processes, short circuiting analysis results.

Patch location. As motivated in Section 3.3, FootPatch heuristically places fix code at the line where Infer reports the violation. For resource and memory leaks, feasible repair locations are constrained by the number of lines at which the resource is no longer in use, but before it is officially dead. In our data, the maximum number of possible correct locations across all 31 fixing patches for resource/memory leaks is 3, while the average is 1.7. This implies little opportunity to vary placement outside of our convention, similar to our motivating example in Figure 1a. For null dereferences, checks may plausibly be placed anywhere between the point at which an object becomes null and its dereference. Our inspection based on Infer's bug reports revealed that the number of locations ranges from 1 to 30, which poses more variability for placement.

Patches reveal more bugs. An especially interesting implication of unifying bug detection and repair is the potential for the latter to extend the capabilities of the former. In our experiments, FootPatch generated patches for the `redis` project that then allowed Infer to find two additional unique bugs. Figure 6 shows a snippet of the code in question. The `fp` file pointer leaks on at least two paths (lines 6 and 12). Before patching, Infer only reports a resource-leak for the variable `fp`, because it "short circuits" its analysis once the first leak is detected. After FootPatch inserts `fclose(fp);` on line 5, Infer reports the second leak on line 11. Rerunning FootPatch yields an additional fix on line 11. To the best of our knowledge, this is the first demonstration that automated patching has the potential to improve static analysis.

4.4 Fixing by semantic effects

Resource and Memory Leaks. Resource leaks often occur when a function returns prematurely due to an error [6, 59]. Figure 7 shows a leak of an unreleased file handle in the `Swoole`[8] project. FootPatch uses the Repair Query $\{\ pvar \mapsto \langle File, Acquired\rangle\ \}\ ?C\ \{\ pvar \mapsto \langle File, Release\rangle\ \}$ from (6), Figure 5, to discover a `close(fd);` elsewhere in the program. This demonstrates the importance of the compatibility check, which guards against applying alternative "close" operations (e.g., `fclose`) by using typing information. The pull request based on the patch in Figure 7 was accepted,[9] an important milestone for *end-to-end* automatic repair of a previously undiscovered bug.

Note that, although conceptually similar to resource leaks, memory leaks deserve separate semantic treatment because they tend to occur in programs written in languages that are not garbage collected. Anecdotally, memory leaks may entail more complex fixes in terms of semantic effects. All resource leak patches conform to strict Repair Specifications (meaning they only affect the heap location of interest). However, fixing fragments for memory leaks may entail extra semantic effects. One example fragment is `swHashMap_node_free(hmap, root);` from the `Swoole` project, which frees a data member `root` that is in the table `hmap`. A necessary precondition to inserting this fragment for freeing `root` is that `hmap` be in scope (which it is, where it is used in our produced patches). We obtain such a fragment by relaxing the Repair Query to allow extra semantic effects. One implication of relaxing repairs is that application may be contextual, and subject to additional compatibility checks (e.g., scope and variable capture) with respect to extra semantic effects. In summary, our results show that FootPatch the majority of leak fixes conform to strict Repair Specifications, but enables more complex fixes when relaxing the repair constraint.

Null dereferences. The FootPatch patch for the null dereference(s) in Figure 2 throws an exception when an object is null. However, in general, multiple semantic fix effects may address a given null dereference: initializing a null object, returning or throwing an exception when an object is null, or predicating execution on a condition that an object is not null. More than one of these forms may be correct with respect to a preserving a non-null property, and in general we cannot decide which one is preferred [35].

We therefore experimented with Repair Specification queries in FootPatch over multiple SIL commands to discover null dereference fixes (i.e., function calls entailing nullness checks, conditional expressions). Our approach alleviates the problem of multiple potential fixes by relying on existing code to guide repair. For example, for `jfreechart`, FootPatch produces 22 patches from a candidate which throws an exception when the object is null. This may be the desirable fix, as witnessed by human-written fixes for the `error-prone` bug [2]. Regardless, from a semantic perspective, FootPatch finds candidate fixes from the existing project that removes the fault with respect to the analysis.

4.5 Limitations and discussion

FootPatch currently fixes resource leaks, memory leaks, and null dereferences; these bugs are a mature focus of Infer's analysis domain and are common in practice [16, 17]. Given both our underlying technique, which addresses general heap-based properties, and the continual addition of new analyses to Infer [4], we expect FootPatch to generalize to, for example, information flow bugs. FootPatch currently requires a simple manual fix specification, generally per bug type. This formulation provides flexibility to address particular attributes over diverse bug classes and languages. The manual effort is competitive with effort required to produce a test per bug, required by dynamic repair techniques. Moreover, we

[8] `Swoole` is the 34th most popular C project on GitHub at the time of writing.

[9] https://github.com/swoole/swoole-src/commit/e12c7db38c9737234695d35d9

recognize an opportunity for automatically inferring fixing effects with the aid of a static analyzer, for future work.

FootPatch does not consider the full diversity of possible fixes for the considered defects, especially for null dereferences (we investigated checking nullness and throwing an exception, accepted ways for fixing these bugs, but not instantiating new objects generally). Our approach currently inspects only semantic patch characteristics, ignoring, for example, string constants in an error message. We leave such considerations to future work.

Beyond false positives, Infer can produce inaccurate fault locations, impacting the validity of FootPatch patches. Approximate locations are acceptable for bug reports, and approximate patches may still be informative [57], but truly automated program repair requires precise locations. This can be addressed practically by improving the accuracy of IL-to-source translation during analysis.

We do not explicitly compare our technique to prior repair techniques. Overall, FootPatch is orthogonal (and thus difficult to compare) to prior dynamic repair techniques. Two of our bug classes, resource and memory leaks, are difficult to test for deterministically, and thus underaddressed in the current repair literature. There does exist work addressing repair of null dereferences, which are easier to expose via tests. We attempted to run and fix the null dereferences in Apache Commons Math considered by NOPOL [61]. However, Infer's analysis skips a number of intermediate calls, and fails to detect the null dereferences covered in the associated test suite. Practically speaking, we find that tests for null dereferences considered in prior work (such as Defects4J [28]) simply cover different null dereference bugs from Infer. This highlights one property of static analysis for repair: analysis may miss bugs that could be covered by tests, but may simultaneously find those in corner cases that humans neglected to test. Related techniques based on verification [35] lack detailed breakdown of bugs to inform a comparative study, and is intended to provide patch suggestions to a developer, which lacks automatic patch application. We are unaware of "quick fix" suggestions from existing static tools [1, 2, 27] that target semantic bugs like those we consider.

5 RELATED WORK

Work in automatic program repair over the past decade predominantly use test cases to validate correctness. Generate-and-validate or heuristic repair techniques explore search spaces of templated repairs as applied to the abstract syntax tree program level; this includes techniques like GenProg [33], RSRepair [50], and HDRepair [32] which traverse the space using classic search algorithms like genetic programming or random search. Other techniques like AE [58] SPR [37] and Prophet [36] use predefined transformation schemas and probabilistic models on the AST to discover and apply candidate syntactic fragments. At a high level, such techniques operate on ASTs to indirectly achieve a desired semantic effect that fixes a bug with respect to a test suite. Constraint- or semantics-based approaches reason about semantics more directly, synthesizing fragments using input-output pairs to codify a notion of program semantics [38, 39, 41, 61]. SearchRepair [29] lies between these approaches, using input-output pairs to search over a semantic encoding of candidate repair fragments (and is perhaps closest in spirit to our approach). These techniques all share the property that they use test cases to define patch correctness and guide a search towards a semantically-desirable fix; as a result, they also require developer labor to specify those tests; are limited to fixing bugs that are deterministically testable, and may overfit to the provided tests [54]. The main point of contrast with our work is that our approach is static, and therefore cannot overfit to provided dynamic witnesses of desired behavior. In the previous body of work, properties are not specified formally, but implied by test cases. FootPatch instead considers a logic-encoding and semantic implications of using a program fragment for satisfying repair specification. We argue that our focus on explicitly codifying semantic effects offers additional protection against patch overfitting.

Verification-based approaches, using formal specifications, have used LTL specifications [26, 56], SAT approaches [23], deductive synthesis [30], contracts [35], and model checking for Boolean programs [24, 52] to perform repair. At a high level, our approach relates to the approach by Logozzo et al. [35] which uses automatically inferred assertions over abstract domains, and relies on an abstract interpretation-based static analyzer to discover faults; however, they do not consider automatically applying patches. In contrast, our approach is new in reasoning over an abstract domain based in Separation Logic (repairing pointer safety violations) and formalizes a mechanism for automatic patch application. Overall, verification-based program repair lack application to common bugs in real-world programs.

6 CONCLUSION

We presented a new static APR technique using Separation Logic to reason about semantic effects of program fragments, including a novel application of local reasoning and the frame rule to find bug-fixing patches from existing code. Our technique overcomes significant challenges compared to test-based repair techniques, including the ability to repair previously undiscovered bugs, bugs that are difficult to expose via testing, and repeated semantic errors. We implemented our approach in a tool called FootPatch that builds on top of a Separation Logic-based analysis to target bug repair for heap-related properties; we demonstrate on resource leaks, memory leaks, and null dereferences, and anticipate that the approach is extendable. FootPatch correctly fixes 55 bugs, including 11 undiscovered bugs in 11 projects. Moreover, FootPatch achieves significant speedup over test-based repair, works on large codebases, and targets multiple source languages. Unlike other formal approaches for program repair, FootPatch works end-to-end on existing code bases and does not require formal annotation or special coding practices. Its reliance on principled semantic reasoning provides additional evidence of generated patch correctness. FootPatch thus represents an important step in bridging the gap between grounded verification techniques and trustworthy automatic program repair for real-world software, opening potentially promising avenues in automatic program improvement.

7 ACKNOWLEDGMENTS

This work is partially supported under NSF grant number CCF-1563797. All statements are those of the authors, and do not necessarily reflect the views of the funding agency.

REFERENCES

[1] 2017. FindBugs Static Analyzer. https://github.com/findbugsproject/findbugs. (2017). Online; accessed 26 August 2017.

[2] 2017. Google Error-prone bug-fixing commit. https://github.com/google/error-prone/commit/3709338. (2017). Online; accessed 16 January 2017.

[3] 2017. Infer bug types. http://fbinfer.com/docs/infer-bug-types.html. (2017). Online; accessed 11 May 2017.

[4] 2017. Infer experimental checkers. http://fbinfer.com/docs/experimental-checkers.html. (2017). Online; accessed 11 May 2017.

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

[6] 2017. Resource Leak in C. http://fbinfer.com/docs/infer-bug-types.html#RESOURCE_LEAK. (2017). Online; accessed 16 January 2017.

[7] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*. 89–98.

[8] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29.

[9] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson R. Murphy-Hill. 2016. From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration. In *IEEE International Conference on Software Maintenance and Evolution (ICSME '16)*. IEEE Computer Society, 211–221.

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

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

[12] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2012. Verification Condition Generation and Variable Conditions in Smallfoot. *CoRR* abs/1204.4804 (2012).

[13] Josh Berdine, Arlen Cox, Samin Ishtiaq, and Christoph M. Wintersteiger. 2012. Diagnosing Abstraction Failure for Separation Logic-Based Analyses. In *Computer Aided Verification (CAV '12)*. 155–173.

[14] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75.

[15] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. *Reversible Debugging Software*. Technical Report. University of Cambridge, Judge Business School.

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

[17] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (NFM '15)*. 3–11.

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

[19] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification (CAV '16)*. 383–401.

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

[21] Dino Distefano, Peter W O'Hearn, and Hongseok Yang. 2006. A Local Shape Analysis Based on Separation Logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 287–302.

[22] Dino Distefano and Matthew J Parkinson J. 2008. jStar: Towards Practical Verification for Java. *ACM Sigplan Notices* 43, 10 (2008), 213–226.

[23] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '11)*. 173–188.

[24] Andreas Griesmayer, Roderick Bloem, and Byron Cook. 2006. Repair of Boolean Programs with an Application to C. *Computer Aided Verification* (2006), 358–371.

[25] Ciera Jaspan, I-Chin Chen, and Anoop Sharma. 2007. Understanding the Value of Program Analysis Tools. In *Object-oriented Programming Systems and Applications (OOPSLA '07)*. 963–970.

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

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *International Conference on Software Engineering (ICSE '13)*. 672–681.

[28] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis (ISSTA '14)*. 437–440.

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

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

[31] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Software Analysis, Evolution, and Reengineering (SANER '16)*. 213–224.

[32] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Software Analysis, Evolution, and Reengineering (SANER '16)*. 213–224.

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

[34] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

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

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

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

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

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

[40] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electronic notes in Theoretical Computer Science* 89, 2 (2003), 44–66.

[41] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. *International Conference on Software Engineering*, 772–781.

[42] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. 2007. Automated Verification of Shape and Size Properties via Separation Logic. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI '07)*. 251–266.

[43] National Institute of Standards and Technology. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Technical Report NIST Planning Report 02-3. NIST. http://www.nist.gov/director/prog-ofc/report02-3.pdf

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

[45] Peter W. O'Hearn. 2015. From Categorical Logic to Facebook Engineering. In *Symposium on Logic in Computer Science*. 17–20.

[46] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *International Workshop on Computer Science Logic (CSL '01)*. 1–19.

[47] Matthew J. Parkinson and Gavin M. Bierman. 2005. Separation Logic and Abstraction. In *Symposium on Principles of Programming Languages (POPL '05)*. 247–258.

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

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

[50] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *International Conference on Software Engineering (ICSE)*. 254–265.

[51] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *International Symposium on Software Testing and Analysis (ISSTA '15)*. 24–36.

[52] Roopsha Samanta, Oswaldo Olivo, and E Allen Emerson. 2014. Cost-Aware Automatic Program Repair. In *Static Analysis Symposium (SAS '14)*. 268–284.

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

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

[55] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the Localness of Software. In *Foundations of Software Engineering (FSE 2014)*. 269–280.
[56] Christian von Essen and Barbara Jobstmann. 2015. Program Repair without Regret. *Formal Methods in System Design* 47, 1 (2015), 26–50.
[57] Westley Weimer. 2006. Patches As Better Bug Reports. In *Generative Programming and Component Engineering (GPCE '06)*. 181–190.
[58] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *Automated Software Engineering (ASE '13)*. 356–366.
[59] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*. 461–476.
[60] Westley Weimer and George C. Necula. 2008. Exceptional Situations and Program Reliability. *ACM Transactions on Programming Languages and Systems* 30, 2, Article 8 (March 2008), 51 pages.
[61] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.
[62] Hongseok Yang and Peter O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. 402–416.