

Lecture 10 – Automated Program Repair

AAA705: Software Testing and Quality Assurance

Jihyeok Park

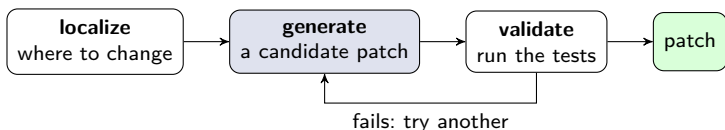


2026 Fall

```
1  int mid(int x, int y, int z) {  
2      int m = z;  
3      if (y < z) {  
4          if (x < y)      m = y;  
5          else if (x < z) m = y; // bug  
6      } else {  
7          if (x > y)      m = y;  
8          else if (x > z) m = x;  
9      }  
10     return m;  
11 }
```

	input	expected	
t_1	(3, 3, 5)	3	✓
t_2	(1, 2, 3)	2	✓
t_3	(3, 2, 1)	2	✓
t_4	(5, 5, 5)	5	✓
t_5	(5, 3, 4)	4	✓
t_6	(2, 1, 3)	2	✗

- Fault localization: line 5 is **rank 1**
- A human writes `m = x`. Can a **program** do it?



- Goal: a program P' that passes **all** the tests
- The tests are the **only specification**
- Today: how to **generate** – and why passing the tests is **not enough**

1. Generate-and-Validate Repair
2. Semantics-Based Repair
3. Patch Overfitting
4. LLM-Based Repair

1. Generate-and-Validate Repair
2. Semantics-Based Repair
3. Patch Overfitting
4. LLM-Based Repair

“Changes to a codebase contain snippets that already exist in the codebase at the time of the change.”¹

```
1  int mid(int x, int y, int z) {  
2      int m = z;  
3      if (y < z) {  
4          if (x < y)      m = y;  
5          else if (x < z) m = y; // bug  
6      } else {  
7          if (x > y)      m = y;  
8          else if (x > z) m = x;  
9      }  
10     return m;  
11 }
```

The fix for line 5 is $m = x$.

It is already on **line 8**.

In real commits, **43%** of a change already exists in the code.

¹[FSE'14] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, F. Sarro. “The Plastic Surgery Hypothesis.”

The fix is already in the program → just move existing code.

GenProg² never writes new code. It only **moves** existing statements:

edit	on mid	tests
delete	delete line 5	4 / 6
insert	insert line 8 after line 5	6 / 6
replace	replace line 5 with line 8	6 / 6

- Edit only lines that the **failing test** runs
- Copy only statements that **already exist**

²[ICSE'09] W. Weimer, T. Nguyen, C. Le Goues, S. Forrest. "Automatically Finding Patches Using Genetic Programming."

$$\text{fitness} = \#\{\text{passing tests kept}\} + 10 \times \#\{\text{failing tests fixed}\}$$

variant	t_1	t_2	t_3	t_4	t_5	t_6	fitness
original	✓	✓	✓	✓	✓	✗	5
delete line 5	✗	✓	✓	✓	✓	✗	4
insert line 8 after line 5	✓	✓	✓	✓	✓	✓	15

Genetic programming: evolve a population of variants.



```
void gcd(int a, int b) {  
    if (a == 0) {  
        printf("%d", b);  
        exit(0);        // inserted  
        a = a - b;      // inserted  
    }  
    while (b != 0)  
        if (a > b) a = a - b;  
        else      b = b - a;  
    printf("%d", a);  
    exit(0);  
}
```

Bug: gcd(0, 55) prints 55,
then **loops forever**.

The fix exit(0) is copied from
the **last line**.

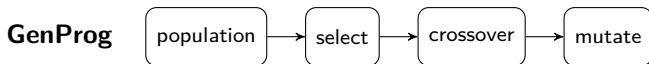
GP also inserted `a = a - b`. It
never runs, so the tests cannot
tell.

Delta debugging keeps only the
edits needed: exit(0).

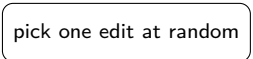
Is the Genetic Algorithm Needed?

GenProg searches with a genetic algorithm. Does the search matter?

RSRepair³: same edits, but pick them **at random**.



RSRepair



```
graph LR; A[pick one edit at random]
```

- Random search needs **fewer** tries on 23 of 24 bugs
- The search method does not matter much. The **search space** does

³[ICSE'14] Y. Qi, X. Mao, Y. Lei, Z. Dai, C. Wang. "The Strength of Random Search on Automated Program Repair."

A Patch No Developer Would Accept

GenProg's random edits pass the tests – but do they make sense?

Mozilla Rhino: `getShort(...)` can return `-1` → array index out of bounds.

```
lhs = strings[getShort(iCode, pc + 1)];           // buggy
lhs = ((Scriptable) lhs).getDefaultValue(null);   // GenProg
if (getShort(iCode, pc + 1) < strings.length &&    // PAR
    getShort(iCode, pc + 1) >= 0)
    lhs = strings[getShort(iCode, pc + 1)];
```

- GenProg's patch passes the tests – by **removing** the array access
- PAR's patch looks like what a **human** would write

PAR⁴: read 60,000+ human patches, write down the common **shapes**.

<pre>// Null Pointer Checker - x.foo(); + if (x != null) x.foo(); // Parameter Replacer - f(a, b); + f(b, a);</pre>	<pre>// Range Checker - a[i] + if (i < a.length) a[i] // Expression Replacer - if (x < y) ... + if (x <= y) ...</pre>
--	---

- 10 templates. Fewer candidates, more **human-like** patches

⁴[ICSE'13] D. Kim, J. Nam, J. Song, S. Kim. "Automatic Patch Generation Learned From Human-Written Patches."

PAR's 10 templates cannot fix mid $\rightarrow$ collect more templates.

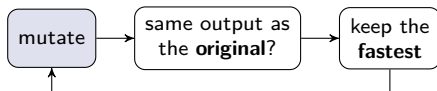
TBar⁵: 35 templates. *Mutate Variable* replaces one variable on mid:

line	patch	tests
4	$m = y \rightarrow m = x$	4 / 6
4	$m = y \rightarrow m = z$	4 / 6
5	$m = y \rightarrow m = z$	4 / 6
5	$m = y \rightarrow m = x$	6 / 6

- A good template at the wrong line is useless
- Given the buggy line, TBar fixes **74** bugs instead of **43**

⁵[ISSTA'19] K. Liu, A. Koyuncu, D. Kim, T. F. Bissyandé. "TBar: Revisiting Template-Based Automated Program Repair."

Genetic improvement⁶: keep the loop, change the fitness to **faster**.



- Bowtie2 (DNA aligner, 50K lines): **70**× faster
- The original program is the **oracle**. Repair has no such oracle

⁶[TEVC'15] W. B. Langdon, M. Harman. "Optimizing Existing Software with Genetic Programming."

1. Generate-and-Validate Repair
2. Semantics-Based Repair
3. Patch Overfitting
4. LLM-Based Repair

GenProg guesses edits and lets the tests vote → ask what the line must compute.

SemFix⁷ on a bug in Tcas (collision avoidance):

```
int is_upward_preferred(int inhibit, int up_sep, int down_sep) {
    int bias;
    if (inhibit) bias = f(inhibit, up_sep, down_sep); // was down_sep
    else        bias = up_sep;
    if (bias > down_sep) return 1;
    else        return 0;
}
```

- Do not guess edits. Ask: **what must f return** so that the tests pass?

⁷[ICSE'13] H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra. "SemFix: Program Repair via Semantic Analysis."

Run each test up to line 3. Then make bias **symbolic** and keep going.

input	expected	constraint on f
$(1, 11, 110)$	1	$f(1, 11, 110) > 110$
$(1, 0, 100)$	0	$f(1, 0, 100) \leq 100$
$(1, -20, 60)$	1	$f(1, -20, 60) > 60$

- The branch `bias > down_sep` turns into the constraint
- All three together are the **repair constraint**

try	f	result
constants only	c	no: $c > 110$ and $c \leq 100$
add “+”	$up_sep + c$	$up_sep + 100$

Check: $11 + 100 > 110$, $0 + 100 \leq 100$, $-20 + 100 > 60$.

- Patch: `bias = up_sep + 100;`
- GenProg **searches** and lets tests vote. SemFix **solves** what the tests demand

```
else if (x < z) m = f(x, y, z);    // line 5
```

test	reaches line 5?	constraint
$t_1 (3, 3, 5) \rightarrow 3$	yes	$f(3, 3, 5) = 3$
$t_6 (2, 1, 3) \rightarrow 2$	yes	$f(2, 1, 3) = 2$
$t_2, \dots, t_5$	no	—

- A constant? $3 \neq 2$ ✗
- A variable? $f = x$ gives 3 and 2 ✓

Nopol – Is This `if` the Bug?

SemFix needs symbolic execution → for `if` conditions, a cheaper trick.

Nopol⁸ fixes only `if` conditions. A bug in gcd (Commons Math):

```
int gcd(int u, int v) {  
    if (u * v == 0) return Math.abs(u) + Math.abs(v);  
    ...                               // the real gcd
```

Run the failing test `gcd(65536, 65536)` → 65536 twice. Do not compute the condition. **Force** the branch:

condition replaced by	result	
true	131072	✗
false	65536	✓

`false` works → this `if` is the place to fix.

⁸[TSE'17] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. Lamelas Marcote, T. Durieux, D. Le Berre, M. Monperrus. "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs."

Angelic value: the value the condition **should** have had for the test to pass.

test	$u \cdot v == 0$ now	should be
$\text{gcd}(0, 5) \rightarrow 5$	true	true
$\text{gcd}(6, 4) \rightarrow 2$	false	false
$\text{gcd}(65536, 65536) \rightarrow 65536$	true (overflow)	false

- Passing tests: keep the current value. Failing test: the angelic value
- Find a condition that gives the last column: $u == 0 \ || \ v == 0$

SemFix throws the original expression away → keep as much of it as possible.

DirectFix⁹: is c in s? The loop misses the last character.

```
1  for (i = 0; i < k; i++)
2    if (s[i] == c) return TRUE;
3  return FALSE;
```

s, c	exp.	now
"ab?", '?'	T	F
"ab?c", '?'	T	T
"!ab", '!'	T	T

Both patches pass all three tests:

```
2    if (c == '?' || c == '!') return TRUE;    // memorized the tests
1  for (i = 0; i <= k; i++)                    // the real fix
```

The first never looks at s: 'b' in "abc" gives FALSE.

⁹[ICSE'15] S. Mechtaev, J. Yi, A. Roychoudhury. "DirectFix: Looking for Simple Program Repairs."

Every part of the original code is a **switch**: keep it, or change it.

```
1  for (i = 0; i < k; i++)  
2    if (s[i] == c) return TRUE;
```

Ask a solver: **all tests must pass**, and **change the fewest switches**.

switches changed	patch	tests
0	–	✗
1	$< \rightarrow \leq$	✓
many	<code>c == '?' c == '!'</code>	✓

The changed switch also tells **where** the bug is.

- SemFix: only **one** line can be unknown
- DirectFix: the **whole program** becomes one formula – too big for large programs

Angelix¹⁰: make **several** suspicious lines unknown at once.

```
if (y < z) {  
    if (x < y)      m = f1(x, y, z);    // line 4, unknown  
    else if (x < z) m = f2(x, y, z);    // line 5, unknown
```

We do not need to know **which** of the two is wrong.

¹⁰[ICSE'16] S. Mechtaev, J. Yi, A. Roychoudhury. "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis."

Run each test. Record only what each unknown **must** return, and the variables there:

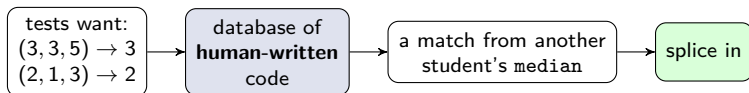
test	unknown	must return	variables there
$t_1 (3, 3, 5) \rightarrow 3$	f2	3	$x = 3, y = 3, z = 5$
$t_2 (1, 2, 3) \rightarrow 2$	f1	2	$x = 1, y = 2, z = 3$
$t_6 (2, 1, 3) \rightarrow 2$	f2	2	$x = 2, y = 1, z = 3$

Solve both together, changing as little as possible (like DirectFix):

- $f1 = y$: already returns 2 $\rightarrow$ **keep**
- $f2 = x$: 3 and 2 $\rightarrow$ **changed**
- The table grows with the **tests**, not the program $\rightarrow$ scales to wireshark (2.8M lines). Also fixed **Heartbleed**

Synthesized code fits only the tests $\rightarrow$ use code that humans wrote.

SearchRepair¹¹: do not synthesize. **Find** code that already does it.



- Works well on small programs like **median-of-three**
- Human code does more than the tests ask $\rightarrow$ passes more **unseen** tests

¹¹[[ASE'15](#)] Y. Ke, K. T. Stolee, C. Le Goues, Y. Brun. "Repairing Programs with Semantic Code Search."

Every fragment in the database is stored as a formula over its inputs and output:

```
if (a < b) r = a;           // (a < b  && r == a)
else      r = b;           // || (a >= b && r == b)
```

Query: map $a \rightarrow x$, $b \rightarrow z$, $r \rightarrow m$. Does the formula hold for the tests at line 5?

test	fragment gives	expected
$t_1: x = 3, z = 5$	3	3 ✓
$t_6: x = 2, z = 3$	2	2 ✓

The solver says **yes** $\rightarrow$ splice the fragment into line 5.

All tools so far trust only the given tests → make new inputs.

CPR¹²: a divide-by-zero in LibTIFF. Fill in **CONDITION**:

```
if (CONDITION) return 0;  
cc = rows * width + 2 * ((rows * width) / (x * y));
```

The only test: $(x, y) = (7, 0)$ must not crash.

CONDITION	$(7, 0)$
$x == 0 \ \ y == 0$	✓
$y == 0$	✓
$x \geq 7$	✓
$y < 1$	✓

All plausible. Which one is right?

¹²[PLDI'21] R. Shariffdeen, Y. Noller, L. Grunske, A. Roychoudhury. "Concolic Program Repair."

Keep **all** candidates. Generate a new input where they **differ** (concolic execution):

CONDITION	(7, 0)	new (0, 5)
$x == 0 \ \ y == 0$	✓	✓
$y == 0$	✓	divides by 0
$x \geq 7$	✓	divides by 0
$y < 1$	✓	divides by 0

- No test says what (0, 5) should return. A **spec** judges it: $x \cdot y \neq 0$ before dividing
- Repeat: new input, cut the patches that break the spec

1. Generate-and-Validate Repair
2. Semantics-Based Repair
3. Patch Overfitting
4. LLM-Based Repair

Two Patches Passed Every Test

```
else if (x < z) {  
    m = y;  
    m = x;      // inserted  
}
```

insert line 8 after line 5

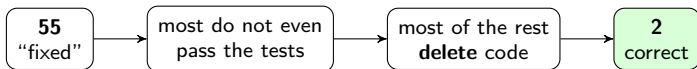
```
else if (x < z) m = x; // was y  
} else {  
    if (x > y)      m = y;  
    else if (x > z) m = y; // was x
```

swap lines 5 and 8

- The swap **broke line 8**. No test runs line 8
- Passes all tests = **plausible**. Plausible $\neq$ **correct**
- The patch **overfits** the tests

Revisiting “55 out of 105”

GenProg reported 55 of 105 real bugs fixed, \$8 each.¹³ Qi et al. read every patch.¹⁴



¹³[ICSE'12] C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer. "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for \$8 Each."

¹⁴[ISSTA'15] Z. Qi, F. Long, S. Achour, M. Rinard. "An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems."

How Did Wrong Patches Pass?

A libtiff test script, in spirit:

```
$ tiffcp input.tif output.tif  
$ echo $?           # the test checks only this: 0 means "pass"
```

- 73 of 78 libtiff tests checked only the **exit code**
- A patch that skips the work and exits with 0 “passes”
- One patch deleted a check and brought back a **security bug** (CVE-2006-2025)

Qi et al. built **Kali** to make the point. It has three edits:

edit	example
redirect a branch	<code>if (c) → if (true)</code> or <code>if (false)</code>
insert a return	<code>return 0;</code> before a statement
remove a statement	delete it

- Kali fixes **as many** bugs correctly as GenProg
- Deleting code is the **cheapest** way to make a failing test pass

Qi et al. read the patches by hand → measure with a second test suite.

Smith et al.¹⁵: repair with one test suite, **judge** with another.

patch	repair suite t_1-t_6	held-out mid(1, 2, 0)
insert line 8 after line 5	6 / 6	1 ✓
swap lines 5 and 8	6 / 6	2 ✗

- Their study used the same kind of program: student **median** code
- A typical GenProg patch passed only **75%** of the held-out tests

¹⁵[FSE'15] E. K. Smith, E. T. Barr, C. Le Goues, Y. Brun. "Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair."

```
1  int mid(int x, int y, int z) {  
2      int m = z;  
3      if (y < z) {  
4          if (x < y)      m = y;  
5          else if (x < z) m = y; // bug  
6      } else {  
7          if (x > y)      m = y;  
8          else if (x > z) m = x;  
9      }  
10     return m;  
11 }
```

No test in t_1 – t_6 runs **line 8**.

So the swap may break it for free.

Add one test that runs line 8 →
the swap **dies**.

- Patch quality follows the **coverage** of the repair tests

Why so many wrong patches? → look at the whole search space.

Long and Rinard¹⁶: count every plausible and every correct patch in the space.

one statement edit

✓ insert line 8

✗ swap 5 and 8

any $v_1 \pm v_2 + c$ on line 5

✓ x

✗ $z - x + 1$

✗ $y - x + 3$

- Real bugs: about **one** correct patch, hundreds of plausible ones
- A bigger space can find **fewer** correct patches

¹⁶[ICSE'16] F. Long, M. Rinard. "An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems."

Many wrong patches just delete code → forbid such edits.

Tan et al.¹⁷: some edits are **almost never** a real fix.

anti-pattern	on mid	
delete a control statement	delete the <code>if</code> on line 5	forbidden
append an early exit	insert <code>return m</code> ; after line 2	forbidden
delete a <code>return</code>	delete line 10	forbidden
–	swap lines 5 and 8	allowed

- Removes cheap, deleting patches
- But the swap still gets through

¹⁷[FSE'16] S. H. Tan, H. Yoshida, M. R. Prasad, A. Roychoudhury. "Anti-patterns in Search-Based Program Repair."

Forbidding catches only obvious cases → rank by what correct patches look like.

Prophet¹⁸: a PHP bug. Both patches pass the tests:

```
if (offset out of bounds) {  
    if (type != 0) return;           // plausible: skips the rest  
    zend_error("Uninitialized string offset");  
    result = ...;  
}  
  
if (offset out of bounds) {  
    if (type != 3)                   // correct: guards only the error  
        zend_error("Uninitialized string offset");  
    result = ...;  
}
```

- Learned from human patches: “a **return** before an assignment” is **rarely correct** → rank it low

¹⁸[[POPL'16](#)] F. Long, M. Rinard. “Automatic Patch Generation by Learning Correct Code.”

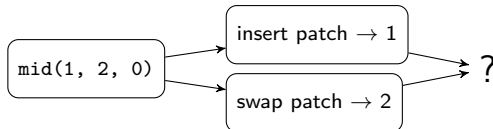
Describe each candidate by **features**:

feature	early return	guarded error
inserts a return	✓	
skips a later assignment	✓	
guards an existing call		✓

- Score = sum of **weights** of the features
- Weights are **learned** so that human-written correct patches score highest
- Run the tests on candidates in score order → the first plausible patch is more often correct

More Tests – Who Says Which Is Right?

Generate a new input. The two patches disagree:



- New inputs are easy. The hard part is the **oracle**
- Two answers: ask a **developer** one small question, or compare with the **original program**

A new input has no expected value → ask a developer about just that input.

DiffTGen¹⁹: toBoolean("tru") must return false.

```
case 3:
  if (ch == 'y') return ...;    // "yes"
  if (ch == 'Y') return ...;    // "YES"  -- Nopol: if (str != null)
```

Generate inputs that run the **changed line** (EvoSuite). Find one where the two programs **differ**: "@es"

	toBoolean("@es")
buggy original	false
Nopol's patch	true

Developer: false → the patch **overfits**. A new test.

¹⁹[ISSTA'17] Q. Xin, S. P. Reiss. "Identifying Test-Suite-Overfitted Patches through Test Case Generation."

No developer to ask → compare with the original program.

PATCH-SIM²⁰: a jKali patch for JFreeChart.

```
public void draw(...) {  
+   if (true) return;      // tests only check "no exception" -> pass  
    ...  
}
```

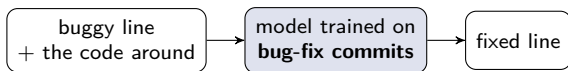
- Idea: a correct patch **changes** the failing test's run and **keeps** the passing tests' runs
- Here the passing test's run changes completely → **rejected**
- More tests (Randoop): label each new input passing-like or failing-like by how **similar** its run is, then apply the same rule

²⁰[ICSE'18] Y. Xiong, X. Liu, M. Zeng, L. Zhang, G. Huang. "Identifying Patch Correctness in Test-Based Program Repair."

1. Generate-and-Validate Repair
2. Semantics-Based Repair
3. Patch Overfitting
4. LLM-Based Repair

Templates are written by hand → learn fixes from bug-fix commits.

SequenceR²¹: translate a buggy line into a fixed line.



```
- masterNode = null;  
+ nonMasterNode = null;    // a rare name, copied from the context
```

- Learns only the kinds of fixes seen in the data

²¹[TSE'21] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshvyanyk, M. Monperrus. "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair."

SequenceR needs bug-fix data → use a model trained on all code.

AlphaRepair²²: a model trained on **all** code. Mask the buggy line:

```
if (y < z) {  
    if (x < y)      m = y;  
    else if (x < z) <MASK>  
} ...  
return m;
```

$$\Pr[m = x;] > \Pr[m = z;] > \Pr[m = y;]$$

- No bug-fix data needed. The code **around** the blank says what fits

²²[FSE'22] C. S. Xia, L. Zhang. "Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning."

AlphaRepair guesses, then checks; the test result is thrown away → feed it back.

ChatRepair²³ on a bug in Closure:

```
Q: Buggy line: if (result != null) {  
    Test error: expected <NoObject> but was <None>  
A: if (!(result instanceof NoObjectType)) {  
Q: Compilation error: cannot find symbol NoObjectType  
A: if (!(result instanceof UnionType)) {  
Q: Still fails the original test  
A: if (!result.isNoType()) {                                -> all tests pass  
Q: Please generate an alternative fix.
```

- The **test result** becomes the next prompt

²³[\[ISSTA'24\]](#) C. S. Xia, L. Zhang. "Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for \$0.42 Each using ChatGPT."

ChatRepair follows a fixed loop → let the model choose its next step.

RepairAgent²⁴: the model calls **tools** and picks the next one itself.

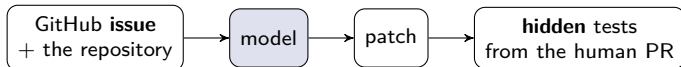
```
state: understand the bug
  > extract_tests()          -> "mid(2, 1, 3): expected 2, got 1"
  > run_fault_localization() -> line 5, line 4, line 3, ...
state: collect information to fix the bug
  > read_range(mid.c, 1, 11)
  > search_code_base("m = ") -> lines 2, 4, 5, 7, 8
state: try to fix the bug
  > write_fix(mid.c, 5, "else if (x < z) m = x;") -> 6 tests passed
```

- Localize, generate, validate are now **actions**, not fixed steps

²⁴[ICSE'25] I. Bouzenia, P. Devanbu, M. Pradel. "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair."

All so far start from a failing test → start from an issue a user wrote.

SWE-bench²⁵: real issues from real Python projects.



- No failing test is given. The model starts from **text**
- Judge: the developer's tests, unseen by the model
- But many passing patches copied the answer from the issue or passed weak tests²⁶: 12.5% → **4%**

²⁵[\[ICLR'24\]](#) C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, K. Narasimhan. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?"

²⁶[\[arXiv'24\]](#) R. Aleithan et al. "SWE-Bench+: Enhanced Coding Benchmark for LLMs."

From research to industry: one bug kind, a human at the end.

Getafix²⁷ (Facebook): the analyzer warns “this may be `null`”. Developers fixed such warnings many times:

```
mListView.clearListeners();  ->  if (mListView == null) return;
                               mListView.clearListeners();
mAdapter.notifyDataSetChanged();  ->  if (mAdapter == null) return;
                                   mAdapter.notifyDataSetChanged();
```

Keep what is the same, turn what differs into a **hole**:

```
h0.h1();                      ->  if (h0 == null) return; h0.h1();
```

²⁷[OOPSLA'19] J. Bader, A. Scott, M. Pradel, S. Chandra. “Getafix: Learning to Fix Bugs Automatically.”

A new warning: mView may be `null`.

```
void onPause() {  
    mView.refresh();  
}
```

// warning here

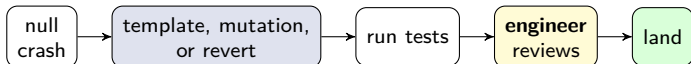
Match the pattern: `h0 = mView, h1 = refresh`. Suggest:

```
void onPause() {  
    if (mView == null) return;  
    mView.refresh();  
}
```

- Several patterns may match → rank them, show only the top one
- A **developer** accepts or rejects it

Getafix fixes analyzer warnings → SapFix fixes crashes.

SapFix²⁸ (Facebook): fix crashes found by automated testing.



- Narrow bug kind + cheap check + a **human** at the end
- The reviewer is the engineer whose change caused the crash

²⁸[ICSE-SEIP'19] A. Marginean et al. "SapFix: Automated End-to-End Repair at Scale."

	where the patch comes from	judge
GenProg	existing statements	tests
TBar	human fix templates	tests
SemFix	a solver	tests
AlphaRepair	a model filling a blank	tests
ChatRepair	a conversation	tests
RepairAgent	an agent	tests

- How patches are made changed completely
- The **judge** never did: plausible $\neq$ correct
- Next: the same models write the **tests**

1. Generate-and-Validate Repair
2. Semantics-Based Repair
3. Patch Overfitting
4. LLM-Based Repair

- LLM-based Software Testing

Jihyeok Park

jihyeok_park@korea.ac.kr

<https://plrg.korea.ac.kr>