Automatic Patch Generation Learned from Human Written Patches

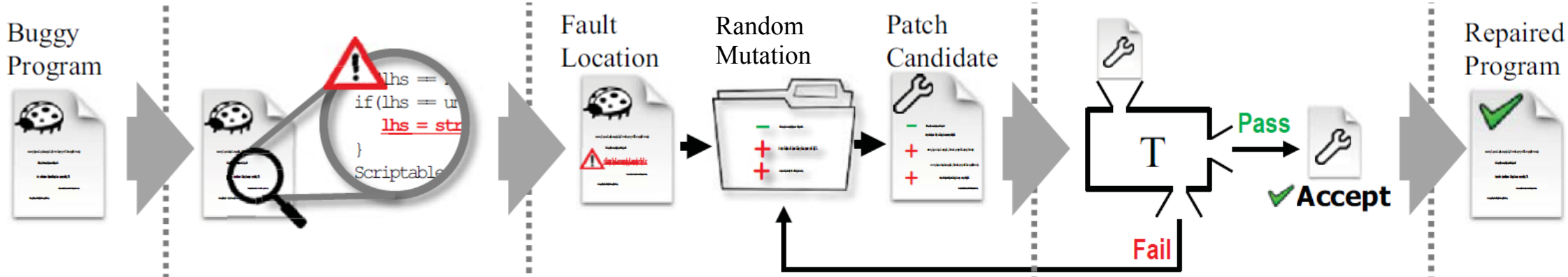
Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim

The Hong Kong University of Science and Technology, China

slide author names omitted for FERPA compliance

Introduction – Automatic Program Repair

- Two types of approaches
  - Testing-based (GenProg)
  - Specification-based
- GenProg



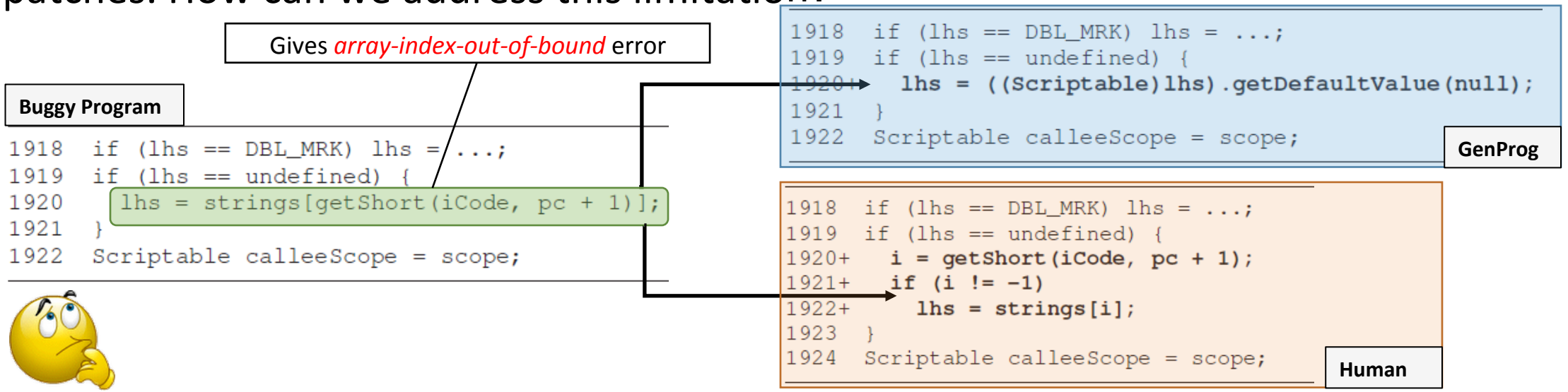
Identify suspicious statements in program (**Fault Localization**)

Perform random mutation and crossover operations to modify the suspicious statements with other program statements to generate program variants

Test program variants on test suit. Keep generating variants unless one passes all test cases of test suit

Problem Statement

- Using random mutation and crossover operations generates many nonsensical patches. How can we address this limitation?



- How about analyzing existing human-written patches to identify common patterns and use them to generate patches?

Creating Patterns – Manual Analysis of Human-written patches

- Analyzed 62,656 Human-written patches from open-source project- Eclipse JDT
- Focus on Semantic rather than Syntactic changes
- Analysis Involved:
  1. Does patch involves any semantic changes?
  2. What is the root cause of the bug and resolution of the corresponding patch?
  3. Group similar patches considering 1 and 2
- Reduce manual inspection time using *groums*

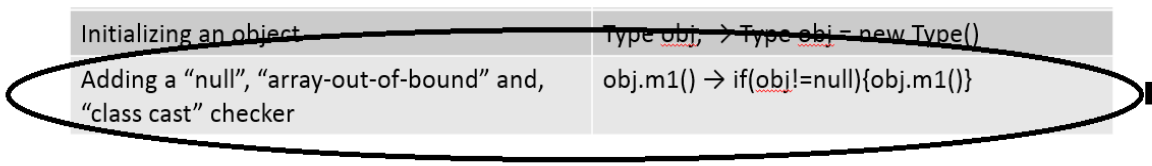
Fix Patterns

Pattern	Example
Altering method parameters	obj.method(v1,v2) → obj.method(v1,v3)
Calling another method with the same parameters	obj.method1(param) → obj.method2(param)
Calling another overloaded method with one more parameter	obj.method(v1) → obj.method(v1,v2)
Changing a branch condition	if(a == b) → if(a == b && c != 0)
Initializing an object	Type obj; → Type obj = new Type()
Adding a “null”, “array-out-of-bound” and, “class cast” checker	obj.m1() → if(obj!=null){obj.m1()}

Fix Template

- Derived from Fix pattern
- Find out the difference between Abstract Syntax Trees (ASTs) of a program before and after applying human patch
- Transform this difference into editing scripts
- Example

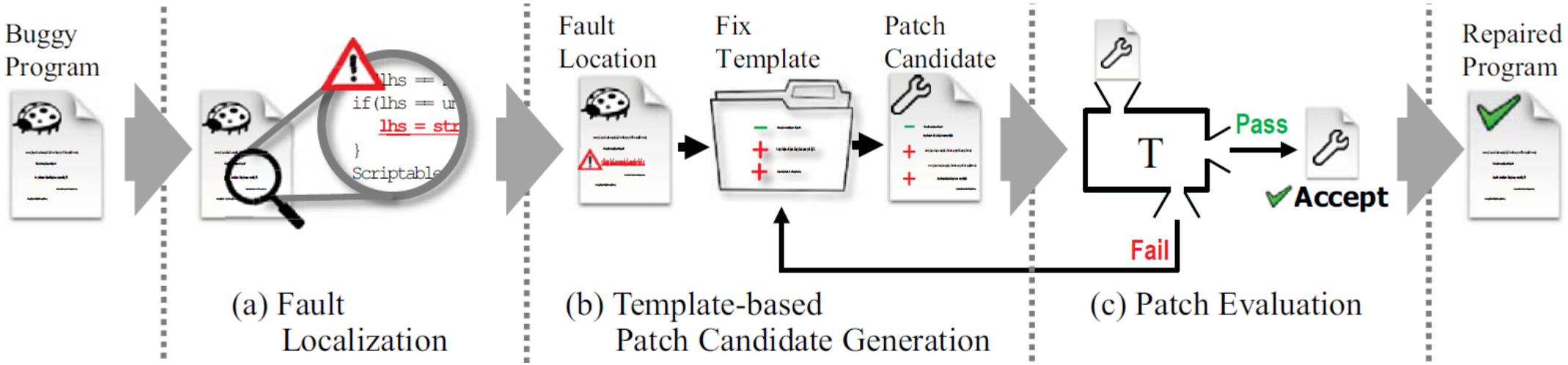
Initializing an object	Type obj; → Type obj = new Type()
Adding a “null”, “array-out-of-bound” and, “class cast” checker	obj.m1() → if(obj!=null){obj.m1()}



```
1  [Null Pointer Checker]
2  P = program
3  B = fault location
4
5  <AST Analysis>
6  C ← collect object references (method invocations,
     field accesses, and qualified names) of B in P
7
8  <Context Check>
9  if there is any object references in C ⇒ continue
10 otherwise ⇒ stop
11
12 <Program Editing>
13 insert an if() statement before B
14
15 loop for all objects in C {
16   insert a conditional expression that checks whether a
       given object is null
17 }
18 concatenate conditions by using AND
19
20 if B includes return statement {
21   negate the concatenated conditional expression
22   insert a return statement that returns a default value
       into THEN section of the if() statement
23   insert B after the if() statement
24 } else {
25   insert B into THEN section of the if() statement
26 }
```

Pattern-Based Automatic Program Repair (PAR)

- Analyze existing human-written patches to identify common *fix patterns*
- Utilize these *fix patterns* to create *Fix templates*
- Use *Fix templates* to automatically fix the bugs in code.



PAR-Example of Bug Repair

```
01  if (kidMatch != -1) return kidMatch;
02  for (int i = num; i < state.parenCount; i++)
03  {
04    state.parens[i].length = 0;
05  }
06  state.parenCount = num;
```

Suspicious Statement obtained using Fault Localization

Buggy Program

<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

Fix Template

```
01  if (kidMatch != -1) return kidMatch;
02  for ( ... )
03  {
04+     if( state != null && state.parens[i] != null)
05          state.parens[i].length = 0;
06  }
07  state.parenCount = num;
```

Repaired Program

Research Questions

- RQ1:How many bugs can the PAR fix? (Fixability)

- RQ2:Are patches generated by PAR sensible? (Acceptability)

RQ1:How many bugs can the PAR fix? (Fixability)

- Applied PAR to 119 real bugs from 5 open source Java projects (Rhino, AspectJ, log4J, Math, Lang, and Collections)
- PAR fixed 27 out of 119 bugs
- GenProg fixed 16 bugs.
- 5 bugs fixed by both techniques
- **PAR can fix more bugs than GenProg**

RQ2:Are patches generated by PAR sensible? (Acceptability)

- Conducted two user studies to compare the patch acceptability (89 students and 164 developers)
- Direct Patch Comparison Rankings (17 students, 68 developers, 5 patches)
  - 1.72(Human-written) vs 1.57 (PAR) vs 2.67(GenProg) in case of students
  - 1.81(Human-written) vs 1.82 (PAR) vs 2.36(GenProg) in case of developers
- Indirect Patch Comparison Results (72 students, 96 developers, 43 patches)
  - PAR acceptable in 305 (49%) out of 621 sessions (PAR: 21% + both: 28%)
  - GenProg acceptable in 108 (32%) out of 344 sessions (GenProg: 20% + both: 12%).
- **Patches generated by PAR are more Acceptable (comparable to human-written patches) than GenProg**

Problem Statement Solution

- Using random mutation and crossover operations generates many nonsensical patches. How can we address this limitation?

Gives *array-index-out-of-bound* error

Buggy Program

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

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

GenProg

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+   i = getShort(iCode, pc + 1);
1921+   if (i != -1)
1922+     lhs = strings[i];
1923  }
1924  Scriptable calleeScope = scope;
```

Human

- **Use PAR**

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+   if (getShort(iCode, pc + 1) < strings.length &&
          getShort(iCode, pc + 1) >= 0)
1921+   {
1922      lhs = strings[getShort(iCode, pc + 1)];
1923+   }
1924  }
1925  Scriptable calleeScope = scope;
```

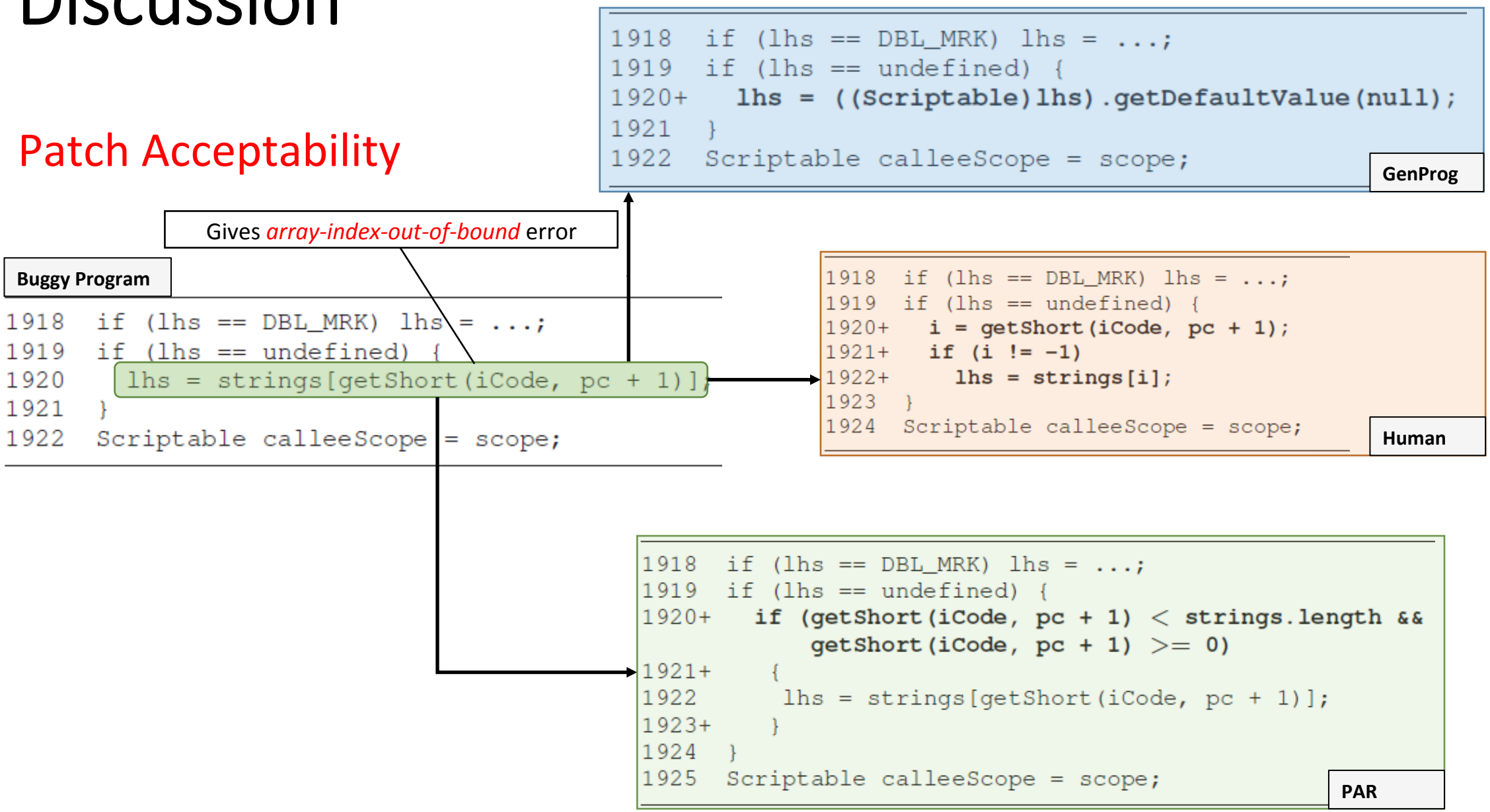
PAR

Major Contributions

- Manual inspection of human-written patches to identify and create common fix patterns.
- A novel patch generation approach – **Pattern-Based Automatic Program Repair (PAR)** that utilizes *Fix templates* derived from *Fix patterns*.
- Empirical evaluation of the approach by applying PAR to 119 real bugs and performing user study involving 89 students and 164 developers.

Discussion

Patch Acceptability



Discussion

Generalizability, Scalability of approach

- *Top 8 Fix patterns identified are found to be used in many (almost 30%) of the real patches*
- *patches of closed-source projects may have different patterns*

Discussion

Types of Bugs fixed, Validity of comparison with GenProg