An Analysis of Patch Plausibility and Correctness of Generate-And-Validate Patch Generation System

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard
MIT CSAIL
Generate-And-Validate Patch Generation Systems

Buggy Program

Candidate patch space

Test suite of test cases

BUG FIXED
Generate-And-Validate Patch Generation Systems

- **GenProg** – Genetic Programming

- **AE** – Adaptive Search

- **RSRepair** – Random Search
All of them report **impressive** results

<table>
<thead>
<tr>
<th></th>
<th>GenProg</th>
<th>AE</th>
<th>RSRepair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark Defects</td>
<td>105</td>
<td>105</td>
<td>24</td>
</tr>
<tr>
<td>Reported Fixed Defects</td>
<td>55</td>
<td>54</td>
<td>24</td>
</tr>
</tbody>
</table>

- Patches generated by these systems are different from human written patch
- No systematic analysis
We analyze the reported patches for these systems

Plausible?
Produce correct outputs for all test cases in the test suite

*All generated patches should be plausible*
## Plausibility

<table>
<thead>
<tr>
<th></th>
<th>GenProg</th>
<th>AE</th>
<th>RSRepair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark Defects</td>
<td>105</td>
<td>105</td>
<td>24</td>
</tr>
<tr>
<td>Reported Fixed Defects</td>
<td>55</td>
<td>54</td>
<td>24</td>
</tr>
<tr>
<td>Defects With Plausible Patches</td>
<td>18</td>
<td>27</td>
<td>10</td>
</tr>
</tbody>
</table>

- **Reason - Weak Proxy**
  - Patch evaluation does not check for correct output
  - `php, libtiff` – check exit code, *not output*
  - Accepted `php` patch: `main(){ exit(0); }`
Analysis of the reported patches for these systems

**Plausible?**
Produce correct outputs for all test cases in the test suite

**Majority of the patches are not plausible**

**Correct?**
Eliminate the defect

*Passing test suite != correctness*
Correctness

<table>
<thead>
<tr>
<th></th>
<th>GenProg</th>
<th>AE</th>
<th>RSRepair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark Defects</td>
<td>105</td>
<td>105</td>
<td>24</td>
</tr>
<tr>
<td>Reported Fixed Defects</td>
<td>55</td>
<td>54</td>
<td>24</td>
</tr>
<tr>
<td>Defects With Plausible Patches</td>
<td>18</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>Defects With Correct Patches</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Developed new test cases that expose defects for all plausible but incorrect patches.
GenProg Statistics

2 Correct
16 Plausible but Incorrect
37 Implausible

Stronger Test Suites?
Will GenProg generate correct patches given new test cases that eliminate incorrect patches?

Fixed Test Scripts?
Will GenProg generate plausible patches given fixed patch evaluation scripts?
Analysis of the reported patches for these systems

**Plausible?**
Produce correct outputs for all test cases in the test suite

Majority of the patches are not plausible

**Correct?**
Eliminate the defect

The overwhelming majority of the patches are not correct

**Do stronger test suites help?**

Rerun GenProg with fixed patch evaluation scripts and new test cases that eliminate incorrect patches
Reexecution of GenProg on Remaining 103 Defects

First Reexecution
Fixed patch evaluation
New test cases
Patches for 2 defects

Second Reexecution
2 additional test cases
Patches for 0 defects

Why?

• Developer patches are not in GenProg search space
• GenProg search space may not contain any correct patch for these 103 defects
• May need richer search space to generate correct patches
Bottom Line For GenProg

• Rerun GenProg with
  – Fixed test scripts
  – Stronger test suites

• GenProg generates patches for only 2 of 105 defects (both patches are correct)
Examples of Correct GenProg Patch (1/2)

Developer

```c
if (y < 1000) {
    PyObject *accept = PyDict_GetItemString(moddict, "accept2day");
    if (accept != NULL) {
        int acceptval = PyObject_IsTrue(accept);
        if (acceptval == -1) return 0;
        if (acceptval) {
            if (0 <= y && y < 69)
                y += 2000;
            else if (69 <= y && y < 100)
                y += 1900;
            else {
                PyErr_SetString(PyExc_ValueError,
                                "year out of range");
                return 0;
            }
        }
    }
    PyErr_WarnEx(PyExc_DeprecationWarning,
                 "Century info guessed for a 2-digit year.
                 ")
    return 0;

else
    return 0;
}
p->tm_year = y - 1900;
p->tm_mon--; 
p->tm_wday = (p->tm_wday + 1) % 7;
```

GenProg

```c
if (y < 1000) {
    PyObject *accept = PyDict_GetItemString(moddict, "accept2day");
    accept = tmp_0;
    if ((unsigned int )accept != (unsigned int )((void *)0)) {
        tmp_1 = PyObject_IsTrue(accept);
        acceptval = tmp_1;
    }
    if (acceptval == -1) {
        return (0);
    } else {
        ...
    }
    if (acceptval) {
        if (0 <= y) {
            if (y < 69) {
                y += 2000;
            } else {
                goto _L;
            }
        } else {
            _L: /\ CIL Label */
            if (69 <= y) {
                y += 1900;
            } else {
                PyErr_SetString(PyExc_ValueError,
                                "year out of range");
                return (0);
            }
        }
        PyErr_SetString(PyExc_DeprecationWarning,
                        "Century info guessed for a 2-digit year.");
        return (0);
    } else {
        PyErr_SetString(PyExc_ValueError,
                        "year out of range");
        return (0);
    }
    if (tmp_2 != 0) {
        return (0);
    } else {
        ...
    }
}
p->tm_year = y - 1900;
p->tm_mon--; 
p->tm_wday = (p->tm_wday + 1) % 7;
```
Examples of Correct GenProg Patch (2/2)

Developer

```
if (offset >= s1_len) {
    php_error_docref0((char const *)(void *)
    "The start position cannot exceed length of string
    RETURN_FALSE;
}

if (len > s1_len - offset) {
    len = s1_len - offset;
}
```

GenProg

```
if (offset >= (long)s1_len) {
    php_error_docref0((char const *)(void *)
    "The start position cannot exceed length of string
    RETURN_FALSE;

while (1) {
    __z__1 = return_value;
    __z__1->value.lval = 0L;
    __z__1->type = (unsigned char)3;
    break;
}

return;
} else {
}
```

```
if (len > (long)s1_len - offset) {
    len = (long)s1_len - offset;
} else {

    if (len) {
        tmp__1 = len;
    } else {
        if ((long)s2_len > (long)s1_len - offset
            tmp__0 = (long)s2_len;
        } else {
            tmp__0 = (long)s1_len - offset;
        }
    }

    tmp__1 = tmp__0;
}
```

```
cmp_len = (unsigned int)tmp__1;
```
All Correct Patches Simply Delete Code
Semantic Analysis

• Analyze all the plausible patches
• Determine if patch is equivalent to single functionality deletion modification

• Results
  – GenProg: 14/18
  – AE: 22/27
  – RSRepair: 8/10
We found a common scenario

• A negative test case exposes the defect
  – Feature is otherwise unexercised
  – The patch simply deletes the functionality
    • Introduces new security vulnerabilities
      (buffer overflows)
    • Disables critical functionality
      (gzip cannot decompress non-zero files)

• Weak test suites
  – May be appropriate for human developers
  – *May not be* appropriate for automatic patch generation systems (at least not by themselves)
If all these patches simply delete functionality

Why not build a patch generation system that ONLY deletes functionality?
We present Kali

- Automatic patch generation system

- Consider the search space that consists of only patches that remove functionality
### Experimental Results of Kali

<table>
<thead>
<tr>
<th></th>
<th>GenProg</th>
<th>AE</th>
<th>RSRepair</th>
<th>Kali</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark Defects</td>
<td>105</td>
<td>105</td>
<td>24</td>
<td>105</td>
</tr>
<tr>
<td>Reported Fixed Defects</td>
<td>55</td>
<td>54</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Defects With Plausible Patches</td>
<td>18</td>
<td>27</td>
<td>10</td>
<td>27</td>
</tr>
<tr>
<td>Defects With Correct Patches</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- Kali is as good as previous systems
  - Much simpler
  - Not need to know the source code file to repair
- Can pinpoint the defective code
- Can provide insight into important defect characteristics.
# Experimental Results of Kali

<table>
<thead>
<tr>
<th></th>
<th>GenProg</th>
<th>AE</th>
<th>RSRepair</th>
<th>Kali</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Benchmark Defects</strong></td>
<td>105</td>
<td>105</td>
<td>24</td>
<td>105</td>
</tr>
<tr>
<td><strong>Reported Fixed Defects</strong></td>
<td>55</td>
<td>54</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td><strong>Defects With Plausible Patches</strong></td>
<td>18</td>
<td>27</td>
<td>10</td>
<td>27</td>
</tr>
<tr>
<td><strong>Defects With Correct Patches</strong></td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Is Automatic Patch Generation A Total Failure?

NO!
Path To Success

• Richer search spaces
• More efficient search algorithms
• Incorporate additional sources of information
  – Correct code from other applications
  – Learned characteristics of human patches
  – Learned invariants
  – Specifications
Promising directions

• Learn invariant from correct execution
  • Patches security vulnerabilities in 9 of 10 defects
  • At least 4 patches are correct

• Solvers
  • NOPOL: F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. *Automatic repair of buggy if conditions and missing preconditions with smt*. CSTVA 2014
Promising directions

• Specifications
  • Etienne Kneuss, Manos Koukoutos and Viktor Kuncak. Deductive Program Repair. CAV 2015

• Correctness evaluation

• Code from another application
### Promising Directions

<table>
<thead>
<tr>
<th>Benchmark Defects</th>
<th>105</th>
<th>105</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defects With Plausible Patches</td>
<td>41</td>
<td>42</td>
</tr>
<tr>
<td>Defects With Correct Patches</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>


Take Aways

• Facts about GenProg/AE/RSRepair
  – These systems fix 2/3/2 of 105 bugs (not 55/54/24)
  – Errors in test scripts and weak test suites
  – Fixed test scripts and stronger test suites do not help

• Paths to success
  – Richer search spaces
  – More efficient search algorithms
  – Incorporate additional sources of information
    • Correct code from other applications (CodePhage)
    • Learned characteristics of human patches (Prophet)
    • Learned invariants (ClearView)
    • Specifications (AutoFixE, Deductive Repair)
Summary

• Evaluation of GenProg, AE and RSRepair
  – Incorrect results
  – Equivalent to functionality elimination
  – Stronger test suites do not help

• Kali
  – Functionality elimination system
  – Help developer better understand the bug
Path to Success for the automatic patch generation systems

• Richer search spaces
• More efficient search algorithms
• Incorporate additional sources of information
  – Correct code from other applications
  – Learned characteristics of human patches
  – Learned invariants
• Better patch evaluation
  – Correctness
  – Understand the negative effects

Questions?