Automated Program Repair

Abhik Roychoudhury
Professor (Provost’s Chair)
National University of Singapore
Comprehensive university with Science, Arts, Engineering, medical, Law, Business, Public Policy, Music, **Computing**, …

Public university 30K undergraduate students, 10K+ graduate students overall. 2500+ faculty members overall, 100+ in Computing (two departments CS and IS).

[http://www.nus.edu.sg/about#corporate-information](http://www.nus.edu.sg/about#corporate-information)
Viewpoint

Search Space Exploration
Avoid over-fitting

Symbolic Execution

KAIST Invited Colloquium 2020

Himalayas 2019
Trustworthy SW

Space of Problems
(Search Problems?)

• Fuzz Testing
  – Feed semi-random inputs to find hangs and crashes

• Continuous fuzzing
  – Incrementally find new “problems” in software

• Crash reproduction
  – Re-construct a reported crash, crashing input not included due to privacy

• Reaching nooks and corners

• Localizing reported observable errors

• Patching reported errors from input-output examples
Trustworthy SW Search Problems

• Random Search
  – Less systematic
  – Easy set-up, execute up to a time budget
  – Use objective function to steer search.

• Symbolic Execution
  – Systematic
  – More involved set-up, solver calls.
  – Use logical formula to steer search.
Use of Random Search - Fuzzing

Input: Seed Inputs $S$

1: $T_X = \emptyset$
2: $T = S$
3: if $T = \emptyset$ then
4: add empty file to $T$
5: end if
6: repeat
7: $t = \text{chooseNext}(T)$
8: $p = \text{assignEnergy}(t)$
9: for $i$ from 1 to $p$ do
10: $t_0 = \text{mutate_input}(t)$
11: if $t_0$ crashes then
12: add $t_0$ to $T_X$
13: else if $\text{isInteresting}(t_0)$ then
14: add $t_0$ to $T$
15: end if
16: end for
17: until timeout reached or abort-signal

Output: Crashing Inputs $T_X$
int \textbf{test\_me}(int \text{Climb}, int \text{Up})\{ \\
\hspace{1em} \text{int sep, upward; } \\
\hspace{1em} \text{if (Climb} > 0)\{ \\
\hspace{2em} \text{sep = Up;} \\
\hspace{1em} \text{else } \{ \text{sep = add100(Up);} \} \\
\hspace{1em} \text{if (sep} > 150)\{ \\
\hspace{2em} \text{upward = 1;} \\
\hspace{1em} \text{else } \{ \text{upward = 0;} \} \\
\hspace{1em} \text{if (upward} < 0)\{ \\
\hspace{2em} \text{abort;} \\
\hspace{1em} \text{else return upward; } \\
\} \\
\} \\
}
int test_me(int Climb, int Up) {
    int sep, upward;
    if (Climb > 0) {
        sep = Up;
    } else {
        sep = add100(Up);
    }
    if (sep > 150) {
        upward = 1;
    } else {
        upward = 0;
    }
    if (upward < 0) {
        abort;
    } else {
        return upward;
    }
}
Fuzzing vs. Symbolic Execution

Bug Finding

- Concolic execution: supporting *real* executions [Directed Automated Random Testing]

- Symbolic execution tree construction e.g. KLEE [Modeling system environment]

- Grey-box fuzz testing for systematic path exploration inspired by concolic execution

AFLFast
Fuzzing vs. Symbolic Execution

Reachability Analysis

Reachability of a location in the program

- Traverse the symbolic execution tree using search strategies e.g. KATCH

Encode it as an optimization problem inside the genetic search of grey-box fuzzing AFLGo

\[
\varphi_1 = (x>y) \land (x+y>10) \\
\varphi_2 = \neg (x>y) \land (x+y>10)
\]
In the absence of formal specifications, analyze the buggy program and its artifacts such as execution traces via various heuristics to glean a specification about how it can pass tests and what could have gone wrong!

**Specification Inference**
(application: self-healing)

• “Behind every large program there is a **small program** waiting to get out”
• C.A.R. Tony Hoare

• Behind every large program there is an **algorithm** waiting to get out
• Leslie Lamport
Program Repair

**Buggy Program**

**Tests**

**Program Repair**

**Patched Program**

**Candidate patches**

Transformation schemas:

\[
\begin{align*}
    x &:= y + 1; \rightarrow x := y - 1; \\
    stmt; \rightarrow \text{if } (x > 0) \text{ stmt}; \\
    \ldots
\end{align*}
\]

**Plausible patch**

(passes all tests)
Repair: Why?

- Maintaining Legacy Software
- Debugging Aid
- Education, Grading in MooCs
- Security Patches
- Self-healing systems, Drones
Search

- Applicability
- Scalability
- Over-fitting

Large program?
Large search space?
Over-fitting
Example

```c
1 int triangle(int a, int b, int c) {
2     if (a <= 0 || b <= 0 || c <= 0)
3         return INVALID;
4     if (a == b && b == c)
5         return EQUILATERAL;
6     if (a == b || b != c) // bug!
7         return ISOSCELES;
8     return SCALENE;
9 }
```

Correct fix
(a == b || b== c || a == c)

Traverse all mutations of line 6 ??

Hard to generate fix since (a ==c) or (c ==a) never appear anywhere else in the program!

<table>
<thead>
<tr>
<th>Test id</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>oracle</th>
<th>Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>INVALID</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>EQUILATERAL</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>ISOSCELES</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>ISOSCELES</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>ISOSCELES</td>
<td>✗</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>SCALANE</td>
<td>✗</td>
</tr>
</tbody>
</table>
# Example

```c
1 int triangle(int a, int b, int c) {
2     if (a <= 0 || b <= 0 || c <= 0)
3         return INVALID;
4     if (a == b && b == c)
5         return EQUILATERAL;
6     if (a == b || b != c) // bug!
7         return ISOSCELES;
8     return SCALENE;
9 }
```

<table>
<thead>
<tr>
<th>Test id</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>oracle</th>
<th>Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>INVALID</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>EQUILATERAL</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>ISOSCELES</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>ISOSCELES</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>ISOSCELES</td>
<td>✗</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>SCALENE</td>
<td>✗</td>
</tr>
</tbody>
</table>

Correct fix

\[(a == b \lor b == c \lor a == c)\]

Automatically generate the constraint

\[f(2,2,3) \land f(2,3,2) \land f(3,2,c) \land \neg f(2,3,4)\]

Solution

\[f(ab,c) = (a == b \lor b == c \lor a == c)\]
Comparison

1. Where to fix, which line?
2. Generate patches in the candidate line
3. Validate the candidate patches against correctness criterion.

Syntax-based Schematic
for e in Search-space{
  Validate e against Tests
}

Semantics-based Schematic
for t in Tests {
  generate repair constraint $\Psi_t$
}
Synthesize e from $\wedge_t \Psi_t$

1. Where to fix, which line(s)?
2. What values should be returned by those lines, e.g. $\langle \text{inp} == 1, \text{ret} == 0 \rangle$
3. What are the expressions which will return such values?
**Specification Inference**

Program

Concrete Execution

Concrete values

Output: Value-set or Constraint

Symbolic execution

Oracle (expected output)

Test input t

\[ \text{var} = f(\text{live-vars}) \triangleq X \]

Concrete Execution

Program

Concrete values

Output: Value-set or Constraint

Symbolic execution

Oracle (expected output)

Test input t

\[ \bigvee_{j \in \text{Paths}} (pc_j \land \text{out}_j \equiv \text{expected_out}(t)) \land \text{f}(t) \equiv X \]

Repair constraint

[ICSE13]
Example

```c
1 int is_upward( int inhibit, int up_sep, int down_sep){
2     int bias;
3     if (inhibit)
4         bias = down_sep; // bias= up_sep + 100
5     else bias = up_sep ;
6     if (bias > down_sep)
7         return 1;
8     else return 0;
9 }
```

<table>
<thead>
<tr>
<th>inhibit</th>
<th>up_sep</th>
<th>down_sep</th>
<th>Observed o/p</th>
<th>Oracle</th>
<th>Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>110</td>
<td>0</td>
<td>1</td>
<td>✗</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>0</td>
<td>1</td>
<td>✗</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
</tbody>
</table>
Example

```c
1 int is_upward(int inhibit, int up_sep, int down_sep) {
2     int bias;
3     if (inhibit)
4         bias = f(inhibit, up_sep, down_sep)  // X
5     else bias = up_sep;
6     if (bias > down_sep)
7         return 1;
8     else return 0;
9 }
```

Symbolic Execution

\[ \forall (pc_j \land out_j == \text{expected_out}(t)) \]
\[ j \in \text{Paths} \] \[ \land \]
\[ f(t) == X \]

\[ \{ (X > 110 \land 1 == 1) \]
\[ \lor (X \leq 110 \land 0 == 1) \]
\[ \}\] \[ \land \]
\[ f(1, 11, 110) == X \]
Example

```c
1 int is_howard( int inhibit, int up_sep, int down_sep ){
2     int bias;
3     if (inhibit)
4         bias = f(inhibit, up_sep, down_sep)
5     else bias = up_sep ;
6     if (bias < down_sep)
7         return 1 ;
8     else return 0 ;
9 }
```

$\forall \{ \text{pc}_j \land \text{out}_j == \text{expected_out}(t) \} \quad j \in \text{Paths}$

$\land$

$f(t) == X$

Repair constraint

Symbolic Execution

$f(1,11,110) > 110$
Example

- Accumulated constraints
  - $f(1, 11, 110) > 110$
  - $f(1, 0, 100) \leq 100$
  - ...

- Find a $f$ satisfying this constraint
  - By fixing the set of operators appearing in $f$

- Candidate methods
  - Search over the space of expressions
  - Program synthesis with fixed set of operators
    - Can also be achieved by second-order constraint solving

- Generated fix
  - $f(inhibit, up\_sep, down\_sep) = up\_sep + 100$
Simplified Workflow, but

Over-fitting
Scalability
Applicability

Tests → Debugging → DSE → Synthesis

MaxSMT solver

Conjure a function which represents minimal change to buggy program.

[ICSE15]

KAIST Invited Colloquium 2020
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>#Programs</th>
<th>Equivalent</th>
<th>Same Loc.</th>
<th>Diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SemFix</td>
<td>44</td>
<td>17%</td>
<td>46%</td>
<td>6.36</td>
</tr>
<tr>
<td>[ICSE13]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DirectFix</td>
<td>44</td>
<td>53%</td>
<td>95%</td>
<td>2.31</td>
</tr>
<tr>
<td>[ICSE15]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test cases: all possible orderings of \(x, y, z\)

### Code snippets

SemFix:
```java
if (x > y)
    if (x > z)
        out = 10;
    else
        out = 20;
else
    out = 30;
return ((x==y)? ((x==z)?10:20): out);
```

DirectFix:
```java
if (x >= y)
    if (x >= z)
        out = 10;
    else
        out = 20;
else
    out = 30;
return out;
```
Workflows

KAIST Invited Colloquium 2020

Conjure a function which represents minimal change to buggy program.

Concise Semantics Signature

Tests → Debugging → DSE → Synthesis

MaxSMT solver

Over-fitting → Scalability

Applicability
**Angelix**

**Angeonic forest:** Patch synthesis specification based on

**Angeonic values**  \{Symbolic Variable name, Constant, State\} Paths, Tests

---

*KAIST Invited Colloquium 2020*
Repair Constraint

- **SemFix work (ICSE 2013)**
  - Example: for an identified expression $e$ to be fixed
    - $[ X > 0 ] \land f(t) == X$ for each test $t$

- **DirectFix work (ICSE 2015)**
  - **Whole Program** as repair constraint
  - Use the principle of minimality to synthesize a minimal patch.

- **Angelix work (ICSE 2016)**
  - Example: for identified expressions $e_1, e_2, \ldots$ to be fixed
    - $[ (X == 1) \lor (X == 2) \lor (X == 3)] \land f(t) == X$ for each test $t$
    - $[ (X == 1 \land Y == 1) \lor (X == 2 \land Y == 2)] \land f(t) == X \land g(t) == Y$ for each test $t$. 
Scalability, Quality

Average time == 32 minutes

<table>
<thead>
<tr>
<th>Subject</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wireshark</td>
<td>2814K</td>
</tr>
<tr>
<td>php</td>
<td>1046K</td>
</tr>
<tr>
<td>gzip</td>
<td>491K</td>
</tr>
<tr>
<td>gmp</td>
<td>145K</td>
</tr>
<tr>
<td>libtiff</td>
<td>77K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Repair tool</th>
<th>Functionality Deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angelix</td>
<td>21%</td>
</tr>
<tr>
<td>SPR</td>
<td>42%</td>
</tr>
</tbody>
</table>

(a) The buggy part of the Heartbleed-vulnerable OpenSSL

(b) A fix generated automatically

(c) The developer-provided repair

```
1  if (hbtype == TLS1 HB REQUEST) {
2     ...  
3     memcpy(bp, pl, payload);
4     ...  
5  }

1  if (hbtype == TLS1 HB REQUEST
2     && payload + 18 < s->s3->rrec.length) {
3         
4     }
```

```
1  if (1 + 2 + payload + 16 > s->s3->rrec.length)
2     return 0;
3     ...  
4  if (hbtype == TLS1 HB REQUEST) {
5     ...  
6  }
7  else if (hbtype == TLS1 HB RESPONSE) {
8     ...  
9  }
10  return 0;
```
Experience of others

• “The core technique in Angelix using symbolic execution and program synthesis works well”.

• “It can potentially suffer from poor fault localization”.

• “With better fault localization, the patch synthesis seems hard to improve in effectiveness”
  – Can still be improved in terms of efficiency

[ICSE16 and its usage]

• [Anecdotal comments only from user groups]
**Specification Inference**

- **Two approaches**
  - Get property of function $f$ via symbolic execution, and synthesize a function $f$ satisfying these properties.
  - *Directly solve for function $f$ by building a second-order symbolic execution engine.*

- **Allow for existentially quantified second order variables.**

- **Restrict their interpretation to a language e.g. linear integer arithmetic**

  $\text{Term} = \text{Var} | \text{Constant} | \text{Term} + \text{Term} | \text{Term} - \text{Term} | \text{Constant} \ast \text{Term}$

- **Example SAT**
  - $\rho(0) > 0 \land \rho(1) \leq 0$
  - Satisfying solution $\rho = \lambda x. 1 - x$
Second order Program Repair

\[ Term = Var \mid Constant \mid Term + Term \mid Term - Term \mid Constant \times Term \]

**Synthesis Specification:**

\[ \exists \rho. \bigvee_i \pi_i \land \text{output} = \text{expected} \]

Solve for \( \rho \) directly
Encoding for Synthesis

\[ \psi_{\text{node}} := \bigwedge_{j \in [1, C]} s_i^j \Rightarrow \text{out}_i = F_j(\text{out}_{i_1}, \text{out}_{i_2}, \ldots, \text{out}_{i_k}) \]

\[ \psi_{\text{choice}} := \text{exactlyOne}(s_i^1, s_i^2, \ldots, s_i^C) \]
First order vs. Second order

<table>
<thead>
<tr>
<th>Subject</th>
<th>FO patch</th>
<th>SO patch</th>
<th>FO paths</th>
<th>SO paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>find.091557f6</td>
<td>-</td>
<td>Correct</td>
<td>400</td>
<td>16</td>
</tr>
<tr>
<td>find.24bf33c0</td>
<td>Plausible</td>
<td>Plausible</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>find.24e2271e</td>
<td>Correct</td>
<td>Correct</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>find.07b941b1</td>
<td>Plausible</td>
<td>Plausible</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>find.e6680237</td>
<td>Correct</td>
<td>Correct</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>find.dbcb10e9</td>
<td>-</td>
<td>Correct</td>
<td>400</td>
<td>14</td>
</tr>
<tr>
<td>find.e1d0a991</td>
<td>Plausible</td>
<td>Plausible</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>grep.55c7f6b6a</td>
<td>Correct</td>
<td>Correct</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>grep.3220317a</td>
<td>Plausible</td>
<td>Plausible</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>grep.db9d6340</td>
<td>Plausible</td>
<td>Plausible</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>grep.c96b0f2c</td>
<td>Plausible</td>
<td>Plausible</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>grep.5fa8c7c9</td>
<td>Correct</td>
<td>Correct</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>grep.54d55bba</td>
<td>-</td>
<td>Plausible</td>
<td>400</td>
<td>26</td>
</tr>
<tr>
<td>Overall</td>
<td>10</td>
<td>13</td>
<td>107</td>
<td>19</td>
</tr>
</tbody>
</table>

GNU Utilities
(Test-based) Program Repair

Syntax-based Schematic

for e in Search-space {
    Validate e against Tests
}

Semantic Schematic

for t in Tests {
    generate repair constraint $\Psi_t$
}

Synthesize e from $\bigwedge_t \Psi_t$
Other Possibilities

Syntax and semantics-based Schematic

for (e ∈ SearchSpace) do
  if (e is unmarked)
    validate e;
  mark all e’ in the same partition as e
done

Test-equivalent for test n=2 since both expressions are evaluated into \{F, T\}

```
for (i = 0; i < n; i++)
  if (i mod 2 == 1)
    printf("1");
```

```
for (i = 0; i < n; i++)
  if (i == 1)
    printf("1");
```

```
x = 1;
if (n > 0) {
    printf(x);
}
```

```
if (n > 0) {
    x = 1;
    printf(x);
}
```

Test-equivalent for test n=2 since
1. if-condition is true
2. If condition does not depend on x
Test-equivalence based repair

```c
scanf("%d", &x);
for (i = 0; i < 10; i++)
    if (x - i > 0)
        printf("1");
    else
        printf("0");
```

Consider all inequalities $ax [\pm] \beta i [\geq=\neq] \gamma$

Sequence of values:  
Equivalence class ($x = 4$):

- $\{T, T, T, T, T, T, T, T, T\}$  
  $\{x > 0, x > 1, \ldots\}$
- $\{T, T, T, T, T, T, T, T, F\}$  
  $\{x - i > -5, \ldots\}$
- $\{T, T, T, T, T, T, T, F, T\}$  
  EMPTY
- $\{T, T, T, T, T, T, F, F, T\}$  
  $\{x - i > -4, \ldots\}$
- $\{T, T, T, T, T, T, F, F, T\}$  
  EMPTY
- $\{T, T, T, T, T, T, F, T, T\}$  
  EMPTY
- $\{T, T, T, T, T, T, F, F, T\}$  
  EMPTY
- $\{T, T, T, T, T, T, F, F, F\}$  
  EMPTY

...
## Effectiveness

<table>
<thead>
<tr>
<th>Subject</th>
<th>F1X</th>
<th>Angelix</th>
<th>Prophet</th>
<th>GenProg</th>
</tr>
</thead>
<tbody>
<tr>
<td>libtiff</td>
<td>5/13</td>
<td>3/10</td>
<td>2/5</td>
<td>0/5</td>
</tr>
<tr>
<td>lighttpd</td>
<td>0/5</td>
<td>-/-</td>
<td>-/4</td>
<td>0/4</td>
</tr>
<tr>
<td>php</td>
<td>6/15</td>
<td>4/10</td>
<td>10/18</td>
<td>2/7</td>
</tr>
<tr>
<td>gmp</td>
<td>2/2</td>
<td>2/2</td>
<td>1/2</td>
<td>0/1</td>
</tr>
<tr>
<td>gzip</td>
<td>2/3</td>
<td>1/2</td>
<td>1/2</td>
<td>0/2</td>
</tr>
<tr>
<td>python</td>
<td>0/5</td>
<td>-/-</td>
<td>0/6</td>
<td>1/3</td>
</tr>
<tr>
<td>wireshark</td>
<td>0/4</td>
<td>0/4</td>
<td>0/4</td>
<td>0/4</td>
</tr>
<tr>
<td>fbc</td>
<td>1/1</td>
<td>-/-</td>
<td>1/1</td>
<td>0/1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>16/49</strong></td>
<td><strong>10/28</strong></td>
<td><strong>15/42</strong></td>
<td><strong>3/27</strong></td>
</tr>
</tbody>
</table>

* Correct/Plausible
# Efficiency

<table>
<thead>
<tr>
<th>Subject</th>
<th>Defects</th>
<th>Tests</th>
<th>Testing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>libtiff</td>
<td>24</td>
<td>78</td>
<td>8.18 sec</td>
</tr>
<tr>
<td>lighttpd</td>
<td>9</td>
<td>295</td>
<td>29.75 sec</td>
</tr>
<tr>
<td>php</td>
<td>44</td>
<td>8471</td>
<td>427.25 sec</td>
</tr>
<tr>
<td>gmp</td>
<td>2</td>
<td>146</td>
<td>53.52 sec</td>
</tr>
<tr>
<td>gzip</td>
<td>5</td>
<td>12</td>
<td>0.43 sec</td>
</tr>
<tr>
<td>python</td>
<td>11</td>
<td>35</td>
<td>156.46 sec</td>
</tr>
<tr>
<td>wireshark</td>
<td>7</td>
<td>63</td>
<td>9.92 sec</td>
</tr>
<tr>
<td>fbc</td>
<td>3</td>
<td>773</td>
<td>240.27 sec</td>
</tr>
</tbody>
</table>

![Time (seconds)](chart.png)

- **F1X**
- **Angelix**
- **Prophet**
- **GenProg**

*KAIST Invited Colloquium 2020*
Application: Security

Number of identified vulnerabilities in 2018: 81915

On average, it took developer 69 days to fix the critical vulnerabilities.
Repair with Fuzzing

[ISSTA19]

Search Space

Overfitted Patches

Correct Patches

Mutators

Refined patch pool

Pool of Patches

Mutated files

Test suite

Input Queue

Deque

Enqueue

Mutators

Test suite (Seeds)

Input Queue

Deque

Enqueue (IsInteresting)
Fix2Fit

Integration of repair into programming environments?

Number of plausible patches that can be reduced if the tests are empowered with more oracles
A scalable binary rewriting tool is of critical importance:
- Binary Repair: (Automatically) fixing bugs at the binary level
- Binary Hardening: Mitigating against undiscovered bugs

Scalable binary rewriting is hard:
- Control flow information must be recovered from the binary.
- Consider the issue of register indirect jumps
  - jmp [R]
  - Content of R is not known at compile-time.
- Even if a patch is synthesized, it cannot be inserted into the binary!!!

The E9Patch approach is highly scalable → can rewrite Chrome/Firefox:

The browser runs stable for benchmarks + various websites including youtube.

E9Patch is first static binary rewriter that can scale to >100MB binaries.

Check out our upcoming PLDI20 paper.
Use program repair in intelligent tutoring systems to give the students' individual attention. *Detailed Study in IIT-Kanpur, India [FSE17, and ongoing]*
Repair in steps

Use program repair in intelligent tutoring systems to give the students’ individual attention.

*Detailed Study in IIT-Kanpur, India*
Use program repair in intelligent tutoring systems to give the students’ individual attention.

**KAIST Invited Colloquium 2020**
Reactory: Teaching Python to 1st year

Reference Solutions

Incorrect Student Program

Test-Suite

(1) Re-factoring

Semantically Equivalent Solutions

(2) Block Repair

Patch

KAIST Invited Colloquium 2020
Experiments at NUS

- **Dataset**
  - Introductory Python programming course at National University of Singapore
  - Credited by 361 students
  - 2,442 correct student submissions
  - 1,783 incorrect student submissions
  - [https://github.com/githubhuyang/refactory/](https://github.com/githubhuyang/refactory/)

- **Baseline**
  - Clara [Gulwani et al, PLDI 2018]
  - [https://github.com/iradicek/CLARA](https://github.com/iradicek/CLARA)

<table>
<thead>
<tr>
<th></th>
<th>Refactory</th>
<th>Clara (baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repair Rate</td>
<td>90.8 %</td>
<td>71.3 %</td>
</tr>
<tr>
<td>Time Taken (seconds)</td>
<td>5.5 s</td>
<td>13.6 s</td>
</tr>
<tr>
<td>Relative Patch Size</td>
<td>0.400</td>
<td>0.560</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Reference Solution</th>
<th>Incorrect Student Program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def search(x, seq):</code></td>
<td><code>def search(e, lst):</code></td>
</tr>
<tr>
<td><code>for i in range(len(seq)):</code></td>
<td><code>for j in range(len(lst)):</code></td>
</tr>
<tr>
<td><code>if x &lt;= seq[i]: return i</code></td>
<td><code>if e &lt;= lst[j]: return j</code></td>
</tr>
<tr>
<td><code>j = j + 1</code></td>
<td><code>else: return len(seq)</code></td>
</tr>
<tr>
<td><code>return len(seq)</code></td>
<td><code>return len(lst) + 1</code></td>
</tr>
</tbody>
</table>
Some Relevant Research Results

Binary Rewriting without Control Flow Recovery,
PLDI 2020

Re-factoring based Program Repair applied to Programming Assignments,
ASE 2019

Crash-Avoiding Program Repair
ISSTA 2019.

Symbolic execution with second order existential constraints
ESEC-FSE 2018.

A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments

Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis (pdf)
ICSE 2016.

DirectFix: Looking for Simple Program Repairs (PDF)
ICSE 2015.

SemFix: Program Repair via Semantic Analysis (pdf)
ICSE 2013.


ACKNOWLEDGEMENT: National Cyber Security Research program from NRF Singapore
http://www.comp.nus.edu.sg/~tsunami/
Summary

Figure taken from:

Automated Program Repair