KATCH: High-Coverage Testing of Software Patches

Paul Marinescu and Cristian Cadar
Imperial College London
Intro

• Manual testing is hard

• Supplement it with automatic testing

• We focus on testing software changes
repository -> KATCH -> coverage

```c
if (k > n) {
  if (k >= n) {
    k = 4;
  }...
```
Example: Testing diffutils

$ ls diffutils
config.sh  build.sh  regression-test.sh

$ cat config.sh
REPO="git://git.savannah.gnu.org/diffutils.git"
DIFFTARGETS="src  lib"
PROGRAMS="src/diff src/diff3 src/sdiff src/cmp"
LIBS="-lrt"

$ katch diffutils 0 100
High-Level Idea

• Synthesize inputs which execute the patch code

• Given a program location (e.g. file name, line number), synthesize an input which executes that location
High-Level Approach

• Concrete/Symbolic execution mix + heuristics

• Seeded with existing inputs from the regression test suites
System Overview

KATCH

Program, Patch, Test suite

Patch Preprocessing → Input Selection

Symbolic Execution

Greedy Exploration
Informed Path Regeneration
Definition Switching

New program inputs
void log(char input) {
    int file = open("access.log", ...);
+    if (input >= ' ' &&
+        input <= '~') {
+        // printable characters
+        write(file, &input, 1);
+    } else {
+        char escinput;
+        escinput = escape(input);
+        write(file, &escinput, 1);
+    }
    close(file);
}
Input Selection

• Rank existing inputs based on how ‘easy’ it is to change them to execute the patch

• Optimization

• Lightweight
Input Selection

Input A – distance 4

Input B – distance 2

Example control-flow graph
Concrete/Symbolic Execution

• Iterative refinement of the initial input

• Get ‘closer’ to the target at each iteration

• Symbolic execution + path selection heuristics
void log(char input) {
    int file = open("access.log", ...);
    if (input >= ‘ ’ &&
        input <= ‘~’) {
        // printable characters
        write(file, &input, 1);
        + } else {
        +   char escinput = escape(input);
        +   write(file, &escinput, 1);
        + }
    close(file);
}
Greedy Exploration Step

1. Greedy step: choose the symbolic branch whose unexplored side is closest to the patch.
2. Explore this side!

Available input: “t” (or any printable char)

```c
void log(char input) {
    int file = open("access.log", ...);
    if (input >= ' ' && input <= '~') {
        // printable characters
        write(file, &input, 1);
    } else {
        + char escinput = escape(input);
        + write(file, &escinput, 1);
    }
}
```
Informed Path Regeneration

```c
void log(char input) {
    if (input >= ' ' && input <= '~') {
        . . .
    } else {
        . . .
    }
}

if (0 == strcmp(request, "GET")) {
    . . .
    for (char* p = request; *p; p++)
        log(*p);
}
```

Available input: “GET”

Greedy step fails!

1. Backtrack to the symbolic branch that disallows this side to be executed
2. Explore the other side of that branch

request[2] ≠ ‘T’
enum escape_t escape;
void log(char input) {
    if (escape == ESCAPE_ALL) {
        . . .
    }
}

opt = getopt_long(argc, argv, ...);
switch (opt) {
    case 'a': escape = ESCAPE_SPACE;
        break;
    case 'b': escape = ESCAPE_ALL;
        . . .
    log(. . .);

Available test: opt = ‘a’

Patch guarded by concrete branch

Backtracking step fails!

1. Find all reaching definitions for the variables involved and try to cover another one
2. Favors definitions that can be statically shown to satisfy target, or unexecuted definitions
Evaluation

Added/modified executable basic blocks

- findutils
- diffutils
- binutils

114 executable patches
1362 targets
Coverage Improvement

- findutils: 60% covered by test suite, 40% covered by KATCH, 0% not covered
- diffutils: 70% covered by test suite, 30% covered by KATCH, 0% not covered
- binutils: 0% covered by test suite, 0% covered by KATCH, 100% not covered
<table>
<thead>
<tr>
<th>Bugs Found</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>15 Crash Bugs</strong></td>
</tr>
<tr>
<td>6 bugs in patch code</td>
</tr>
<tr>
<td>5 bugs close to patch code</td>
</tr>
<tr>
<td>4 bugs unknown causal relation</td>
</tr>
</tbody>
</table>
Bugs Found

- Already fixed in the last version: 2
- Reported and pending: 1
- Reported and fixed: 12
Automatic Patch Testing

Practical autonomous testing system

Coverage improvement and bug finding

Short artifact* presentation on Friday
http://srg.doc.ic.ac.uk/projects/katch/

*Successfully evaluated by the ESEC/FSE artifact evaluation committee
Selected Related Work

• Directed Test Suite Augmentation (APSEC’09, FSE’10)

• Directed Symbolic Execution (SAS’11)

• Differential Symbolic Execution (FSE’08)

• Directed Incremental Symbolic Execution (PLDI’11)
# Heuristic Contribution

<table>
<thead>
<tr>
<th>Suite</th>
<th>Greedy</th>
<th>Greedy+IPR</th>
<th>Greedy+DS</th>
<th>KATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>findutils</td>
<td>74</td>
<td>85</td>
<td>78</td>
<td>85</td>
</tr>
<tr>
<td>diffutils</td>
<td>25</td>
<td>29</td>
<td>49</td>
<td>63</td>
</tr>
<tr>
<td>binutils</td>
<td>70</td>
<td>121</td>
<td>76</td>
<td>135</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>169</strong></td>
<td><strong>235</strong></td>
<td><strong>203</strong></td>
<td><strong>283</strong></td>
</tr>
</tbody>
</table>

IPR = Informed Path Regeneration  
DS = Definition Switching