Upcoming

- Homework 3 due April 18
- Literature review due today April 11

1

Repairing Automated Repair

2

Repairing Automated Repair

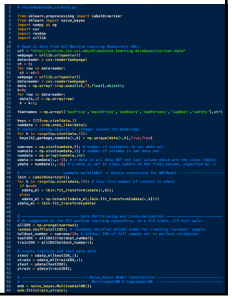
3

Cobra Effect

4

What do cobras have to do with automated program repair?

repairing python programs?

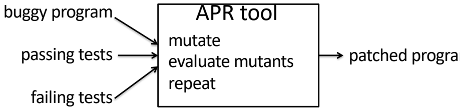


```
# thinkModelCode_carData.py

from sklearn.preprocessing import LabelBinarizer
from sklearn import naive_bayes
import numpy as np
import csv
import random
import urllib

# Read in data from UCI Machine Learning Repository URL:
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/car/car.data"
webpage = urllib.urlopen(url)
datareader = csv.reader(webpage)
ct = 0;
for row in datareader:
 ct = ct+1
webpage = urllib.urlopen(url)
datareader = csv.reader(webpage)
data = np.array(-1*np.ones((ct,7),float),object);
k=0;
for row in datareader:
 data[k,:] = np.array(row)
 k = k+1;

featnames = np.array(['buyPrice','maintPrice','numDoors','numPersons','lugBoot','safety'],str)

keys = [[]]*np.size(data,1)
numdata = -1*np.ones_like(data);
# convert string objects to integer values for modeling:
for k in range(np.size(data,1)):
 keys[k],garbage,numdata[:,k] = np.unique(data[:,k],True,True)

numrows = np.size(numdata,0); # number of instances in car data set
numcols = np.size(numdata,1); # number of columns in car data set
numdata = np.array(numdata,int)
xdata = numdata[:,:-1]; # x-data is all data BUT the last column which are the class labels
ydata = numdata[:,-1]; # y-data is set to class labels in the final column, signified by -1

# ---------------- numdata multilabel -> binary conversion for NB-Model ----------------
lbin = LabelBinarizer();
for k in range(np.size(xdata,1)): # loop thru number of columns in xdata
 if k==0:
  xdata_ml = lbin.fit_transform(xdata[:,k]);
 else:
  xdata_ml = np.hstack((xdata_ml,lbin.fit_transform(xdata[:,k])))
ydata_ml = lbin.fit_transform(ydata)


# ---------------------- Data Partitioning and Cross-Validation ----------------------
# As suggested by the UCI machine learning repository, do a 2/3 train, 1/3 test split
allIDX = np.arange(numrows);
random.shuffle(allIDX); # randomly shuffles allIDX order for creating 'holdout' sample
holdout_number = numrows/10; # holdout 10% of full sample set to perform validation
testIDX = allIDX[0:holdout_number];
trainIDX = allIDX[holdout_number:];

# create training and test data sets
xtest = xdata_ml[testIDX,:];
xtrain = xdata_ml[trainIDX,:];
ytest = ydata[testIDX];
ytrain = ydata[trainIDX];

# ------------------------------ Naive_Bayes Model Construction ------------------------------
# ------------------------------  MultinomialNB & ComplementNB  ------------------------------
mnb = naive_bayes.MultinomialNB();
mnb.fit(xtrain,ytrain);
```

5

Automated Program Repair

basic idea:



buggy program

passing tests

failing tests

APR tool

mutate
evaluate mutants
repeat

patched program

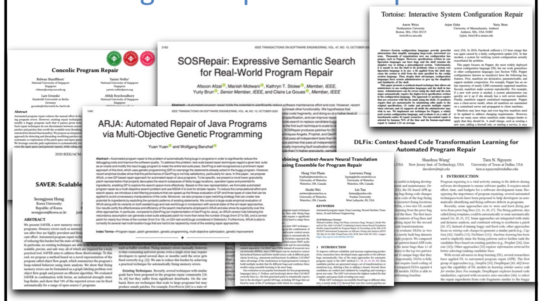
6

the many repair tools

ClearView [Perkinds et al. 2009] GenProg [Weimer et al. 2009]
Prophet [Long and Rinard 2015] SPR [Long and Rinard 2015]
TDS [Perelman et al. 2014]
Par [Kim et al. 2013] AE [Weimer et al. 2013]
SemFix [Nguyen et al. 2013] AutoFix-E [Wei et al. 2010]
[Carzaniga et al. 2010] [Carzaniga et al. 2013]
[Jin et al. 2011] Coker and Hafiz et al. 2013]
[Debroy and Wong et al. 2010] [Lin and Ernst et al. 2004]
[Forrest et al. 2009] [Novark et al. 2007] [Demsky et al. 2006]

7

Program repair techniques


Tortoise: Interactive System Configuration Repair

Concolic Program Repair

SOSRepair: Expressive Semantic Search for Real-World Program Repair

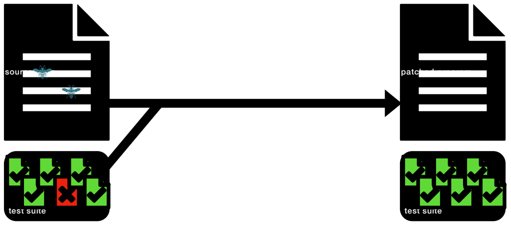
ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming

DLFix: Context-based Code Transformation Learning for Automated Program Repair

SAVER: Scalable

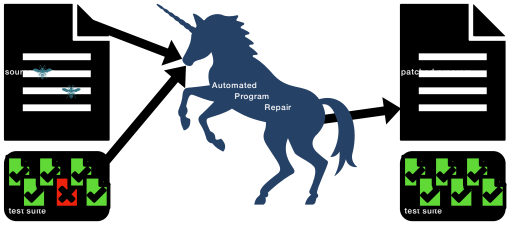
8

Automated program repair



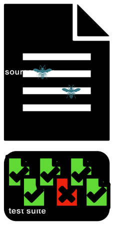
9

Automated program repair

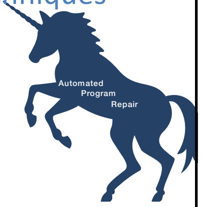


10

Program repair techniques

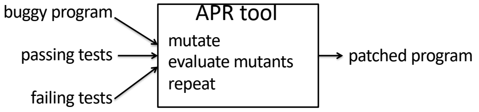


- Tweak the program
- Check if tests pass
- If not, repeat



11

Potential problem


buggy program

passing tests

failing tests

APR tool

mutate

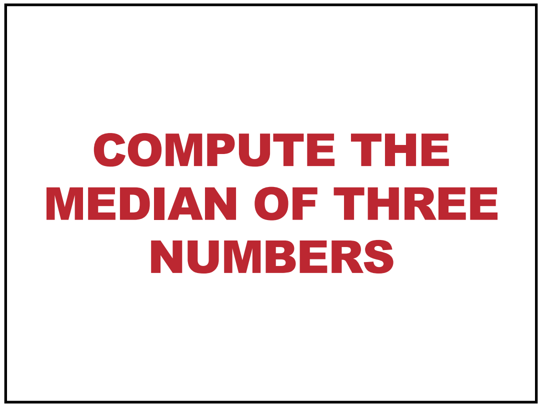
evaluate mutants

repeat

patched program

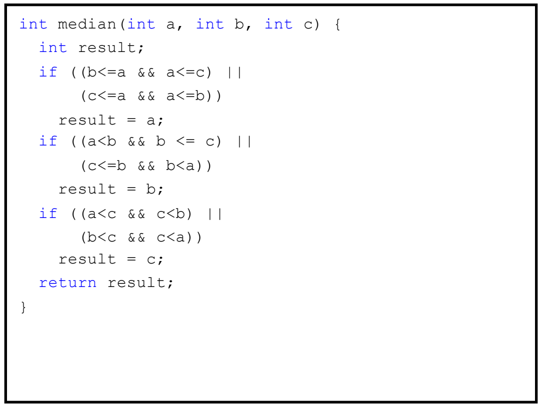
the patched program may pass all given tests, but break other functionality

12



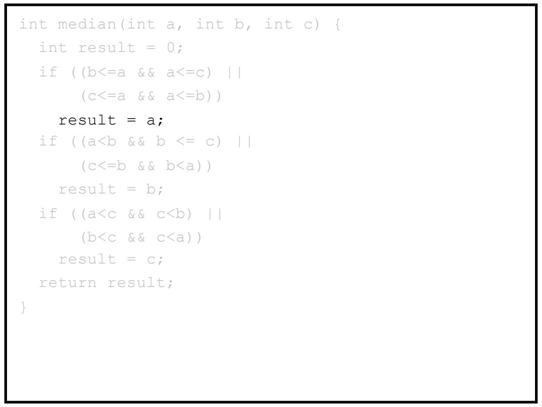
COMPUTE THE MEDIAN OF THREE NUMBERS

13



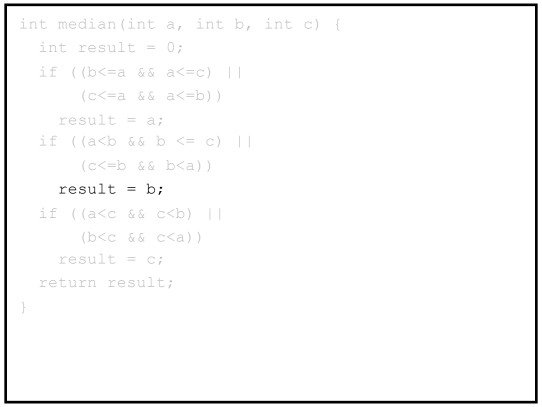
```
int median(int a, int b, int c) {
  int result;
  if ((b<=a && a<=c) ||
      (c<=a && a<=b))
    result = a;
  if ((a<b && b <= c) ||
      (c<=b && b<a))
    result = b;
  if ((a<c && c<b) ||
      (b<c && c<a))
    result = c;
  return result;
}
```

14



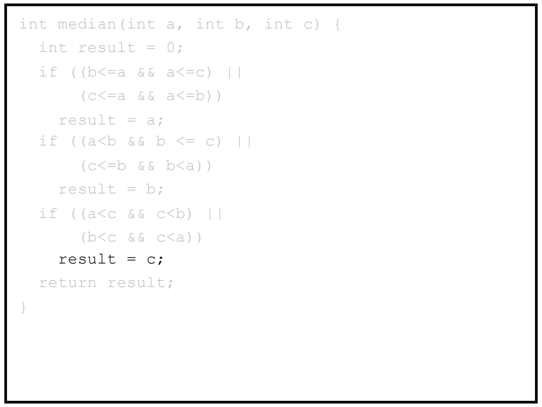
```
int median(int a, int b, int c) {
  int result = 0;
  if ((b<=a && a<=c) ||
      (c<=a && a<=b))
    result = a;
  if ((a<b && b <= c) ||
      (c<=b && b<a))
    result = b;
  if ((a<c && c<b) ||
      (b<c && c<a))
    result = c;
  return result;
}
```

15



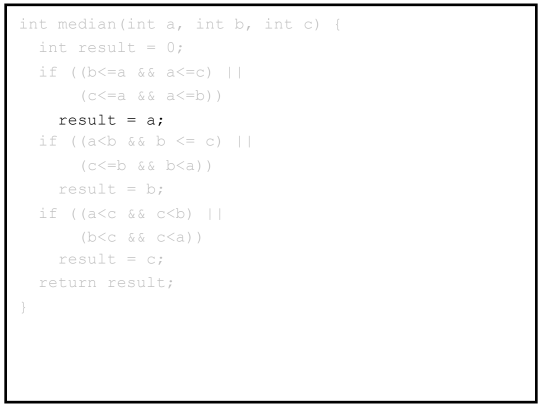
```
int median(int a, int b, int c) {
  int result = 0;
  if ((b<=a && a<=c) ||
      (c<=a && a<=b))
    result = a;
  if ((a<b && b <= c) ||
      (c<=b && b<a))
    result = b;
  if ((a<c && c<b) ||
      (b<c && c<a))
    result = c;
  return result;
}
```

16



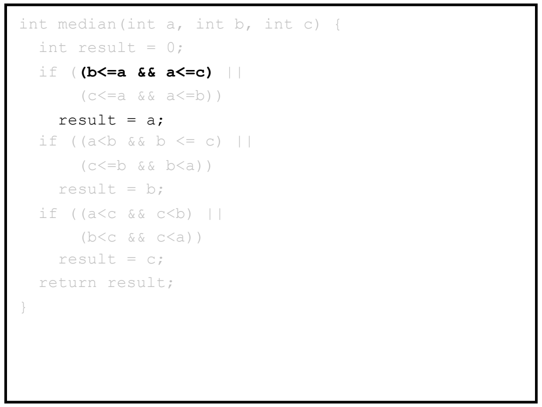
```
int median(int a, int b, int c) {
  int result = 0;
  if ((b<=a && a<=c) ||
      (c<=a && a<=b))
    result = a;
  if ((a<b && b <= c) ||
      (c<=b && b<a))
    result = b;
  if ((a<c && c<b) ||
      (b<c && c<a))
    result = c;
  return result;
}
```

17



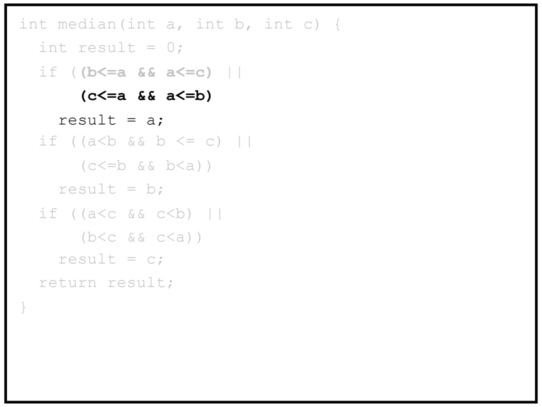
```
int median(int a, int b, int c) {
  int result = 0;
  if ((b<=a && a<=c) ||
      (c<=a && a<=b))
    result = a;
  if ((a<b && b <= c) ||
      (c<=b && b<a))
    result = b;
  if ((a<c && c<b) ||
      (b<c && c<a))
    result = c;
  return result;
}
```

18



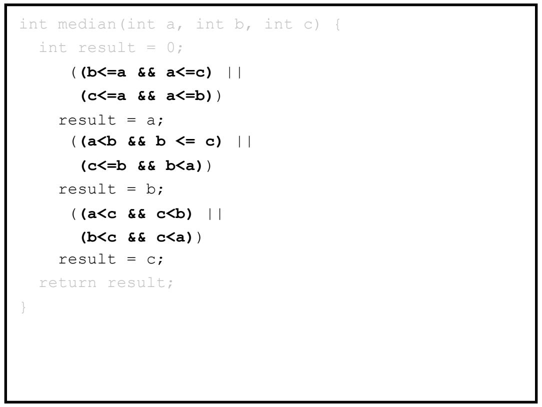
```
int median(int a, int b, int c) {
  int result = 0;
  if ((b<=a && a<=c) ||
      (c<=a && a<=b))
    result = a;
  if ((a<b && b <= c) ||
      (c<=b && b<a))
    result = b;
  if ((a<c && c<b) ||
      (b<c && c<a))
    result = c;
  return result;
}
```

19



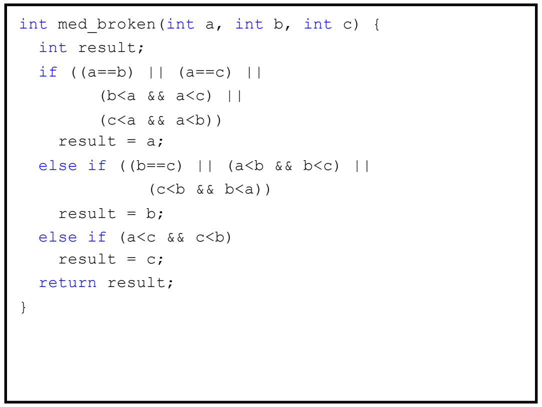
```
int median(int a, int b, int c) {
  int result = 0;
  if ((b<=a && a<=c) ||
      (c<=a && a<=b)
    result = a;
  if ((a<b && b <= c) ||
      (c<=b && b<a))
    result = b;
  if ((a<c && c<b) ||
      (b<c && c<a))
    result = c;
  return result;
}
```

20



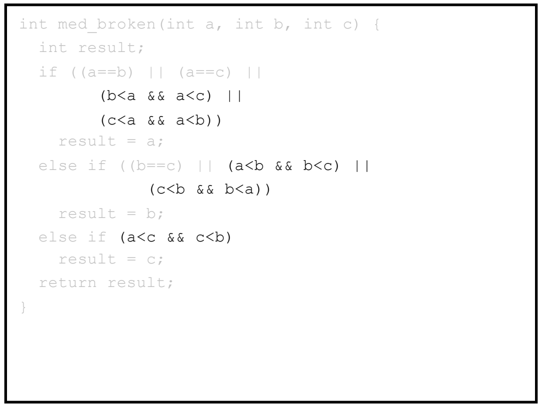
```
int median(int a, int b, int c) {
  int result = 0;
     ((b<=a && a<=c) ||
      (c<=a && a<=b))
    result = a;
     ((a<b && b <= c) ||
      (c<=b && b<a))
    result = b;
     ((a<c && c<b) ||
      (b<c && c<a))
    result = c;
  return result;
}
```

21



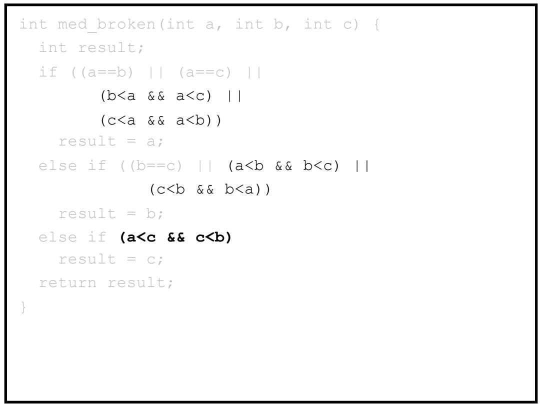
```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        (b<a && a<c) ||
        (c<a && a<b))
    result = a;
  else if ((b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  else if (a<c && c<b)
    result = c;
  return result;
}
```

22



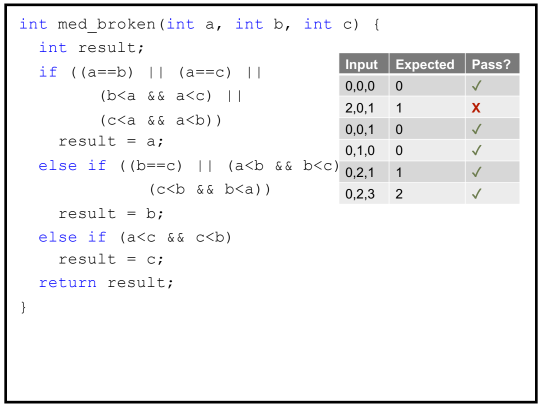
```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        (b<a && a<c) ||
        (c<a && a<b))
    result = a;
  else if ((b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  else if (a<c && c<b)
    result = c;
  return result;
}
```

23



```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        (b<a && a<c) ||
        (c<a && a<b))
    result = a;
  else if ((b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  else if (a<c && c<b)
    result = c;
  return result;
}
```

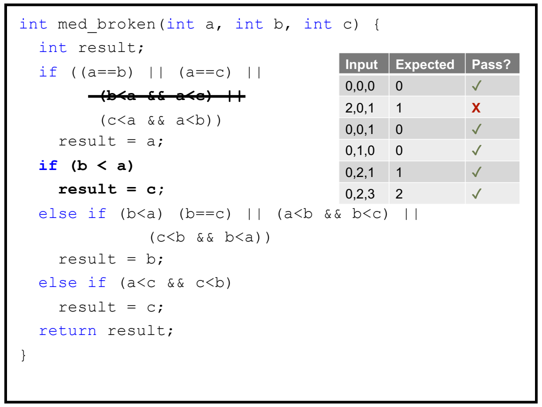
24



```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        (b<a && a<c) ||
        (c<a && a<b))
    result = a;
  else if ((b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  else if (a<c && c<b)
    result = c;
  return result;
}
```

Input	Expected	Pass?
0,0,0	0	✓
2,0,1	1	X
0,0,1	0	✓
0,1,0	0	✓
0,2,1	1	✓
0,2,3	2	✓

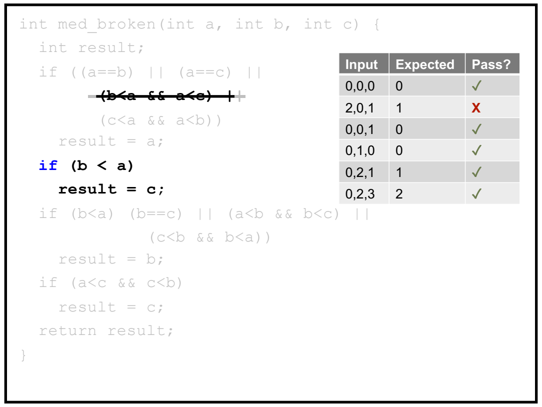
25



```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        ~~(b<a && a<c) ||~~
        (c<a && a<b))
    result = a;
  if (b < a)
    result = c;
  else if (b<a) (b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  else if (a<c && c<b)
    result = c;
  return result;
}
```

Input	Expected	Pass?
0,0,0	0	✓
2,0,1	1	X
0,0,1	0	✓
0,1,0	0	✓
0,2,1	1	✓
0,2,3	2	✓

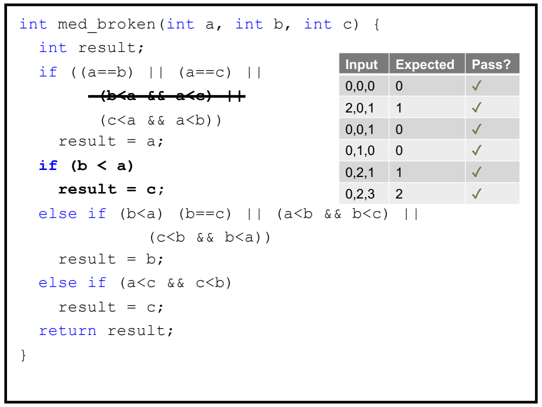
26



```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        ~~(b<a && a<c) ||~~
        (c<a && a<b))
    result = a;
  if (b < a)
    result = c;
  if (b<a) (b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  if (a<c && c<b)
    result = c;
  return result;
}
```

Input	Expected	Pass?
0,0,0	0	✓
2,0,1	1	X
0,0,1	0	✓
0,1,0	0	✓
0,2,1	1	✓
0,2,3	2	✓

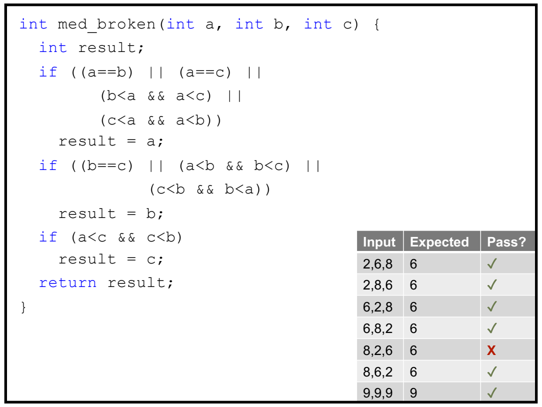
27



```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        ~~(b<a && a<c) ||~~
        (c<a && a<b))
    result = a;
  if (b < a)
    result = c;
  else if (b<a) (b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  else if (a<c && c<b)
    result = c;
  return result;
}
```

Input	Expected	Pass?
0,0,0	0	✓
2,0,1	1	✓
0,0,1	0	✓
0,1,0	0	✓
0,2,1	1	✓
0,2,3	2	✓

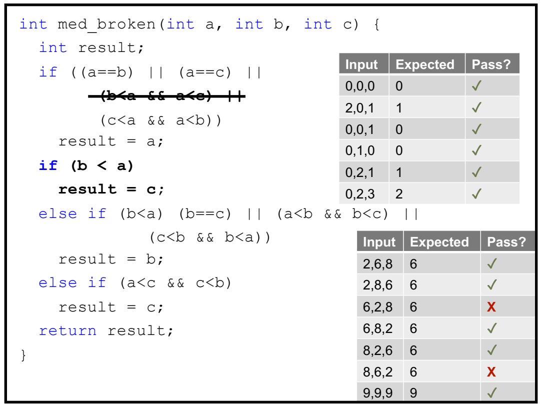
28



```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        (b<a && a<c) ||
        (c<a && a<b))
    result = a;
  if ((b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  if (a<c && c<b)
    result = c;
  return result;
}
```

Input	Expected	Pass?
2,6,8	6	✓
2,8,6	6	✓
6,2,8	6	✓
6,8,2	6	✓
8,2,6	6	X
8,6,2	6	✓
9,9,9	9	✓

29



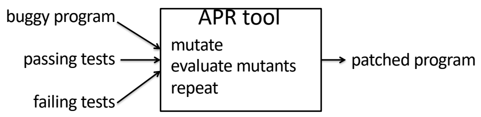
```
int med_broken(int a, int b, int c) {
  int result;
  if ((a==b) || (a==c) ||
        ~~(b<a && a<c) ||~~
        (c<a && a<b))
    result = a;
  if (b < a)
    result = c;
  else if (b<a) (b==c) || (a<b && b<c) ||
             (c<b && b<a))
    result = b;
  else if (a<c && c<b)
    result = c;
  return result;
}
```

Input	Expected	Pass?
0,0,0	0	✓
2,0,1	1	✓
0,0,1	0	✓
0,1,0	0	✓
0,2,1	1	✓
0,2,3	2	✓

Input	Expected	Pass?
2,6,8	6	✓
2,8,6	6	✓
6,2,8	6	X
6,8,2	6	✓
8,2,6	6	✓
8,6,2	6	X
9,9,9	9	✓

30

Potential solution



Use an independent test suite to measure quality of the patch

31

Focus of prior evaluations

- Most evaluations are interested in whether tools work
  - produce patches
- Some interest in other factors
  - human acceptance of patches [Durieux et al. 2015] [Fry et al. 2012] [Kim et al. 2013]
  - plausibility [Qi et al. 2015]
  - ...but these don't fully assess functional correctness
- No evaluations test functional correctness of repair outputs independently of repair inputs

32

What do we need?

- We need bugs with 2 test suites
  - and the test suites need to be good

Why?

- it's hard enough to find one good test suite, good luck finding programs with two

33

Make your own!

http://repairbenchmarks.cs.umass.edu

998 student-written buggy C programs

- simple (very small)
- have 2 test suites
  - white-box (generated by KLEE)
  - black-box (written by instructor)

Some programs fail some wb tests, others bb tests, others, some of both

34

RQ1: What is the base incidence of overfitting?

Give a repair tool the buggy program and the black-box test suite, try to repair it, see what fraction of the white-box tests the patches pass.

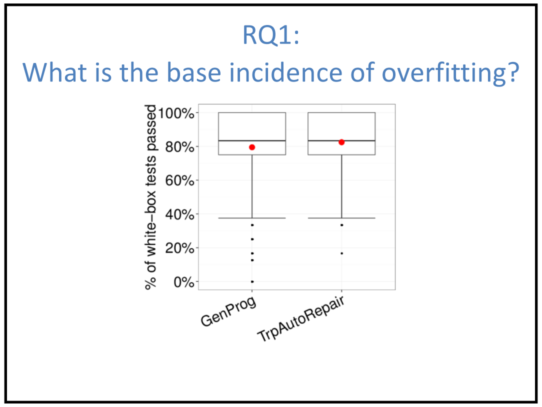
35

RQ1: What is the base incidence of overfitting?

but first, how often can we actually generate patches?

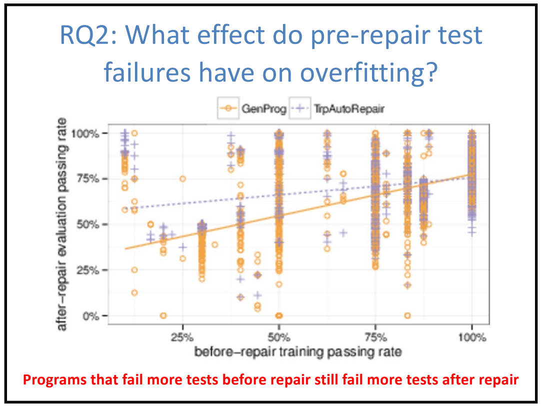
repair tool	patch production %
GenProg	466/778 = 59.9%
TrpAutoRepair	444/778 = 57.1%

36



RQ1: What is the base incidence of overfitting?

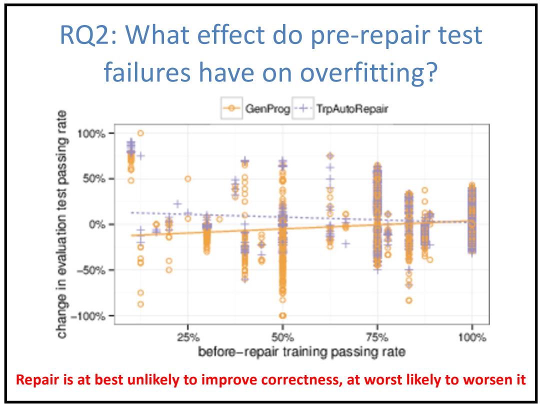
37



RQ2: What effect do pre-repair test failures have on overfitting?

Programs that fail more tests before repair still fail more tests after repair

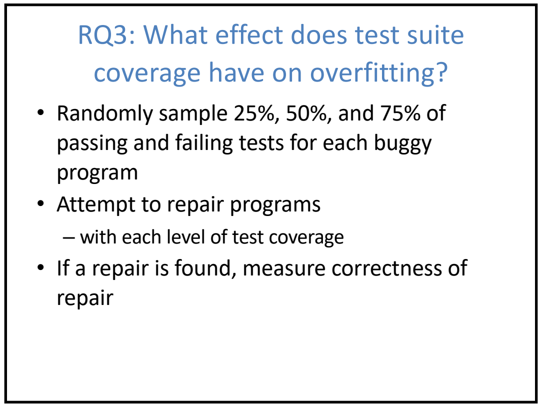
38



RQ2: What effect do pre-repair test failures have on overfitting?

Repair is at best unlikely to improve correctness, at worst likely to worsen it

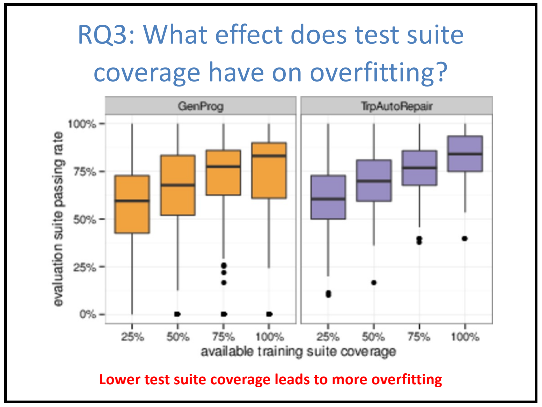
39



RQ3: What effect does test suite coverage have on overfitting?

- Randomly sample 25%, 50%, and 75% of passing and failing tests for each buggy program
- Attempt to repair programs
  - with each level of test coverage
- If a repair is found, measure correctness of repair

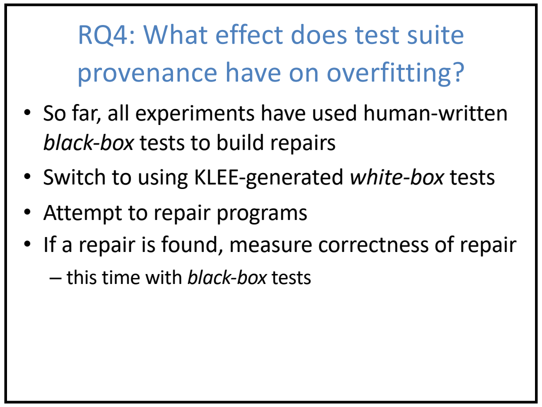
40



RQ3: What effect does test suite coverage have on overfitting?

Lower test suite coverage leads to more overfitting

41

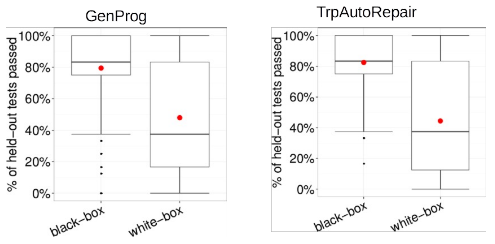


RQ4: What effect does test suite provenance have on overfitting?

- So far, all experiments have used human-written *black-box* tests to build repairs
- Switch to using KLEE-generated *white-box* tests
- Attempt to repair programs
- If a repair is found, measure correctness of repair
  - this time with *black-box* tests

42

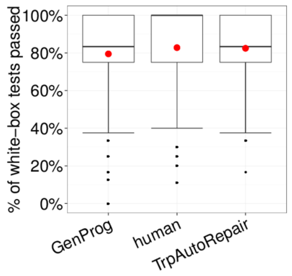
RQ4: What effect does test suite provenance have on overfitting?



Automatically generated tests produced significantly buggier repairs compared to human-written tests

43

RQ4: Do tools do better than novices?



44

Summary of that study

- Overfitting is a real concern
  - median patch for either tool passed only 75% of evaluation suite
- Overfitting is hard to avoid
  - minimization doesn't help on this dataset
  - N-version voting only works in extreme cases
- Program repair is harder for buggier programs, but likely to break more correct programs
- Novice developers don't significantly beat repair tools

45

How well does APR work?

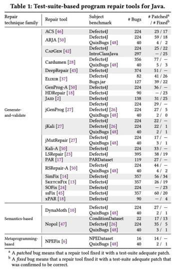
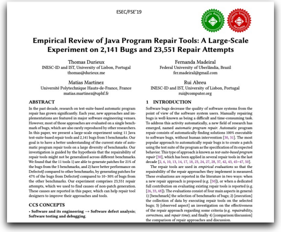


Table 1: Test-suite-based program repair tools for Java.

Repair technique family	Repair tool	Subject benchmark	# Bugs	# Patched[a] / # Fixed[b]
Generate-and-validate	ACS [46]	Defects4J	224	23 / 17
	ARJA [50]	Defects4J	224	59 / 18
		QuixBugs [48]	40	4 / 2
	CapGen [42]	Defects4J	224	25 / 22
		IntroClassJava	297	– / 25
	Cardumen [28]	Defects4J	356	77 / –
		QuixBugs [48]	40	5 / 3
	DeepRepair [43]	Defects4J	374	51 / –
	Elixir [37]	Defects4J	82	41 / 26
		Bugs.jar	127	39 / 22
	GenProg-A [50]	Defects4J	224	36 / –
	HDRepair [18]	Defects4J	90	– / 23
	Jaid [2]	Defects4J	138	31 / 25
	jGenProg [27]	Defects4J	224	29 / –
		Defects4J [26]	224	27 / 5
		QuixBugs [48]	40	2 / 0
	jKali [27]	Defects4J	224	22 / –
		Defects4J [26]	224	22 / 1
		QuixBugs [48]	40	2 / 1
	jMutRepair [27]	Defects4J	224	17 / –
		QuixBugs [48]	40	3 / 1
	Kali-A [50]	Defects4J	224	33 / –
	LSRepair [23]	Defects4J	395	38 / 19
	PAR [17]	PARDataset	119	27 / –
	RSRepair-A [50]	Defects4J	224	44 / –
		QuixBugs [48]	40	4 / 2
	SimFix [14]	Defects4J	357	56 / 34
	SketchFix [13]	Defects4J	357	26 / 19
	SOFix [24]	Defects4J	224	– / 23
	ssFix [45]	Defects4J	357	60 / 20
	xPAR [18]	Defects4J	90	– / 4
Semantics-based	DynaMoth [10]	Defects4J	224	27 / –
		QuixBugs [48]	40	2 / 1
	Nopol [47]	ConditionDataset	22	17 / 13
		Defects4J [26]	224	35 / 5
		QuixBugs [48]	40	3 / 1
Metaprogramming-based	NPEFix [6]	NPEDataset	16	14 / –
		QuixBugs [48]	40	2 / 1

[a] A *patched* bug means that a repair tool fixed it with a test-suite adequate patch.
[b] A *fixed* bug means that a repair tool fixed it with a test-suite adequate patch that was confirmed to be correct.



ESEC/FSE'19

Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts

Thomas Durieux
INESC-ID and IST, University of Lisbon, Portugal
thomas@durieux.me

Fernanda Madeiral
Federal University of Uberlândia, Brazil
fer.madeiral@gmail.com

Matias Martinez
Université Polytechnique Hauts-de-France, France
matias.martinez@uphf.fr

Rui Abreu
INESC-ID and IST, University of Lisbon, Portugal
rui@computer.org

ABSTRACT

In the past decade, research on test-suite-based automatic program repair has grown significantly. Each year, new approaches and implementations are featured in major software engineering venues. However, most of those approaches are evaluated on a single benchmark of bugs, which are also rarely reproduced by other researchers. In this paper, we present a large-scale experiment using 11 Java test-suite-based repair tools and 2,141 bugs from 5 benchmarks. Our goal is to have a better understanding of the current state of automatic program repair tools on a large diversity of benchmarks. Our investigation is guided by the hypothesis that the repairability of repair tools might not be generalized across different benchmarks. We found that the 11 tools 1) are able to generate patches for 21% of the bugs from the 5 benchmarks, and 2) have better performance on Defects4J compared to other benchmarks, by generating patches for 47% of the bugs from Defects4J compared to 10–30% of bugs from the other benchmarks. Our experiment comprises 23,551 repair attempts, which we used to find causes of non-patch generation. These causes are reported in this paper, which can help repair tool designers to improve their approaches and tools.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis; Software testing and debugging.**

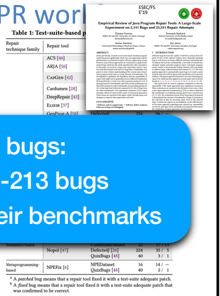
1 INTRODUCTION

Software bugs decrease the quality of software systems from the point of view of the software system users. Manually repairing bugs is well-known as being a difficult and time-consuming task. To address this activity automatically, a new field of research has emerged, named *automatic program repair*. Automatic program repair consists of automatically finding solutions 100% executable to software bugs, without human intervention [30, 31]. The most popular approach to automatically repair bugs is to create a patch using the test suite of the program as the specification of its expected behavior. This type of approach is known as *test-suite-based program repair* [30], which has been applied in several repair tools in the last decade [2, 6, 10, 13, 14, 17, 18, 23, 24, 27, 28, 37, 42, 43, 45–47, 50].

The repair tools are used in *empirical evaluations* so that the repairability of the repair approaches they implement is measured. These evaluations are reported in the literature in two ways: when a new repair approach is proposed (e.g. [50]), or when a dedicated full contribution on evaluating existing repair tools is reported (e.g. [26, 33, 48]). The evaluations consist of four main aspects in general: 1) [benchmark] the selection of benchmarks of bugs; 2) [execution] the collection of data by executing repair tools on the selected bugs; 3) [observed aspect] an investigation on the effectiveness of the repair approach regarding some criteria (e.g. *repairability*, *correctness*, and *repair time*); and finally 4) [comparison/discussion] the comparison of repair approaches and discussion.

46

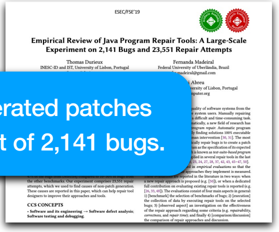
How well does APR work?



Results on 11 tools, 2,141 bugs:
- Generate patches for 15-213 bugs
- Evaluations overfit to their benchmarks

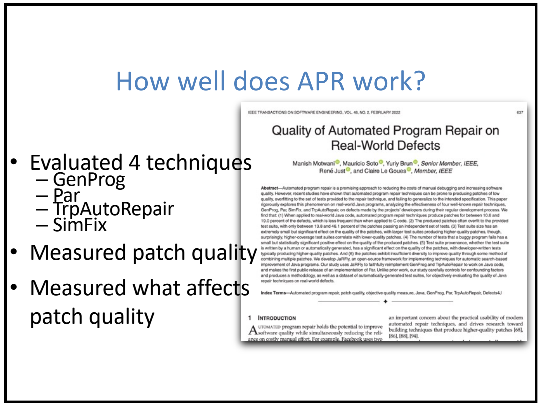
47

How well does APR work?



11 tools generated patches on 15-213 out of 2,141 bugs.

48



How well does APR work?

- Evaluated 4 techniques
  - GenProg
  - Par
  - TrpAutoRepair
  - SimFix
- Measured patch quality
- Measured what affects patch quality

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 48, NO. 2, FEBRUARY 2022 637

Quality of Automated Program Repair on Real-World Defects

Manish Motwani, Mauricio Soto, Yuriy Brun, *Senior Member, IEEE*, René Just, and Claire Le Goues, *Member, IEEE*

Abstract—Automated program repair is a promising approach to reducing the costs of manual debugging and increasing software quality. However, recent studies have shown that automated program repair techniques can be prone to producing patches of low quality, overfitting to the set of tests provided to the repair technique, and failing to generalize to the intended specification. This paper rigorously explores this phenomenon on real-world Java programs, analyzing the effectiveness of four well-known repair techniques, GenProg, Par, SimFix, and TrpAutoRepair, on defects made by the projects' developers during their regular development process. We find that: (1) When applied to real-world Java code, automated program repair techniques produce patches for between 10.6 and 19.0 percent of the defects, which is less frequent than when applied to C code. (2) The produced patches often overfit to the provided test suite, with only between 13.8 and 46.1 percent of the patches passing an independent set of tests. (3) Test suite size has an extremely small but significant effect on the quality of the patches, with larger test suites producing higher-quality patches, though, surprisingly, higher-coverage test suites correlate with lower-quality patches. (4) The number of tests that a buggy program fails has a small but statistically significant positive effect on the quality of the produced patches. (5) Test suite provenance, whether the test suite is written by a human or automatically generated, has a significant effect on the quality of the patches, with developer-written tests typically producing higher-quality patches. And (6) the patches exhibit insufficient diversity to improve quality through some method of combining multiple patches. We develop JaRFly, an open-source framework for implementing techniques for automatic search-based improvement of Java programs. Our study uses JaRFly to faithfully reimplement GenProg and TrpAutoRepair to work on Java code, and makes the first public release of an implementation of Par. Unlike prior work, our study carefully controls for confounding factors and produces a methodology, as well as a dataset of automatically-generated test suites, for objectively evaluating the quality of Java repair techniques on real-world defects.

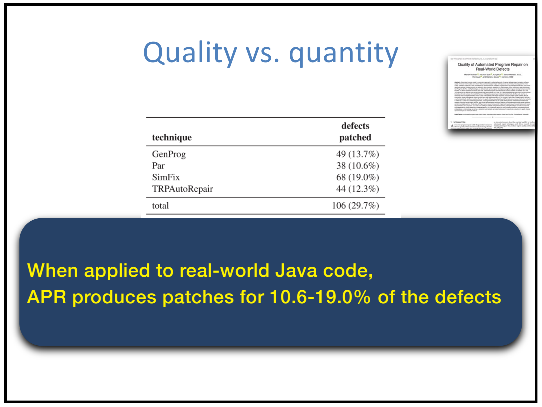
Index Terms—Automated program repair, patch quality, objective quality measure, Java, GenProg, Par, TrpAutoRepair, Defects4J

1 INTRODUCTION

AUTOMATED program repair holds the potential to improve software quality while simultaneously reducing the reliance on costly manual effort. For example, Facebook uses two

an important concern about the practical usability of modern automated repair techniques, and drives research toward building techniques that produce higher-quality patches [68], [86], [88], [94].

49

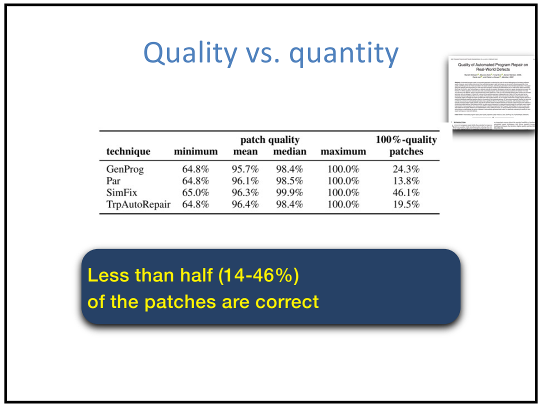


Quality vs. quantity

technique	defects patched
GenProg	49 (13.7%)
Par	38 (10.6%)
SimFix	68 (19.0%)
TRPAutoRepair	44 (12.3%)
total	106 (29.7%)

When applied to real-world Java code,
APR produces patches for 10.6-19.0% of the defects

50

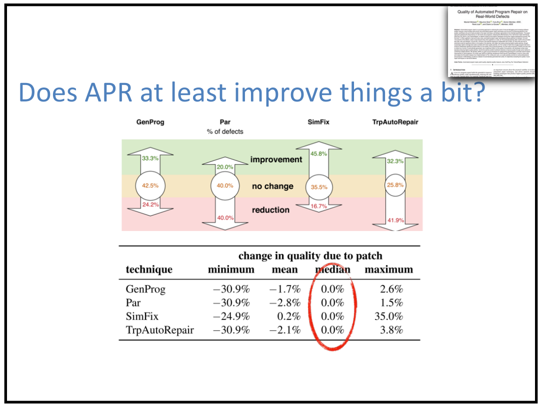


Quality vs. quantity

technique	minimum	patch quality mean	median	maximum	100%-quality patches
GenProg	64.8%	95.7%	98.4%	100.0%	24.3%
Par	64.8%	96.1%	98.5%	100.0%	13.8%
SimFix	65.0%	96.3%	99.9%	100.0%	46.1%
TrpAutoRepair	64.8%	96.4%	98.4%	100.0%	19.5%

Less than half (14-46%)
of the patches are correct

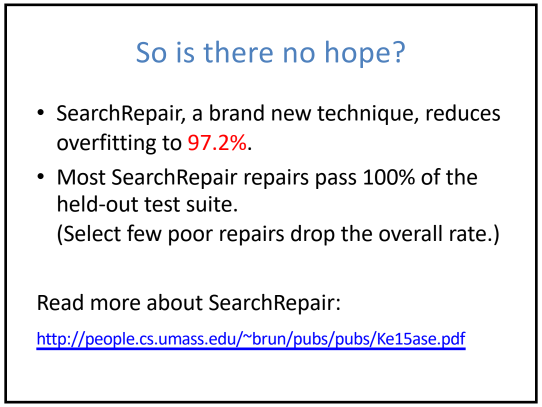
51



Does APR at least improve things a bit?

technique	minimum	change in quality due to patch mean	median	maximum
GenProg	−30.9%	−1.7%	0.0%	2.6%
Par	−30.9%	−2.8%	0.0%	1.5%
SimFix	−24.9%	0.2%	0.0%	35.0%
TrpAutoRepair	−30.9%	−2.1%	0.0%	3.8%

52



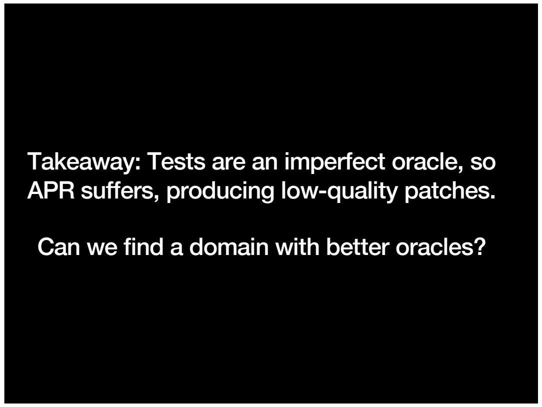
So is there no hope?

- SearchRepair, a brand new technique, reduces overfitting to 97.2%.
- Most SearchRepair repairs pass 100% of the held-out test suite.
  (Select few poor repairs drop the overall rate.)

Read more about SearchRepair:

http://people.cs.umass.edu/~brun/pubs/pubs/Ke15ase.pdf

53



Takeaway: Tests are an imperfect oracle, so APR suffers, producing low-quality patches.

Can we find a domain with better oracles?

54