Anti-patterns in Search-Based Program Repair

Shin Hwei Tan[*,†] Hiroaki Yoshida[‡] Mukul R. Prasad[‡] Abhik Roychoudhury[†]
[†]National University of Singapore, Singapore
[‡]Fujitsu Laboratories of America, Inc., Sunnyvale, CA, USA
{shinhwei,abhik}@comp.nus.edu.sg
{hyoshida,mukul}@us.fujitsu.com

ABSTRACT

Search-based program repair automatically searches for a program fix within a given repair space. This may be accomplished by retrofitting a generic search algorithm for program repair as evidenced by the GenProg tool, or by building a customized search algorithm for program repair as in SPR. Unfortunately, automated program repair approaches may produce patches that may be rejected by programmers, because of which past works have suggested using human-written patches to produce templates to guide program repair. In this work, we take the position that we will *not* provide templates to guide the repair search because that may unduly restrict the repair space and attempt to overfit the repairs into one of the provided templates. Instead, we suggest the use of a set of anti-patterns — a set of generic forbidden transformations that can be enforced on top of any search-based repair tool. We show that by enforcing our *anti-patterns*, we obtain repairs that localize the correct lines or functions, involve less deletion of program functionality, and are mostly obtained more efficiently. Since our set of anti-patterns are generic, we have integrated them into existing search based repair tools, including GenProg and SPR, thereby allowing us to obtain higher quality program patches with minimal effort.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging; Search-based software engineering;** •**Computing methodologies** → *Genetic programming;*

Keywords

Debugging, fault localization, and repair

1. INTRODUCTION

Automated program repair techniques have gained prominence in recent years [7, 11, 28, 27]. These techniques bear the promise of automatically suggesting fixes to "easy-to-fix" programming errors, thereby relieving substantial burden from programmers on the manual effort of debugging and generating fixes.

A major challenge in automated program repairs arises from the "incomplete specification" of intended behavior. Indeed, any repair technique tries to patch errors so as to achieve the intended behavior. Yet, in reality, the intended behavior is incompletely specified, often through a set of test cases. Thus, repair methods attempt to patch a given buggy program, so that the patched program passes all tests in a given test-suite T. Unsurprisingly, this may not only lead to incomplete fixes but the patched program may also end up *introducing* new errors, because the patched program may fail tests outside T, which were previously passing [26].

Several recent research articles have pointed out the pitfalls of using test-suites as specification to drive program repair [23, 26]. Furthermore, if the test oracles of the tests in the test-suite are not strong enough, simple program modifications, such as deletion of program functionality, have been shown to be sufficient to repair programs [23]. The situation presents us with an unenviable dilemma — we want to avoid incomplete or incorrect fixes but it is not practical to assume the presence of formal specifications to drive program repair towards correct fixes.

In this paper, we propose to use *anti-patterns* to help alleviate the problem of incorrect or incomplete fixes resulting from program repair. We present our technique in the context of *search-based program repair* systems, also referred to as "generate-and-validate" systems in the literature [11]. These repair tools seek to repair a buggy program (one failing at least one test in a given test-suite T) by searching among possible fixes by applying fix templates. A proposed fix is "validated" if it passes all the tests in the given test-suite T. One key problem faced in the resulting fixes is that they often boil down to program modifications like deletion of functionality — which, though sufficient to pass tests in given test-suite T, may fail tests outside T and can be unacceptable to developers in general.

Our main idea is simple — for any search-based repair technique which is searching for a plausible repair, we define a set of *anti-patterns* that essentially capture disallowed modifications to the buggy program. In other words, even if such a modification results in the modified program passing all tests in the given test-suite, we do not count them as repairs. Our set of *anti-patterns* is generic and does not vary across application domains.

[*]This author was an intern at Fujitsu Laboratories of America during part of this work.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950295

Conceptually, our idea is different from the strategy of using human patch templates to guide program repair [7]. Generally speaking, the use of human patch templates is geared towards producing patches close to human patches — the underlying assumption being that by going close to human patches, we will avoid incorrect or incomplete fixes. However, this requires providing human patch templates, which is limited by a fixed set of templates, and hence the produced repair may overfit the provided set of templates. Furthermore, there is a strong assumption that by fitting patches to human patch templates, we have a greater chance of the fix being accepted by developers — an assumption that may or may not be true (*e.g.*, [16] argues that "fix acceptability may be an unanswerable question").

Instead of gearing our repair towards human patches by providing human patch templates, we ask ourselves the following research question — *is it possible to drive the repair search towards correct and complete fixes, simply by providing a generic set of* anti-patterns*?* Many of our *anti-patterns* are at the level of the control flow graph — forbidding certain manipulations to the control flow graph. A few of the *anti-patterns* involve assignments affecting branch outcomes and one anti-pattern forbids adding tautologies as branch conditions. Overall, our *anti-patterns* are generic.

Furthermore, and more importantly, we are not proposing a separate repair method based on *anti-patterns*. Our proposed set of *anti-patterns* can be integrated into any existing search-based repair tool, and we can then compare the repair produced after enforcing *anti-patterns* with the original repairs produced by the search-based repair tool. Indeed, we propose a small set of *anti-patterns* and we have integrated them into two existing search based program repair tools: GenProg [28] and SPR [11].

Any automated program repair system is driven by a correctness criterion (to which we repair to), and since formal specifications are usually absent, test-suites are used as correctness criteria. As a fully automated derivation of a formal correctness criterion is often impossible, our *anti-patterns* are not meant to solve the problem of deriving better correctness criteria. Instead, the value of our *anti-patterns* lies in their ability to provide more precise repair hints to developers [6], which is illustrated through our evaluation on patch quality. We evaluated our *anti-patterns* on 86 real bugs obtained from 12 subjects. Results from our experiments indicate that *anti-patterns* could lead search-based program repair tools to producing patches that localizes better by isolating either the correct line or the correct function. Moreover, *anti-patterns* could also reduce the destructive effect of search-based repair tools by producing patches that removes less functionality. Our *anti-patterns* also provide considerable amount of speedup in obtaining the final repair because our *anti-patterns* prune the repair search space. All experimental data are available at the following web site: *https://anti-patterns.github.io/search-based-repair/*.

2. RELATED WORK

Search-based Repair. One of the earliest and most well-known representative of search-based repair systems, GenProg [10], uses genetic programming to search its repair space. Its repairs consist of code snippets copied from elsewhere in the program. RSRepair [22] searches a similar repair space but using random search instead. AE [28] uses a deterministic search strategy but exploits program equivalence relations to prune equivalent patches. However, recent work [23] has shown that the vast majority of patches produced by GenProg, RSRepair, and AE are semantically equivalent to functionality deletion and that their Kali repair tool, that exclusively relies on a small set of functionality deletion repair schemas, can achieve similar results. PAR [7] re-uses GenProg's basic search strategy but proposes a set of 10 specialized repair templates manually derived from human-written patches and show that they are more successful in identifying correct and meaningful patches than GenProg. *relifix* [27] addresses the repair of software regression errors in a search-based repair framework. It proposes the use of previous versions of a buggy program and a set of novel contextual repair operators, operating on multiple versions of the program, to define the repair space. SPR [11] recognizes bug-fixes based on manipulation or insertion of conditional statements as an important subset of machine-generatable patches. It proposes a heuristic strategy to search the repair space defined by its rich set of repair schemas (patch patterns instead of *anti-patterns*). A key innovation is the use of abstract repair conditions to evaluate and prune away the vast majority of condition repair candidates (staging) before concretizing and evaluating them against the full test suite. The recent work of Prophet [13] uses machine learning to characterize previously-known correct human patches, and uses the learned characterization to prioritize candidate repairs, in an effort to avoid plausible patches — incorrect patches that pass the test suite. However, for the majority of bugs, when the correct repair is outside the search space of the tool, such plausible patches might still be generated.

Our approach shares PAR's goal of generating more meaningful patches and SPR and Prophet's goal of avoiding plausible but incorrect patches. However, it is in essence orthogonal to all of the above search-based repair approaches in that it seeks to analyze a repair schema, as instantiated, in the context of a specific program and bug, and disallow ones that result in "unreasonable" changes to the original program's behavior. Our *anti-patterns* form a simple set of "do-not" rules (in the style of software development or software design anti-patterns[8]), which can guide an automated repair tool. Thus, our approach is rather independent of the repair schemas and search strategies being used and can be integrated into any of the above repair tools.

Oracle-based Repair. SemFix [17] uses symbolic execution to capture and implicity express the values that an expression under repair must assume under various test executions. It then uses program synthesis to synthesize a repaired statement compatible with this "oracle". MintHint [6] also uses symbolic execution to synthesize an oracle but then uses statistical analysis to guide the choice for a repair. NOPOL [3] addresses the repair of branch conditions and uses instrumented test-suite executions to synthesize an oracle, which is then converted into a suitable SMT formula representing feasible repairs and solved to generate a patch. DirectFix [14] generates *minimal* repairs to obtain human-readable and comprehensible repairs. The idea is to encode the problem as a partial maximum satisfiability problem over SMT formulas (partial maxSAT) and solve it using a suitably modified SMT solver. Angelix [15] solves the scalability problems of DirectFix by using a lightweight repair constraint. Since the above techniques do not directly tackle the problem of the incompleteness of the repair speci-

Table 1: Prevalence of Anti-patterns in Plausible Patches

	Anti-delete CFG exit node				Anti-delete Control Statement		Anti-delete Single-statement CFG	Anti-delete Set-Before-If	Anti-delete Loop-Counter Update	Anti-append Early Exit			Anti-append Trivial Conditions	
	Delete exit	Delete return	Delete goto	Delete error code	Delete if-statement	Delete loop	Delete only statement within if	Delete condition	Delete loop counter update	Insert early return	Insert early exit	Insert early goto	Insert Tautology	Insert Contradiction
GenProg	4.00%	8.00%	2.00%	14.00%	28.00%	6.00%	4.00%	4.00%	2.00%	2.00%	0%	2.00%	0%	0%
SPR	0%	7.14%	7.14%	14.29%	10.71%	21.43%	7.14%	7.14%	3.57%	7.14%	3.57%	3.57%	7.14%	39.29%
Average	2.00%	7.57%	4.57%	14.14%	19.36%	13.71%	5.57%	5.57%	2.79 %	4.57%	1.79%	2.79%	3.57%	19.65%

fication, i.e., the test-suite, they could in theory benefit from our technique. However, the appropriate integration of our technique in these tools is non-obvious and would require further investigation.

Specification-based Repair. AutofixE [19] uses developer-provided software contracts (e.g., invariants, pre-conditions, and post-conditions) to automatically repair faulty Eiffel classes. Gopinath et al.'s approach encodes constraints on program behavior derived from a specification as a SAT formula and uses the SAT solution to construct a repair [5]. More recently, Samanta et al. proposed the notion of cost-aware program repair whereby repairs are generated on a Boolean abstraction of a C program, based on a cost model, and then concretized back to a repaired C program [24]. However, human intervention is required to define the cost model and for the concretization. As in the case of oracle-based techniques, our ideas could benefit specification-based repair techniques as well but would require further research.

Domain-Specific Repair. Automatic repair techniques have also been developed for specific application domains such as for data-structure repair [4], the Vejovis system for repairing DOM-related faults in JavaScript code [18], and the ClearView system to fix security vulnerabilities [20].

Patterns and Anti-patterns. *Anti-patterns* are "commonly occurring solutions to a problem that generate decidedly negative consequences" [2]. We refer to our proposed set of forbidden transformations as *anti-patterns* because they capture transformations which are solutions to the repair problem, but generate decidedly negative consequences, such as introducing regressions. One notable difference between repair patterns in Pattern-based Automatic program Repair (i.e, PAR [7]) and our *anti-patterns* is that repair patterns are derived from common bug fix patterns in human generated patches, whereas our *anti-patterns* are derived from machine generated patches. Nevertheless, as a subset of SPR "transformation schemas" (e.g., "Add Initialization") are essentially repair patterns, as in PAR [7], checking the effectiveness of *anti-patterns* in SPR helps us to understand the characteristic of patches produced by a pattern-based approach. Prior studies on the common patterns of incorrect human fixes [21, 25, 29] are orthogonal to our work, as our study focuses on the common problems in machine-generated patches.

3. PREVALENCE OF ANTI-PATTERNS

Although various search-based program repair techniques [22, 27, 28] show promising results in generating a large number of patches, prior studies show that most of these patches are often only plausible but incorrect [23]. Specifically, SPR generates 28 out of 40 (i.e, 70%) plausible but incorrect patches, while GenProg generates 50 out of 53 (i.e., 94.33%) plausible patches, for the GenProg benchmarks [10].

To better understand the nature of the plausible patches (see Definition 2 for definition of plausible patches), we performed a manual inspection on all the machine-generated patches produced by SPR and GenProg (including plausible and correct patches) as well as on the correct developer-provided patches for these bugs. Specifically, we manually analyzed each patch and attempted to answer two questions:

Q1: What makes a given patch plausible? Why is it incorrect (i.e., does not capture the semantics of the developer-provided patch)?

Q2: Do the plausible patches, as a whole, share any common syntactic features that explain their "plausibility" as well as distinguish them from the pool of correct patches (human as well as machine generated)?

The aim was to find a compact set of syntactic features that are independent of the repair templates used by the tool. Table 1 shows the results of our manual inspection. Interestingly, our manual inspection identified a set of 14 simple features, shown in the second row of Table 1, one or more of which appear in *each* of the plausible patches produced by SPR and GenProg and almost none of the correct patches [1]. They correspond to various modifications to the control flow or the data flow of the program. Each column corresponds to a specific feature and denotes the percentage of plausible patches bearing that feature. Note that our calculation considers the fact that one plausible patch may exhibit several features. For example, *Delete if-statement* (column 6) deletes an if statement and appears in 28% of plausible GenProg patches and 10.71% of plausible SPR patches. Similarly, *Insert Tautology* (column 14) inserts a trivial tautological condition into the program and appears in 7.14% of the plausible SPR patches but none of the plausible GenProg patches. We further generalize and consolidate these 14 features into 7 transformations, shown in the first row of Table 1, which we chose to further develop as *anti-patterns* in our approach.

Table 2 lists the *anti-patterns* that we identified through our manual inspection [2]. In each example patch in Table 2 and in patches presented throughout the paper, code with a leading "-" denotes statements deleted by the patch, while code with a leading "+" marks statements added. Code without any leading symbol denotes unchanged statements.

4. HOW ANTI-PATTERNS MAY HELP

We illustrate how *anti-patterns* can improve program repair by showing two examples of fixes in two projects. The first example shows the effect of applying *anti-patterns* for improvement in fix localization, while the second example demonstrates the benefit of *anti-patterns* in reducing patch generation time.

Example 1: Improving fix localization. Listing 9 shows the GenProg patch for `findutils-84aef0ea-07b941b1` generated by GenProg. The GenProg patch deletes the only return statement in `parse_noop(...)`. This patch violates our *Anti-delete CFG exit node* pattern. In contrast, our version of GenProg modified with *anti-patterns*,

[1]Except for one PHP defect and one Python defect.

Table 2: Our proposed set of *anti-patterns* with examples that illustrate the usage of each antipattern

Anti-patterns	Example

A1: Anti-delete CFG exit node. This pattern disallows removal of return statements, exit calls, functions with the word "error" (i.e., ignoring letter case), and assertions.	**Ex1:** The example below shows a patch generated by GenProg for `libtiff-8f6338a-4c5a9ec`. The patch removes the erroneous exit call.
	`static void BadPPM(char* file) {`
	`  fprintf(stderr, "%s: Not a PPM file.\n", file);`
	`-  exit(-2);`
	`}`
	Listing 1: Example patch for Anti-delete CFG exit node

A2: Anti-delete Control Statement
This pattern disallows removal of control statements, e.g., if-statements, switch-statements, and loops.
	Ex2: The example below shows a patch generated by GenProg for `php-307931-307934`. The patch removes the whole if-then-else statement that checks for the return value of a function call.
	`call_result = call_user_function_ex(...);`
	`- if (call_result == SUCCESS && retval != NULL && ...) {`
	`-  if (SUCCESS == statbuf_from_array(...))`
	`-  ret = 0;`
	`- } else if (call_result == FAILURE) {`
	`-  php_error_docref(...);`
	`-  }`
	Listing 2: Example patch for Anti-delete Control Statement

A3: Anti-delete Single-statement CFG
This pattern disallows deletion of the statement within a CFG node that has only one statement.
	Ex3: The example below presents a candidate patch generated by GenProg for `libtiff-90d136e4-4c66680f`. The patch removes the statement that assigns the return value of 1 which indicates a failure.
	`fail:{`
	`- ret = 1;`
	`}`
	Listing 3: Example patch for Anti-delete Single-statement CFG

A4: Anti-delete Set-Before-If
This pattern disallows deletion of a variable definition if the variable in the definition is used in subsequent if-statement.
	Ex4: The example shows a GenProg generated candidate patch for `libtiff-d13be72c-ccadf48a`. The patch removes the statement that stores the value of the expression `EstimateStripByteCounts(...)<0`.
	`- tmp = EstimateStripByteCounts(tif, dir, dircount)<0;`
	`if(tmp!=0)`
	`  goto bad;`
	Listing 4: Example patch for Anti-delete Set-Before-If

A5:Anti-delete Loop-Counter Update
Although more sophiscated techniques are needed to ensure termination in patched program, we implement an approximation of this pattern by disallowing deletion of an assignment statement inside a loop if the set of variables used in the terminating condition intersects with the set of variables used in the LHS of the assignment statement.
	Ex5: The example belows shows a patch that delete the increment statement within a loop.
	`while( x> 5)`
	`-  x++;`
	Listing 5: Example: Anti-delete Loop-Counter Update

A6: Anti-append Early Exit
This pattern disallows insertion of return statement and goto statement at any location except for after the last statement in a CFG node.
	Ex6: The example shows a SPR's patch for `php-308262-308315`. The patch adds a conditional return statement before a function call that throws an error.
	`+ if ((type != 0))`
	`+  return;`
	`zend_error((1<<3L),"Uninitialized string offset:",...);`
	Listing 6: Example patch for Anti-append Early Exit

A7: Anti-append Trivial Conditions
This expression-level pattern disallows insertion of trivial conditions. A condition is trivial iff (1) it is either true or false constant (e.g., if (0), if (1)), (2) it is evaluated to true or false by any assignment of the program variables (e.g., if(x || y || !y)), and (3) it is always evaluated to true or false by any values that program variables can take according to results of static analysis (e.g., if(x || y != 0) in which y is initialized).
	Ex7: The following example shows two SPR patches for `lighttpd-2661-2662`. The patch in Listing 7 appends the condition `!(1)` to the existing condition, which is semantically equivalent to disabling the branch containing the continue statement. The patch in Listing 8 loosens the original condition with the expression `1`, which is semantically equivalent to deleting the condition `(fields->size==0)`.
	`-if ((fmap[j].key != format->ptr[i + 1]))`
	`+if ((fmap[j].key != format->ptr[i + 1]) && !(1))`
	`  continue;`
	Listing 7: Example patch with contradiction for Anti-append Trivial Conditions
	`- if ((fields->size == 0)) {`
	`+ if ((fields->size == 0) || (1)) {`
	`  fields->size = 16;`
	`  fields->used = 0;`
	`  fields->ptr = malloc(fields->size * sizeof(format_field *));`
	`}`
	Listing 8: Example patch with tautology for Anti-append Trivial Conditions

Table 3: Problems in search-based program repairs and the corresponding *anti-patterns* aim to solve these problems

Problem	*Anti-patterns*
Weak Oracle. Instead of checking for the actual output of a program, developers may validate the outcome of a failing test by relying on the exit status or assertions of the program. Such statements serve as proxies for verifying the correctness of a program, and thus, they should not be manipulated by machine-generated patches. However, such restrictions are not imposed on automatically generated patches. In fact, patches that simply remove such statements may be more preferable for test-driven program repair techniques as they can be generated faster [23].	***A1: Anti-delete CFG exit node.***
Inadequate Test Coverage. If the program under test has low code coverage, test-driven program repair tools could incorrectly remove a logical block of statements as they are seen as redundant code to the test suite. This may lead to regressions in the patched program [26]	***A2: Anti-delete Control Statement*** ***A3: Anti-delete Single-statement CFG*** ***A4: Anti-delete Set-Before-If***
Mask Existing Vulnerabilities. A patched program may mask previously exposed vulnerability by removing certain branches through implicit data-flow.	***A4: Anti-delete Set-Before-If***
Non-termination. Program repair tools may incorrectly remove a loop update statement, causing infinite loop in the patched program. If no timeout is specified, search-based repair tools may spend the entire repair session to validate the patched program. Worst still, such patches could be mistakenly treated as a repair if the test only checks if an error is thrown within a time limit.	***A5:Anti-delete Loop-Counter Update***
Trivial Patch. An incorrectly patched program may bypass an important functionality or an error check through insertions of premature exit calls. The worst scenario happens when repair tools produce trivial patches that simply insert return-statements based on the value of the expected output of the failing test (e.g., a trivial patch that insert `if(test1) return expected-out;`)	***A6: Anti-append Early Exit***
Functionality Removal Repair tools like SPR may produce patches that are semantically equivalent to functionality removal by inserting tautological condition or contradition. A tautology will cause the elimination of the check condition while a contradiction will cause the entire branch to be removed.	***A7: Anti-append Trivial Conditions***

```
// GenProg AE patch for findutils-84aef0ea-07b941b1
static boolean parse_noop (const struct parser_table*
     entry, char **argv, int *arg_ptr)
{
  (void) entry;
 - return parse_true(get_noop(), argv, arg_ptr);
}
```

Listing 9: Example patch generated by GenProg

```
// mGenProg patch for findutils-84aef0ea-07b941b1
static boolean
insert_regex (char **argv, int *arg_ptr, const struct
     parser_table *entry, int regex_options)
{...
   if (error_message)
     error (1, 0, "%s", error_message);
   (*arg_ptr)++;
-   our_pred->est_success_rate =
     estimate_pattern_match_rate(argv[*arg_ptr], 1);
   return true;

// Developer patch for findutils-84aef0ea-07b941b1
insert_regex (char **argv, int *arg_ptr, const struct
     parser_table *entry, int regex_options)
{...
   if (error_message)
     error (1, 0, "%s", error_message);
-  (*arg_ptr)++;
   our_pred->est_success_rate =
        estimate_pattern_match_rate(argv[*arg_ptr], 1);
+  (*arg_ptr)++;
   return true;
```

Listing 10: Example patches generated by mGenProg and Findutils developer

called mGenProg, removes the statement that assigns the return value of `estimate_pattern_match_rate` to the field `our_pred->est_success_rate`. Meanwhile, Findutils developer moved the statement `(*arg_ptr)++` to the location after the problematic statement `our_pred->est_success_rate=estimate_pattern_match_rate(...);` that throws "Segmentation fault" error due to out-of-bound access of the `argv` array (Listing 10). We argue that in this example, our mGenProg patch is preferable to GenProg's because (1) mGenProg localizes the correct function compared to the GenProg patch, which is applied inside a completely different function `parse_noop();` (2) mGenProg

```
// GenProg, mGenProg & Developer patch for php
     -309111-309159
if ((p = memchr(s, '?', (ue - s)))) {
    pp = strchr(s, '#');
    if (pp && pp < p) {
+   if (pp - s) {
+      ret->path = estrndup(s, (pp-s));
+      php_replace_controlchars_ex(ret->path, (pp - s));
+   }
            p = pp;
            goto label_parse;
    }
}
```

Listing 11: Example patch generated by GenProg, mGenProg and PHP developers

correctly pinpoints the function call that causes the error, while the GenProg patch completely removes the functionality encapsulated by the `parse_noop()` function. This example shows that *anti-patterns* can improve fix localization and eliminate nonsensical patches that remove functionality.

Example 2: Accelerating program repair. Listing 11 shows a patch that inserts a conditional statement that can be copied from other places within the same file. While both GenProg and mGenProg generate the patch in listing 11 that is in fact equivalent to the correct patch, mGenProg takes only 13.7 hours compared to 20.6 hours taken by GenProg (i.e., mGenProg achieves a 20.6/13.7 = 1.5x speedup). Thus, if the correct repair can be found within the repair space, our *anti-patterns* can serve as a search-space pruning mechanism that reduces the time taken to find the correct repair through filtering of invalid patches.

Table 3 shows the common problems in the patches generated by search-based program repair tools together with the *anti-patterns* that solve these problems.

5. INTEGRATING ANTI-PATTERNS

We integrate our *anti-patterns* directly into two search-based repair tools (i.e., GenProg AE [28] and SPR [11]).

Procedure isAntipattern shows our *anti-patterns* filtering algorithm. The function *isSingleCFGStmt*(*E.stmtk*) corresponds to the *Anti-delete Single-Statement CFG* pattern. Similarly, *isSetBefIf*(*E.stk*) checks for the *Anti-delete Set-*

Procedure isAntipattern

Input: P': Program
Input: M: Transformations functions
Output: isAnti: indicates if M violates any *anti-paterns*

```
1  isAnti ← false;
2  if M.type == delete then
3  |  if isSingleCFGStmt(M.stmtk) then
4  |  |  isAnti ← true;
5  |  else if isExitNode(M.stk) || isCondition(M.stk) then
6  |  |  isAnti ← true;
7  |  else if isAssignment(M.stmtk) then
8  |  |  isAnti ← isSetBfIf(M.stk) || isSetInLoop(M.stk);
9  |  end
10 else if M.type == append then
11 |  isAnti ← isExitNode(M.stk) ∧ ¬isLastStmt(M);
12 end
13 return isAnti
```

M.stk: the AST node type of M.
M.type: the edit type of M

Before-If pattern, while $isSetInLoop(E.stk)$ corresponds to the *Anti-delete Loop-Counter Update* pattern. The function $isCondition(E.stk)$ indicates whether an edit E involves a conditional statement, which is used in the *Anti-delete Control-Statement*. The function $isExitNode(E.stk)$ checks if the statement in Edit E is a CFG exit node. Both *Anti-Delete CFG exit node* and *Anti-append Early Exit* use this function. The function $isLastStmt(E)$ checks if a statement will be inserted as the last statement in a CFG block to fulfill the requirement for the *Anti-append Early Exit* pattern. As many search-based approaches [7, 11, 27, 28] are based on evolutionary algorithm [9] in which a population is reproduced, evaluated, and selected, we recommend integrating our *anti-patterns* filtering algorithm before the initial population is generated to reduce the time spent in evaluating each individual in a population.

Modification of SPR and GenProg. Algorithm 1 shows the pseudo-code of the mSPR repair generation algorithm. We implement our *anti-patterns* on two parts of mSPR: (1 – first box). For candidate repairs that do not require condition synthesis, we apply similar modifications to mSPR and mGenProg (refer to Procedure isAntipattern). (2 – second box). For candidate repairs that require condition synthesis in mSPR, we apply the *Anti-append Trivial Conditions* pattern to each synthesized condition. The function $isTrivialCondition(c)$ checks if the given condition c is a trivial condition (refer to Table 2 for definition of trivial conditions). As our modifications on GenProg is similar to the changes on mSPR for repairs that do not require condition synthesis, we leave out the details for mGenProg.

Algorithm 1: mSPR Repair generation algorithm

Input: P: Program
Input: positive and negative test cases NegT and PosT
Input: M: Transformation functions.
Output: the repaired program P' or $\emptyset$ if failed

```
for P' in M(P) do
|  if ¬isAntipattern(P, M) then
|  |  M' ← M' ∪ P;
|  end
end
1
2  for P' in M'(P) do
3  |  if p' contains abstc then
4  |  |  C ← CondSynthesis(P', NegT, PosT);
   |  |  for c in C do
   |  |  |  if ¬isTrivialCondition(c) then
   |  |  |  |  C ← C/c;
   |  |  |  |  if Test(P'[c/abstc],NegT, PosT) then
   |  |  |  |  |  return P'[c/abstc]
   |  |  |  |  end
   |  |  |  end
   |  |  end
5  |  |
6  |  else if Test(P', NegT, PosT) then
7  |  |  return P'
8  |  end
9  end
```

$CondSynthesis(P, NegT, PosT)$: searches for a sequence of values in P that pass all tests in NegT and PosT. The output of this function is C — the set of all synthesized conditions in the repair space.
$P[c/abstc]$: the result of replacing every occurrence of *abstc* in P with the condition c.
$Test(P, NegT, PosT)$: check if the program P passes all tests.

6. EXPERIMENTS

We compare the effectiveness of *anti-patterns* on GenProg and SPR using two sets of benchmarks: (1) the CoREBench benchmarks [1] and (2) the GenProg benchmarks [10]. We use the CoREBench benchmarks for the evaluation set because it contains real errors in widely used C programs. Although our manual inspection for deriving anti-patterns in Section 3 was based on plausible patches from the GenProg benchmarks, this study used just *one* generated patch per buggy program. A recent study has shown that the typical repair search space for these bugs contains up to thousands of plausible patches [12]. Thus, we feel it is still meaningful to study the impact of *anti-patterns* on the complete repair space of these bugs. Our evaluation studies the following research questions:

RQ1 How do *anti-patterns* affect the quality of patches generated by search-based program repair tools?

RQ2 How many nonsensical patches can our *anti-patterns* eliminate to reduce manual inspection costs?

RQ3 When our modified tools produce the same patch, what is the speedup that we achieve?

RQ4 How does the use of *anti-patterns* compare to an approach that simply prohibits deletion?

Table 6: Subject Programs and Their Basics Statistics

Subjects	Description	kLoC	Tests
coreutils	File, Shell and Text manipulation Utility	83.1	4772
findutils	Directory Searching Utility	18.0	1054
grep	Pattern Matching Utility	9.4	1582
make	Program executable generation utilities	35.3	528
php	Programming Language	1046	8471
libtiff	Image Processing Library	77	78
python	Programming Language	407	35
gmp	Math Library	145	146
gzip	Data Compression Utility	491	12
wireshark	Network Packet Analyzer	2814	63
fbc	Compiler	97	773
lighthttpd	Web Server	62	295

6.1 Experimental Setup

We evaluate the effects of *anti-patterns* on 49 defects from the CoREBench benchmarks and at least 37 defects from the GenProg benchmarks. We exclude some versions in our evaluation due to specific technical difficulties, such as benchmarks that require specific system configurations to be built. Specifically, we exclude 21 defects from the CoREBench benchmarks. For the GenProg benchmarks, we manage to reproduce the bugs for 42 defects in the original GenProg experiment and 37 defects in the original SPR experiments.

Table 4: Overall Results on GenProg (AE) versus mGenProg (mAE)

Subjects	Same Patch	Different Patch									Average Speedup (Same Patch)
		Localizes Better				Less Functionality Removal	No Repair		Others		
		Localizes Correct Line		Localizes Correct Function but Incorrect Line							
		AE	mAE	AE	mAE		AE	mAE	AE	mAE	
coreutils	0	0	0	4	4	5	0	0	5	0	-
findutils	4	0	4	2	1	1	0	1	5	0	1.11
grep	4	0	2	3	2	1	0	0	2	0	1.30
make	2	0	1	3	2	0	0	0	0	0	1.77
php	10	1	1	0	2	6	0	0	8	0	2.08
libtiff	3	0	4	3	1	5	0	3	10	0	1.13
python	1	0	0	0	0	0	0	0	0	0	0.98
gmp	-	-	-	-	-	-	-	-	-	-	-
gzip	1	0	0	0	0	0	0	0	0	0	1.12
wireshark	0	0	3	0	0	0	0	0	3	0	-
fbc	-	-	-	-	-	-	-	-	-	-	-
lighthttpd	1	0	0	0	0	1	0	0	1	0	1.85
Total	10+16=26	0+1=1	7+8=15	12+3=15	9+3=12	7+12=19	0+0=0	1+3=4	12+22=34	0+0=0	1.39+1.43=1.42

Table 5: Overall Results on SPR versus mSPR

Subjects	Same Patch	Different Patch									Average Speedup (Same Patch)
		Localizes Better				Less Functionality Removal	No Repair		Others		
		Localizes Correct Line		Localizes Correct Function but Incorrect Line							
		SPR	mSPR	SPR	mSPR		SPR	mSPR	SPR	mSPR	
coreutils	6	0	0	2	2	3	0	0	3	0	1.56
findutils	6	1	2	1	0	1	0	0	1	0	1.62
grep	5	0	1	3	3	2	0	0	3	0	2.15
make	0	0	0	2	2	1	0	0	1	0	-
php	15	0	2	2	0	0	0	0	0	0	1.96
libtiff	2	1	1	1	0	1	0	1	1	0	2.10
python	2	0	0	1	1	0	0	0	0	0	1.50
gmp	2	0	0	0	0	0	0	0	0	0	1.42
gzip	1	0	1	0	0	0	0	0	1	0	1.08
wireshark	3	0	1	1	0	0	0	0	0	0	1.85
fbc	-	-	-	-	-	-	-	-	-	-	-
lighthttpd	0	0	2	1	0	2	0	0	3	0	-
Total	17+25=42	1+1=2	3+7=10	8+6=14	7+1=8	7+3=10	0+0=0	0+1=1	8+5=13	0+0=0	1.78+1.65=1.69

Table 6 lists information about these subjects. The first four rows of the table list the details for the four CoREBench subjects while the remaining rows show relevant statistics about the GenProg subjects. For each bug, we run GenProg, mGenProg, SPR and mSPR to produce repairs.

Many of our *anti-patterns* block functionality deletion, so it is natural to ask if the same effect could be achieved by simply disallowing deletion in repair. To answer RQ4, we implement a customized version of GenProg, called dGenProg, where we disallow the usage of the deletion mutation operator. We reuse the same parameters listed in previous work [10] for running GenProg. One significant difference is that we switch to the deterministic adaptive search algorithm (AE) [28] to control potential randomness. Each run of GenProg, mGenProg, dGenProg, SPR, and mSPR terminates either after all candidate repairs have been evaluated or when a patch is found (i.e., each tool runs to completion without timeout). All experiments for GenProg, mGenProg, and dGenProg were performed by distributing the load on 20 virtual machines with single-core Intel Xeon 2.40GHz processor and 19GB of memory. All experiments for SPR and mSPR were performed on a 12-core Intel Xeon E5-2695 2.40Ghz processor and 408GB of memory.

After collecting all the repairs, we manually inspect each of these patches and compare the quality of patches generated by GenProg versus mGenProg, mGenProg versus dGenProg, and SPR versus mSPR.

Definition 1. We measure the quality of patches generated by search-based repair tools using the criteria defined below:

(Q1) Same Patch. A generated repair is considered "Same Patch" if both the original tool and the modified tool generate exactly the same repair.

(Q2) Localizes Correct Line. A generated repair is considered "Localizes Correct Line" if both the generated patch and the human patch generate repairs that modify the same line. For example, we categorize the mGenProg patch in Listing 10 as "Localizes Correct Line".

(Q3) Localizes Correct Function but Incorrect Line. A generated repair is considered a patch that localizes the correct function if both the generated patch and the human patch modify statements within the same function.

(Q4) Removes Less Functionality. A generated repair is considered a repair that removes less functionality if the repair removes or skips over (e.g., by inserting return) fewer lines of source code from the original program.

(Q5) No Repair. We label a benchmark as "No Repair" when the original tool generates a repair but the modified tool has iterated through the entire repair space and produce no final patch.

We categorize the patch quality of each repair according to the order listed above (i.e, we first check if a patch is "Same Patch" and only categorize a patch as "Removes Less Functionality" if it does not satisfy other more preferable criteria (e.g, "Localizes Correct Function but Incorrect Line"). We eliminate the potential discrepancies on categorization by ensuring that each defined criteria can be measured through comparisons of the syntactic differences between two patches.

Table 7: Patch Correctness Analysis Result on mGenProg and mSPR

Subjects	GenProg		mGenProg		SPR		mSPR	
	Correct	Plausible	Correct	Plausible	Correct	Plausible	Correct	Plausible
coreutils	0	9	0	9	0	11	0	11
findutils	0	11	0	10	0	9	0	9
grep	0	9	0	9	0	11	0	11
make	0	5	0	5	0	3	0	3
php	2	17	1	18	8	9	9	8
libtiff	0	16	0	13	1	4	1	3
python	1	0	1	0	1	2	1	2
gmp	0	0	0	0	1	1	1	1
gzip	0	1	0	1	1	1	1	1
wireshark	0	3	0	3	0	4	0	4
fbc	0	0	0	0	0	0	0	0
lighthttpd	0	2	0	2	0	4	0	4
Total	0+3=3	34+39=73	0+2=2	33+37=70	0+12=12	34+25=59	0+13=13	34+23=57

Each column in Tables 4, 5 and 9 corresponds to the criteria defined above. The "Others" column denotes the cases where the patch does not fulfill any of the defined criteria. Numbers in the last row in Tables 4, 5, and 9 are of the form $x + y = z$, where x represents the number of patches in the CoREBench benchmarks, y denotes the number of patches in the GenProg benchmarks, and z is the total number of patches in both benchmarks.

We also manually classify and compute the number of correct repairs and the number of plausible repairs.

Definition 2. We use the definition below for our patch correctness analysis:

Correct Repair. A repair r is a *correct* repair if (1) r passes all test cases in the test suite and (2) r is semantically equivalent to the repair issued by the developer.

Plausible Repair. A repair r is a *plausible* repair if (1) r passes all test cases in the test suite but (2) r is **not** semantically equivalent to the repair issued by the developer.

6.2 Evaluation on CoREBench benchmarks

The first four rows of table 4, 5, 8, and 9 show the evaluation results for the CoREBench benchmarks.

6.2.1 Patch Quality (RQ1)

Table 4 shows that both GenProg and mGenProg produce the same patch for 10 defects in the CoREBench benchmarks. mGenProg could localize the correct line in 7 more defects than GenProg. mGenProg also generates patches that remove less functionality in 7 defects.

Table 5 shows that both SPR and mSPR produce the same patch in 17 defects. mSPR localizes the correct line in 2 more defects than SPR. For 7 defects, mSPR generates patches that removes less functionality.

Table 7 shows the overall patch correctness analysis results for GenProg, mGenProg, SPR, and mSPR for each subject. GenProg generates 34 plausible patches while mGenProg produces 33 plausible patches for the CoREBench benchmarks. Specifically, mGenProg does not produce any repair for `findutils-e8bd5a2c-66c536b` because the patch violates our *anti-patterns*. Both SPR and mSPR generate 34 plausible patches for the CoREBench benchmarks.

Improvement on fix localization. Our results show that *anti-patterns* could lead both mGenProg and mSPR to producing patches that localize either the correct line or the correct function. *Anti-pattern*-enhanced techniques may achieve this improvement because *anti-patterns* may filter all invalid repairs on a given location, forcing fixes to be generated at other locations. We claim that the ability to localize more precisely is important because when the repair tools fail to generate the correct repair, the next best thing is to check whether they can still generate hints that may lead developers to the repair faster.

Less functionality removal. Under the presence of weak oracles [23], search-based repair tools may generate patches that pass the test suite by removing untested functionality. Our results shows that *anti-patterns* help in producing patches that remove less functionality and thus reduce the potential destructive effects of generated patches.

Comparison between mGenProg and mSPR. Our *anti-patterns* integration achieves greater improvement of patch quality on GenProg compared to SPR. We think that this difference may be due to SPR being innately restricted by its set of transformation schemas, which contain transformations that are often used in human patches.

Predominance of Plausible Patches. Both GenProg and SPR do not generate any correct patch for the CoREBench benchmarks. One possible explanation is that the defects in the CoREBench have higher error complexity than other benchmarks. Thus more substantial patches are required to fix the errors in these benchmarks [1]. These results also agree with our earlier observation (in Section 3) that there is a clear predominance of plausible but incorrect patches among all automatically generated patches.

RQ1: *anti-patterns* direct repair tools towards generating patches that pinpoint the buggy location more accurately. *Anti-patterns* also reduce the potential destructive effect of automatically generated patches by producing patches that remove less functionality.

6.2.2 Reducing Manual Inspection Cost (RQ2)

SPR may produce multiple patches in one repair session due to the use of batch compilation and its staged repair algorithm. Given several candidate repairs, developers need to manually inspect and verify each individual patch.

Figure 1 shows the total number of patches generated by SPR versus mSPR for the CoREBench subjects. As SPR and mSPR can produce multiple patches for a given bug, Figure 1 reports the total number of patches, while the data in Tables 4, 5 and 7 uses a single, best patch (according to the order in Definition 1) among all generated patches for a particular bug. Overall, SPR generates 87 patches while mSPR only generates 54 patches. Our patch analysis reveals that all 33 additional patches generated by SPR (not generated by mSPR) are indeeed plausible but incorrect patches.

Discussion on Number of Plausible Patches. Though prior evaluation of search-based repair [28] focuses on measuring the number of successful repairs, our results show that mSPR actually produces less number of candidate repairs than SPR because some of the plausible patches produced by SPR are actually nonsensical patches that are eliminated

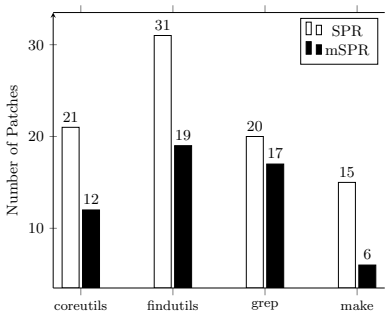

Figure 1: Number of Patches Found by SPR vs. mSPR

by our *anti-patterns*. Producing less plausible patches could save the time spent on manual filtering of invalid patches, which would eventually be rejected by developers.

RQ2: *Anti-patterns* reduce manual inspection cost by eliminating nonsensical candidate repairs.

6.2.3 Speedup (RQ3)

The "Average Speedup (Same Patch)" column in Tables 4, 5, and 9 denotes the average speedup obtained when we only considered the subjects where both the original tool and the modified tool produce the same patch. We use the formula below for our speedup calculation (*Repair Time* is defined as the time taken for a repair to be generated):

$$Repair\ Time\ Speedup = \frac{Original\ Repair\ Time}{Modified\ Repair\ Time} \quad (1)$$

When GenProg and mGenProg produce the same patch, mGenProg obtain an average repair time speedup of 1.39x while mSPR obtain an average repair time speedup of 1.78x, for the CoREBench benchmarks.

Table 8: Subject Programs and Repair Space Reduction Results for mGenProg and mSPR

Subject	Repair Space Reduction(%)	
	mGenProg	mSPR
coreutils	43	22
findutils	47	19
grep	38	32
make	37	36
php	37	20
libtiff	43	61
python	31	26
gmp	-	9
gzip	43	31
wireshark	42	35
fbc	-	-
lighthttpd	41	15
Average	41, 40, 40	27, 28, 28

Table 8 shows the overall reduction in the total number of repair candidates generated for mGenProg and mSPR. The last row is of the form x, y, z where x denotes average for CoREBench subjects, y denotes average for GenProg subjects and z denotes the average for all subjects. We calculate the "Repair Space Reduction" according to the formula below (where $TotC$ refers to the total number of repair candidates within the entire repair search space):

$$Repair\ Space\ Reduction = (1 - \frac{Modified\ TotC}{Orig\ TotC}) * 100 \quad (2)$$

On average, mGenProg achieves 41% repair space reduction compared to GenProg while mSPR obtains 27% repair space reduction compared to SPR for CoREBench subjects.

Discussion on Speedup. Tables 4 and 5 show that by enforcing *anti-patterns*, we produce patches faster due to repair space reduction shown in Table 8. One conceptual argument against the idea of *anti-patterns* may be that it might make the repair search unduly inefficient. These results show that it is not so. In fact, the anti-patterns skip "irrelevant" parts of the repair space (i.e., repairs that causes undesirable behavior, such as the deletion of the symptoms of a bug).

RQ3: *Anti-patterns* reduce the overall repair time by pruning significant portion of the repair space.

6.2.4 Comparison with dGenProg (RQ4)

Table 9 shows the results for dGenProg versus mGenProg for the CoREBench benchmarks. While both mGenProg and dGenProg produce 12 same patches, mGenProg localizes better compared to dGenProg in seven more subjects than dGenProg. *Although dGenProg explicitly prohibits deletions, our results show that mGenProg actually removes less functionality in five subjects compared to dGenProg.* Our analysis reveals that dGenProg may produce patches that skip over many source lines of code by introducing early return. For `make-73e7767f-d584d0c1`, mGenProg localizes the correct line while dGenProg do not produce any repair.

When mGenProg and dGenProg produce the same patch, mGenProg achieves an overall speedup of 1.20x over dGenProg in the CoREBench benchmarks.

Improvement over dGenProg. Our results on the CoRE Bench benchmarks show that GenProg with *anti-patterns* produce patches of better quality and faster than GenProg that simply prohibits deletions.

RQ4: *Anti-patterns* produce patches of better quality and faster than an tool that simply prohibits deletions.

6.3 Evaluation on GenProg benchmarks

The 5-12th rows of Tables 4, 5, 8, and 9 show the experimental results for the GenProg benchmarks. Tables 4 and 5 illustrate that our *anti-patterns* achieve similar improvement on patch quality on the GenProg benchmarks. In particular, mGenProg localizes better than GenProg in seven more defects. mGenProg also removes less functionality in 12 defects. In contrast, mSPR removes less functionality than SPR in three defects on GenProg benchmarks.

Table 7 shows that GenProg produces 3 correct repairs and 39 plausible repairs while mGenProg produces two correct repairs and 37 plausible repairs for the GenProg benchmarks. mGenProg does not generate any repair for three subjects due to their violations of *anti-patterns*. Instead of producing correct repair as in GenProg, mGenProg only generates plausible repairs for `php-309892-309910` because the correct repair actually involves deletion of a if-statement, which violates our *Anti-delete Control Statement* pattern. In contrast, mSPR produces one more correct patch than SPR and 23 plausible repairs. Specifically, mSPR produces correct patch for `php-308262-308315` while SPR only generates plausible patch for this version. For `libtiff-086036-1ba752`, mSPR does not generate any repair while SPR generates patch with trivial condition that disables a branch. As the correct repair for this libtiff defect requires modifications of multiple statements, our analysis reveals that the correct repair is indeed outside of SPR's repair space.

We also achieve similar reduction on repair time on the GenProg benchmarks, as in the CoREBench benchmarks.

***Restrictiveness of* anti-patterns.** Another conceptual argument against the idea of *anti-patterns* may be that *anti-*

Table 9: Overall Results on mGenProg (mAE) versus dGenProg(dAE)

Subjects	Same Patch	Different Patch									Average Speedup (Same Patch)
		Localizes Better									
		Localizes Correct Line		Localizes Correct Function but Incorrect Line		Less Functionality Removal	No Repair		Others		
		dAE	mAE	dAE	mAE		dAE	mAE	dAE	mAE	
coreutils	4	0	0	1	1	3	0	0	3	0	1.80
findutils	3	0	5	1	1	0	0	0	5	0	1.03
grep	2	0	1	3	3	2	0	0	4	1	0.79
make	3	0	1	1	1	0	1	0	0	0	1.18
Total	12	0	7	6	6	5	1	0	12	1	1.20

patterns will be overly restrictive and will rule out any repair in many cases whereas, if an existing search-based tool produces some repair, it still helps the developers to some extent. Our results on the GenProg benchmarks show that *anti-patterns* are not overly restrictive and, in the few cases where it ruled out any repair, indeed no valid repair existed.

Weak Proxies. Our experiments for SPR and mSPR use the updated proxies in previous work [23], which modifies the test harness and the developer test script for `php` and `libtiff`. In contrast, we reuse the weak proxies for our experiments on GenProg and mGenProg. We used the weak oracles for GenProg and the strong oracles for SPR because they are provided together with the original tool distribution. If we compare the row 5-6 of Tables 4 and 5 in which different set of proxies are used, we observe that having a stronger proxy does not help SPR substantially in terms of fix localization. Indeed the improvement of mSPR over SPR in terms of localizing the correct line, is similar to the improvement of mGenProg over GenProg.

Discussion on Patch Correctness. Our results show that enforcing anti-patterns does not necessarily lead to patches that are exactly equivalent to the human patches. This is not entirely unexpected, because *we only mark a generated patch as correct, if it is near identical to the developer provided patch.* Our repair method is driven by a suite of test cases and aims to pass the test-suite while not inserting any of the anti-patterns. It frees the developers from providing different human patch patterns for different defect classes, exception types, vulnerabilities, etc. Nevertheless, mSPR still generates one more correct repair than SPR while mGenProg generates one plausible repair that removes a branch from the original program.

Anti-patterns provides repair space reduction, yet are not overly restrictive.

7. THREATS TO VALIDITY

We identify the threats to validity of our experiments.

Set of *anti-patterns*. Our *anti-patterns* merely represent bug, tool, and language agnostic patterns that we found frequently occurred in bad patches and seldom in correct ones. Though our experimental results show that our proposed *anti-patterns* are effective in eliminating invalid patches, we do not claim that our proposed set is a "complete" set.

Search. We terminate the search for repairs in both GenProg and SPR after a repair has been found, due to limited resources. While both tools support full exploration that may generate similar patches as in our modified versions, such exploration may also lead to increase in the number of invalid patches and longer manual inspection time. As "we use the deterministic adaptive search algorithm (AE) to control potential randomness" (Section 6.1), we will re-evaluate the savings for the stochastic algorithm in future work.

Patch Correctness Analysis. While we tried to assess repair quality across multiple dimensions, our check for semantic equivalence is inherently incomplete and many fixes exist for a particular fault. Our conservative patch analysis classifies a patch as "correct" only when near identical to the human patch. Hence, the number of repairs reported as "correct" may be an underestimate because a plausible patch marked as not correct could very well be semantically equivalent to the developers' provided patch.

Generality of *anti-patterns*. As we only evaluate the effect of *anti-patterns* on CoREBench benchmarks [1] and the GenProg benchmarks [10], our *anti-patterns* may have different effects on other benchmarks. Nevertheless, our experimental results show that *anti-patterns* provide similar benefits at least in both these benchmarks.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed integrating *anti-patterns* to search-based program repair. Our experimental results show that by enforcing *anti-patterns*, we produce patches with more pleasant properties, such as patches that delete less functionality, and localize better. Tools integrated with *anti-patterns* also could generate patches faster due to repair space reduction. A recent study [12] shows the abundance of plausible patches and sparsity of correct patches — thereby arguing for rich specifications (beyond test-suites) to guide the repair process. Our results indicate that our *anti-patterns*, while they are not correctness specifications, form one such set of specifications whose enforcement can improve patch quality.

While in this work we explicitly specified a set of anti-patterns as prohibited code transformations, in future, it is feasible to implicitly specify anti-patterns as selected "code smells". Thus, during the repair search, any program modification that produces a program with a bad code smell could be effectively prohibited.

Our work opens the possibility of adapting the idea of *anti-patterns* to other search-based software engineering activities beyond program repair. For example, specific code *anti-patterns* identifying energy hot-spots may be employed for energy reduction.

In future, we are unlikely to have programming environments that automatically patch all errors without sufficient intervention or domain knowledge. Meanwhile, it might be possible to have programming environments, which attempt to patch programs so as to pass a given test-suite and point the developers to likely error locations and likely fixes. Our proposal of *anti-patterns* is a step in this direction.

Acknowledgments. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

9. REFERENCES

[1] M. Böhme and A. Roychoudhury. CoREBench: Studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 105–115, New York, NY, USA, 2014. ACM.

[2] W. H. Brown, R. C. Malveau, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998.

[3] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 30–39, New York, NY, USA, 2014. ACM.

[4] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 233–244, New York, NY, USA, 2006. ACM.

[5] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using sat. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188. Springer, 2011.

[6] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 266–276, New York, NY, USA, 2014. ACM.

[7] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[8] A. Koenig. Patterns and antipatterns. *Journal of Object Oriented Programming*, 8(1), 1995.

[9] J. R. Koza. *Genetic Programming: On the Programming of computers by Means of Natural Selection*. MIT Press, 1992.

[10] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[11] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[12] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 702–713, New York, NY, USA, 2016. ACM.

[13] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[14] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE 2015, pages 448–458. ACM, 2015.

[15] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[16] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[18] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for javascript faults. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 837–847, New York, NY, USA, 2014. ACM.

[19] Y. Pei, C. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *Software Engineering, IEEE Transactions on*, 40(5):427–449, May 2014.

[20] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[21] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes-extended abstract. In *International Workshop on Mining Software Repositories (MSR 2004), International Conference on Software Engineering*, pages 90–94, 2004.

[22] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, 2014.

[23] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. ACM.

[24] R. Samanta, O. Olivo, and E. Emerson. Cost-aware automatic program repair. In M. Müller-Olm and H. Seidl, editors, *Static Analysis Symposium*, volume 8723 of *Lecture Notes in Computer Science*, pages 268–284. Springer International Publishing, 2014.

[25] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software*

Repositories, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[26] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Bergamo, Italy*, 2015.

[27] S. H. Tan and A. Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE 2015, pages 471–482. ACM, 2015.

[28] W. Weimer, Z. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013.

[29] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.