A Fact of a Software Life

Not just “a”.....

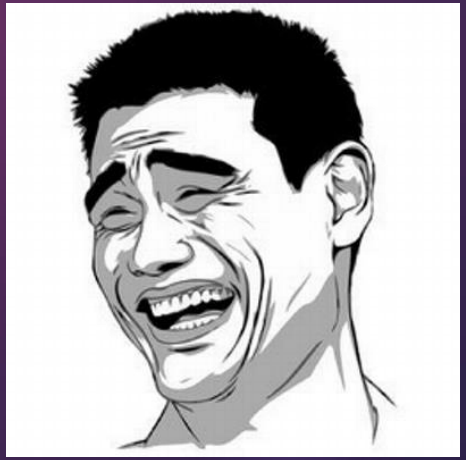
- Windows 2000 shipped with more than 63,000 KNOWN bugs
- In 2005, almost 300 bugs were appearing daily in Mozilla, according to one of its developer

Manual Repair?

Not just “a”.....

- Windows 2000 shipped with more than 63,000 KNOWN bugs
- In 2005, almost 300 bugs were appearing daily in Mozilla, according to one of its developer

Manual Repair?



Automatic Repair!

GenProg (one of several others)

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920    lhs = strings[getShort(iCode, pc + 1)];
1921  }
1922  Scriptable calleeScope = scope;
```

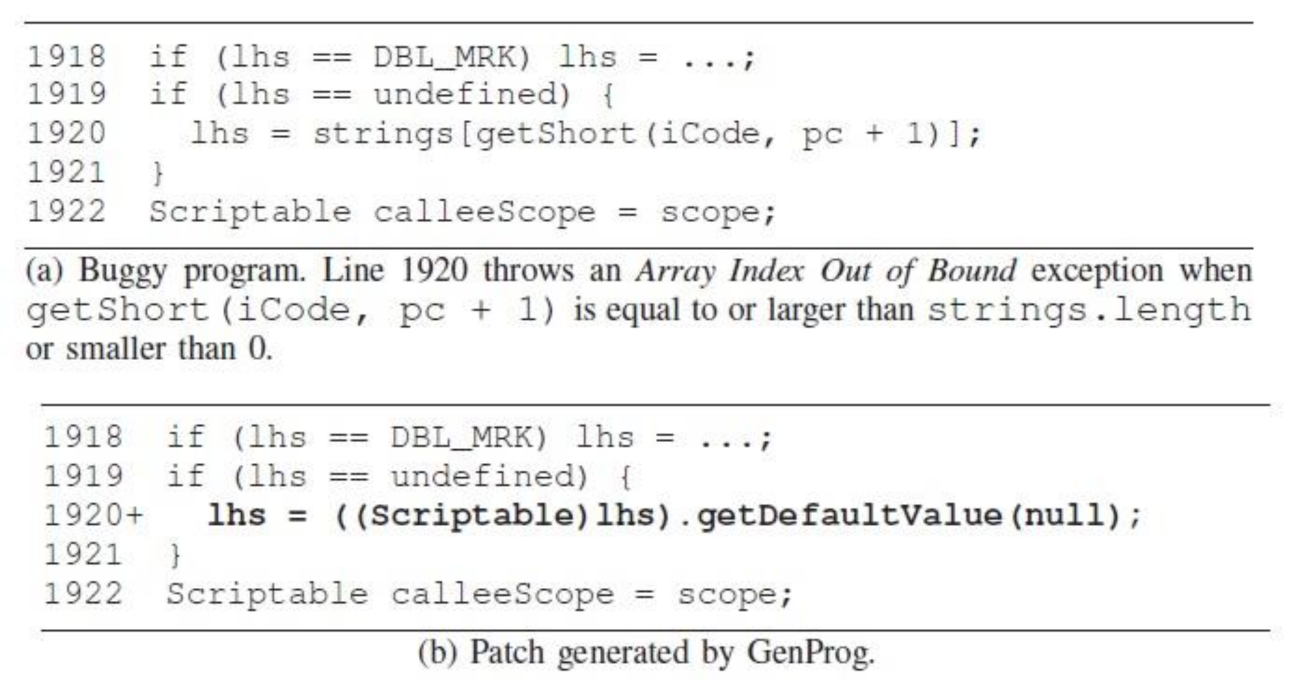
(a) Buggy program. Line 1920 throws an *Array Index Out of Bound* exception when `getShort(iCode, pc + 1)` is equal to or larger than `strings.length` or smaller than 0.

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+   lhs = ((Scriptable)lhs).getDefaultValue(null);
1921  }
1922  Scriptable calleeScope = scope;
```

(b) Patch generated by GenProg.

Automatic Repair

GenProg (HW!)



```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920    lhs = strings[getShort(iCode, pc + 1)];
1921  }
1922  Scriptable calleeScope = scope;
```

(a) Buggy program. Line 1920 throws an *Array Index Out of Bound* exception when `getShort(iCode, pc + 1)` is equal to or larger than `strings.length` or smaller than 0.

```
1918   if (lhs == DBL_MRK) lhs = ...;
1919   if (lhs == undefined) {
1920+    lhs = ((Scriptable)lhs).getDefaultValue(null);
1921   }
1922   Scriptable calleeScope = scope;
```

(b) Patch generated by GenProg.

presenter name(s) removed for FERPA considerations

Automatic Patch Generation Learned from Human-Written Patches

DONGSUN KIM, JAECHANG NAM,
JAEWOO SONG, AND SUNGHUN KIM

Hong Kong University of Science and technology, China

Pattern-based Automatic Program Repair (PAR)

- Manual Observations on Human Written Patches

  Fix Patterns!

- PAR

  Fix Patterns -> Fix Templates -> Automatic Patches

- Empirical Evaluation

  Test PAR on 119 Real Bugs

Manual Observations of Human Written Patches

- Patch Collection – 62,656 human written patches from Eclipse JDT
- Common Patches Mining
  - Identify patches as additive, subtractive, or altering
  - Examine root causes of bugs & how patches specifically resolved the bugs

```
3561    public ITextHover getCurrentTextHover() {
3562+     if (fTextHoverManager== null)
3563+      return null;
3564      return fTextHoverManager.getCurrentTextHover();
3565    }
```

  - Group similar patches into common patterns

Common Fix Patterns!

Top eight patterns cover almost 30% of all patches observed!

TABLE I: Common fix patterns identified from Eclipse JDT's patches.

Fix Patterns
Altering method parameters
Calling another method with the same parameters
Calling another overloaded method with one more parameter
Changing a branch condition
Adding a null checker
Initializing an object
Adding an array bound checker
Adding a class-cast checker

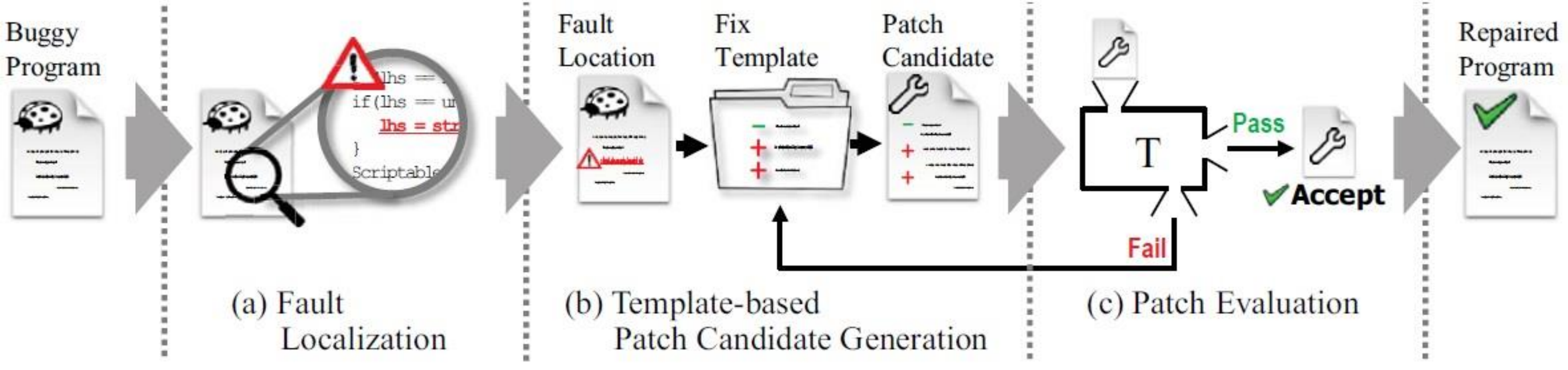
Common Fix Patterns!

Pattern	Example	Description
Altering method parameters	obj.method(v1,v2) -> obj.method(v1, v3)	Pattern gives the appropriate parameters to the method
Calling another method with the same parameters	obj.method1(param) -> obj.method2(param)	Pattern changes the callee in a method call statement
Changing a branch condition	If (a == b) -> If(a == b && c != 0)	Pattern modifies branch condition in conditional statements. Often just add or remove a term to/from a predicate

PAR – In Action

- (1) Identifies fault locations
- (2) Uses fix templates to generate program variants
- (3) Evaluates program variants by fitness function (computes number of passing tests of patch candidate)
- (4) If candidate passes all tests, then SUCCESS!

    ELSE Repeat (2) & (3)



PAR

Algorithm 1: Patch generation using fix templates in PAR.

Input : fitness function *Fit*: Program $\rightarrow \mathbb{R}$
Input : *T*: a set of fix templates
Input : *PopSize*: population size
Output: *Patch*: a program variant that passes all test cases

```
let Pop ← initialPopulation(PopSize);
repeat
    let Pop ← apply(Pop,T);
    let Pop ← select(Pop,PopSize,Fit);
until ∃ Patch in Pop that passes all test cases;
return Patch
```

Fault Localization

- Statistical fault localization based on test cases
- Assumes that a statement visited by failing tests is more likely to be a defect than other statements
  - (1) Executes two groups of tests: passing and failing
  - (2) Records the statement coverage of both test case groups
    - (a) Covered by both groups
    - (b) Covered only by Passing group
    - (c) Covered only by Failing group
    - (d) Not covered by either group

Assign a value of 0.1 to statements in (a), 1.0 to (c), 0.0 otherwise

Fix Template

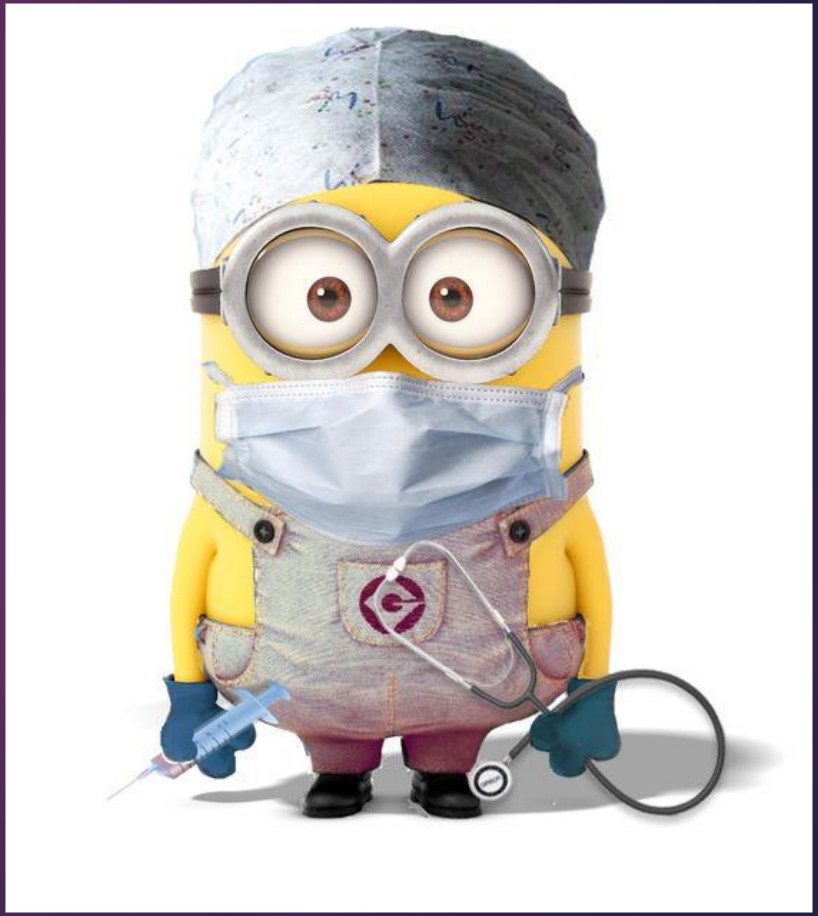
```
[Null Pointer Checker]
P = program
B = fault location

<AST Analysis>
C ← collect object references (method invocations,
   field accesses, and qualified names) of B in P

<Context Check>
if there is any object references in C ⇒ continue
otherwise ⇒ stop

<Program Editing>
insert an if() statement before B

loop for all objects in C {
   insert a conditional expression that checks whether a
      given object is null
}
concatenate conditions by using AND

if B includes return statement {
   negate the concatenated conditional expression
   insert a return statement that returns a default value
      into THEN section of the if() statement
   insert B after the if() statement
} else {
   insert B into THEN section of the if() statement
}
```

Fig. 4: Null pointer checker fix template. This template inserts an `if()` statement checking whether objects are `null`.

- AST Analysis – Scans program's AST and analyzes fault location and adjacent location
- Context Check – Examines whether the program can be edited by a template by inspecting the analyzed AST
- Program Editing – If possible, rewrite the program's AST based on the script in the template

Fitness Evaluation

- Fitness Function
  - (Program variant, test cases) -> Compute value representing the number of passing test cases of the variant
  - Resulting fitness value used for evaluating and comparing program variants in a population
  - Based on fitness values of program variants, PAR chooses program variants by tournament selection



Examples:

```
if (kidMatch != -1) return kidMatch;
for (int i = num; i < state.parenCount; i++)
{
   state.parens[i].length = 0;
}
state.parenCount = num;
```

(a) **Buggy Program**: the underlined statement is a fault location.

⇓

<Null Pointer Checker>

INPUT: `state.parens[i].length = 0;`

1. Analyze: Extract obj refer → `state, state.parens[i]`
2. Context Check: object references?: PASS
3. Edit: INSERT

```
   ...
   ...
+ if( state != null && state.parens[i] != null ) {
          state.parens[i].length = 0;
+ }
   ...
   ...
```

OUTPUT: `a new program variant`

⇓

```
if (kidMatch != -1) return kidMatch;
for ( ... )
{
   if( state != null && state.parens[i] != null)
      state.parens[i].length = 0;
}
state.parenCount = num;
```

(b) **After applying a fix template**: a patch generated by PAR. As shown in the fix template, corresponding statements have been edited.

Fig. 5: Real example of applying a fix template to `NativeRegExp.java` to fix Rhino Bug #76683.

Examples:

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920    lhs = strings[getShort(iCode, pc + 1)];
1921  }
1922  Scriptable calleeScope = scope;
```

(a) Buggy program. Line 1920 throws an *Array Index Out of Bound* exception when `getShort(iCode, pc + 1)` is equal to or larger than `strings.length` or smaller than 0.

```
1918   if (lhs == DBL_MRK) lhs = ...;
1919   if (lhs == undefined) {
1920+    lhs = ((Scriptable)lhs).getDefaultValue(null);
1921   }
1922   Scriptable calleeScope = scope;
```

(b) Patch generated by GenProg.

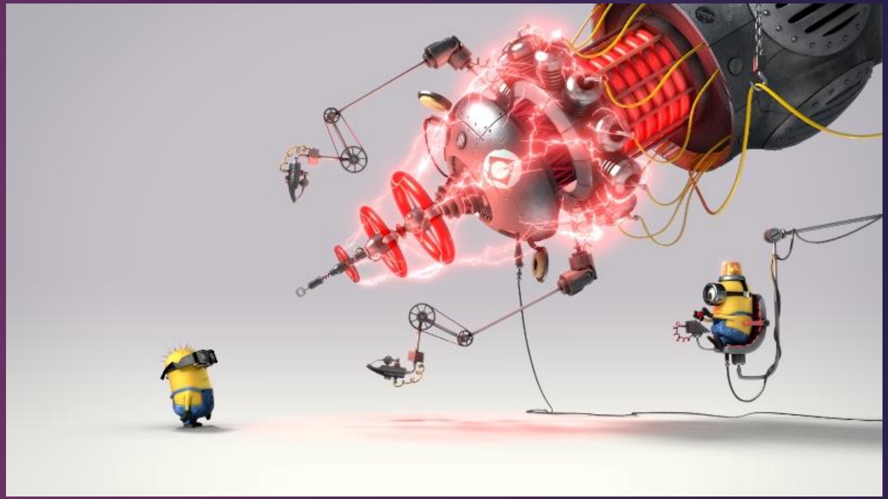
Examples:

```
  if (lhs == DBL_MRK) lhs = ...;
  if (lhs == undefined) {
+   if (getShort(iCode, pc + 1) < strings.length &&
        getShort(iCode, pc + 1) >= 0)
+   {
      lhs = strings[getShort(iCode, pc + 1)];
+   }
  }
  Scriptable calleeScope = scope;
```

(d) Patch generated by PAR.

Evaluation of PAR

- Two Research Questions
  - RQ1: (Fixability) How many bugs are fixed successfully?
  - RQ2: (Acceptability) Which approach can generate more acceptable bug patches?



Experimental Design

TABLE III: Data set used in our experiments. "LOC" (Lines of code) and "# statements" represent the size of each subject. "# test cases" is the number of test cases used for evaluating patch candidates generated by PAR.

Subject	# bugs	LOC	# statements	# test cases	Description
Rhino	17	51,001	35,161	5,578	interpreter
AspectJ	18	180,394	139,777	1,602	compiler
log4j	15	27,855	19,933	705	logger
Math	29	121,168	80,764	3,538	math utils
Lang	20	54,537	40,436	2,051	helper utils
Collections	20	48,049	35,335	11,577	data utils
Total	119	483,004	351,406	25,051	

- Collected 119 bugs from open source projects
- Applied both PAR and GenProg to each bug to generate patches
- Examined how many bugs were successfully fixed by each (RQ1)
- Conducted user study to compare patch quality (RQ2)

Experimental Design

- All six projects written in Java
- All six projects commonly used in literature and have well maintained bug report
- Randomly selected 15-29 bugs per project
- For each bug collected all available test cases
- Conducted 100 runs for each bug per approach. Total runs (100 * 119 * 2 = 23,800)
- Each run stopped when it took more than 10 generations or 8 hours, which meant it failed at creating a successful patch.

RQ1 Fixability

TABLE IV: Patch generation results. Among 119 bugs, PAR successfully fixed 27 bugs while GenProg was successful for only 16 bugs. Note that 5 bugs were fixed by both approaches. We used these 5 bugs in our comparative study for acceptability evaluation.

Subject	# bugs	# bugs fixed by GenProg	# bugs fixed by PAR	# bugs fixed by both
Rhino	17	7	6	4
AspectJ	18	0	9	0
log4j	15	0	5	0
Math	29	5	3	1
Lang	20	1	0	0
Collections	20	3	4	0
Total	119	16	27	5

- **PAR 27   GenProg 16**
- 5 bugs fixed by both but patch for each were different. To be used for RQ2
- Fix patterns were generated from Eclipse JDT but applied to bugs of other projects
- Only used limited number of fix patterns. Can improve fixability?
- GenProg 10% in Java vs 50% in C?

RQ2 (Acceptability)

- Formulated two null hypotheses
  - $H1_0$: Patches generated by PAR and GenProg have no acceptability difference from each other.
  - $H2_0$: Patches generated by PAR have no acceptability difference from human-written patches.
- The corresponding alternative hypotheses are:
  - $H1_a$: PAR generates more acceptable patches than GenProg.
  - $H2_a$: Patches generated by PAR are more acceptable than human-written patches.

RQ2 (Acceptability)

- Subjects
  - Two groups (CS students and Developers)
  - 17 software engineering graduate students with Java experience
  - 68 developers (online developer communities and software companies)
- Study Design
  - Five sessions.
  - One of five bugs per session fixed by both PAR and GenProg
  - Each session explained bug in detail
  - Session listed three anonymized patches (human, PAR, GenProg)
  - Participant asked to compare and rank

RQ2 (Acceptability)

TABLE V: Average rankings evaluated by 17 students (standard deviation is shown in parentheses). The lower values indicate that the patch obtained higher rankings on average by the evaluators.

Bugs	Human	PAR	GenProg
Math #280	**1.33** (0.62)	2.27 (0.59)	2.40 (0.83)
Rhino #114493	2.00 (0.54)	**1.33** (0.62)	2.67 (0.72)
Rhino #192226	**1.47** (0.64)	1.67 (0.62)	2.67 (0.72)
Rhino #217379	1.69 (0.70)	**1.50** (0.63)	2.81 (0.40)
Rhino #76683	2.13 (0.51)	**1.07** (0.26)	2.80 (0.41)
Average	1.72 (0.67)	1.57 (0.68)	2.67 (0.64)

- Result – Students
  - PAR patches consistently ranked higher than GenProg patches
  - Average ranking of PAR patches = 1.57 (SD = 0.68)
  - Average ranking of GenProg patches = 2.67 (SD= 0.64)
  - Ranking differences between Par and GenProg are statistically significant (p-value =0.000 < 0.05)
  - Based on results, reject null hypothesis H1o for student group

RQ2 (Acceptability)

TABLE VI: Average rankings evaluated by 68 developers (standard deviation is shown in parentheses). The lower values indicate that the patch obtained higher rankings on average by the evaluators.

Bugs	Human	PAR	GenProg
Math #280	**1.92** (0.76)	2.00 (0.82)	2.08 (0.95)
Rhino #114493	**1.60** (0.63)	2.40 (0.74)	2.00 (0.93)
Rhino #192226	2.00 (0.68)	**1.79** (0.98)	2.21 (0.80)
Rhino #217379	**1.62** (0.77)	1.69 (0.63)	2.69 (0.63)
Rhino #76683	1.92 (0.64)	**1.23** (0.43)	2.85 (0.38)
Average	1.81 (0.70)	1.82 (0.80)	2.36 (0.90)

- Result – Developers
  - Similarly PAR ranked higher than GenProg patches except one
  - Average ranking of PAR patches = 1.82 (SD = 0.80)
  - Average ranking of GenProg patches = 2.36 (SD= 0.90)
  - Ranking differences between Par and GenProg are statistically significant (p-value =0.016 < 0.05)
  - Based on results, reject null hypothesis H1o for developer group

RQ2 (Acceptability)

- Results of comparative studies:
  - PAR patches consistently have higher rankings than GenProg patches
  - Results are statistically significant
  - Implication? PAR can generate more acceptable patches than GenProg
  - Ranking differences not statistically significant between Par and human written patches
  - Implication? Patches generated by PAR are comparable to human written patches

RQ2 (Acceptability)

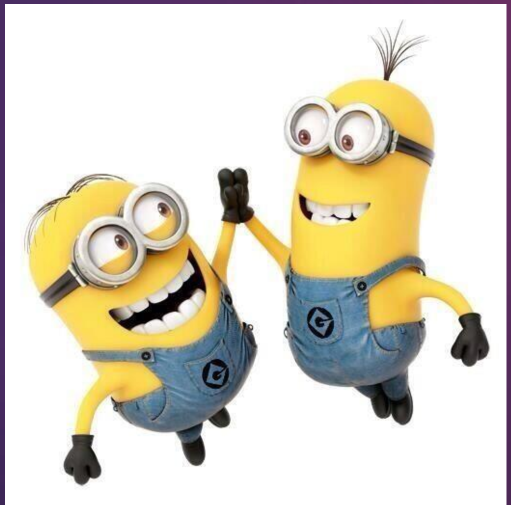
TABLE VII: Indirect patch comparison results.

Selection	# response
PAR	130 (21%)
Both	175 (28%)
Human	229 (37%)
Not Sure	87 (14%)
Total	621 (100%)

(a) PAR comparison results.

Selection	# response
GenProg	68 (20%)
Both	40 (12%)
Human	176 (51%)
Not Sure	60 (17%)
Total	344 (100%)

(b) GenProg comparison results.



- Indirect Patch Comparison
  - Compare acceptability of all 43 patches (27 PAR, 16 GenProg) to human-written patches
  - Web online Survey
  - Each session showed anonymized patches (one human written and one corresponding PAR or GenProg)
  - Patches generated by PAR more acceptable (21% + 28%) than GenProg patches (20% + 12%)

What about the other 92?

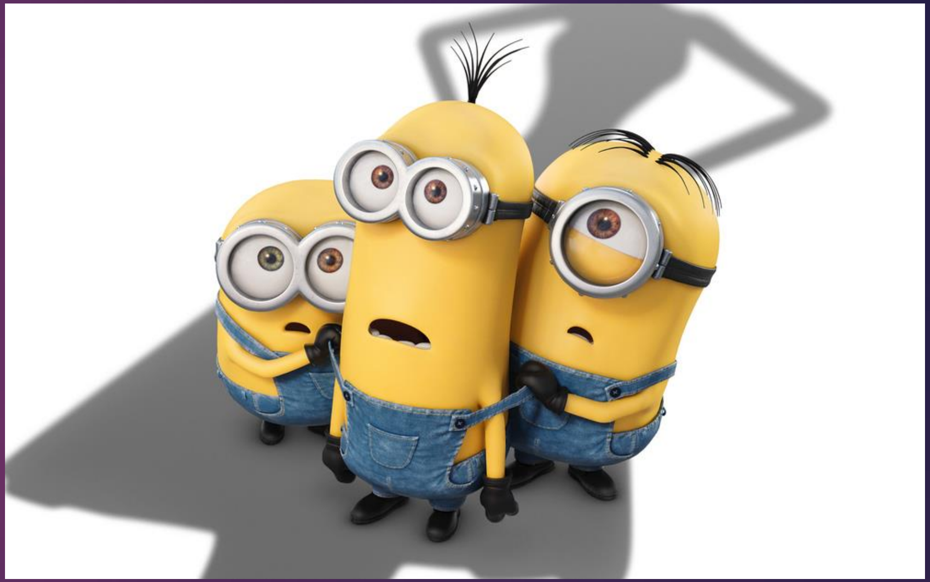
TABLE VIII: Causes of unsuccessful patches.

Cause	# of bugs
Branch condition	26 (28%)
No matching pattern	66 (72%)

- 92 out of 119 bugs not patched
- Branch conditions (28%)
  - Cannot generate predicates to satisfy branch conditions at fault locations by using fix templates
- No matching pattern (72%)
  - Cannot generate a patch because no fix template has appropriate editing scripts

Threats to Validity

- Systems are all open source projects
  - Patches of closed-source projects may have different patterns
- Some user studies participants may not be qualified
  - Could not verify qualifications of developers



....Conclusion!

- Manually inspected human-written patches and discovered common fix patterns
- Used fix patterns to generate automatic patches (PAR)
- Evaluated the patches against GenProg patches and human written patches
- PAR was more successful than GenProg generated 27 successful patches vs 16 by GenProg. PAR patches comparable to human-written patches.

- Future Work
  - Automatic fix template mining
  - Balance test case generation

Questions:

- Quiz!
- What is the scientific question? the answer?
- What's the key new idea that allows answering it?
- How do you measure the success of the answer?

Questions:

Can we identify additional Threats to Validity?

Questions:

- Which one do you think is more efficient? GenProg or PAR?

Questions:

Patch hunting within program

vs

Patch hunting outside of program

Which is better?

Questions:

- PAR vs GenProg
- Apples to apples comparison?

Sources used:

- “Automatic Patch Generation Learned from Human-Written Patches” by Kim et al
- “Automatically Finding Patches Using Genetic Programming” by Weimer et al
- “A Critical Review of “Automatic Patch Generation Learned from Human-Written Patches”: Essay on the Problem Statement and the Evaluation of Automatic Software Repair” by Martin Monperrus
- Several Minions and otherwise images from throughout the web
- One slide from Professor Yuriy Brun’s class!