Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2015-008 March 11, 2015

Staged Program Repair in SPR

Fan Long and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Staged Program Repair in SPR

Fan Long and Martin Rinard
MIT EECS & CSAIL
{fanl, zichaoqi, sarachour, rinard}@csail.mit.edu

ABSTRACT

We present SPR, a new program repair system that uses *condition synthesis* to instantiate *transformation schemas* to repair program defects. SPR's staged repair strategy combines a rich space of potential repairs with a targeted search algorithm that makes this space viably searchable in practice. This strategy enables SPR to successfully find correct program repairs within a space that contains many meaningful and useful patches. The majority of these correct repairs are not within the search spaces of previous automatic program repair systems.

1. INTRODUCTION

We present a new program repair system, SPR, that uses a novel *staged repair* strategy to generate and efficiently search a space of candidate program transformations. After using fault localization to identify target statements to transform [41], the first stage selects a (non-semantics preserving) parameterized transformation schema. The second stage instantiates the parameter to obtain a transformation that it applies to the target statement. SPR validates the transformation against a test suite of test cases. The suite contains *positive test cases*, for which the program already produces correct outputs, and *negative test cases*, which expose a defect that causes the program to produce incorrect outputs. SPR searches the transformation space to find a transformation that produces correct outputs for all test cases.

Transformation-based repair exhibits an inherent tradeoff between the size and sophistication of the transformation search space and the time required to search the space — larger spaces may contain more correct repairs but may take longer to search. Staged repair promotes the development of targeted schema instantiation algorithms that enable the system to efficiently search the space of potential repairs. SPR's first stage contains transformation schemas that take conditions as parameters. SPR's second stage uses a *condition synthesis* algorithm to find successful conditions for these parameters. The following SPR schemas take a condition as a parameter and rely on condition synthesis to efficiently find the condition:

- **Condition Refinement:** Given a target if statement, SPR transforms the condition of the if statement by conjoining or disjoining an additional (synthesized) target condition to the original if condition.
- **Condition Introduction:** Given a target statement, SPR transforms the program so that the statement executes only if a (synthesized) target condition is true.
- **Conditional Control Flow Introduction:** SPR inserts a new control flow statement (return, break, or goto an existing label) that executes only if a (synthesized) target condition is true.

SPR stages the condition synthesis as follows. It first finds branch directions that deliver correct results on all inputs in the test suite. It then generates a target condition that successfully approximates these branch directions:

- **Semantics-Preserving Target Conditions:** Each transformation schema has a target condition value that preserves the semantics of the program. For Condition Transformation, the semantics-preserving value is true for conjoined conditions and false for disjoined conditions. For Condition Introduction, the semantics-preserving value is true (so that the target statement always executes). For Conditional Control-Flow Introduction, the semantics-preserving value is false (so that the introduced control flow statement never executes).
- **Target Condition Value Search:** SPR applies the schema to the program, using an abstract target condition for the parameter. SPR then repeatedly reexecutes the transformed program on the negative test cases. On each execution it generates a sequence of 0/1 values for the abstract condition (the sequence contains one value for each execution of the abstract condition). The goal is to find sequences of target condition values that produce correct outputs for all negative test cases.
- **Instrumented Executions:** SPR executes instrumented versions of the program on both negative and positive test cases. For each negative test case, it uses the discovered target condition values that produce the correct output. For each positive test case, it uses the semantics-preserving target condition values. For each execution of the abstract target condition, the instrumentation records a mapping from the values of variables in the surrounding context to the corresponding target condition value.
- **Condition Generation:** The goal is to obtain a symbolic condition, over the variables in the surrounding context, that generates the same target condition values as those recorded in the successful instrumented executions. In practice, when a correct repair exists in the search space, it is almost always possible to realize this goal. In general, however, a correct repair

may not need to exactly match the target condition values [36]. SPR therefore does not require an exact match — it works with the recorded mappings to generate symbolic conditions that maximize the number of generated target condition values that are the same as the target condition values recorded in the previous instrumented executions (in many but not all cases these symbolic conditions match all of the recorded target condition values).

- **Condition Test:** SPR instantiates the schema with the generated condition, applies the transformation, and runs the transformed program to determine if transformed program produces correct outputs for all test cases. If not, SPR proceeds on to test the next generated condition in the sequence.

Staging condition synthesis as target condition value search followed by condition generation is the key to efficient condition synthesis. In practice, well over 99% of the schema applications are rejected in the target value search stage (when the target value search fails to find values that produce correct results), eliminating the need to perform any condition search at all for the overwhelming majority of schema applications. And when the target value search succeeds, the resulting values enable SPR to quickly find conditions that are likely to deliver a successful repair.

In addition to the condition-oriented schemas, SPR also contains several schemas that focus on changing the values of target program variables.

- **Insert Initialization:** For each identified statement, SPR generates repairs that insert a memory initialization statement before the identified statement.

- **Value Replacement:** For each identified statement, SPR generates repairs that either 1) replace one variable with another, 2) replace an invoked function with another function, or 3) replace a constant with another constant.

- **Copy and Replace:** For each identified statement, SPR generates repairs that copy an existing statement to the program point before the identified statement and then apply a Value Replacement transformation.

The rationale for the Copy and Replace modification is to exploit redundancy in the program — many successful program modifications can be constructed from code that already exists in the program [9]. The goal of the Replace part of Copy and Replace is to obtain a rich repair search space that includes variable replacement to enable copied code to operate successfully in a new naming context. The success of these modifications is consistent with previous work that shows that, without replacement, only approximately 10% of developer changes can be fully derived from existing statements without modification [27, 9, 17].

1.1 Experimental Results

We evaluate SPR on 69 real world defects and 36 functionality changes from the repositories of seven large real world applications, libtiff, lighttpd, the PHP interpreter, fbc, gzip, wireshark, and python. This is the same benchmark set used to evaluate several previous automatic patch generation systems [24, 39].[1] The SPR search space contains transformations that correctly repair 19 of the 69 defects (and 1 of the 36 functionality changes). For 11 of these 19 defects (and 1 of the 36 functionality changes), SPR finds a correct repair (i.e., a repair that completely eliminates the defect for all inputs, see Section 3) as the first repair in the search space that delivers correct outputs for all test cases. These correct repairs semantically match subsequent repairs provided by human developers. For comparison, GenProg [24] finds correct patches for only 1 of the 69 defects [33] and AE [39] finds correct patches for only 2 of the 69 defects [33]. These results are consistent with the fact that the correct SPR transformations for 17 of the 19 defects are outside the GenProg/AE search space. Kali, an automatic patch generation system that only deletes functionality, also finds 2 correct patches [33].

We define a repair to be *plausible* if it produces correct results for all of the inputs in the test suite used to validate the repair (a plausible repair may be incorrect — it may produce incorrect outputs for inputs not in the test suite). SPR generates plausible repairs for 37 of the 69 defects (and 3 of the 36 functionality changes). For comparison, GenProg [24] generates plausible patches for only 16 of the 69 defects (and 2 of the 36 functionality changes) [33]. AE [39] generates plausible patches for only 25 of the 69 defects (and 2 of the 36 functionality changes) [33].[2] Kali also generates plausible patches for 25 defects (and 2 functionality changes) [33].

These results highlight the success of SPR's staged approach and the synergistic relationship between its transformation schemas and condition synthesis algorithm. The transformation schemas generate a rich transformation space with many useful repairs. The condition synthesis algorithm provides the efficient search algorithm required to make this rich search space viably searchable in practice.

1.2 Contributions

This paper makes the following contributions:

- **Staged Repair:** It introduces a staged repair technique for generating and efficiently searching rich repair spaces that contain many useful repairs.

- **Search Space:** It presents a set of transformation schemas that 1) generate a search space with many useful repairs and 2) synergistically enable the development of a staged repair system that uses condition synthesis to efficiently search the generated space.

[1]Papers for these previous systems report that this benchmark set contains 105 defects [24, 39]. Our analysis of the commit logs and applications indicates that 36 of these defects correspond to deliberate functionality changes. That is, for 36 of these defects, there is no actual defect to repair. We evaluate SPR on all 105 defects/changes, but report results separately for the actual defects and functionality changes.

[2]Because of errors in the patch evaluation scripts, the GenProg and AE papers report incorrect results [33]. Specifically, the GenProg paper reports patches for 55 of the 105 defects/changes. But for 37 of these 55 defects/changes, none of the reported patches produces correct outputs even for the inputs in the test suite used to validate the patches [24, 33]. Similarly, for 27 of the 54 defects/changes reported in the AE result tar file [39, 33], none of the reported patches produces correct outputs for the inputs in the test suite used to validate the patches [24, 39, 33].

- **Condition Synthesis:** It presents a novel condition synthesis algorithm. This algorithm first uses instrumented executions to discover branch directions that produce correct outputs for all inputs in the test suite. It then generates symbolic conditions that successfully approximate the set of branch directions. This condition synthesis algorithm enables SPR to efficiently search the space of conditions in the SPR search space.

- **Experimental Results:** It presents experimental results that characterize the effectiveness of SPR in automatically finding correct repairs for 11 out of 69 benchmark defects and plausible repairs for 37 of these defects. The results also show that the SPR search space contains correct repairs for 19 of the 69 defects. We discuss several extensions to the SPR search space and identify the correct repairs that each extension would bring into the search space.

Copying statements and manipulating conditionals are not new ideas in program repair. But by themselves, these techniques fail to deliver anything close to a successful program repair system [24, 32, 39, 11, 33]. A more targeted approach may produce more successful repairs [22, 26, 31], but only within a limited and potentially very fragile scope. SPR, with its unique combination of powerful transformation schemas and an efficient search algorithm driven by condition synthesis, shows one way to productively overcome the limitations of previous systems and automatically generate repairs for a significant number of defects.

2. EXAMPLE

We next present an example that illustrates how SPR repairs a defect in the PHP interpreter. The PHP interpreter before 5.3.7 (or svn version before 309580) contains an error (PHP bug #54283) in its implementation of the DatePeriod object constructor [5]. If a PHP program calls the DatePeriod constructor with a single `NULL` value as the parameter (e.g., `DatePeriod(NULL)`), the PHP interpreter dereferences an uninitialized pointer and crashes.

Figure 1 presents simplified source code (from the source file `ext/date/php_date.c`) that is responsible for this error. The code in Figure 1 presents the C function inside the PHP interpreter that implements the DatePeriod constructor. The PHP interpreter calls this function to handle DatePeriod constructor calls in PHP programs.

A PHP program can invoke the DatePeriod constructor with either three parameters or a single string parameter. If the constructor is invoked with three parameters, one of the two calls to `zend_parse_parameter_ex()` on lines 9-15 will succeed and set `interval` to point to a created DateInterval PHP object. These two calls leave `isostr_len` and `isostr` unchanged. If the constructor is invoked with a single string parameter, the third call to `zend_parse_parameter_ex()` on lines 17-18 parses the parameters and sets `isostr` to point to a created PHP string object. `isostr_len` is the length of this string.

The then clause in lines 35-38 is designed to process calls with one parameter. The defect is that the programmer assumed (incorrectly) that all one parameter calls will set `isostr_len` to a non-zero value. But if the constructor is called with a null string, `isostr_len` will be zero. The if condition at line 34 misclassifies the call as a three parameter call and executes the else clause in lines 40-44. In this

```
1  // Creates new DatePeriod object.
2  PHP_METHOD(DatePeriod, __construct) {
3    php_period_obj   *dpobj;
4    char *isostr = NULL;
5    int   isostr_len = 0;
6    zval *interval;
7    ...
8    // Parse (DateTime,DateInterval,int)
9    if (zend_parse_parameters_ex(..., &start, date_ce_date,
10     &interval, date_ce_interval, &recurrences,
11     &options)==FAILURE) {
12     // Parse (DateTime, DateInterval, DateTime)
13     if (zend_parse_parameters_ex(..., &start, date_ce_date,
14       &interval, date_ce_interval, &end, date_ce_date,
15       &options)==FAILURE) {
16       // Parse (string)
17       if (zend_parse_parameters_ex(...,&isostr,
18         &isostr_len, &options)==FAILURE) {
19         php_error_docref(...,"This constructor accepts"
20          " either (DateTime, DateInterval, int) OR"
21          " (DateTime, DateTimeInterval, DateTime)"
22          " OR (string) as arguments.");
23         ...
24         return;
25       } } }
26   dpobj = ...;
27   dpobj->current = NULL;
28 // repair transformation schema
29 /* if (isostr_len || abstract_cond() ) */
30 // instantiated repair. abstract_code() -> (isostr != 0)
31 /* if (isostr_len || (isostr != 0)) */
32 // developer patch
33 /* if (isostr)*/
34   if (isostr_len) {
35     // Handle (string) case
36     date_period_initialize(&(dpobj->start), &(dpobj->end),
37       &(dpobj->interval), &recurrences, isostr, isostr_len);
38     ...
39   } else {
40     // Handle (DateTime,...) cases
41     /* pass uninitialized 'interval' */
42     intobj  = (php_interval_obj *)
43       zend_object_store_get_object(interval);
44     ...
45   }
46   ...
47 }
```

Figure 1: Simplified Code for PHP bug #54283

case `interval` is uninitialized and the program will crash when the invoked `zend_object_store_get_object()` function dereferences `interval`.

We apply SPR to automatically generate a repair for this error. Specifically, we give SPR:

- **Program to Repair:** Version 309579 of the PHP source code (this version contains the error).

- **Negative Test Cases:** Test cases that expose the error — i.e., test cases that PHP version 30979 does not pass but the repaired version should pass. In this example there is a single negative test case.

- **Positive Test Cases:** Test cases that prevent regression — i.e, test cases that the version 30979 already passes and that the patched code should still pass. In this example there are 6974 positive test cases.

Error Localization: SPR compiles the PHP interpreter with additional profiling instrumentation to produce execution traces. It then executes this profiling version of PHP on both the negative and positive test cases. SPR observes that the negative test case always executes the statement at

lines 42-43 in Figure 1 while the positive test cases rarely execute this statement. SPR therefore identifies the enclosing if statement (line 34) as a high priority repair target.

First Stage: Select Transformation Schema: SPR selects transformation schemas to apply to the repair target. One of these schemas is a Condition Refinement schema that loosens the if condition by disjoining an abstract condition `abstract_cond()` to the if condition.

Second Stage: Condition Synthesis: SPR uses condition synthesis to instantiate the abstract condition `abstract_cond()` in the selected transformation schema.

- **Target Condition Value Search:** SPR replaces the target if statement on line 34 with the transformation schema on line 29. The transformation schema takes an abstraction condition `abstract_cond()` as a parameter. SPR links PHP against a SPR library that implements `abstract_cond()`. Note that if `abstract_cond()` always returns 0, the semantics of PHP does not change.

  SPR searches for a sequence of return values from `abstract_cond()` that causes PHP to produce the correct result for the negative test case. SPR repeatedly executes PHP on the negative test case, generating a different sequence of 0/1 return values from `abstract_cond()` on each execution. In the example, flipping the return value of the last invocation of `abstract_cond()` from 0 to 1 produces the correct output.

- **Instrumented Reexecutions:** SPR instruments the code to record, for each invocation of `abstract_cond()`, a mapping from the values of local variables, accessed global variables, and values accessed via pointers in the surrounding context to the `abstract_cond()` return value. SPR reexecutes the negative test case with the sequence of `abstract_cond()` return values that produces the correct output. It also reexecutes the positive test cases with an all 0 sequence of `abstract_cond()` return values (so that these reexecutions preserve the correct outputs for the positive test cases).

- **Condition Generation:** SPR uses the recorded mappings to generate a symbolic condition that approximates the mappings. In the example, `isostr` is never 0 in the negative test case execution. In the positive test case executions, `isostr` is always zero when `abstract_cond()` is invoked (note that || is a short circuit operator). SPR therefore generates the symbolic condition `(isostr != 0)` as the parameter.

- **Condition Validation:** SPR reexecutes the PHP interpreter with `abstract_cond()` replaced with `(isostr != 0)`. The PHP interpreter passes all test cases and SPR has found a successful repair (lines 30-31 in Figure 1).

Note that the official patch from the PHP developer in version 309580 replaces `isostr_len` with `isostr` (line 33 in Figure 1). At this program point, `isostr_len` is zero whenever `isostr` is zero. The SPR repair is therefore functionally equivalent to the official patch from the PHP developer.

c := `1` | `0` | c_1 `&&` c_2 | c_1 `||` c_2 | `!`c_1 | `(`c_1`)` | v`==`*const*
simps := v `=` v_1 `op` v_2 | v `=` c | `print` v | v `=` `read`
ifs := `if (`c`)` ℓ_1 ℓ_2
absts := `if (`c `&& !abstc)` ℓ_1 ℓ_2 | `if (`c `|| abstc)` ℓ_1 ℓ_2
s := `skip` | `stop` | *simps* | *ifs* | *absts*
v, v_1, v_2 ∈ `Variable` *const* ∈ `Int` ℓ_1, ℓ_2 ∈ `Label`
c, c_1, c_2 ∈ `CondExpr` s ∈ `Stmt` *ifs* ∈ `IfStmt`
simps ∈ `SimpleStmt` *absts* ∈ `AbstCondStmt`

Figure 2: The language statement syntax

3. DESIGN

The current implementation of SPR works with applications written in the C programming language. We use a simple imperative core language (Section 3.1) to present the key concepts in the SPR repair algorithm. Section 3.2 presents the transformation schemas as they apply to the core language. Section 3.3 presents the core repair algorithm, including condition synthesis. Section 3.4 discusses how we extend the core algorithm to handle C.

3.1 Core Language

Language Syntax: Figure 2 presents the syntax of the core language that we use to present our algorithm. A program in the language is defined as $\langle p, n\rangle$, where p : Label → Statement maps each label to the corresponding statement, n : Label → Label maps each label to the label of the next statement to execute in the program. ℓ_0 is the label of the first statement in the program.

The language in Figure 2 contains arithmetic statements and if statements. An if statement of the form "`if (`c`)` ℓ_1 ℓ_2" transfers the execution to ℓ_1 if c is 1 and transfers the execution to ℓ_2 if c is 0. The language uses if statements to encode loops. A statement of the form "v `= read`" reads an integer value from the input and stores the value to the variable v. A statement of the form "`print` v" prints the value of the variable v to the output. Conditional statements that contain an abstract condition `abstc` (i.e., `AbstCondStmt`) are temporary statements that the algorithm may introduce into a program during condition synthesis. Such statements do not appear in the original or repaired programs.

Operational Semantics: A program state $\langle \ell, \sigma, I, O, D, R, S\rangle$ is composed of the current program point (a label ℓ), the current environment that maps each variable to its value (σ : Variable → Int), the remaining input (I), the generated output (O), a sequence of future abstract condition values (D), a sequence of recorded abstract condition values (R), and a sequence of recorded environments for each abstract condition execution (S). I and O are sequences of integer values (i.e., Sequence(Int)). D and R are sequences of zero or one values (i.e., Sequence(0 | 1)). S is a sequence of environments (i.e., Sequence(Variable → Int)).

Figure 3 presents the small step operational semantics of our language for statements without abstract conditions. "∘" in Figure 3 is the sequence concatenation operator. The notation "$\sigma \vdash c \Rightarrow x$" indicates that the condition c evaluates to x under the environment σ. D, R, and S are unchanged in these rules because these statements do not contain an abstract condition.

3.2 Transformation Schemas

$$\frac{p(\ell) = \texttt{skip}}{\langle \ell, \sigma, I, O, D, R, S\rangle \models\!\![\ \langle p, n\rangle\ ]\!\!\Rightarrow \langle n(\ell), \sigma, I, O, D, R, S\rangle} \qquad \frac{p(\ell) = \texttt{stop}}{\langle \ell, \sigma, I, O, D, R, S\rangle \models\!\![\ \langle p, n\rangle\ ]\!\!\Rightarrow \langle \texttt{nil}, \sigma, I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = v\ \texttt{=}\ const}{\langle \ell, \sigma, I, O, D, R, S\rangle \models\!\![\ \langle p, n\rangle\ ]\!\!\Rightarrow \langle n(\ell), \sigma[\mathrm{v} \mapsto const], I, O, D, R, S\rangle} \qquad \frac{p(\ell) = v\ \texttt{=}\ v_1\ \texttt{op}\ v_2 \qquad x = \sigma(v_1)\ \mathrm{op}\ \sigma(v_2)}{\langle \ell, \sigma, I, O, D, R, S\rangle \models\!\![\ \langle p, n\rangle\ ]\!\!\Rightarrow \langle n(\ell), \sigma[\mathrm{v} \mapsto x], I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = v\ \texttt{=}\ \texttt{read} \qquad I = x \circ I\prime}{\langle \ell, \sigma, I, O, D, R, S\rangle \models\!\![\ \langle p, n\rangle\ ]\!\!\Rightarrow \langle n(\ell), \sigma[\mathrm{v} \mapsto x], I\prime, O, D, R, S\rangle} \qquad \frac{p(\ell) = \texttt{print}\ v}{\langle \ell, \sigma, I, O, D, R, S\rangle \models\!\![\ \langle p, n\rangle\ ]\!\!\Rightarrow \langle n(\ell), \sigma, I, O \circ \sigma(v), D, R, S\rangle}$$

$$\frac{p(\ell) = \texttt{if}\ (c)\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1}{\langle \ell, \sigma, I, O, D, R, S\rangle \models\!\![\ \langle p, n\rangle\ ]\!\!\Rightarrow \langle \ell_1, \sigma, I, O, D, R, S\rangle} \qquad \frac{p(\ell) = \texttt{if}\ (c)\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0}{\langle \ell, \sigma, I, O, D, R, S\rangle \models\!\![\ \langle p, n\rangle\ ]\!\!\Rightarrow \langle \ell_2, \sigma, I, O, D, R, S\rangle}$$

Figure 3: Small step operational semantics for statements without abstract condition

$$\begin{array}{lcl}
\mathrm{M}(\langle p, n\rangle) & = & \mathrm{M}_{\mathrm{IfStmt}}(\langle p, n\rangle) \cup \mathrm{M}_{\mathrm{Stmt}}(\langle p, n\rangle) \\
\mathrm{M}_{\mathrm{IfStmt}}(\langle p, n\rangle) & = & \cup_{\ell \in \mathit{TargetL}(\langle p,n\rangle), p(\ell) \in \mathrm{IfStmt}}(\mathrm{M}_{\mathrm{Tighten}}(\langle p, n\rangle, \ell) \cup \mathrm{M}_{\mathrm{Loosen}}(\langle p, n\rangle, \ell)) \\
\mathrm{M}_{\mathrm{Stmt}}(\langle p, n\rangle) & = & \cup_{\ell \in \mathit{TargetL}(\langle p,n\rangle)}(\mathrm{M}_{\mathrm{Control}}(\langle p, n\rangle, \ell) \cup \mathrm{M}_{\mathrm{Init}}(\langle p, n\rangle, \ell) \cup \mathrm{M}_{\mathrm{Guard}}(\langle p, n\rangle, \ell) \cup \mathrm{M}_{\mathrm{Rep}}(\langle p, n\rangle, \ell) \cup \mathrm{M}_{\mathrm{CopyRep}}(\langle p, n\rangle, \ell)) \\
\mathrm{M}_{\mathrm{Tighten}}(\langle p, n\rangle, \ell) & = & \{\langle p[\ell \mapsto \text{if } (c\ \texttt{\&\& !abstc})\ \ell_1\ \ell_2], n\rangle\}, \text{ where } p(\ell) = \text{if } (c)\ \ell_1\ \ell_2 \\
\mathrm{M}_{\mathrm{Loosen}}(\langle p, n\rangle, \ell) & = & \{\langle p[\ell \mapsto \text{if } (c\ \texttt{||}\ \texttt{abstc})\ \ell_1\ \ell_2], n\rangle\}, \text{ where } p(\ell) = \text{if } (c)\ \ell_1\ \ell_2 \\
\mathrm{M}_{\mathrm{Control}}(\langle p, n\rangle, \ell) & = & \{\langle p[\ell\prime \mapsto p(\ell)][\ell\prime\prime \mapsto \texttt{stop}][\ell \mapsto \text{if } (0\ \texttt{||}\ \texttt{abstc})\ \ell\prime\prime\ n(\ell)], n[\ell\prime \mapsto n(\ell)][\ell \mapsto \ell\prime][\ell\prime\prime \mapsto \ell\prime]\rangle\} \\
\mathrm{M}_{\mathrm{Guard}}(\langle p, n\rangle, \ell) & = & \{\langle p[\ell\prime \mapsto p(\ell)][\ell \mapsto \texttt{if}\ (1\ \texttt{\&\&}\ \texttt{!abstc})\ \ell\prime\ n(\ell)], n[\ell\prime \mapsto n(\ell)]\rangle\} \\
\mathrm{M}_{\mathrm{Init}}(\langle p, n\rangle, \ell) & = & \{\langle p[\ell\prime \mapsto p(\ell)][\ell \mapsto v\ \texttt{=}\ 0], n[\ell\prime \mapsto n(\ell)][\ell \mapsto \ell\prime]\rangle \mid \forall v \in \mathit{Vars}(p(\ell))\} \\
\mathrm{M}_{\mathrm{Rep}}(\langle p, n\rangle, \ell) & = & \{\langle p[\ell \mapsto s], n\rangle \mid s \in \mathrm{RepS}(p, p(\ell))\} \\
\mathrm{M}_{\mathrm{CopyRep}}(\langle p, n\rangle, \ell) & = & \{\langle p[\ell\prime \mapsto p(\ell)][\ell \mapsto s], n[\ell\prime \mapsto n(\ell)][\ell \mapsto \ell\prime]\rangle, \\
 & & \ \langle p[\ell\prime \mapsto p(\ell)][\ell \mapsto s\prime)], n[\ell\prime \mapsto n(\ell)][\ell \mapsto \ell\prime]\rangle \mid \forall s \in \mathit{SimpleS}(\langle p, n\rangle),\ \forall s\prime \in \mathrm{RepS(p, s)}\} \\
\mathrm{RepS}(p, v\ \texttt{=}\ v_1\ \texttt{op}\ v_2) & = & \{v\prime\ \texttt{=}\ v_1\ \texttt{op}\ v_2, v\ \texttt{=}\ v\prime\ \texttt{op}\ v_2, v\ \texttt{=}\ v_1\ \texttt{op}\ v\prime \mid \forall v\prime \in \mathit{Vars}(p)\} \\
\mathrm{RepS}(p, v\ \texttt{=}\ const) & = & \{v\prime\ \texttt{=}\ const, v\ \texttt{=}\ const\prime \mid \forall v\prime \in \mathit{Vars}(p),\ \forall const\prime \in \mathit{Consts}(p)\} \\
\mathrm{RepS}(p, v\ \texttt{=}\ \texttt{read}) & = & \{v\prime\ \texttt{=}\ \texttt{read} \mid \forall v\prime \in \mathit{Vars}(p)\} \\
\mathrm{RepS}(p, \texttt{print}\ v) & = & \{\texttt{print}\ v\prime \mid \forall v\prime \in \mathit{Vars}(p)\} \\
\mathrm{RepS}(p, s) & = & \emptyset, \text{ where } s \notin \texttt{SimpleStmt}
\end{array}$$

Figure 4: The program transformation function M. Note that $\ell\prime$ and $\ell\prime\prime$ are fresh labels.

Figure 4 presents our program transformation function M. It takes a program $\langle p, n\rangle$ and produces a set of candidate modified programs after transformation schema application. In Figure 4 $\mathit{TargetL}(\langle p, n\rangle)$ is the set of labels of target statements to transform. Our error localization algorithm (Section 3.4) identifies this set of statements. $\mathit{SimpleS}(p)$ denotes all simple statements (i.e. `SimpleStmt`) in p. $\mathit{Vars}(p)$ and $\mathit{Vars}(s)$ denote all variables in the program p and in the statement s, respectively. $\mathit{Consts}(p)$ denotes all constants in p.

The transformation function in Figure 4 applies condition refinement schemas ($\mathrm{M}_{\mathrm{Tighten}}$, $\mathrm{M}_{\mathrm{Loosen}}$) to all target if statements. It applies the condition introduction schema ($\mathrm{M}_{\mathrm{Guard}}$) and conditional control flow introduction schema ($\mathrm{M}_{\mathrm{Control}}$) to all target statements. These four schemas introduce an abstract condition into the generated candidate programs that will be handled by condition synthesis.

The transformation function also applies the insert initialization schema ($\mathrm{M}_{\mathrm{Init}}$), value replacement schema ($\mathrm{M}_{\mathrm{Rep}}$), and copy and replace schema ($\mathrm{M}_{\mathrm{CopyRep}}$) to all target statements. Note that $\mathrm{RepS}(p, s)$ is an utility function that returns the set of statements generated by replacing a variable or a constant in s with other variables or constants in p.

3.3 Main Algorithm with Condition Synthesis

Figure 5 presents our main repair generation algorithm with condition synthesis. Given a program $\langle p, n\rangle$, a set of positive test cases PosT, and a set of negative test cases NegT, the algorithm produces a repaired program $\langle p\prime, n\prime\rangle$ that passes all test cases. $\mathrm{Exec}(\langle p, n\rangle, I, D)$ at lines 13 and 21 produces the results of running the program $\langle p, n\rangle$ on the input I given the future abstract condition value sequence D. $\mathrm{Test}(\langle p, n\rangle, \mathit{NegT}, \mathit{PosT})$ at lines 29 and 35 produces a boolean to indicate whether the program $\langle p, n\rangle$ passes all test cases. See Figure 6 for relevant definitions.

The algorithm enumerates all transformed candidate programs in the search space derived from our transformation function $\mathrm{M}(\langle p, n\rangle)$ (line 1). If the candidate program does not contain an abstract condition, the algorithm simply uses Test to validate the program with test cases (lines 35-36). Otherwise the algorithm applies condition synthesis in two stages, condition value search and condition generation.

Condition Value Search: We augment the operational semantics in Figure 3 to handle statements with an abstract condition. Figure 7 presents the additional rules. The first two rules specify the case where the result of the condition does not depend on the abstract condition (the semantics implements short-circuit conditionals). In this case the execution is transfered to the corresponding labels with D, R, and S unchanged. The third and the fourth rules specify the case where there are no more future abstract condition values in D for the abstract condition `abstc`. These rules use the semantics-preserving value for the abstract condition `abstc`, with R and S appropriately updated. The last four rules specify the case where D is not empty. In this case the execution continues as if the abstract condition returns the next value in the sequence D, with R, and S updated accordingly.

For each negative test case, the algorithm in Figure 5 searches a sequence of abstract condition values with the

Input : original program $\langle p, n\rangle$
Input : positive and negative test cases $NegT$ and $PosT$, each is a set of pairs $\langle I, O\rangle$ where I is the test input and O is the expected output.
Output: the repaired program $\langle p\prime, n\prime\rangle$, or $\emptyset$ if failed

```
1  for ⟨p′, n′⟩ in M(⟨p, n⟩) do
2      if p′ contains abstc then
3          R′ ⟵ ϵ
4          S′ ⟵ ϵ
5          for ⟨I, O⟩ in NegT do
6              ⟨O′, R, S⟩ ⟵ Exec(⟨p′, n′⟩, I, ϵ)
7              cnt ⟵ 0
8              while O′ ≠ O and cnt ≤ 10 do
9                  if cnt = 10 then
10                     D ⟵ 1 ∘ 1 ∘ 1 ∘ 1 ⋯
11                 else
12                     D ⟵ Flip(R)
13                 ⟨O′, R, S⟩ ⟵ Exec(⟨p′, n′⟩, I, D)
14                 cnt ⟵ cnt + 1
15             if O ≠ O′ then
16                 skip to the next candidate ⟨p′, n′⟩
17             else
18                 R′ ⟵ R′ ∘ R
19                 S′ ⟵ S′ ∘ S
20         for ⟨I, O⟩ in PosT do
21             ⟨O′, R, S⟩ ⟵ Exec(⟨p′, n′⟩, I, ϵ)
22             R′ ⟵ R′ ∘ R
23             S′ ⟵ S′ ∘ S
24         C ⟵ {}
25         for σ in S′ do
26             C ⟵ C ∪ {(v == const), !(v == const) |
               ∀v∀const, such that σ(v) = const}
27         cnt ⟵ 0
28         while C ≠ ∅ and cnt < 20 do
29             let c ∈ C maximizes F(R′, S′, c)
30             C ⟵ C/{c}
31             if Test(⟨p′[c/abstc], n′⟩, NegT, PosT) then
32                 return ⟨p′[c/abstc], n′⟩
33             cnt ⟵ cnt + 1
34     else
35         if Test(⟨p′, n′⟩, NegT, PosT) then
36             return ⟨p′, n′⟩
37 return ∅
```

Figure 5: Repair generation algorithm with condition synthesis

goal of finding a sequence of values that generates the correct output for the test case (lines 5-19). Flip is an utility function that enables SPR to explore different abstract condition value sequences (see Figure 6). The algorithm (line 13) executes the program with different future abstract condition value sequences D to search for a sequence that passes each negative test case. If the algorithm cannot find such a sequence for any negative test case, it will move on to the next current candidate program (line 16).

$$\text{Exec}(\langle p, n\rangle, I, D) = \begin{cases} \langle O, R, S\rangle & \exists I\prime, O, R, S, \text{such that } \langle \ell_0, \sigma_0, I, \epsilon, D, \epsilon, \epsilon\rangle =[\![\langle p, n\rangle]\!]\Rightarrow \langle \texttt{nil}, \sigma, I\prime, O, D\prime, R, S\rangle \\ \bot & \text{otherwise} \end{cases}$$

$$\text{Test}(\langle p, n\rangle, NegT, PosT) = \begin{cases} \texttt{False} & \exists \langle I, O\rangle \in (NegT \cup PosT), \text{such that } \text{Exec}(\langle p, n\rangle, I, \epsilon) = \langle O\prime, R, S\rangle,\ O \neq O\prime \\ \texttt{True} & \textit{otherwise} \end{cases}$$

$$F(\epsilon, \epsilon, c) = 0 \qquad \frac{\sigma \vdash c \Rightarrow x}{F(x \circ R, \sigma \circ S, c) = F(R, S, c) + 1}$$

$$\frac{\sigma \vdash c \Rightarrow (1 - x)}{F(x \circ R, \sigma \circ S, c) = F(R, S, c)}$$

$$\text{Flip}(\epsilon) = \epsilon \qquad \frac{R = R\prime \circ 0}{\text{Flip}(R) = R\prime \circ 1} \qquad \frac{R = R\prime \circ 1}{\text{Flip}(R) = \text{Flip}(R\prime)}$$

Figure 6: Definitions of Exec**,** Test**,** Flip**, and** F

SPR tries a configurable number (in our current implementation, 11) of different abstract condition value sequences for each negative test case in the loop (lines 8-14). At each iteration (except the last) of the loop, the algorithm flips the last non-zero value in the previous abstract condition value sequence (see Flip definition in Figure 6). In the last iteration SPR flips all abstract condition values to 1 (line 10 in Figure 5) (note that the program may executes an abstract condition multiple times).

The rationale is that, in practice, if a negative test case exposes an error at an if statement, either the last few executions of the if statement or all of the executions take the wrong branch direction. This empirical property holds for all defects in our benchmark set.

If a future abstract condition value sequence can be found for every negative test case, the algorithm concatenates the found sequences $R\prime$ and the corresponding recorded environments to $S\prime$ (lines 18-19). The algorithm then executes the candidate program with the positive test cases and concatenates the sequences and the recorded environments as well (lines 22-23). Note that for positive cases the algorithm simply returns zero for all abstract conditions, so that the candidate program has the same execution as the original program.

Condition Generation: The algorithm enumerates all conditions in the search space and evaluates each condition against the recorded condition values (R$\prime$) and environments ($S\prime$). It counts the number of recorded condition values that the condition matches. Our current condition space is the set of all conditions of the form `(`v `==` *const*`)` or `!(`v `==` *const*`)` such that $\exists \sigma \in S\prime.\sigma(v) = const$. It is straightforward to extend this space to include comparison operators ($<, \leq, \geq, >$) and a larger set of logical expressions. For our benchmark set of defects, the relatively simple SPR condition space contains a remarkable number of correct repairs, with extensions to this space delivering relatively few additional correct repairs (see Section 4.5).

We define F(R$\prime$, S$\prime$, c) in Figure 6, which counts the number of branch directions for the condition c that match the recorded abstract condition values $R\prime$ given the recorded environments $S\prime$. The algorithm enumerates a configurable number (in our current implementation, 20) of the top conditions that maximize $F(R\prime, S\prime, c)$ (lines 27-33). The al-

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{\&\& !abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0}{\langle \ell, \sigma, I, O, D, R, S \rangle =\!\!\![\ \langle p, n \rangle\ ]\!\!\!\Rightarrow \langle \ell_2, \sigma, I, O, D, R, S \rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{||}\ \texttt{abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1}{\langle \ell, \sigma, I, O, D, R, S \rangle =\!\!\![\ \langle p, n \rangle\ ]\!\!\!\Rightarrow \langle \ell_1, \sigma, I, O, D, R, S \rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{\&\& !abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1}{\langle \ell, \sigma, I, O, \epsilon, R, S \rangle =\!\!\![\ \langle p, n \rangle\ ]\!\!\!\Rightarrow \langle \ell_1, \sigma, I, O, \epsilon, R \circ 0, S \circ \sigma \rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{||}\ \texttt{abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0}{\langle \ell, \sigma, I, O, \epsilon, R, S \rangle =\!\!\![\ \langle p, n \rangle\ ]\!\!\!\Rightarrow \langle \ell_2, \sigma, I, O, \epsilon, R \circ 0, S \circ \sigma \rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{\&\& !abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1 \qquad D = 0 \circ D'}{\langle \ell, \sigma, I, O, D, R, S \rangle =\!\!\![\ \langle p, n \rangle\ ]\!\!\!\Rightarrow \langle \ell_1, \sigma, I, O, D', R \circ 0, S \circ \sigma \rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{||}\ \texttt{abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0 \qquad D = 0 \circ D'}{\langle \ell, \sigma, I, O, D, R, S \rangle =\!\!\![\ \langle p, n \rangle\ ]\!\!\!\Rightarrow \langle \ell_2, \sigma, I, O, D', R \circ 0, S \circ \sigma \rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{\&\& !abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1 \qquad D = 1 \circ D'}{\langle \ell, \sigma, I, O, D, R, S \rangle =\!\!\![\ \langle p, n \rangle\ ]\!\!\!\Rightarrow \langle \ell_2, \sigma, I, O, D', R \circ 1, S \circ \sigma \rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{||}\ \texttt{abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0 \qquad D = 1 \circ D'}{\langle \ell, \sigma, I, O, D, R, S \rangle =\!\!\![\ \langle p, n \rangle\ ]\!\!\!\Rightarrow \langle \ell_1, \sigma, I, O, D', R \circ 1, S \circ \sigma \rangle}$$

Figure 7: Small step operational semantics for if statements with abstract condition

gorithm then validates the transformed candidate program with the abstract condition replaced by the generated condition c (lines 31-32). $p[c/\texttt{abstc}]$ denotes the result of replacing every occurrence of `abstc` in p with the condition c.

Enumerating all conditions in the space is feasible because the overwhelming majority of the candidate transformed programs will not pass the condition value search stage. SPR will therefore perform the condition generation stage very infrequently and only when there is some evidence that transforming the target condition may actually deliver a correct repair. In our experiments, SPR performs the condition generation stage for less than 1% of the candidate transformed programs that contain an abstract condition. (see Section 4.4).

Alternate Condition Synthesis Techniques: It is straightforward to implement a variety of different condition synthesis techniques. For example, it is possible to synthesize complete replacements for conditions of if statements (instead of conjoining or disjoining new conditions to existing conditions). The condition value search would start with the sequence of branch directions at that if statement with the original condition, then search for a sequence of branch directions that would generate correct outputs for all negative inputs. Condition generation would then work with the recorded branch directions to deliver a new replacement condition.

The effectiveness of condition value search in eliminating unpromising conditions enables the SPR condition generation algorithm to simply enumerate and test all conditions in the condition search space. It is of course possible to apply arbitrarily sophisticated condition generation algorithms, for example by leveraging modern solver technology [29]. One issue is that there may be no condition that exactly matches the recorded sequences of environments and branch directions. Even if this occurs infrequently (as we would expect in practice), requiring an exact match may eliminate otherwise correct repairs. An appropriate solver may therefore need to generate approximate solutions.

3.4 Extensions for C

We have implemented SPR in C++ using the clang compiler front-end [1]. Clang contains a set of APIs for manipulating the AST tree of a parsed C program, which enables SPR to generate a repaired source code file without dramatically changing the overall structure of the source code. Existing program repair tools [24, 32, 39] often destroy the structure of the original source by inlining all header files and renaming all local variables in their generated repaired source code. Preserving the existing source code structure helps developers understand and evaluate the repair and promotes the future maintainability of the application.

C Program Support: SPR extends the algorithm in Section 3.3 to support C programs. SPR applies the transformation function separately to each function in a C program. When SPR performs variable replacement or condition synthesis, it considers all variables (including local, global, and heap variables) that appear in the current transformed function. During condition generation, SPR also searches existing conditions c that occur in the same enclosing compound statement (in addition to conditions of the form (v `==` *const*) and `!`(v `==` *const*) described above in Section 3.3).

When SPR inserts control statements, SPR considers `break`, `return`, and `goto` statements. When inserting `return` statements, SPR generates a repair to return each constant value in the returned type that appeared in the enclosing function. When inserting `goto` statements, SPR generates a repair to jump to each already defined label in the enclosing function. When SPR inserts initialization statements, SPR considers to call `memset()` to initialize memory blocks. When SPR copies statements for C programs, SPR considers to copy compound statements in addition to simple statements, as long as the copied code can fit into the new context.

To represent a abstract condition, SPR inserts a function call `abstract_code()` into the modified condition. When SPR tests a candidate transformed program with abstract conditions, SPR links the program with its runtime library, which contains an implementation of `abstract_cond()`. The `abstract_cond()` in the library implements the semantics specified in Figure 7.

Error Localizer: The SPR error localizer first recompiles the given application with additional instrumentation. It inserts a call back before each statement in the source code to record a positive counter value as the timestamp of the statement execution. SPR then invokes the recompiled application on all the positive and negative test cases.

For a statement s and a test case i, $r(s,i)$ is the recorded execution timestamp that corresponds to the last timestamp from an execution of the statement s when the application runs with the test case i. If the statement s is not executed at all when the application runs with the test case i, then $r(s,i) = 0$.

We use the notation NegT for the set of negative test cases and PosT for the set of positive test cases. SPR computes

three scores $a(s)$, $b(s)$, $c(s)$ for each statement s:

$$a(s) = |\{i \mid r(s,i) \neq 0,\ i \in \text{NegT}\}|$$
$$b(s) = |\{i \mid r(s,i) = 0,\ i \in \text{PosT}\}|$$
$$c(s) = \Sigma_{i \in \text{NegT}} r(s,i)$$

A statement s_1 has higher priority than a statement s_2 if $\text{prior}(s_1, s_2) = 1$, where prior is defined as:

$$prior(s_1, s_2) = \begin{cases} 1 & a(s_1) > a(s_2) \\ 1 & a(s_1) = a(s_2), b(s_1) > b(s_2) \\ 1 & a(s_1) = a(s_2), b(s_1) = b(s_2), \\ & c(s_1) > c(s_2) \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, SPR prioritizes statements that 1) are executed with more negative test cases, 2) are executed with fewer positive test cases, and 3) are executed later during executions with negative test cases.

SPR runs the above error localization algorithm over the whole application including all of its C source files. If the user specifies a source file to repair, SPR computes the intersection of the top 5000 statements in the whole application with the statements in the specified source code file. PCR uses the statements in this intersection as the identified statements to repair. Unlike GenProg [24] and AE [39], SPR can also operate completely automatically. If the user does not specify any such source file, SPR identifies the top 200 statements in the whole application (potentially from different source files) as the potential modification targets.

The SPR error localizer also ranks if statements based on the statements inside their then and else clauses. Specifically, it gives an if statement the rank of the highest ranking statement in its then or else clauses. The rationale is that changing the condition of these enclosing if statement will control the execution of the identified statement and therefore may repair the program.

Repair Test Order: SPR tests each of the generated candidate repairs one by one (line 1 in Figure 5). SPR empirically sets the test order as follows:

1. SPR first tests repairs that change only a branch condition (e.g., tighten and loosen a condition).
2. SPR tests repairs that insert an if-statement before a statement s, where s is the first statement of a compound statement (i.e., C code block).
3. SPR tests repairs that insert an if-guard around a statement s.
4. SPR tests repairs that insert a memory initialization.
5. SPR tests repairs that insert an if-statement before a statement s, where s is not the first statement of a compound statement.
6. SPR tests repairs a) that replace a statement or b) that insert a non-if statement (i.e., generated by $\text{M}_{\text{CopyRep}}$) before a statement s where s is the first statement of a compound statement.
7. SPR finally tests the remaining repairs.

Intuitively, SPR prioritizes repairs that contain conditionals. With abstract conditions that SPR later synthesizes, each condition value search stands in for multiple potential repairs. SPR also prioritizes repairs that insert a statement before the first statement of a compound statement (i.e., a code block), because inserting statements at other locations is often semantically equivalent to such repairs.

If two repairs have the same tier in the previous list, their test orders are determined by the rank of the two corresponding original statements (which two repairs are based on) in the list returned by the error localizer.

Batched Compilation: When SPR tests candidate repairs, compilations of the repaired application may become the performance bottleneck for SPR. To reduce the time cost of compilations, SPR merges similar candidate repairs into a single combined repair with a branch statement. A global integer environment variable controls the branch statement, so that the batched repair will be equivalent to each individual candidate repair, when the environment variable takes a corresponding constant value. SPR therefore only needs to recompile the merged repair once to test each of the individual candidate repairs.

Test Case Evaluation Order: SPR always first tests each candidate repair with the negative test cases. Empirically, negative test cases tend to eliminate invalid repairs more effectively than positive test cases. Furthermore, whenever a positive test case eliminates a candidate repair, SPR will record this positive test case and prioritize this test case for the future candidate repair evaluation.

Repairs for Code Duplicates: Programs often contain duplicate or similar source code, often caused, for example, by the use of macros or code copy/paste during application development. SPR detects such duplicates in the source code. When SPR generates repairs that modify one of the duplicates, it also generates additional repairs that propagate the modification to the other duplicates.

4. EXPERIMENTAL RESULTS

We evaluate SPR on a benchmark set containing 69 defects and 36 functionality changes drawn from seven large open source applications, libtiff [4] (a TIFF image processing library and toolkit), lighttpd [3] (a popular open source HTTP server), the PHP interpreter [6] (an open source interpreter for PHP scripts), gmp (a multiple percision arithmetic library), gzip (a popular compression toolkit), python (the standard Python languague implementation), wireshark (a popular network package analyzer), and fbc (an open source Basic compiler) [2, 24]. We address the following questions:

1. **Plausible Repairs:** How many plausible repairs can SPR generate for this benchmark set?
2. **Correct Repairs:** How many correct repairs can SPR generate for this benchmark set?
3. **Design Decisions:** How do the various SPR design decisions affect the ability of SPR to generate plausible and correct repairs?
4. **Previous Systems:** How does SPR compare with previous systems on this benchmark set?

4.1 Methodology

Reproduce the Defects/Changes: For each of the seven benchmark applications, we collected the defects/changes, test harnesses, test scripts, and test cases used in a previous study [2]. We modified the test scripts and test harnesses to eliminate various errors [33]. For

App	LoC	Tests	Defects/ Changes	Plausible				Correct				Init Time	SPR Search Time	SPR(NoF) Search Time
				SPR	SPR NoF	Gen Prog	AE	SPR	SPR NoF	Gen Prog	AE			
libtiff	77k	78	8/16	5/0	5/0	3/0	5/0	1/0	1/0	0/0	0/0	2.4m	54.0m	71.4m
lighttpd	62k	295	7/2	4/1	2/1	4/1	3/1	0/0	0/0	0/0	0/0	7.2m	144.1m	182.3m
php	1046k	8471	31/13	17/1	13/1	5/0	7/0	8/0	7/0	1/0	2/0	13.7m	156.2m	186.2m
gmp	145k	146	2/0	2/0	2/0	1/0	1/0	1/0	1/0	0/0	0/0	7.5m	128m	374.5m
gzip	491k	12	4/1	2/0	2/0	1/0	2/0	1/0	1/0	0/0	0/0	4.2m	33.5m	29.5m
python	407k	35	9/2	2/1	2/1	0/1	2/1	0/1	0/1	0/1	0/1	31.1m	237.7m	163.0m
wireshark	2814k	63	6/1	4/0	4/0	1/0	4/0	0/0	0/0	0/0	0/0	58.8m	32.2m	40.3m
fbc	97k	773	2/1	1/0	1/0	1/0	1/0	0/0	0/0	0/0	0/0	8m	15m	53m
Total			69/36	37/3	31/3	16/2	25/2	11/1	10/1	1/1	2/1			

Table 1: Overview of SPR Repair Generation Results

libtiff we implemented only partially automated repair validation, manually filtering the final generated repairs to report only plausible repairs [33]. We then reproduced each defect/change (except the fbc defects/changes) in our experimental environment, Amazon EC2 Intel Xeon 2.6GHz Machines running Ubuntu-64bit server 14.04. fbc runs only in 32-bit environments, so we use a virtual machine with Intel Core 2.7Ghz running Ubuntu-32bit 14.04 for the fbc experiments.

Apply SPR: For each defect/change, we ran SPR with a time limit of 12 hours. We terminate SPR when either 1) SPR successfully finds a repair that passes all of the test cases or 2) the time limit of 12 hours expires with no generated SPR repair. To facilitate the comparison of SPR with previous systems, we run SPR twice for each defect: once without specifying a source code file to repair, then again specifying the same source code file to repair as previous systems [2, 24, 39]. [3]

Inspect Repair Correctness: For each defect/change, we manually inspect all of the repairs that SPR generates. We consider a generated repair *correct* if 1) the repair completely eliminates the defect exposed by the negative test cases so that no input will be able to trigger the defect, and 2) the repair does not introduce any new defects.

We also analyze the developer patch (when available) for each of the 40 defects/changes for which SPR generated plausible repairs. Our analysis indicates that the developer patches are, in general, consistent with our correctness analysis: 1) if our analysis indicates that the SPR repair is correct, then the repair has the same semantics as the developer patch and 2) if our analysis indicates that the SPR repair is not correct, then the repair has different semantics from the developer patch.

We acknowledge that, in general, determining whether a specific repair corrects a specific defect can be difficult (or in some cases not even well defined). We emphasize that this is not the case for the repairs and defects that we consider in this paper. The correct behavior for all of the defects is clear, as is repair correctness and incorrectness.

4.2 Summary of Experimental Results

Table 1 summarizes our benchmark set and our experimental results. All of the generated SPR repairs are available [25]. Column 1 (App) presents the name of the benchmark application. Column 2 (LoC) presents the size of the benchmark application measured in the number of source code lines. Column 3 (Tests) presents the number of test cases. Column 4 (Defects/Changes) presents the number of defects/changes we considered in our experiments. Each entry is of the form X/Y, where X is the number of defects and Y is the number of changes.

Each entry in Column 5 (Plausible SPR) is of the form X/Y, where X is the number of defects and Y is the number of changes for which SPR generates a plausible repair. Column 6 (Plausible SPR NoF) presents the corresponding numbers for SPR running without a specified source code file to repair. For comparison, Columns 7-8 present the corresponding results for GenProg [24] and AE [39].[4] Columns 9-12 present the corresponding results for correct repairs.

SPR generates plausible repairs for at least 12 more defects than GenProg and AE (37 for SPR vs. 16 for GenProg and 25 for AE). Even with no specified source code file to repair (note that GenProg and AE require the user to provide this information), SPR is able to generate plausible repairs for 31 defects. The GenProg result tar file [2] reports results from 10 different GenProg executions with different random number seeds. For some defects some of the patches are correct while others are not. We count the defect as patched correctly by GenProg if any of the patches for that defect are correct. Our results show that SPR generates correct repairs for at least 9 more defects than GenProg and AE (11 for SPR vs. 1 for GenProg and 2 for AE). Even when the target source file is not specified, SPR still generates correct repairs for 10 defects.

Column 13 (Init Time) in Table 1 presents the average time SPR spent to initialize the repair process, which includes compiling the application and running the error localizer. Column 14 (SPR Search Time) presents the average execution time of SPR on all defects/changes for which SPR generates repairs. Column 15 (SPR NoF Search Time) presents the average execution time for the runs where we do not specify a source code file to repair. Our experimental results show that for those defects for which SPR generates a repair, SPR will generate the first repair in less than 2 hours on average.

4.3 SPR Repair Generation Results

[3]Previous systems require the user of the system to identify a source code file to patch [2, 24, 39]. This requirement reduces the size of the search space but eliminates the ability of these systems to operate automatically without user input. SPR imposes no such restriction — it can operate fully automatically across the entire source code base. If desired, it can also work with a specified source code file to repair.

[4]Note that due to errors in the repair evaluation scripts, at least half of the originally reported patches from the GenProg and AE papers do not produce correct results for the inputs in the test suite used to validate the patches [33]. See our previous work on the analysis of GenProg and AE patches for details [33].

Defect/ Change	SPR				SPR(No File Name Info)				Gen Prog	AE	SPR Time	SPR (NoF) Time
	Search Space	Gen At	Correct At	Result	Search Space	Gen At	Correct At	Result				
libtiff-ee2ce-b5691	139321	5630	5630	Correct	49423	2167	2167	Correct	No	Gen	162m	134m
libtiff-d13be-ccadf	39207	94	94	Gen	14717	68	68	Gen	Gen	Gen	46m	52m
libtiff-90d13-4c666	142938	5764	-	Gen	50379	2070	-	Gen	No	Gen	273m	133m
libtiff-5b021-3dfb3	85812	21	7526	Gen	62981	35	6894	Gen	Gen	Gen	6m	8m
libtiff-08603-1ba75	38812	46	-	Gen	16415	80	-	Gen	Gen	Gen	30m	30m
lighttpd-1794-1795	62517	37	-	Gen				No	Gen	Gen	84m	>12h
lighttpd-1806-1807	5902	5	-	Gen				No	Gen	Gen	281m	>12h
lighttpd-1913-1914	37783	69	-	Gen	20363	5	-	Gen	Gen	No	248m	298m
lighttpd-2661-2662	21539	47	81	Gen	14218	545	-	Gen	Gen	Gen	55m	190m
php-307562-307561	10927	968	968	Correct	40419	1614	1614	Correct	No	No	32m	412m
php-307846-307853	51579	4842	4842	Correct	261609	1786	1786	Correct	No	No	132m	429m
php-307931-307934	25045	3	-	Gen				No	Gen	Gen	164m	>12h
php-308262-308315	94127	36	12854	Gen				No	No	No	152m	>12h
php-308525-308529	7067	160	-	Gen				No	No	Gen	294m	>12h
php-308734-308761	2886	252	252	Correct	8137	1920	1920	Correct	No	No	180m	385m
php-309111-309159	15867	66	5747	Gen	27262	101	8638	Gen	No	Correct	50m	83m
php-309516-309535	50522	3975	3975	Correct				No	No	No	113m	>12h
php-309579-309580	8451	11	11	Correct	218782	45	45	Correct	No	No	20m	81m
php-309688-309716	6962	462	534	Gen	40750	563	4383	Gen	No	No	18m	113m
php-309892-309910	12662	4	4	Correct	166491	56	56	Correct	Correct	Correct	37m	77m
php-309986-310009	10977	30	-	Gen	54392	37	-	Gen	Gen	Gen	85m	369m
php-310011-310050	2140	153	1252	Gen	39634	20	10773	Gen	Gen	Gen	429m	267m
php-310370-310389	3865	33	-	Gen	45305	29	-	Gen	No	No	91m	100m
php-310673-310681	16079	1215	-	Gen	13859	312	-	Gen	Gen	Gen	132m	56m
php-310991-310999	294623	127	127	Correct	133670	50	50	Correct	No	No	149m	107m
php-311346-311348	33620	22	22	Correct	18121	3	3	Correct	No	No	68m	45m
gmp-13420-13421	14744	2242	2242	Correct	19652	3088	3088	Correct	No	No	236m	363m
gmp-14166-14167	4581	9	-	Gen	10217	13	-	Gen	Gen	Gen	20m	23m
gzip-a1d3d4-f17cbd	46113	942	942	Correct	18106	260	260	Correct	No	Gen	38m	31m
gzip-3fe0ca-39a362	20522	60	-	Gen	21353	3	-	Gen	Gen	Gen	29m	28m
python-69223-69224	11955	794	-	Gen	24113	77	-	Gen	No	Gen	564m	238m
python-70098-70101	27674	46	-	Gen	13238	899	-	Gen	No	Gen	90m	172m
wshark-37112-37111	8820	80	-	Gen	17704	41	-	Gen	Gen	Gen	48m	36m
wshark-37172-37171	47406	223	-	Gen	25531	153	-	Gen	No	Gen	28m	38m
wshark-37172-37173	47406	175	-	Gen	25531	270	-	Gen	No	Gen	24m	40m
wshark-37284-37285	53196	313	-	Gen	27723	345	-	Gen	No	Gen	29m	47m
fbc-5458-5459	506	4	4	Gen	4611	6	54	Gen	Gen	Gen	15m	53m
lighttpd-2330-2331	24919	6	-	Gen	21910	106	-	Gen	Gen	Gen	51m	59m
php-311323-311300	32009	122	-	Gen	211717	93	-	Gen	No	No	666m	83m
python-69783-69784	12691	67	67	Correct	24331	58	58	Correct	Correct	Correct	59m	79m

Table 2: SPR Results for Each Generated Plausible Repair

SPR generates plausible repairs for 40 defects/changes in our benchmark set. Table 2 presents detailed information for each defect/change for which SPR generates a plausible repair.[5] The first 37 rows present results for defects, the last 3 rows present results for functionality changes. Column 1 (Defect/Change) is in the form of X-Y-Z, where X is the name of the application that contains the defect/change, Y is the defective version, and Z is the reference repaired version. Columns 2-5 in Table 2 present results from the SPR runs where we specified the target source file to repair (as in the GenProg and AE runs). Columns 6-9 present results from the SPR runs where we do not specify a source code file to repair.

Search Space: Columns 2 and 6 (Search Space) in Table 2 present the total number of candidate repairs in the SPR search space. A number X in Columns 3 and 7 (Gen At) indicates that the first generated plausible patch is the Xth candidate patch in SPR's search space.

Columns 4 and 8 (Correct At) present the rank of the first correct repair in the SPR search space (if any). A "-" indicates that there is no correct repair in the search space.

[5]This set is a superset of the set of defects/changes for which GenProg/AE generate plausible patches.

Note that even if the correct repair is within the SPR search space, SPR may not generate this correct repair — the SPR search may time out before SPR encounters the correct repair, or SPR may encounter a plausible but incorrect repair before it encounters the correct repair.

Comparison With GenProg and AE: Columns 5 and 9 (Result) present for each defect whether the SPR repair is correct or not. Columns 10 and 11 present the status of the GenProg and AE patches for each defect. "Correct" in the columns indicates that the tool generated a correct patch. "Gen" indicates that the tool generated a plausible but incorrect patch. "No" indicates that the tool does not generate a plausible patch for the corresponding defect.

Our experimental results show that whenever GenProg or AE generates a plausible patch for a given defect, so does SPR. For one defect, AE generates a correct patch when SPR generates a plausible but incorrect repair. For this defect, the SPR search space contains a correct repair, but the SPR search algorithm encounters the plausible but incorrect repair before it encounters the correct repair. For the remaining defects for which GenProg or AE generate a correct repair, so does SPR. SPR generates plausible repairs for 21 more defects than GenProg and 12 more defects than

Defect/ Change	Repair Type	Condition Value Search	
		On	Off
php-307562-307561	Replace†	0/58	5.3X
php-307846-307853	Add Init†	0/198	2.5X
php-308734-308761	Guarded Control†‡	8/44	5.7X
php-309516-309535	Add Init†	0/207	3.1X
php-309579-309580	Change Condition†‡	1/24	15.6X
php-309892-309910	Delete	1/63	50X
php-310991-310999	Change Condition†	4/486	324.4X*
php-311346-311348	Redirect Branch†	2/283	141.2X*
libtiff-ee2ce5-b5691a	Add Control†‡	3/1099	8.9X*
gmp-13420-13421	Replace†‡	0/339	5.1X*
gzip-a1d3d4-f17cbd	Copy and Replace†‡	0/269	11.7X
python-69783-69784	Delete	2/169	35.9X
php-308262-308315	Add Guard†‡	N/A	1.2X
php-309111-309159	Copy‡	N/A	5.2X
php-309688-309716	Change Condition†‡	N/A	7.2X
php-310011-310050	Copy and Replace†‡	N/A	3.2X
libtiff-d13be-ccadf	Change Condition†	N/A	211.1X
libtiff-5b021-3dfb3	Replace†	N/A	3.1X
lighttpd-2661-2662	Guarded Control†‡	N/A	48.1X
fbc-5458-5459	Change Condition†‡	N/A	22.4X

Table 3: SPR Repair Type and Condition Synthesis Results

AE. SPR generates correct repairs for 10 more defects than GenProg and 9 more defects than AE.

4.4 Correct Repair Analysis

The SPR repair search space contains correct repairs for 20 defects/changes. Table 3 classifies these 20 correct repairs. The first 12 of these 20 are the first plausible repair that SPR encounters during the search. The classification highlights the challenges that SPR must overcome to generate these correct repairs. Column 1 (Defect/Change) presents the defect/change.

Modification Operators: Column 2 (Repair Type) presents the repair type of the correct repair for each defect. "Add Control" indicates that the repair inserts a control statement with no condition. "Guarded Control" indicates that the repair inserts a guarded control statement with a meaningful condition. "Replace" indicates that the repair modifies an existing statement using value replacement to replace an atom inside it. "Copy and Replace" indicates that the repair copies a statement from somewhere else in the application using value replacement to replace an atom in the statement. "Add Init" indicates that the repair inserts a memory initialization statement. "Delete" indicates that the repair simply removes statements (this is a special case of the Conditional Guard modification in which the guard condition is set to false). "Redirect Branch" indicates that the repair removes one branch of an if statement and redirects all executions to the other branch (by setting the condition of the if statement to true or false). "Change Condition" indicates that the repair changes a branch condition in a non-trivial way (unlike "Delete" and "Redirect Branch"). "Add Guard" indicates that the repair conditionally executes an existing statement by adding an if statement to enclose the existing statement.

A "†" in Column 2 indicates that the SPR repair for this defect is outside the search space of GenProg and AE (for 17 out of the 20 defects/changes, the SPR repair is outside the GenProg and AE search space). A "‡" in the column indicates that the SPR repair for this defect is outside the search space of PAR with the eight templates from the PAR paper [22]. For 11 of the 20 defects/changes, the SPR repair is outside the PAR search space.

For php-307846-307853, php-308734-308761, php-309516-309535, libtiff-ee2ce5b7-b5691a5a, and lighttpd-2661-2662, the correct repairs insert control statements or initialization statements that do not appear elsewhere in the source file. For php-307562-307561, gmp-13420-13421, gzip-a1d3d4-f17cbd, php-310011-310050, and libtiff-5b021-3dfb3 the SPR repairs change expressions inside the copied or replaced statements. For php-309579-309580, php-310991-310999, php-311346-311348, php-308262-308315, php-309688-309716, libtiff-d13be-ccadf, and fbc-5458-5459 the SPR generated repairs change the branch condition in a way which is not equivalent to deleting the whole statement. These repairs are therefore outside the search space of GenProg and AE, which only copy and remove statements.

The SPR correct repairs for php-308734-308761 (add "break"), libtiff-ee2ce5b7-b5691a5a (add "goto"), gzip-a1d3d4-f17cbd (add "assignment"), php-309111-309159 (copy a statement block), and php-310011-310050 (add a function call) are outside the PAR search space because no template in PAR is able to add goto, break, assignment, function call, or compound statements into the source code. The SPR correct repair for gmp-13420-13421 is also outside the PAR search space, because the repair replaces a subexpression inside an assignment statement and no template in PAR supports such an operation. The SPR correct repair for php-309579-309580 is outside the PAR search space because the repair changes a branch condition (See Section 2), but the inserted clause "isostr" does not appear elsewhere in branch conditions in the application. The PAR template "Expression Adder" collects the added condition clause only from other conditions in the program. The correct repairs for php-308262-308315, php-309688-309716, lighttpd-2661-2662, and fbc-5458-5459 are outside the PAR search space because they require a condition that does not already exist in the program.

Condition Synthesis: Each entry in Column 3 (Condition Value Search On) is of the form X/Y. Here Y is the total number of repair schema applications that contain an abstract target condition. X is the number of these schema applications for which SPR discovers a sequence of abstract condition values that generate correct inputs for all outputs. SPR performs the condition generation search for only these X schema applications. These results highlight the effectiveness of SPR's staged condition synthesis algorithm — over 99.4% of the schema applications are discarded before SPR even attempts to find a condition that will repair the program. We note that, for all defects except php-310991-310999, SPR's condition generation algorithm is able to find an exact match for the recorded abstract condition values. For php-310991-310999, the correct generated condition matches all except one of the recorded abstract condition values. We attribute the discrepancy to the ability of the program to generate a correct result for both branch directions [36].

Column 4 (Condition Value Search Off) presents how many times more candidate repairs SPR would need to consider if SPR turned off condition value search and performed condition synthesis by simply enumerating and testing all conditions in the search space. These results show that SPR's staged condition synthesis algorithm significantly reduces the number of candidate repairs that SPR needs to

validate, in some cases by a factor of over two orders of magnitude. Without staged condition synthesis, SPR would be unable to generate repairs for 4 defects within 12 hours.

4.5 Search Space Extensions

The current SPR repair space contains repairs for 19 of the 69 defects. Extending the SPR condition space to include comparison operations ($<, \leq, \geq, >$) would bring repairs for an additional two defects into the repair space (lighttpd-1913-1914 and python-70056-70059). Extending the repair space to include repairs that apply two transformation schemas (instead of only one as in the current SPR implementation) would bring repairs for another two defects into the space (php-308525-308529 and gzip-3fe0ca-39a362). Extending the Copy and Replace schema instantiation space to include more sophisticated replacement expressions would bring repairs for six more defects into the search space (php-307914-307915, php-311164-311141, libtiff-806c4c9-366216b, gmp-14166-14167, python-69934-69935, and fbc-5556-5557). Combining all three of these extensions would bring an additional six more defects into the search space (php-307687-307688, php-308523-308525, php-309453-309456, php-310108-310109, lighttpd-1948-1949, and gzip-3eb609-884ef6). Repairs for the remaining 34 defects require changes to or insertions of at least three statements.

All of these extensions come with potential costs. The most obvious cost is the difficulty of searching a larger repair space. A more subtle cost is that increasing the search space may increase the number of plausible but incorrect repairs and make it harder to find the correct repair. It is straightforward to extend SPR to include comparison operators. The feasibility of supporting the other extensions is less clear.

5. LIMITATIONS

The data set considered in this paper was selected not by us, but by the GenProg developers in an attempt to obtain a large, unbiased, and realistic benchmark set [24]. The authors represent the study based on this data set as a "Systematic Study of Automated Program Repair" and identify one of the three main contributions of the paper as a "systematic evaluation" that "includes two orders of magnitude more" source code, test cases, and defects than previous studies [24]. Moreover, the benchmark set was specifically constructed to "help address generalizability concerns" [24]. Nevertheless, one potential threat to validity is that our results may not generalize to other applications, defects, and test suites.

SPR only applies one transformation at each time it generates a candidate repair. Repairs that modify two or more statements are not in SPR's search space. It is unclear how to combine multiple transformations and still efficiently explore the enlarged search space. We note that previous tools [24, 32] that apply multiple mutations produce only semantically simple patches. The overwhelming majority of the patches are incorrect and equivalent to simply deleting functionality [33].

6. RELATED WORK

ClearView: ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [31]. It deploys monitors that detect crashes, illegal control transfers and out of bounds write defects. In response, it selects a nearby invariant that the input that triggered the defect violates, and generates patches that take a repair action when the invariant is violated. Subsequent executions enable ClearView to determine if 1) the patch eliminates the defect while 2) preserving desired benign behavior. ClearView generates patches that can be applied directly to a running program without requiring a restart.

ClearView was evaluated by a hostile Red Team attempting to exploit security vulnerabilities in Firefox [31]. The Red Team developed attacks that targeted 10 Firefox vulnerabilities and evaluated the ability of ClearView to automatically generate patches that eliminated the vulnerability. For 9 of these 10 vulnerabilities, ClearView is able to generate patches that eliminate the vulnerability and enable Firefox to continue successful execution [31].

SPR differs from ClearView in both its goal and its technique. SPR targets software defects that can be exposed by supplied negative test cases, which are not limited to just security vulnerabilities. SPR operates on a search space derived from its modification operators to generate candidate patches, while ClearView generates patches to enforce violated invariants.

GenProg, RSRepair, and AE: GenProg [40, 24] uses a genetic programming algorithm to search a space of patches, with the goal of enabling the application to pass all considered test cases. RSRepair [32] changes the GenProg algorithm to use random modification instead. AE [39] uses a deterministic patch search algorithm and uses program equivalence relations to prune equivalent patches during testing.

Previous work shows that, contrary to the design principle of GenProg, RSRepair, and AE, the majority of the reported patches of these three systems are implausible due to errors in the patch validation [33]. Further semantic analysis on the remaining plausible patches reveals that despite the surface complexity of these patches, an overwhelming majority of these patches are equivalent to functionality elimination [33]. The Kali patch generation system, which only eliminates functionality, can do as well [33].

Unlike GenProg [24], RSRepair [32], and AE [39], which only copy statements from elsewhere in the program, SPR defines a set of novel modification operators that enables SPR to operate on a search space which contains meaningful and useful repairs. SPR then uses its condition synthesis technique to efficiently explore the search space. Our results show that SPR significantly outperforms GenProg and AE in the same benchmark set. The majority of the correct repairs SPR generates in our experiments are outside the search space of GenProg, RSRepair, and AE.

PAR: PAR [22] is another prominent automatic patch generation system. PAR is based on a set of predefined human-provided patch templates. We are unable to directly compare PAR with SPR because, despite repeated requests to the authors of the PAR paper over the course of 11 months, the authors never provided us with the patches that PAR was reported to have generated [22]. Monperrus found that PAR fixes the majority of its benchmark defects with only two templates ("Null Pointer Checker" and "Condition Expression Adder/Remover/Replacer") [28].

In general, PAR avoids the search space explosion problem because its human supplied templates restrict its search

space. However, the PAR search space (with the eight templates in the PAR paper [22]) is in fact a subset of the SPR search space. Moreover, the difference is meaningful — the SPR correct patches for at least 11 of our benchmark defects are outside the PAR search space (see Section 4.4). This result illustrates the fragility and unpredictability of using fixed patch templates.

SemFix and MintHint: SemFix [29] and MintHint [19] replace the potential faulty expression with a symbolic value and use symbolic execution techniques [10] and SMT solvers to find a replacement expression that enables the program to pass all test cases. SemFix and MintHint are evaluated only on applications with less than 10000 lines of code. In addition, these techniques cannot generate fixes for statements with side effects.

Debroy and Wong: Debroy and Wong [11] present a mutation-based patch generation technique. This technique either replaces an existing arithmetic operator with another operator or negates the condition of an if or while statement. In contrast, SPR uses more sophisticated and effective modification operators and search algorithms. In fact, none of the correct repairs in SPR' search space for the 19 defects are within the Debroy and Wong search space.

NOPOL: NOPOL [12] is an automatic repair tool focusing on branch conditions. It identifies branch statement directions that can pass negative test cases and then uses SMT solvers to generate repairs for the branch condition. A key difference between SPR and NOPOL is that SPR introduces abstract condition semantics and uses target condition value search to determine the value sequence of an abstract condition, while NOPOL simply assumes that the modified branch statement will always take the same direction during an execution. In fact, this assumption is often not true when the branch condition is executed multiple times for a test case (e.g., php-308734-308761 and php-310991-310999), and NOPOL will fail to generate a correct patch.

SPR also differs from NOPOL in the two following ways. 1) NOPOL focuses only on patches that change conditions, while SPR can generate repairs for a broader class of defects (php-307562-307561, php-307846-307853, php-309516-309535, libtiff-ee2ce-b5691, gmp-13420-13421, and gzip-a1d3d-f17cb). 2) NOPOL was evaluated on two small Java programs (each with less than 5000 lines of code) and two artificial examples in [12], while we evaluate SPR on 105 real world defects in seven C applications with more than one million lines in total.

Fix Safety-Policy Violation: Weimer [38] proposes a patch generation technique for safety policy violation errors. This technique takes a DFA-like specification that describes the safety policy. For an execution trace that violates the policy, it finds a nearest accepting trace from the offending execution trace for the DFA specification. It then generates a patch that forces the program to produce the identified accepting trace instead of the trace that violates the policy. The goal is not to obtain a correct patch — the goal is instead to produce a patch that helps give a human developer insight into the nature of the defect.

In contrast, SPR does not require human-supplied specifications and can work with any defect (not just safety policy violations) that can be exposed by negative test cases. Unlike SPR, Weimer's technique does not attempt to repair branch conditions and simply uses path constraints as branch conditions to guard its modifications to minimize the patch impact on normal traces.

Domain Specific Repair Generation: Other program repair systems include VEJOVIS [30] and Gopinath et al. [18], which applies domain specific techniques to repair DOM-related faults in JavaScript and selection statements in database programs respectively. AutoFix-E [37] repairs program faults with human-supplied specifications called contracts. SPR differs from all of this previous research in that it focuses on generating fixes for general purpose applications without human-supplied specifications.

6.1 Targeted Repair Systems

Researchers have developed a variety of repair systems that are targeted at specific classes of errors.

Failure-Oblivous Computing: Failure-oblivious computing [34] checks for out of bounds reads and writes. It discards out of bounds writes and manufactures values for out of bounds reads. This eliminates data corruption from out of bounds writes, eliminates crashes from out of bounds accesses, and enables the program to continue execution along its normal execution path.

Failure-oblivious computing was evaluated on five errors in five server applications. The goal was to enable servers to survive inputs that trigger the errors and continue on to successfully process other inputs. For all five systems, the implemented system realized this goal. For two of the five errors, failure-oblivious computing completely eliminates the error and, on all inputs, delivers the same output as the official developer patch that corrects the error (we believe these patches are correct).

Bolt: Bolt [23] attaches to a running application, determines if the application is in an infinite loop, and, if so, exits the loop. A user can also use Bolt to exit a long-running loop. In both cases the goal is to enable the application to continue useful execution. Bolt was evaluated on 13 infinite and 2 long-running loops in 12 applications. For 14 of the 15 loops Bolt delivered a result that was the same or better than terminating the application. For 7 of the 15 loops, Bolt completely eliminates the error and, on all inputs, delivers the same output as the official developer patch that corrects the error (we believe these patches are correct).

RCV: RCV [26] enables applications to survive null dereference and divide by zero errors. It discards writes via null references, returns zero for reads via null references, and returns zero as the result of divides by zero. Execution continues along the normal execution path.

RCV was evaluated on 18 errors in 7 applications. For 17 of these 18 errors, RCV enables the application to survive the error and continue on successfully process the remaining input. For 11 of the 18 errors, RCV completely eliminates the error and, on all inputs, delivers either identical (9 of 11 errors) or equivalent (2 of 11 errors) outputs as the official developer patch that corrects the error (we believe these patches are correct).

APPEND: APPEND [15] proposes to eliminate null pointer exceptions in Java by applying recovery techniques such as replacing the null pointer with a pointer to an initialized instance of the appropriate class. The presented examples illustrate how this technique can effectively eliminate null pointer exceptions and enhance program survival.

Data Structure Repair: Data structure repair enables applications to recover from data structure corruption er-

rors [14]. Data structure repair enforces a data structure consistency specification. This specification can be provided by a human developer or automatically inferred from correct program executions [13].

Self-Stabilizing Java: Self-Stabilizing Java uses a type system to ensure that the impact of any errors are eventually flushed from the system, returning the system back to a consistent state and promoting successful future execution [16].

6.2 Horizontal Code Transfer

Horizontal code transfer automatically locates correct code in one application, then transfers that code into another application [35]. This technique has been applied to eliminate otherwise fatal integer overflow, buffer overflow, and divide by zero errors and shows enormous potential for leveraging the combined talents and labor of software development efforts worldwide, not just for eliminating errors but for (potentially automatically) combining and improving software in a broad range of ways.

Horizontal gene transfer is the transfer of genetic material between individual cells [21, 8]. Examples include plasmid transfer (which plays a major role in acquired antibiotic resistance [8]), virally-mediated gene therapy [20], and the transfer of insect toxin genes from bacteria to fungal symbionts of grasses [7]. There are strong analogies between horizontal code transfer and horizontal gene transfer — in both cases functionality is transferred from a donor to a recipient, with significant potential benefits to the recipient. The fact that horizontal gene transfer is recognized as significant factor in the evolution of many forms of life hints at the potential that horizontal code transfer may offer for software systems.

7. CONCLUSION

The difficulty of generating a search space rich enough to correct defects while still supporting an acceptably efficient search algorithm has significantly limited the ability of previous automatic patch generation systems to generate successful patches [24, 39]. SPR's novel combination of staged program repair, transformation schemas, and condition synthesis highlight how a rich program repair search space coupled with an efficient search algorithm can enable successful automatic program repair.

8. REFERENCES

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[2] GenProg | Evolutionary Program Repair - Univeristy of Virginia. `http://dijkstra.cs.virginia.edu/genprog/`.

[3] Home - Lighttpd - fly light. `http://www.lighttpd.net/`.

[4] LibTIFF - TIFF library and utilities. `http://www.libtiff.org/`.

[5] PHP: DatePeriod::__construct - Manual. `http://php.net/manual/en/dateperiod.construct.php`.

[6] PHP: Hypertext Preprocessor. `http://php.net/`.

[7] K. Ambrose, A. Koppenhofer, and F. Belanger. Horizontal gene transfer of a bacterial insect toxin gene into the epichloe fungal symbionts of grasses. *Scientific Reports*, 4, July 2014.

[8] M. Barlow. What Antimicrobial Resistance Has Taught Us About Horizontal Gene Transfer. *Methods in Molecular Biology*, 532:397–411, 2009.

[9] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 306–317, Hong Kong, China, November 2014.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[11] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[12] 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.

[13] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244, 2006.

[14] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[15] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, pages 47–56, 2008.

[16] Y. H. Eom and B. Demsky. Self-stabilizing java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 287–298, 2012.

[17] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10', pages 147–156, New York, NY, USA, 2010. ACM.

[18] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 243–253, New York, NY, USA, 2014. ACM.

[19] 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.

[20] M. A. Kay, J. C. Glorioso, and L. Naldini. Viral vectors for gene therapy: the art of turning infectious

agents into vehicles of therapeutics. *Nat Med*, 7(1):33–40, Jan. 2001.

[21] P. J. Keeling and J. D. Palmer. Horizontal gene transfer in eukaryotic evolution. *Nature Reviews Genetics*, 9(8), 8 2008.

[22] 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. IEEE Press, 2013.

[23] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450. ACM, 2012.

[24] 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 *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 3–13. IEEE Press, 2012.

[25] F. Long and M. Rinard. Staged Program Repair in SPR (Supplementary Material). `http://hdl.handle.net/1721.1/95963`.

[26] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14', pages 227–238, New York, NY, USA, 2014. ACM.

[27] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 492–495, New York, NY, USA, 2014. ACM.

[28] 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, pages 234–242, New York, NY, USA, 2014. ACM.

[29] 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*, ICSE '13', pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[30] 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.

[31] 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. ACM, 2009.

[32] 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, pages 254–265, New York, NY, USA, 2014. ACM.

[33] Z. Qi, F. Long, S. Achour, and M. Rinard. An anlysis of patch plausibility and correctness for generate-and-validate patch generation systems. MIT-CSAIL-TR-2015-003.

[34] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[35] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, P. Piselli, and M. Rinard. Automatic error elimination by multi-application code transfer. Technical Report MIT-CSAIL-TR-2014-024, 2014.

[36] N. Wang, M. Fertig, and S. Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03', pages 56–, Washington, DC, USA, 2003. IEEE Computer Society.

[37] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10', pages 61–72, New York, NY, USA, 2010. ACM.

[38] W. Weimer. Patches as better bug reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06', pages 181–190, New York, NY, USA, 2006. ACM.

[39] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE'13*, pages 356–366, 2013.

[40] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09', pages 364–374. IEEE Computer Society, 2009.

[41] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.