Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2015-004 February 12, 2015

# Automatic Program Repair with Condition Synthesis and Compound Mutations

Fan Long, Zichao Qi, Sara Achour, and Martin Rinard

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

# Automatic Program Repair with Condition Synthesis and Compound Mutations

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

## ABSTRACT

We present PCR, a new automatic patch generation system. PCR uses a new *condition synthesis* technique to efficiently discover logical expressions that generate desired control-flow transfer patterns. Presented with a set of test cases, PCR deploys condition synthesis to find and repair incorrect if conditions that cause the application to produce the wrong result for one or more of the test cases. PCR also leverages condition synthesis to obtain a set of *compound modifications* that generate a rich, productive, and tractable search space of candidate patches.

We evaluate PCR on a set of 105 defects from the GenProg benchmark set. For 40 of these defects, PCR generates *plausible patches* (patches that generate correct outputs for all inputs in the test suite used to validate the patch). For 12 of these defects, PCR generates correct patches that are functionally equivalent to developer patches that appear in subsequent versions. For comparison purposes, GenProg generates plausible patches for only 18 defects and correct patches for only 2 defects. AE generates plausible patches for only 27 defects and correct patches for only 3 defects.

## 1. INTRODUCTION

Large software systems are known to include many more defects than human developers are able to triage, understand, and repair [34]. Generate-and-validate systems start with a test suite of inputs, at least one of which exposes a defect. They then generate candidate patches in an attempt to find a (hopefully correct) patch that produces correct outputs for all inputs in the test suite [27, 34, 21, 28, 33].

Recent papers on several generate-and-validate systems report impressive results — they claim to fix over half of the bugs in a large benchmark set of bugs [21, 28, 33]. Unfortunately, because of experimental error in the patch validation, these reported results are wrong [29]. In fact, at least half of the reported patches do not generate correct results *even for the inputs used to validate the generated patches.* Moreover, the overwhelming majority of the patches are *not correct* — at the end of the day, these systems generate correct patches for less than 3% of the bugs in the benchmark set [29]. Experimental results from an implemented automatic patch generation system show that generating patches that simply delete functionality can do as well [29]. These facts do not inspire confidence in the overall viability of the generate-and-validate approach to automatic patch generation.

But of course, these are far from the only generate-and-validate patch generation systems. ClearView, for example, deploys a more targeted set of modification operations, induced patch space, and search algorithm to successfully produce patches that eliminate Firefox security vulnerabilities [27]. These results suggest that the problem lies not with the generate-and-validate approach *per se*, but with the specific set of modification operators, induced patch space, and search algorithms used in previous systems [21, 28, 33].

### 1.1 PCR

We present a new program repair system, Predicate Compound Repair (PCR). PCR deploys a novel set of modification operators and a novel search algorithm to obtain a tractable search space rich in useful patches. PCR works with a set of *positive test cases*, for which the program already produces correct outputs, and a set of *negative test cases*, for which the program produces incorrect outputs. The goal is to obtain a patch that preserves the correct behavior for the positive test cases and changes the behavior for negative test cases so that the patched program produces correct outputs for all test cases.

PCR first uses standard defect localization techniques to identify candidate statements to modify. PCR then uses the following modification operators to obtain candidate patches. It applies each candidate patch in turn as it is generated to determine if it produces correct outputs on all test cases.

- **Condition Tightening:** For each identified branch statement, PCR generates patches that conjoin the branch condition with an additional (synthesized by PCR) clause to tighten the condition.
- **Condition Loosening:** For each identified branch statement, PCR generates patches that disjoin the branch condition with an additional (synthesized by PCR) clause to loosen the condition.
- **Conditional Guard:** For each identified statement, PCR generates patches that insert an if statement that guards the statement, so that the statement executes only when a (synthesized by PCR) condition is true.
- **Conditional Control-Flow Insertion:** At the program point before each identified statement, PCR generates patches that insert a control-flow statement (return, break, or goto an existing label) guarded by an if statement, so that the control-flow statement executes only when a (synthesized by PCR) condition is true.
- **Insert Initialization:** For each identified statement, PCR generates patches that insert a memory initialization statement before the identified statement.

- **Value Replacement:** For each identified statement, PCR generates patches 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, PCR generates patches that copy an existing statement to the program point before the identified statement and then replace a value in the copied statement with another valid value.

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 [7]. The goal of the Replace part of Copy and Replace is to obtain a rich patch 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 [23, 7, 16]. This fact may also provide insight into the poor performance of previous patch generation systems [21, 33, 28] (which only copy statements without expression-level modifications).

Many of PCR's modifications introduce new if statements with new conditions. These modifications help PCR create a rich space of potentially correct patches. But this search space is effective in large part because it contains such a large and flexible set of potential conditions. PCR therefore deploys a condition synthesis algorithm that enables it to quickly find productive conditions within this very large search space. The goal of the condition synthesis algorithm is to derive a condition that:

- **Preserves Positive Test Case Behavior:** For each positive test case, the new condition should preserve the branch direction for all executions of the condition so that the execution remains unchanged and the patched program produces the correct output for the positive test case.

- **Changes Negative Test Case Behavior:** For each negative test case, the new condition should change at least one branch direction so that the program produces a different (ideally the correct) output for the negative test case.

Instead of naively enumerating and testing all possible possible conditions, PCR's condition synthesis algorithm performs the following steps:

- **Negative Test Case Executions:** PCR runs an instrumented version of the program on each negative test case to determine 1) the number of times the target condition executes and 2) the direction the target branch takes on each execution.

- **Flip Negative Test Case Directions:** PCR reexecutes an instrumented version of the program on each negative test case. On each execution, it selects one or more branch directions to flip. It then observes the resulting output to see if it is correct.

- **Instrumented Executions:** If PCR finds a sequence of branch directions that produce correct outputs for the negative test cases, it performs instrumented executions of the program on both negative and positive test cases. For each negative test case, is uses the discovered branch directions that produce the correct output.

  For each instrumented execution, PCR records a mapping from the values of the variables in the context surrounding the condition to the resulting branch directions for each execution of the condition.

- **Condition Synthesis:** The goal is to synthesize a symbolic condition, over the variables in the surrounding context, that produces the same branch directions as those recorded in the successful instrumented executions. PCR synthesizes a sequence of symbolic conditions that maximize the number of branch directions that match the branch directions in the instrumented executions.

- **Condition Synthesis Test:** PCR applies the synthesized condition and runs the patched program to determine if the synthesized condition enables the program to produce correct outputs for all test cases. If so, PCR has found a patch. If not, PCR proceeds on to test the next condition in the generated sequence of conditions.

## 1.2 Experimental Results

We evaluate PCR on 105 real world defects from 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 generate-and-validate systems [21, 33]. We say that a patch is *plausible* if it generates correct results for all of the inputs in the test suite used to validate the patch. PCR generates plausible patches for 40 of the 105 defects in the benchmark set. For 12 of these detects, PCR generates correct patches which are functionally equivalent to the developer patch in the subsequent fixed version. For comparison, GenProg [21] generates plausible patches for only 18 defects and correct patches for only 2 defects. AE [33] generates plausible patches for only 27 defects and correct patches for only 3 defects [29]. Kali, an automatic patch generation system that only deletes functionality [29], generates plausible patches for 27 defects and correct patches for 3 defects on this benchmark set.

## 1.3 Reasons for Success

We manually analyzed the correct patches that PCR generates for the 12 defects in our experiments. We attribute much of the PCR success to the following reasons:

- **Modification Operators:** PCR's modification operators generate a rich patch search space with many useful patches. The richness of the search space can be seen in comparison with the previous GenProg and AE search space — for 10 out of the 12 defects, the generated PCR correct patches lie outside the GenProg and AE patch search spaces. See Section 4.4.

- **Condition Synthesis:** The condition synthesis technique enables PCR to efficiently explore its rich patch search space. For the 12 defects, the condition synthesis reduces the number of candidate patches that PCR needs to validate by an order of magnitude on

average. In fact, without the condition synthesis PCR would be unable to find correct patches for 4 out of the 12 defects within the 12 hour PCR timeout. See Section 4.4.

## 1.4 Comparison with PAR

PAR [19] 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 PCR 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 [19]. Monperrus [24] found that PAR fixes the majority of its benchmark defects with only two templates ("Null Pointer Checker" and "Condition Expression Adder/Remover/Replacer").

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 [19]) is in fact a subset of the PCR search space. Moreover, the difference is meaningful — the PCR correct patches for at least 5 defects in our experiments are outside the PAR search space (see Section 4.4). This result illustrates the fragility and unpredictability of using fixed patch templates.

## 1.5 Contributions

This paper makes the following contributions:

- **Modification Operators:** It presents a set of novel modification operators that generate a productive search space rich in useful patches.
- **Condition Synthesis:** It presents a novel condition synthesis algorithm which enables PCR to efficiently search the rich PCR space of conditions for candidate patches.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of PCR in automatically generating patches for 105 defects from the GenProg benchmark set [21]. PCR generates plausible patches for 40 of these defects and correct patches for 12 of these defects. These results show that PCR substantially outperforms previous patch generation systems that have been evaluated on this benchmark set [21, 33, 28].
- **Rationale and Insight:** It discusses the reasons why PCR is able to generate so many successful patches in comparison with previous systems. This discussion provides insight into important issues in automatic patch generation, specifically the inadequacy of general ideas such as copying code or targeting conditionals and how the it is the specific design of the modification operations and search algorithm that determines success or failure.

PCR is dramatically more successful in repairing defects than previous systems [28, 21, 33]. This success highlights the importance of the specific modification operations and the interactions between the resulting generated patch search space and the algorithm used to search that space. General ideas such as copying statements and manipulating conditionals may be easy to come by but, in the absence of more insightful concepts such as condition synthesis and replacement, fail to deliver anything close to a successful patch generation system [21, 28, 33, 9]. A more targeted approach may produce successful patches [19, 22, 27], but only within a limited and potentially very fragile scope. PCR, with its unique combination of powerful modification operations and an efficient search algorithm driven by condition synthesis, overcomes many of the disappointing limitations of previous systems [21, 28, 33, 9].

```
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(...)==FAILURE) {
10     // Parse (DateTime, DateInterval, DateTime)
11     if (zend_parse...(...)==FAILURE) {
12       // Parse (stirng)
13       if (zend_parse_parameters_ex(...,&isostr,
14            &isostr_len, &options)==FAILURE) {
15         php_error_docref(...,"This constructor\
16          accepts either\
17          (DateTime, DateInterval, int) OR\
18          (DateTime, DateInterval, DateTime)\
19          OR (string) as arguments.");
20         ...
21         return;
22       } } }
23   dpobj = ...;
24   dpobj->current = NULL;
25 // candidate fix template
26 /* if (isostr_len || abstract_cond() ) */
27 // final generated fix
28 /* if (isostr_len || isostr) */
29 /* if (isostr)*/ // official fix
30   if (isostr_len) {
31     // Handle (string) case
32     date_period_initialize(&(dpobj->start),
33       &(dpobj->end), &(dpobj->interval),
34       &recurrences, isostr, isostr_len);
35     ...
36   } else {
37     // Handle (DateTime,...) cases
38     /* pass uninitialized 'interval' */
39     intobj  = (php_interval_obj *)
40       zend_object_store_get_object(interval);
41     ...
42   }
43   ...
44 }
```

**Figure 1: Simplified Code for PHP bug #54283**

The rest of this paper is organized as follows. Section 2 presents a motivating example of PCR. Section 3 presents the technical detail of PCR. Section 4 evaluates PCR. We discuss related work in Section 5 and then conclude in Section 6.

# 2. EXAMPLE

We next present a motivating example of using PCR to generate a patch for 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]. A PHP program that calls the DatePeriod constructor with a single `NULL` value as the parameter (e.g., `DatePeriod(NULL)`) can cause the PHP interpreter to access invalid memory address and crash.

Figure 1 presents the simplified source code from the source file `ext/date/php_date.c` that is responsible for this error. The code in Figure 1 is from the corresponding 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. Lines 9-22 in Figure 1 parse the parameters. Note that if the PHP program invokes the constructor with a single `NULL` value, the `zend_parse_parameter_ex()` function will not return FAILURE. Instead, it will set the variable `isostr` to `NULL` and the variable `isostr_len` to zero.

The if statement at line 30 in Figure 1 checks the value of `isostr_len` to determine whether the current invocation has three parameters or only one parameter. However, if the invocation contains a single NULL value as the parameter, the variable `isostr_len` will be zero and the invocation will be misclassified as having three parameters and take an incorrect branch (lines 37-41). This causes the program to pass a potentially uninitialized pointer `interval` to the function `zend_object_store_get_object()` at line 40, which eventually causes the function to access invalid memory address.

We apply PCR to automatically generate a patch for this error. Specifically, we provide PCR:

1. **Program to Patch:** The PHP interpreter source code repository at version 309579 which contains this error;

2. **Negative Test Scripts:** A set of negative test scripts that expose the error (i.e., test scripts that the PHP version 30979 cannot pass but the code after applying the generated patch should pass);

3. **Positive Test Scripts:** A set of positive test scripts that prevent regression (i.e, test scripts that the version 30979 already passes and that the patched code should still pass).

PCR then performs following steps:

- **Error Localization:** PCR compiles the given PHP interpreter version with additional profiling instrumentations which produces the execution traces. It then executes this profiling version of PHP with all test scripts in both the negative test script set and the positive test script set. PCR observes that lines 39-40 in Figure 1 are always executed with negative test scripts but are rarely executed with positive test scripts. PCR therefore recognizes this statement and its enclosing if statement condition (line 30) are potential modification targets.

- **Generate Candidate Patch:** PCR searches patches that modify the source code around the location of the previously identified potential target statements. At the 11th attempt for this error, PCR loosen the branch condition of the if statement at line 30 with an undetermined abstract condition `abstract_cond()` (see lines 25-26).

- **Candidate Fix Testing:** PCR then compiles the patched source code together with a PCR runtime library that implements an algorithm that searches over possible combinations of the inserted abstract condition. PCR tests the patched source code with test scripts in both the negative and positive test script sets. The PCR runtime library determines that the candidate patch may pass all test scripts, given that an appropriate abstract condition could be properly synthesized.

- **Branch Condition Synthesize:** The PCR branch condition synthesis algorithm observes that for all executions with negative test scripts, the variable `isostr` is not zero at the program point of the abstract condition and for all executions with positive test scripts, the variable `isostr` is zero (note that || is a short circuit operator). PCR therefore determines that `(isostr != 0)` is a potential concrete condition that can replace the abstract condition.

  PCR compiles and tests the PHP interpreter again with the abstract condition being replaced with `(isostr != 0)`. The PHP interpreter passes all test scripts and therefore PCR outputs the result patch (lines 27-28 in Figure 1) to the user.

Note that the official patch from the PHP developer in the version 309580 replaces `isostr_len` with `isostr` (line 29 in Figure 1), but `isostr_len` is always zero when `isostr` is zero at the branch condition. The PCR generated patch is therefore functionally equivalent to the official patch from the PHP developer.

# 3. DESIGN

The current implementation of PCR works with applications written in the C programming language. PCR consists of three parts, the *error localizer*, the *main search algorithm*, and the *condition synthesizer*. Given the application source code, a set of *negative test cases* that can expose the error the user wants to fix, and a set of *positive test cases* that the original application source code already passes, the error localizer produces a ranked list of statements that are potentially responsible for the error.

The main search algorithm generates a set of candidate patches that modify the original program around the statements identified by the error localizer. It then tests each of the generated candidate fixes until it finds a candidate patch that can pass all given test cases.

For a candidate patch that modifies a branch condition, the PCR main search algorithm inserts an *abstract condition* as the placeholder instead of enumerating all possible concrete conditions. For such a candidate patch, the condition synthesizer will synthesize the concrete condition to replace the abstract condition to generate the final patch.

## 3.1 Error Localizer

The PCR 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. PCR 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

**Input** : original program prog
**Input** : potential target statement set $S$
**Output**: a set of candidate programs $P$

1 $P \longleftarrow \emptyset$
2 **for** $s$ **in** stmts($prog$) **do**
3  **if** $s$ *is IF-STATEMENT* **then**
4  **if** $s \in S$ *or* thenstmts($s$) $\cap S \neq \emptyset$ **then**
5  $S_{\text{new}} \longleftarrow$ **genLooseIfCond**($s$)
6  $P \longleftarrow P \cup$ replaceStmt(prog, $s$, $S_{\text{new}}$)
7  **if** $s \in S$ *or* elsestmts($s$) $\cap S \neq \emptyset$ **then**
8  $S_{\text{new}} \longleftarrow$ **genTightIfCond**($s$)
9  $P \longleftarrow P \cup$ replaceStmt(prog, $s$, $S_{\text{new}}$)
10  **if** $s \in S$ **then**
11  $S_{\text{repl}} \longleftarrow$ **genGuardedStmt**($s$)
12  $S_{\text{repl}} \longleftarrow S_{\text{repl}} \cup$ **genReplace1Atom**($s$)
13  $P \longleftarrow P \cup$ replaceStmt(prog, $s$, $S_{\text{repl}}$)
14  $S_{\text{ins}} \longleftarrow$ **genInitStmt**($s$)
15  $S_{\text{ins}} \longleftarrow S_{\text{ins}} \cup$ **genGuardedControl**($s$)
16  $S_{\text{ins}} \longleftarrow S_{\text{ins}} \cup$ **genStmtToAdd**($s$)
17  $P \longleftarrow P \cup$ insertStmt(prog, $s$, $S_{\text{ins}}$)

**Figure 2: The main search algorithm of PCR**

at all when the application runs with the test case $i$, then $r(s,i) = 0$.

We use the notation Neg for the set of negative test cases and Pos for the set of positive test cases. PCR computes three scores $a(s)$, $b(s)$, $c(s)$ for each statement $s$:

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

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

PCR 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 patch, PCR 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 patch. Unlike GenProg [21] and AE [33], PCR can also operate completely automatically. If the user does not specify any such source file, PCR returns the top 200 statements in the whole application (potentially from different source files) as the potential modification targets.

## 3.2 Search Algorithm

PCR uses the clang front-end [1] to transform the program source code into a clang AST tree in which each node may represent a declaration, a definition, or a statement. Each node may have multiple sub-nodes to represent hierarchical relationships within the code (e.g., a compound statement contains multiple sub-statements). PCR generates patches that modify only one AST statement node in the clang AST tree of the program (including patches that modify compound statement nodes, which represent code blocks).

**Generate Candidate Patches:** Figure 2 presents the PCR candidate patch generation algorithm. Given the original program prog and a set of potential target statements $S$ identified by the error localizer, the algorithm produces a set of candidate patched programs $P$.

Note that stmts(prog) in line 2 denotes the set of statements in prog. thenstmts($s$) in line 4 denotes the set of statements in the "then" branch of the if-statement $s$. elsestmts($s$) in line 7 denotes the set of statements in the "else" branch of the if-statement $s$. replaceStmt(prog, $s$, $S$) (at lines 6, 9, 13) is an utility routine, which takes a program prog, a statement $s$, and a set of new statements $S$ and returns a set of programs, in which each program is a duplicate of prog with the statement $s$ being replaced by a corresponding new statement in $S$. insertStmt(prog, $s$, $S$) (at line 17) is an utility routine similar to replaceStmt except that the corresponding new statement is inserted before $s$ in each duplicate program in the returned set instead of replacing $s$.

There are seven boldfaced subroutines in the algorithm (see lines 5, 8, 11, 12, 14, 15, and 16 in the Figure 2) whose names start with **gen**. These subroutines implement *compound modifications* that PCR supports. Each of these subroutines takes a statement $s$ and returns a set of statements that PCR will use to replace or insert before the statement $s$. These subroutines work as follows:

- **Tighten or Loosen an If Condition:** For an if-statement $s$, if $s$ or any statement in the then branch of $s$ is in the potential target statement set identified by the error localizer, PCR will generate a patched program with the branch condition in the if-statement $s$ being tightened (see lines 4-6 in Figure 2). The **genTightIfCond**($s$) routine produces a set that contains a single duplicate statement of the if statement $s$. The routine appends to the duplicate statement an additional "and" clause with an undetermined abstract condition to the branch condition (i.e., appending `&& abstract_cond()`). PCR similarly calls **genLooseIfCond**($s$) to generate a patch with loosen branch condition (see lines 7-9).
- **Add If Guard for a Statement:** For each statement $s$ that the error localizer identifies as potential modification target. PCR generates a patch in which the statement $s$ is executed only if the program state satisfies an undetermined abstract condition (see line 11). The routine **genGuardedStmt**($s$) returns a set that contains only the statement `if (abstract_cond()) { s; }`.
- **Add Memory Initialization:** For each identified statement $s$ and each pointer $p$ that $s$ dereferences, PCR generates a patch in which the entire memory region that $p$ points to is set to zero before executing $s$ (see line 16). The size of the memory region is determined by the type signature of the pointer $p$. The routine **genInitStmt**($s$) returns a set of statements that contains a corresponding `memset()` call initialization statement for each pointer $p$ used in $s$.

- **Add a Guarded Control Statement:** For each identified statement $s$, PCR generates patches that insert guarded control flow statements before $s$ (line 15). The routine **genGuardedControl**($s$) returns a set of if-guarded control flow statements that PCR can insert before $s$ for this kind of patches. Again, the conditions of the inserted if-statements are undetermined abstract conditions. The current implementation of PCR considers the following control statements: 1) return statements that return void or constant integers that appear in the same function, 2) break statements inside a loop, 3) and goto statements that jump to an existing label in the same function.

- **Replace a Statement:** For each identified statement $s$, PCR generates patches in which one of the program values in $s$ is replaced by another value of the same type (line 12). In the current implementation, PCR considers 1) replacing a variable with another local variable, 2) replacing a constant enumerated value with another constant value in the same enumeration, 3) and replacing a function in a call expression with another function that has the same type signature. **genReplace1Atom**($s$) returns the set of statements formed by replacing exactly one program value in $s$. PCR then generates a candidate patch by replacing $s$ with each statement in the returned set of **genReplace1Atom**($s$) (see lines 12 and 13).

- **Add a Statement:** For each identified statement $s$, PCR generates patches in which a new statement $s_{\text{new}}$ is inserted before $s$. The routine **genStmtsToAdd**($s$) computes the set of statements that PCR will add before $s$. It first collects all simple statements (excluding complicated statements that may contain substatements) that appear elsewhere in the same source file. For each collected simple statement, it performs the replacement modifications similar to the routine genReplace1Atom that we discussed previously. For each collected simple statement and each modified statement we obtained, if the statement is semantically valid to insert before $s$, the routine will add it to the result set. PCR then generates a patch by inserting each statement in the returned set of **genStmtsToAdd**($s$) before $s$ (lines 16 and 17).

Note that PCR does not include any modifications that delete a statement. Statement deletion is simply a special case of the compound modification that adds an if-guard for the statement. If the branch condition of the if-guard is false, this modification effectively removes the statement.

**Patch Test Order:** PCR tests each of the generated candidate patches one by one. PCR empirically sets the patch test order as follows:

1. PCR first tests patches that changes only a branch condition (e.g., tighten and loosen a condition).
2. PCR tests patches 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. PCR tests patches that insert an if-guard around a statement $s$.
4. PCR tests patches that insert a memory initialization.
5. PCR tests patches that insert an if-statement before a statement $s$, where $s$ is not the first statement of a compound statement.
6. PCR tests patches a) that replace a statement or b) that insert a non-if statements (i.e., generated by **genStmtsToAdd**()) before a statement $s$ and $s$ is the first statement of a compound statement.
7. PCR finally tests the remaining patches.

Intuitively, PCR prioritizes patches that contain conditionals. With abstract conditions that PCR later synthesizes, each such patch stands in for multiple potential concrete patches. PCR also prioritizes patches 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 patches.

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

## 3.3 Test and Condition Synthesizer

**Test Runtime:** PCR patches the application with each candidate patch and runs the patched application with all considered test scripts to determine whether the candidate patch is potentially valid or not. For patches that do not contain abstract conditions, this testing process is straightforward. If the compiled patched application passes all considered test scripts, PCR has found a plausible patch that it accepts.

Note that as an optimization, PCR tests the application with negative test scripts first. We observed that most invalid patches fail on at least one of the negative test scripts. PCR can skip the rest of testing process as soon as the patched program fails on any of the test scripts.

For candidate patches that contain abstract conditions, PCR applies the patch to the application and compiles the patched application with the PCR runtime library, which implements the function stub `abstract_cond()` for the abstract condition. The implementation of the `abstract_cond()` in the library is actually an algorithm that searches possible output combinations of the abstract condition that can pass test scripts.

Note that for each positive test script, always returning the same zero/one sequence as the execution of the original unpatched application will cause the application to produces the same correct results (if the application executes deterministically). For each negative test script, PCR runs the patched application multiple times. In the first run, the function `abstract_cond()` returns the same zero/one sequence as in the original unpatched execution. In each successive run, the PCR runtime library flips the return value of the last unflipped invocation of `abstract_code()` to search for a sequence of values that enable the application to produce the correct result. In the last run, the runtime library simply flips all return values.

In the current implementation, the PCR runtime tries 10 runs for each negative test script. If PCR is able to find a combination that passes each negative test script, the PCR test runtime will determine that this is a potentially valid patch.

If the test process succeeds, the PCR test runtime records the list of output value combinations of the abstract condi-

tion that make the application to pass each test script. PCR also records the value of all active local variables, global variables, and heap variables each time the function stub of the abstract condition in the test runtime is invoked. The test runtime passes this information to the condition synthesizer.

**Condition Synthesizer:** The PCR condition synthesizer operates similar to a program invariant detector [15]. Unlike an invariant detector, which aims to find invariants that are true at the specific program point in every execution, the PCR condition synthesizer aims to find condition expressions that match the return values from the test runtime. PCR considers the following integer or pointer type program values:

1. local variables,
2. global variables or heap variables that are used in the same code block of the abstract condition,
3. and subexpressions in the same statement of the abstract condition.

The condition synthesizer currently considers condition expressions that combine these values with equal signs, unequal signs, and constant values.

The condition synthesizer produces a sequence of concrete condition expressions. Each expression maximizes the number of matched return values from the test runtime. If there is a tie between two expressions, the synthesizer considers the expression with a simpler form first (e.g., PCR considers that `(!a)` simpler than `a != 3` or `a != b`).

PCR replaces the abstract condition in the patch with the produced concrete condition and runs all considered test scripts again with the new fix. If the patched application passes all of the test scripts, PCR outputs the patch to the user. Otherwise, the condition synthesizer will try again with another condition expression. In our current implementation, the synthesizer tries 20 condition expressions before it moves on to the next candidate patch.

## 3.4 Optimizations

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

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

**Patches for Code Duplicates:** Programs often contain duplicate or similar source code. This is often caused by the usage of macros or code copy-paste during application development. PCR detects such code clones in the source code. When PCR generates patches that modify one of the cloned code snippets, PCR also generate additional patches that propagate the modification to the other duplicates of the modified snippet.

## 3.5 Implementation

We have implemented PCR in C++ using the clang compiler front-end [1]. We choose clang because it has a AST tree interface that contains APIs that support source code rewrites. This enables PCR to generate a patched source code file without dramatically changing the overall structure of the source code. Existing program repair tools [21, 28, 33] often destroy the structure of the original source by inlining all header files and renamining all local variables in their generated patched source code. Preserving the existing source code structure helps developers understand and evaluate the patch and promotes the future maintainability of the application.

# 4. EXPERIMENTAL RESULTS

We evaluate PCR on the defects in the GenProg benchmark set, which contains 105 real world defects [21]. These defects are 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).

We address the following questions:

1. **Plausible Patches:** How many plausible patches can PCR generate for this benchmark set?
2. **Correct Patches:** How many correct patches can PCR generate for this benchmark set?
3. **Design Decisions:** How do the various PCR design decisions (condition synthesis, value replacement) affect the ability of PCR to generate plausible and correct patches?
4. **Previous Systems:** How does PCR compare with previous patch generation systems on this benchmark suite?

## 4.1 Methodology

**Collect and Reproduce the Defects:** For each of the seven benchmark applications, we collected (from the GenProg website [2]) the set of defects, test harnesses, test scripts, and test cases that the GenProg authors used in their experiments. We modified the test scripts and test harnesses to eliminate various errors [29]. For libtiff we implemented only partially automated patch validation, manually filtering the final generated patches to report only plausible patches [29].

For each defect, we collected the specific version of the application that corresponds to the defect and the corresponding reference patched version where this defect has been patched by the developer in the repository. We then reproduced each defect (except the fbc defects) in our experimental environment, Amazon EC2 Intel Xeon 2.6GHz Machines running Ubuntu-64bit server 14.04. The fbc application only runs in 32-bit environments, so we use a virtual machine with Intel Core 2.7Ghz running Ubuntu-32bit 14.04 for the fbc experiments.

| App | LoC | Tests | Defects | Plausible | | | | Correct | | | | Init Time | PCR Search Time | PCR(NoF) Search Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Gen Prog | AE | PCR | PCR (NoF) | Gen Prog | AE | PCR | PCR (NoF) | | | |
| libtiff | 77k | 78 | 24 | 3 | 5 | 5 | 5 | 0 | 0 | 1 | 1 | 2.4m | 54.0m | 71.4m |
| lighttpd | 62k | 295 | 9 | 5 | 4 | 5 | 3 | 0 | 0 | 0 | 0 | 7.2m | 144.1m | 182.3m |
| php | 1046k | 8471 | 44 | 5 | 7 | 18 | 14 | 1 | 2 | 8 | 7 | 13.7m | 156.2m | 186.2m |
| gmp | 145k | 146 | 2 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 7.5m | 128m | 374.5m |
| gzip | 491k | 12 | 5 | 1 | 2 | 2 | 2 | 0 | 0 | 1 | 1 | 4.2m | 33.5m | 29.5m |
| python | 407k | 35 | 11 | 1 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 31.1m | 237.7m | 163.0m |
| wireshark | 2814k | 63 | 7 | 1 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 58.8m | 32.2m | 40.3m |
| fbc | 97k | 773 | 3 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 8m | 15m | 53m |
| Total | | | 105 | 18 | 27 | 40 | 34 | 2 | 3 | 12 | 11 | | | |

**Table 1: Overview of PCR Patch Generation Results**

Each defect has a *positive test case set* on which both the version with the defect and the reference patched version of the application produce correct outputs and a *negative test case set* on which the version with the defect does not produce correct output. The goal is to generate a plausible patch so that the patched application produces correct outputs on all test cases in both of the two test case sets.

**Apply PCR:** For each defect, we ran PCR with a time limit of 12 hours. We terminate PCR when either 1) PCR successfully finds a patch that passes all of the test cases or 2) the time limit of 12 hours expires with no generated PCR patch.

GenProg and AE require the user of the system to identify a source code file to patch. This requirement reduces the size of the search space but eliminates the ability of these three systems to operate automatically without user input. PCR does not impose this limitation — it is capable of operating fully automatically across the entire source code base. To compare PCR with previous systems, we run PCR twice for each defect: once without specifying a source code file to patch, then again specifying the same source code file to patch as specified in the GenProg experiments.

**Inspect Patch Correctness:** For each defect, we manually inspected all of the patches that PCR generates. We consider a generated patch *correct* if 1) the patch completely eliminates the defect exposed by the negative test cases so that no input will be able to trigger the defect, and 2) the patch does not introduce any new defects.

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

We acknowledge that, in general, determining whether a specific patch 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 patches and defects that we consider in this paper. The correct behavior for all of the defects is clear, as is patch correctness and incorrectness.

**Compare with GenProg and AE Systems:** We compared the PCR patches with the patches that GenProg and AE generate. In this patch comparison, we leverage our previous plausibility and correctness analysis for the GenProg and AE patches [29]. The comparison is straightforward: PCR generates plausible patches for at least 13 more defects (40 for PCR vs. 18 for GenProg and 27 for AE) and correct patches for at least 9 more defects (12 for PCR vs. 2 for GenProg and 3 for AE).

We next inspected each correct patch that PCR generated and identified the corresponding PCR modification operator that PCR used to generate each patch. Our analysis of the patches indicates that the correct PCR patches for 10 out of the 12 defects are outside the GenProg and AE search spaces. PCR's novel modification operators enable the generation of these patches. See Section 4.4 for more details.

We next evaluated the condition synthesis technique by turning off the condition synthesis and rerunning PCR on these 12 defects. The resulting executions search the same space, but simply enumerate all conditions in the space (instead of using condition synthesis to accelerate the search). Our results show that turning off condition synthesis causes PCR to explore substantially more patches, in some cases over two orders of magnitude more patches. See Section 4.4 for more details.

## 4.2 Summary of Experimental Results

Table 1 summarizes our benchmark set and our experimental results. The first column "App" presents the name of the benchmark application. The second column "LoC" presents the size of the benchmark application measured in the number of source code lines. The third column "Tests" presents the number of developer supplied test cases that are used in the GenProg benchmark set and in our experiments. The fourth column "Defects" presents the number of defects we considered in our experiments.

The fifth column "Plausible GenProg" in Table 1 presents the number of defects for which GenProg successfully generated at least one plausible patch. The sixth column "Plausible AE" presents the number of defects for which AE successfully generated at least one plausible patch. Note that due to experimental error in the patch evaluation scripts, at least half of the originally reported patches from the GenProg and AE papers are implausible [29]. See our previous work on the analysis of GenProg and AE patches for details [29].

The seventh and eighth columns present the number of defects for which PCR generates a plausible patch. "PCR" corresponds to the runs where we specified the target source file to patch (as in the GenProg and AE runs), while "PCR(NoF)" corresponds to the runs where we do not provide PCR with a target source file to patch so that PCR executes fully automatically over the entire source code base of the application. PCR generates plausible patches for at least 13 more defects than GenProg and AE (40 for PCR v.s. 18 for GenProg and 27 for AE). Even with no specified source code file to patch (note that GenProg and AE require the user to provide this

| Defect | PCR | | | | PCR(No File Name Info) | | | | Gen Prog | AE | PCR Time | PCR (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-2330-2331 | 24919 | 6 | - | Gen | 21910 | 106 | - | Gen | Gen | Gen | 51m | 59m |
| 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-311323-311300 | 32009 | 122 | - | Gen | 211717 | 93 | - | Gen | No | No | 666m | 83m |
| 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-69783-69784 | 12691 | 67 | 67 | Correct | 24331 | 58 | 58 | Correct | Correct | Correct | 59m | 79m |
| 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 |

**Table 2: PCR Results for Each Generated Plausible Patch**

information), PCR is able to generate plausible patches for 34 defects.

The ninth column "Correct GenProg" presents the number of defects for which GenProg generates a correct patch. The GenProg result tar file available on the GenProg website [2] reports results from 10 different GenProg executions with different random number seeds. For some defects some of the generated patches are correct while others are not. We count the defect as patched correctly by GenProg if any of the generated patches for that defect are correct. The tenth column "Correct AE" presents the number of defects for which the generated AE patch is correct.

The eleventh and twelfth columns present the number of defects for which the generated PCR patch is correct. "PCR" corresponds to the runs where we provided PCR with the target source file to patch (as in the GenProg and AE runs), while "PCR(NoF)" corresponds to the runs where we do not provide PCR with a target source code file. This result highlights PCR's ability to generate correct patches. Our results show that PCR generates correct patches for at least 9 more defects than GenProg and AE (12 for PCR v.s. 2 for GenProg and 3 for AE). Even when the target source file is not specified, PCR still generates correct patches for 11 defects.

The thirteenth column "Init Time" in Table 1 presents the average time PCR spent to initialize the repair process, which includes compiling the whole application and running the error localizer. The fourteenth column "PCR Search Time" presents the average execution time of PCR on all defects of the application for which PCR can generate patches. The fourteenth column "PCR(NoF) Search Time" presents the average execution time for the runs where we do not specify source code file to patch. Our experimental results show that for those defects for which PCR generates a patch, PCR will generate the first patch in less than 2 hours on average.

## 4.3 PCR Patch Generation Results

PCR generates plausible patches for 40 defects in our benchmark sets. Table 2 presents detailed information for each defect for which PCR generates a patch. The first column "Defect" is in the form of "X-a-b", where "X" is the name of the application that contains the defect, "a" is the defective version, and "b" is the reference patched version. Columns 2-5 in Table 2 present results from the PCR runs where we specified the target source file to patch (as in the GenProg and AE runs). Columns 6-9 present results from

the PCR runs where we do not specify a source code file to patch.

**Search Space:** The second column and the sixth column "Search Space" in Table 2 present the total number of candidate patches in the PCR patch search space. The third column and the seventh column "Gen At" presents the total number of candidate patches that the PCR test runtime evaluates before PCR generates the patch.

For most defects, PCR generates patches after evaluating less than 1000 candidate patches. This is because 1) 28 out of 40 patches that PCR generates are related to branch statement conditions, 2) and the PCR condition synthesizer enables PCR to efficiently explore the search space of such patches.

The fourth column and the eighth column "Correct At" present the rank of the first correct patch in the PCR search space (if any). A "-" indicates that there is no correct patch in the search space. For 20 out of the total 40 defects, if the source code file to patch is specified and for 17 out of the total 34 defects if the source code file to patch is not specified, there is at least one correct patch inside the PCR search space.

Note that even if the correct patch is within the PCR search space, PCR may not generate this correct patch — the PCR search may time out before PCR encounters the correct patch, or PCR may encounter a plausible but incorrect patch before it encounters the correct patch. Stronger test suites with additional test cases may expose more defects in the candidate patches and (by eliminating otherwise plausible patches or more quickly eliminating implausible patches) enable PCR to find more correct patches.

**Comparison With GenProg and AE:** The fifth column and the ninth column"Result" presents for each defect whether the PCR patch is correct or not. The tenth column "GenProg" presents the status of the GenProg generated patch for each defect. The eleventh column "AE" presents the status of the AE generated patch for each defect. "Correct" in the column 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 PCR. For one defect, AE generates a correct patch when PCR generates a plausible but incorrect patch. For this defect, the PCR search space contains a correct patch, but the PCR search algorithm encounters the plausible but incorrect patch before it encounters the correct patch. For the remaining defects for which GenProg or AE generate a correct patch, so does PCR. PCR generates plausible patches for 22 more defects than GenProg and 13 more defects than AE. PCR generates correct patches for 10 more defects than GenProg and 9 more defects than AE.

## 4.4 PCR Design Choices

To evaluate the impact of various PCR design choices, we analyzed the 12 correct patches that PCR generates. Table 3 classifies the 12 correct patches that PCR generates. This classification highlights the challenges that PCR must overcome to generate these correct patches. The first column "Defect" presents the defect.

| Defect | Fix Type | Fix Evaluation (No Synthesis) |
|---|---|---|
| php-307562-307561 | Replace† | 5.3X |
| php-307846-307853 | Add Init† | 2.5X |
| php-308734-308761 | Add Guarded Control†‡ | 5.7X |
| php-309516-309535 | Add Init† | 3.1X |
| php-309579-309580 | Change Cond†‡ | 15.6X |
| php-309892-309910 | Delete | 50X |
| php-310991-310999 | Change Cond† | 324.4X* |
| php-311346-311348 | Redirect Branch† | 141.2X* |
| libtiff-ee2ce5-b5691a | Add Control†‡ | 8.9X* |
| gmp-13420-13421 | Replace†‡ | 5.1X* |
| gzip-a1d3d4-f17cbd | Copy and Replace†‡ | 11.7X |
| python-69783-69784 | Delete | 35.9X |

**Table 3: PCR results on each correct patch**

**Modification Operators:** The second column "Fix Type" presents the fix type of the correct patch for each defect. "Add Control" indicates that the generated patch inserts a control statement with no condition. "Add Guarded Control" indicates that the generated patch inserts a guarded control statement with a meaningful condition. "Replace" indicates that the generated patch modifies an existing statement using value replacement to replace an atom inside it. "Copy and Replace" indicates that the generated patch copies a statement from somewhere else in the application using value replacement to replace an atom in the statement. "Add Init" indicates that the generated patch inserts a memory initialization statement. "Delete" indicates that the generated patch 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 generated patch removes one branch of an if statement and redirects all executions to the other branch (by setting the condition in the if statement to true or false). "Change Cond" indicates that the generated patch changes a branch condition in a non-trivial way (unlike "Delete" and "Redirect Branch").

A "†" in the second column indicates that the PCR patch for this defect is outside the search space of GenProg and AE (for 10 out of the 12 defects, the PCR patch is outside the GenProg and AE search space). A "‡" in the column indicates that the PCR patch for this defect is outside the search space of PAR with the eight templates from the PAR paper [19] For 5 out of the 12 defects, the PCR patch is outside the PAR search space.

For php-307846-307853, php-308734-308761, php-309516-309535, and libtiff-ee2ce5b7-b5691a5a, the PCR patches insert control statements or initialization statements that do not appear elsewhere in the source file. For php-307562-307561, gmp-13420-13421, and gzip-a1d3d4-f17cbd, the PCR patches change expressions inside the copied or replaced statements. For php-309579-309580, php-310991-310999, and php-311346-311348, the PCR generated patches change the branch condition in a way which is not equivalent to deleting the whole statement. These patches are therefore outside the search space of GenProg and AE, which only copy and remove statements.

The PCR correct patches for php-308734-308761 (add "break"), libtiff-ee2ce5b7-b5691a5a (add "goto"), and gzip-a1d3d4-f17cbd (add "assignment") are outside the PAR search space because no template in PAR is able to add goto, break, or assignment statements into the source code. The PCR correct patch for gmp-13420-13421 is also outside the PAR search

space, because the patch replaces a subexpression inside an assignment statement and no template in PAR supports such an operation. The PCR correct patch for php-309579-309580 is outside the PAR search space because the patch 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.

**Condition Synthesis:** The third column "Fix Evaluation (No Synthesis)" presents the increased number of candidate patches PCR needs to consider before the correct patch if PCR turns off condition synthesis and naively enumerates all possible conditions. A "*" in the column indicates that PCR would be unable to generate the patch within 12 hours if the condition synthesis is turned off. Our results show that condition synthesis significantly reduces the number of candidate patches that PCR needs to validate, in some cases by a factor of over two orders of magnitude. Without condition synthesis, PCR would be unable to generate patches for 4 out of the 12 defects within 12 hours.

# 5. RELATED WORK

**ClearView:** ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [27]. 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 [27]. 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 [27].

PCR differs from ClearView in both its goal and its technique. PCR targets software defects that can be exposed by supplied negative test cases, which are not limited to just security vulnerabilities. PCR 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 [34, 21] 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 [28] changes the GenProg algorithm to use random modification instead. AE [33] uses a deterministic patch search algorithm and uses program equivalence relations to prune equivalent patches during testing.

Previous work [29] 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 experimental error in the patch validiation. Further semantic analysis [29] 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. In fact, an implemented naive patch generation system that only eliminates functionality can, on the same benchmark set, produce plausible patches and correct patches for at least as many defects [29].

Unlike GenProg [21], RSRepair [28], and AE [33], which only copy statements from elsewhere in the program, PCR defines a set of novel modification operators that enables PCR to operate on a search space which contains meaningful and useful patches. PCR then uses its condition synthesis technique to efficiently explore the search space. Our results show that PCR significantly outperforms GenProg and AE in the same benchmark set. The majority of the correct patches PCR generates in our experiments are outside the search space of GenProg, RSRepair, and AE.

**SemFix and MintHint:** SemFix [25] and MintHint [18] replace the potential faulty expression with a symbolic value and use symbolic execution techniques [8] and SMT solvers to find a replacement expression that can enable 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 [9] presents 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, PCR uses more sophisticated and effective modification operators and search algorithms. In fact, none of the 12 correct patches that PCR generates are within the Debroy and Wong search space.

**NOPOL:** NOPOL [10] 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. PCR differs from NOPOL in the following ways. 1) To make the search space tractable, NOPOL assumes that a branch statement in the fixed program will always take the same direction during the execution. This assumption is often not true when the branch condition is executed multiple times for a test case (php-308734-308761 and php-310991-310999). In this case NOPOL will fail to generate a patch. The PCR condition synthesis algorithm, of course, does not have this limitation. 2) NOPOL focuses only on patches that change conditions, while PCR can generate patches 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). 3) NOPOL was evaluated on two small Java programs (each with less than 5000 lines of code) and two artificial examples in [10], while we evaluate PCR on 105 real world defects in seven C applications with more than one million lines in total.

**Fix Safety-Policy Violation:** Weimer [32] proposed 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, PCR 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 PCR, 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 [26] and Gopinath et al. [17], which applies domain specific techniques to repair DOM-related faults in JavaScript and selection statements in database programs respectively. AutoFix-E [31] repairs program faults with human-supplied specifications called contracts. PCR differs from all of this previous research in that it focuses on generating fixes for general purpose applications without human-supplied specifications.

## 5.1 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 [30] 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 [20] 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 [22] 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 [13] 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 errors [12]. 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 [11].

**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 [14].

## 5.2 Discussion

These results highlight the importance of the specific modification operators and search algorithm in obtaining a successful patch generation system. General concepts such as copying statements [21, 28, 33] and targeting conditionals [9], by themselves, fail to deliver sufficiently rich and tractable patch search spaces with useful patches. Consider, in contrast, the success of the more sophisticated ClearView invariant enforcement approach and the PCR combination of compound mutations and condition synthesis. The success of these two more sophisticated approaches indicates that it is the tight interplay between the operations that generate the patch search space and the algorithms that search that space that is important. It is this sophisticated interplay, and not general concepts such as copying statements or focusing on conditionals, that makes or breaks a given automatic patch generation system.

# 6. CONCLUSION

The difficulty of generating a patch search space rich enough to correct defects while still supporting a search algorithm efficient enough to find the patches in an acceptable amount of time has significantly limited the ability of previous automatic patch generation systems [21, 33]. PCR's novel set of modification operators and efficient search algorithm based on condition synthesis highlight how a synergistic combination of modification operators and search algorithm can enable successful automatic patch generation.

# 7. 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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.