Staged Program Repair with Condition Synthesis

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

ABSTRACT

We present SPR, a new program repair system that combines *staged program repair* and *condition synthesis*. These techniques enable SPR to work productively with a set of *parameterized transformation schemas* to generate and efficiently search a rich space of program repairs. Together these techniques enable SPR to generate correct repairs for over five times as many defects as previous systems evaluated on the same benchmark set.

Categories and Subject Descriptors

D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging

Keywords

Program repair, Staged repair, Condition synthesis

1. INTRODUCTION

Despite decades of effort, defect triage and correction remains a central concern in software engineering. Indeed, modern software projects contain so many defects, and the cost of correcting defects remains so large, that projects typically ship with a long list of known but uncorrected defects. Consequences of this unfortunate situation include pervasive security vulnerabilities and the diversion of resources that would be better devoted to other, more productive, activities.

Automatic program repair holds out the promise of significantly reducing the time and effort required to deal with software defects. In the last decade researchers have developed automatic techniques that have been demonstrated to successfully correct targeted but important classes of defects such as out of bounds accesses [33, 30, 34], integer overflow errors [34], null pointer dereferences [11, 23], infinite loops [18], memory leaks [27], and data structure corruption errors [9, 10, 8]. But impoverished search spaces and inefficient search algorithms have crippled the ability of previous systems to generate correct patches for more general classes of defects [20, 37, 32].

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786811

1.1 Staged Program Repair (SPR)

We present SPR, a new program repair system that uses a novel *staged program repair* strategy to efficiently search a rich search space of candidate repairs. Three key techniques work synergistically together to enable SPR to generate successful repairs for a range of software defects. Together, these techniques enable SPR to generate correct repairs for over five times as many defects as previous systems evaluated on the same benchmark set:

- **Parameterized Transformation Schemas:** SPR deploys a set of general transformation schemas, each of which implements a strategy designed to generate repairs that correct an identified class of defects. Because each schema is parameterized, it represents a large class of program transformations. Together, these schemas enable SPR to work with a rich search space that contains many successful repairs for common defects.

- **Target Value Search:** Given a parameterized transformation schema, SPR uses *target value search* to quickly determine if there is *any* parameter value that will enable the schema to produce a successful repair. If not, SPR rejects the schema and bypasses all of the many repairs that the schema can generate.

- **Condition Synthesis:** Many of the SPR transformation schemas take a logical condition as a parameter. The SPR condition synthesis algorithm first uses target value search to obtain constraints that any such logical condition should satisfy. It then works with these constraints to synthesize logical expressions that enable the transformation schema to generate a successful repair.

A key insight is that staging the repair process as transformation selection, target value search, and condition synthesis enables SPR to immediately bypass the overwhelming majority of candidate repairs and focus the search on the most promising regions of the space. Our results highlight the effectiveness of this strategy — staged repair can deliver two orders of magnitude reduction in the size of the space that SPR explores. Staged program repair is therefore critical in enabling SPR to work productively with large repair search spaces that contain many useful repairs.

1.2 Experimental Results

We evaluate SPR on 69 real world defects and 36 functionality changes from the repositories of eight real world applications. This is the same benchmark set used to evaluate several previous automatic patch generation systems [20,

37, 32].[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, SPR generates the correct repair as the first repair to validate (i.e, produce correct outputs for all test cases in the test suite). These correct repairs semantically match corresponding repairs provided by human developers. For comparison, previous systems generate correct patches for only one [20] or two [37, 32] of the 69 defects in this benchmark set.

A repair is *plausible* if it produces correct outputs for all of the test cases in the test suite (a plausible repair may be incorrect — it may produce incorrect outputs for test cases not in the test suite). SPR generates plausible repairs for 38 of the 69 defects (and 3 of the 36 functionality changes). For comparison, previous systems generate plausible patches for only 16 [20] or 25 [37, 32] of the 69 defects in this benchmark set.[2]

These results highlight the success of SPR's staged approach and the synergistic relationship between its three novel techniques. Parameterized transformation schemas produce a rich repair space that contains many useful repairs. Target value search enables the efficient search algorithm required to make this rich search space viably searchable in practice. And condition synthesis delivers the logical expressions required to successfully instantiate the transformation schemas to obtain successful repairs.

1.3 Repair Prioritization and Prophet

SPR uses a set of heuristics to prioritize the order in which it validates repairs (see Section 3.4). The goal is to obtain a sequence of plausible repairs in which the correct repair appears as early as possible (and ideally the first repair in the sequence to validate). The SPR heuristics are reasonably effective at satisfying this goal — for 11 of the 19 relevant defects the correct repair is the first to validate. But there is room for improvement.

Prophet [21] searches the same repair space as SPR, but works with a large corpus of correct repairs from human developers. It processes this corpus to learn a probabilistic model that assigns a probability to each candidate repair in the search space. This probability indicates the likelihood that the repair is correct. It then uses this model to prioritize its search of the repair space. The results show that Prophet's learned repair correctness model outperforms SPR's heuristics — for 15 of the 19 defects, Prophet finds the correct repair as the first repair to validate. This result highlights how leveraging information available in existing large software development projects can significantly improve our ability to automatically manipulate large software systems.

[1]Papers for these previous systems report that this benchmark set contains 105 defects [20, 37]. 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, previous papers report incorrect results [20, 37, 32]. Specifically, previous papers report patches for 55 [20] and 54 [37] of the 105 defects/changes. But for 37 of these 55 defects/changes [20, 32] and 27 of these 54 defects/changes [37, 32] none of the reported patches produces correct outputs for the test cases in the test suite used to validate the patches [20, 37, 32].

1.4 Contributions

This paper makes the following contributions:

- **Staged Program Repair:** It introduces staged program repair a new technique for automatically 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.
- **Condition Synthesis:** It presents a novel condition synthesis algorithm. This algorithm first uses target value search to obtain constraints that would enable the repaired program to produce correct outputs for all test cases in the test suite. It then generates logical 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.

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 a defect (PHP bug #54283) in its implementation of the DatePeriod object constructor [3]. 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 contains this defect. 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 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 PHP string object. `isostr_len` is the string length.

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_cond() -> (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 defect. Specifically, we give SPR:

- **Program to Repair:** Version 309579 of the PHP source code (this version contains the defect).
- **Negative Test Cases:** Test cases that expose the defect — 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.

Defect 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 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 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)`. PHP passes all test cases and SPR has found a successful repair (lines 30-31 in Figure 1).

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.

3. DESIGN

SPR starts with a program with a defect and a test suite. The suite contains *positive test cases*, for which the program already produces correct outputs, and *negative test cases*, which expose the defect that causes the program to produce incorrect outputs. The goal is to generate a repair that enables the program to pass the supplied test suite (i.e, produce correct outputs for all test cases in the test suite).

```
c := 1 | 0 | c1 && c2 | c1 || c2 | !c1 | (c1) | v==const
prts := print v | print const
simps := v = v1 op v2 | v = const | v = read | prts
ifs   := if (c) ℓ1 ℓ2
absts := if (c && !abstc) ℓ1 ℓ2 | if (c || abstc) ℓ1 ℓ2
         | print abstval
s     := skip | stop | simps | ifs | absts
v, v1, v2 ∈ Variable  const ∈ Int  ℓ1, ℓ2 ∈ Label
c, c1, c2 ∈ CondExpr  s ∈ Stmt  ifs ∈ IfStmt
prts ∈ PrintStmt  simps ∈ SimpleStmt
absts ∈ AbstStmt
```

Figure 2: The language statement syntax

SPR first uses fault localization to identify target statements to transform [39, 15, 5]. It then stages the search for a successful repair as follows. It first selects a transformation schema to apply to a target statement. The result is a candidate repair template, which may contain an abstract expression as the template parameter. SPR then uses target value search to determine if there is any parameter value that would instantiate the candidate repair template to deliver a successful repair. If so, SPR synthesizes candidate parameter values and attempts to validate each resulting repair in turn.

3.1 Core Language

Language Syntax: Figure 2 presents the syntax of the core language that we use to present our algorithm. The current implementation of SPR works with applications written in the C programming language. See Section 3.4 for the extension of our algorithm to handle C.

A program is a pair $\langle p, n\rangle$, where p : Label $\rightarrow$ Statement maps each label to the corresponding statement and n : Label $\rightarrow$ Label maps each label to the label of the next statement to execute. 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 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. Statements that contain an abstract expression (i.e., `AbstStmt`) are temporary statements that the algorithm may introduce into a program during the repair algorithm. Such statements do not appear in the original or repaired programs.

Program State: A program state $\langle \ell, \sigma, I, O, D, R, S\rangle$ contains the current program point (a label ℓ), the current environment that maps each variable to its value (σ : Variable $\rightarrow$ Int), the remaining input (I), and the generated output (O). I is a sequence of integer values (i.e., Sequence(Int)). O is a sequence of integer and abstract values (i.e., Sequence(Int $\cup$ {`abstval`})).

To support the extension to programs with abstract expressions, the program state also contains 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 expression execution (S). 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 $\rightarrow$ Int)).

The first three rules in Figure 3 present the operational semantics of input read, if, and print statements. "$\circ$" 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. Note that for programs that do not contain abstract expressions, D, R, and S are unchanged. See our technical report [22] for the rules for other kinds of statements.

3.2 Transformation Schemas

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. $TL(\langle p, n\rangle)$ is the set of target statement labels to transform. Our error localizer (Section 3.4) identifies this set of statements. $SimpleS(p)$ denotes all simple statements (i.e. `SimpleStmt`) in p. $Vars(p)$ and $Vars(s)$ denote all variables in the program p and in the statement s, respectively. $Consts(p)$ denotes all constants in p. 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. Specifically, SPR works with the following transformation schemas:

- **Condition Refinement:** Given a target if statement, SPR transforms the condition of the if statement by conjoining or disjoining an abstract condition to the original if condition ($M_{Tighten}$ and M_{Loosen}).
- **Condition Introduction:** Given a target statement, SPR transforms the program so that the statement executes only if an abstract condition is true (M_{Guard}).
- **Conditional Control Flow Introduction:** SPR inserts a new control flow statement (return, break, or goto an existing label) that executes only if an abstract condition is true ($M_{Control}$).
- **Insert Initialization:** For each identified statement, SPR generates repairs that insert a memory initialization statement before the identified statement (M_{Init}).
- **Value Replacement:** For each identified statement, SPR generates repairs that replace either 1) one variable with another, 2) an invoked function with another, or 3) a constant with another constant (M_{Rep}).
- **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 (M_{CpRep}).

Note that the transformation schemas $M_{Tighten}$, M_{Loosen}, M_{Guard}, and $M_{Control}$ introduce an abstract condition into the generated candidate programs. The transformation schemas M_{Rep} and M_{CpRep} may introduce a print statement with an abstract expression. These abstract conditions and expressions will be handled by the repair algorithm later.

3.3 Staged Repair with Condition Synthesis

Figure 6 presents our main staged repair 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. Exec$(\langle p, n\rangle, I, D)$ at lines 6, 13, 21, and 37 produces the results of running the program $\langle p, n\rangle$ on the input I given the future abstract condition value sequence D. Test$(\langle p, n\rangle, NegT, PosT)$ at lines 31, 40, and 42 produces

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

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

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

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

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

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

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

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

Figure 3: Small step operational semantics

$$\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 TL(\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 TL(\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{CpRep}}(\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{ || 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{ || 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 \&\& !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{CpRep}}(\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{ = }\mathit{const}) & = & \{v\prime\texttt{ = }\mathit{const}, v\texttt{ = }\mathit{const}\prime \mid \forall v\prime \in \mathit{Vars}(p), \forall \mathit{const}\prime \in \mathit{Consts}(p)\} \\
\mathrm{RepS}(p, v\texttt{ = read}) & = & \{v\prime\texttt{ = read} \mid \forall v\prime \in \mathit{Vars}(p)\} \\
\mathrm{RepS}(p, \mathit{prts}) & = & \{\texttt{print abstval}\}, \text{where } \mathit{prts} \in \texttt{PrintStmt} \\
\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.

a boolean to indicate whether the program $\langle p, n\rangle$ passes all test cases. See Figure 5 for the definitions of Exec and Test.

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 expression, the algorithm simply uses Test to validate the program (lines 42-43). If the candidate program contains an abstract condition, the algorithm applies condition synthesis (lines 3-33) in two stages, target condition value search and condition generation.

Target Condition Value Search: We augment the operational semantics of the core language to handle statements with an abstract condition. The last four rules in Figure 3 present the semantics of "`if (`c `&& abstc)` $\ell_1\ \ell_2$". The rules for "`if (`c `|| abstc)` $\ell_1\ \ell_2$" are similar [22].

The fifth rule in Figure 3 specifies 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 sixth rule specifies the case where there are no more future abstract condition values in D for the abstract condition `abstc`. This rule uses the semantics-preserving value for the abstract condition `abstc`, with R and S appropriately updated. The last two 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 6 searches a sequence of abstract condition values with the 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 flips the last non-zero value in a given value sequence (see Figure 5 for the formal definition). 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, it moves on to the next candidate program (line 16).

Note that the program may execute an abstract condition multiple times. 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 sequence. In the last iteration SPR flips all abstract condition values to one (line 10 in Figure 6).

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). It 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 ($\mathrm{R}\prime$) and environments ($S\prime$). It counts the number of recorded condition values that the condition matches. Our current condition space is the

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

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

$$F(\epsilon, \epsilon, c) = 0$$

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

$$V(\epsilon, \epsilon, S, C) = C \qquad \frac{O\prime \neq O \qquad O\prime = \epsilon \text{ or } O = \epsilon}{V(O\prime, O, S, C) = \emptyset} \qquad \frac{S = \sigma \circ S_0 \qquad O\prime = x \circ O_0\prime \qquad O = y \circ O_0 \qquad x = y}{V(O\prime, O, S, C) = V(O_0\prime, O_0, S_0, C)}$$

$$\frac{O\prime = x \circ O_0\prime \qquad O = y \circ O_0 \qquad x \neq y \qquad x \neq \texttt{abstval}}{V(O\prime, O, S, C) = \emptyset} \qquad \frac{S = \sigma \circ S_0 \qquad O\prime = \texttt{abstval} \circ O_0\prime \qquad O = y \circ O_0 \qquad C\prime = V(O_0\prime, O_0, S_0, C)}{V(O\prime, O, S, C) = \{v \mid \sigma(v) = y, v \in C\prime\} \cup \{y \mid y \in C\prime\}}$$

Figure 5: Definitions of Exec**,** Test**,** Flip**,** F**, and** V

```
Input  : original program ⟨p, n⟩
Input  : positive and negative test cases NegT and PosT, each
         is a set of pairs ⟨I, O⟩ where I is the test input and O
         is the expected output.
Output: the repaired program ⟨p′, n′⟩, or ∅ 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 if p′ contains abstval then
35         C ⟵ Variable ∪ Int
36         for ⟨I, O⟩ in NegT do
37             ⟨O′, _, S⟩ ⟵ Exec(⟨p′, n′⟩, I, ϵ)
38             C ⟵ V(O′, O, S, C)
39         for val in C do
40             if Test(⟨p′[val/abstval], n′⟩, NegT, PosT) then
41                 return ⟨p′[val/abstval], n′⟩
42     else if Test(⟨p′, n′⟩, NegT, PosT) then
43         return ⟨p′, n′⟩
44 return ∅
```

Figure 6: Repair generation algorithm with condition synthesis

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 current SPR condition space contains a remarkable number of correct repairs, with extensions to this space delivering relatively few additional correct repairs (see Section 4.4).

F(R′, S′, c) in Figure 6 is an utility function that counts the number of branch directions for the condition c that match the recorded abstract condition values $R\prime$ given the recorded environments $S\prime$. See Figure 5 for the formal definition. 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 algorithm 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 do not pass the target 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 deliver a correct repair. In our experiments, SPR performs the condition generation stage for less than 1% of the candidate programs that contain an abstract condition (see Section 4.3).

Staged Repair for Print Statements: We augment the semantics to support print statements with abstract expressions as shown as the fourth rule in Figure 3. When such a print statement is executed, a special token `abstval` is appended to the output sequence O to represent an undetermined value.

At the first stage, the algorithm executes the program with such abstract print statements on each of the negative test cases (lines 35-38). The algorithm compares the result output sequence $O\prime$ with the expected output sequence O. This comparison uses the utility function $V(O\prime, O, S, C)$ to compute a set of concrete values that, after replacing the abstract expression, enable the repaired program to pass the test case. See Figure 5 for the formal definition of V. If the first stage succeeds, the second stage replaces the abstract expression with the computed concrete values and validates the resulting repair (lines 39-41).

3.4 Extensions for C

C Program Support: We have implemented SPR in C++ using the clang compiler front-end [1]. 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 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*) or `!(`*v* `==` *const*`)` described in Section 3.3).

When SPR inserts control statements, SPR generates repairs that include `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 generates repairs that call `memset()` to initialize memory blocks. When SPR copies statements, SPR generates repairs that copy compound statements in addition to simple statements, as long as the copied code can fit into the new context. SPR also extends its staged repair algorithm to constant string literals in C print statements (e.g., `printf()`). See our technical report [22] for implementation details.

Error Localizer: The SPR error localizer 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 test cases and produces a prioritized list that contains target statements to modify based on the recorded timestamp values. 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. Our technical report [22] presents the error localization algorithm.

Repair Test Order: SPR validates the generated candidate repairs one by one (line 1 in Figure 6). SPR prioritizes the generated patches 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 with abstract print statements.
5. SPR tests repairs that insert a memory initialization.
6. SPR tests repairs that insert an if-statement before a statement s, where s is not the first statement of a compound statement.
7. SPR tests repairs a) that replace a statement or b) that insert a non-if statement (i.e., generated by M_{CpRep}) before a statement s where s is the first statement of a compound statement.
8. SPR finally tests the remaining repairs.

If two repairs have the same tier in the previous list, their validation order is determined by the rank of the two corresponding original statements (which the two repairs modify) in the list returned by the error localizer.

4. EXPERIMENTAL RESULTS

We evaluate SPR on a benchmark set containing 69 defects and 36 functionality changes drawn from eight large open source applications, libtiff, lighttpd, the PHP interpreter, gmp, gzip, python, wireshark, and fbc [2, 20]. We address the following questions:

- **Repair Generation:** How many correct/plausible repairs can SPR generate for this benchmark set?
- **Design Decisions:** How do the various SPR design decisions affect the ability of SPR to generate repairs?
- **Previous Systems:** How does SPR compare with previous systems on this benchmark set?

4.1 Methodology

Reproduce the Defects/Changes: For each of the eight 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 [32]. For libtiff we implemented only partially automated patch validation, manually filtering the final generated repairs to report only plausible repairs [32]. 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. 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, 20, 37]. [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 test case 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 defects/changes for which SPR generated plausible repairs. Our analysis indicates that the developer patches are 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 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.

[3]Previous systems require the user of the system to identify a source code file to patch [2, 20, 37]. 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.

Table 1: Overview of SPR Repair Generation Results

App	LoC	Tests	Defects/ Changes	Plausible				Correct				Init Time	SPR Time	SPR WSF Time
				SPR	SPR WSF	Gen Prog	AE	SPR	SPR WSF	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	10.8m	18.0m
lighttpd	62k	295	7/2	3/1	4/2	4/1	3/1	0/0	0/0	0/0	0/0	7.2m	111.0m	175.0m
php	1046k	8471	31/13	16/1	19/1	5/0	7/0	9/0	9/0	1/0	2/0	13.7m	119.5m	141.3m
gmp	145k	146	2/0	2/0	2/0	1/0	1/0	1/0	1/0	0/0	0/0	7.5m	94.0m	70.5m
gzip	491k	12	4/1	2/0	2/0	1/0	2/0	0/0	1/0	0/0	0/0	4.2m	10.5m	17.5m
python	407k	35	9/2	5/1	3/1	0/1	2/1	0/0	0/1	0/1	0/1	31.1m	137.0m	284.8m
wireshark	2814k	63	6/1	4/0	4/0	1/0	4/0	0/0	0/0	0/0	0/0	58.8m	23.5m	24.3m
fbc	97k	773	2/1	1/0	1/0	1/0	1/0	0/0	0/0	0/0	0/0	8.0m	49m	32m
Total			69/36	38/3	40/4	16/2	25/2	11/0	12/1	1/1	2/1			

4.2 Summary of Experimental Results

Table 1 summarizes our benchmark set and our experimental results. 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 WSF) presents the corresponding numbers for SPR running with a specified source code file to repair. For comparison, Columns 7-8 present the corresponding results for GenProg [20] and AE [37].[4] Columns 9-12 present the corresponding results for correct repairs.

Even with no specified source code file, SPR generates plausible repairs for at least 13 more defects than GenProg and AE (38 for SPR vs. 16 for GenProg and 25 for AE; GenProg and AE require the user to provide this information). The GenProg result tar file [2] reports results from 10 different GenProg executions with different random seeds. We count the defect as patched correctly by GenProg if any of the patches for that defect is correct. Our results show that SPR generates correct repairs for at least nine more defects than GenProg and AE (11 for SPR vs. one for GenProg and two for AE), even when the target source file is not specified.

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 Time) presents the average execution time of SPR on all defects/changes for which SPR generates repairs. Column 15 (SPR WSF Time) presents the average execution time for the runs where we specify a source code file to repair. When SPR generates a repair, it does so in less than two hours on average.

4.3 Correct Repair Analysis

When the target source file is not specified, the SPR repair search space contains correct repairs for 20 defects/changes. Table 2 classifies these 20 correct repairs. The first 11 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) contains entries of the form 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.

[4]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 test cases in the test suite used to validate the patches [32]. See previous work on the analysis of GenProg and AE patches for details [32].

Table 2: SPR Repair Type and Condition Synthesis Results

Defect/ Change	Repair Type	Condition Value Search	
		On	Off
php-307562-307561	Replace†	0/139	6.3X
php-307846-307853	Add Init†	0/126	3.2X
php-307914-307915	Replace Print†‡	0/188	67.5X
php-308734-308761	Guarded Control†‡	6/257	4.5X
php-309516-309535	Add Init†	0/133	5.8X
php-309579-309580	Change Condition†‡	1/64	34.0X
php-309892-309910	Delete	1/144	25.5X
php-310991-310999	Change Condition†	4/101	68.9X
php-311346-311348	Redirect Branch†	1/89	42.4X
libtiff-ee2ce5-b5691a	Add Control†‡	3/294	94.1X
gmp-13420-13421	Replace†‡	0/515	3.7X
php-308262-308315	Add Guard†‡	N/A	6.9X
php-309111-309159	Copy‡	N/A	2.9X
php-309688-309716	Change Condition†‡	N/A	4.0X
php-310011-310050	Copy and Replace†‡	N/A	4.5X
libtiff-d13be-ccadf	Change Condition†	N/A	121.4X
libtiff-5b021-3dfb3	Replace†	N/A	5.5X
gzip-a1d3d4-f17cbd	Copy and Replace†‡	N/A	5.0X
python-69783-69784	Delete	N/A	40.6X
fbc-5458-5459	Change Condition†‡	N/A	18.8X

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 non-print statement using value replacement to replace an atom inside it. "Replace Print" indicates that the repair replaces an existing print statement via staged repair. "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 an initialization statement. "Delete" indicates that the repair removes statements (this is a special case of the Condition Introduction 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 statement.

A "†" in Column 2 indicates that the SPR repair for this defect is outside the search space of GenProg and AE (17 out of the 20 defects/changes). 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 [17] (11 out of the 20 defects/changes). See our technical report [22] for the analysis details.

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 outputs for all test cases. 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 99.2% of the schema applications are discarded before SPR even attempts to find a condition that will repair the program. 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 values. We attribute the discrepancy to the ability of the program to generate a correct result for both branch directions [35].

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.

4.4 Search Space Extensions

The current SPR repair space contains repairs for 19 of the 69 defects. Increasing the threshold of the error localization ranked list from 200 to 2000 would bring a repair for an additional defect into the search space. Extending the SPR condition space to include comparison operations ($<, \leq, \geq, >$) would bring repairs for an additional two defects into the repair space. 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 three defects into the space. Extending the Copy and Replace schema instantiation space to include more sophisticated replacement expressions would bring repairs for four more defects into the search space. Combining all three of these extensions would bring an additional six more defects into the search space. 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 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 [20]. Nevertheless, one potential threat to validity is that our results may not generalize to other defects and test suites.

SPR applies one transformation at each time it generates a candidate repair. It is unclear how to combine multiple transformations and still efficiently explore the enlarged space. Note that previous tools [20, 31] that apply multiple transformations produce only semantically simple patches. The overwhelming majority of the patches are incorrect and equivalent to simply deleting functionality [32].

6. RELATED WORK

CodePhage: Horizontal code transfer automatically locates correct code in one application, then transfers that code into another application [34]. 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.

ClearView: ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [30]. 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 to enforce the invariant.

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 vulnerabilities. SPR operates on a search space derived from its transformation schemas to generate repairs, while ClearView generates patches to enforce violated invariants.

GenProg, RSRepair, AE, and Kali: GenProg [38, 20] 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 [31] changes the GenProg algorithm to use random search instead. AE [37] uses a deterministic 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 [32]. 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 [32]. The Kali patch generation system, which only eliminates functionality, can do as well [32].

Unlike GenProg [20], RSRepair [31], and AE [37], 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 [17] 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 [17]. Monperrus found that PAR fixes the majority of its benchmark defects with only two templates ("Null Pointer Checker" and "Condition Expression Adder/Remover/Replacer") [25].

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 [17]) is in fact a subset of the SPR search space. Moreover, the difference is meaningful — the SPR correct repairs for at least 11 of our benchmark defects are outside the PAR search space (see Section 4.3). This result illustrates the fragility and unpredictability of using fixed patch templates.

SemFix and MintHint: SemFix [26] and MintHint [16] replace the faulty expression with a symbolic value and use symbolic execution techniques [4] 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.

Repair Model and Repair Shape: Martinez and Monperrus [24] propose to stage the program repair in three steps, error localization, selecting a repair shape, and repair synthesis. They also mine the past human repairs to obtain a probabilistic distribution of different repair shapes. The SPR transformation schemas, in contrast, enable SPR to preliminarily validate the result candidate patch templates and significantly reduce the number of candidate patches SPR needs to consider.

Debroy and Wong: Debroy and Wong [6] present a transformation-based patch generation technique. This technique either replaces an arithmetic operator with another operator or negates a condition. In contrast, SPR uses more sophisticated and effective transformations and search algorithms. None of the correct repairs in SPR's search space for the 19 defects are within the Debroy and Wong search space.

NOPOL: NOPOL [7, 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). In this case NOPOL will fail to generate a correct patch.

AutoFixE: AutoFix-E [36, 29] operates with a set of fix schemas to repair Eiffel programs with human-supplied specifications called contracts. SPR differs from AutoFixE in that it requires no specification and uses the staged program repair strategy to efficiently explore the search space.

Deductive Program Repair: Deductive Program Repair formalizes the program repair problem as a program synthesis problem, using the original defective program as a hint [19]. It replaces the expression to repair with a synthesis hole and uses a counterexample-driven synthesis algorithm to find a patch that satisfies a formal specification. SPR, in contrast, works with large real world applications, where formal specifications are typically not available.

Domain Specific Repair Generation: Other program repair systems include VEJOVIS [28] and Gopinath et al. [14], which applies domain specific techniques to repair DOM-related faults in JavaScript and selection statements in database programs respectively. SPR differs from all of this previous research in that it focuses on generating fixes for general purpose applications without human-supplied specifications.

Failure-Oblivious Computing: Failure-oblivious computing [33] 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.

RCV: RCV [23] 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.

Bolt: Bolt [18] 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.

Cyclic Memory Allocation: Cyclic memory allocation eliminates memory leaks by cyclically allocating objects out of a fixed-size buffer [27].

Data Structure Repair: Data structure repair enables applications to recover from data structure corruption errors [9, 10]. It enforces a data structure consistency specification. This specification can be provided by a developer or automatically inferred from correct program executions [8].

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 [13].

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 [20, 37, 32]. SPR's novel combination of staged program repair, parameterized transformation schemas, target value search, and condition synthesis highlight how a rich program repair search space coupled with an efficient search algorithm can enable successful automatic program repair.

8. ACKNOWLEDGEMENTS

We would like to thank Zichao Qi and Sara Anchor for their valuable help on the experiments. We also thank the anonymous reviewers for their insightful comments. This research was supported by DARPA (Grant FA8650-11-C-7192).

Table 3: SPR Results for Each Generated Plausible Repair Obtained via the Replication Package

Defect/ Change	SPR				SPR(With Specified File Name)				Gen Prog	AE	SPR Time	SPR (WSF) Time
	Search Space	Gen At	Correct At	Result	Search Space	Gen At	Correct At	Result				
libtiff-ee2ce-b5691	67202	283	283	Correct	228157	1012	1012	Correct	No	Gen	9m	26m
libtiff-d13be-ccadf	71632	7	7	Gen	207237	116	116	Gen	Gen	Gen	13m	15m
libtiff-90d13-4c666	69955	296	-	Gen	232963	1029	-	Gen	No	Gen	9m	27m
libtiff-5b021-3dfb3	156766	40	17875	Gen	190979	24	19092	Gen	Gen	Gen	4m	4m
libtiff-08603-1ba75	74108	54	-	Gen	209076	70	-	Gen	Gen	Gen	19m	18m
lighttpd-1806-1807	19895	-	-	No	9662	30	-	Gen	Gen	Gen	-	676m
lighttpd-1913-1914	31120	8	-	Gen	58347	33	-	Gen	Gen	No	175m	174m
lighttpd-1948-1949	25037	152	-	Gen	87815	2	-	Gen	No	No	73m	51m
lighttpd-2661-2662	22608	951	-	Gen	26544	5	81	Gen	Gen	Gen	150m	36m
php-307562-307561	22851	1672	1672	Correct	10825	959	959	Correct	No	No	192m	28m
php-307846-307853	17585	1852	1852	Correct	49643	4470	4470	Correct	No	No	51m	108m
php-307914-307915	230811	198	198	Correct	3300	140	140	Correct	No	No	34m	30m
php-307931-307934	57714	-	-	No	30234	8	-	Gen	Gen	Gen	-	174m
php-308262-308315	70577	-	-	No	14725	39	2071	Gen	No	No	-	186m
php-308323-308327	33295	-	-	No	3965	3	-	Gen	No	No	-	34m
php-308525-308529	33933	121	-	Gen	9798	73	-	Gen	No	Gen	236m	362m
php-308734-308761	10390	2179	2179	Correct	5425	464	464	Correct	No	No	237m	174m
php-309111-309159	40641	74	12312	Gen	25794	52	9235	Gen	No	Correct	70m	61m
php-309516-309535	21016	1189	1189	Correct	51074	3683	3683	Correct	No	No	39m	100m
php-309579-309580	40106	4	4	Correct	8736	18	18	Correct	No	No	45m	37m
php-309688-309716	45498	361	4206	Gen	5779	253	501	Gen	No	No	29m	15m
php-309892-309910	27030	15	15	Correct	13925	6	6	Correct	Correct	Correct	62m	50m
php-309986-310009	45339	27	-	Gen	9934	5	-	Gen	Gen	Gen	409m	82m
php-310011-310050	52498	12	14611	Gen	2242	210	1382	Gen	Gen	Gen	301m	232m
php-310370-310389	44436	50	-	Gen	3640	2	-	Gen	No	No	103m	89m
php-310673-310681	21746	483	-	Gen	25704	1685	-	Gen	Gen	Gen	26m	157m
php-310991-310999	69387	12	12	Correct	454797	351	351	Correct	No	No	101m	219m
php-311346-311348	5099	21	21	Correct	7847	29	29	Correct	No	No	36m	68m
gmp-13420-13421	41681	6278	6278	Correct	26995	4024	4024	Correct	No	No	173m	129m
gmp-14166-14167	25539	4	-	Gen	6336	2	-	Gen	Gen	Gen	15m	12m
gzip-a1d3d4-f17cbd	33426	573	7898	Gen	75715	1089	1089	Correct	No	Gen	5m	19m
gzip-3fe0ca-39a362	70059	60	-	Gen	79556	100	-	Gen	Gen	Gen	16m	16m
python-69223-69224	44745	38	-	Gen	32949	1640	-	Gen	No	Gen	218m	526m
python-69368-69372	57051	69	-	Gen	20401	-	-	No	No	No	30m	-
python-69709-69710	47738	88	-	Gen	395	-	-	No	No	No	37m	-
python-70019-70023	17209	2646	-	Gen	86032	4194	-	Gen	No	No	420m	476m
python-70098-70101	17809	1485	-	Gen	42666	30	-	Gen	No	Gen	51m	83m
wshark-37112-37111	23874	2	-	Gen	22794	12	-	Gen	Gen	Gen	22m	32m
wshark-37172-37171	53801	307	-	Gen	112514	506	-	Gen	No	Gen	26m	22m
wshark-37172-37173	53801	307	-	Gen	112514	506	-	Gen	No	Gen	22m	21m
wshark-37284-37285	57946	315	-	Gen	124408	537	-	Gen	No	Gen	24m	22m
fbc-5458-5459	5847	13	54	Gen	716	3	9	Gen	Gen	Gen	49m	35m
lighttpd-1794-1795	21068	-	-	No	114801	74	-	Gen	Gen	Gen	-	70m
lighttpd-2330-2331	39200	29	-	Gen	46194	67	-	Gen	Gen	Gen	46m	43m
php-311323-311300	915018	16	-	Gen	75550	4	-	Gen	No	No	61m	620m
python-69783-69784	38144	48	52	Gen	21209	50	50	Correct	Correct	Correct	66m	54m

9. REPLICATION PACKAGE

We provide a full replication package for SPR at `http://groups.csail.mit.edu/pac/spr/`. The package contains a public AMI image for reproducing all of the experiments except fbc and a VMWare image for reproducing the fbc experiments. To reproduce the SPR experiment on a defect, start an EC2 instance or a VM and follow the step-by-step instructions inside the package.

Results: Table 3 summarizes the results available via the replication package for the 46 defects/changes for which SPR generates a plausible repair. [5] The first 42 rows present results for defects, the last 4 rows present results for functionality changes. Column 1 (Defect/Change) presents the defect/change. Columns 2-5 present results from SPR running without a specified source file to repair. Columns 6-9 present results from SPR running with a specified source file to repair.

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

Columns 2 and 6 (Search Space) 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). 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.

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

[4] 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.

[5] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11', pages 121–130, New York, NY, USA, 2011. ACM.

[6] 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.

[7] 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.

[8] 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.

[9] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

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

[11] 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.

[12] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *CoRR*, abs/1505.07002, 2015.

[13] 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.

[14] 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.

[15] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11', pages 437–446, New York, NY, USA, 2011. ACM.

[16] 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.

[17] 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.

[18] 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.

[19] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer-Aided Verification (CAV)*, 2015.

[20] 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.

[21] F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful patches. Technical Report MIT-CSAIL-TR-2015-027, 2015.

[22] F. Long and M. Rinard. Staged Program Repair in SPR. Technical Report MIT-CSAIL-TR-2015-008, 2015.

[23] 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.

[24] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[25] 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.

[26] 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.

[27] H. H. Nguyen and M. C. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th International*

Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007, pages 15–30, 2007.

[28] 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.

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

[30] 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.

[31] 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.

[32] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2015*, 2015.

[33] 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.

[34] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 43–54, 2015.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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