Automatic Patch Generation

Claire {Le~Goues}

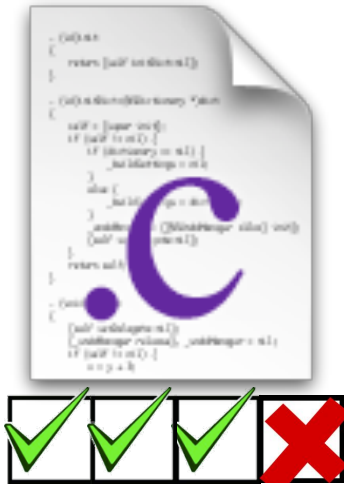
PWLConf; September 15, 2016

Carnegie Mellon University

institute for SOFTWARE RESEARCH

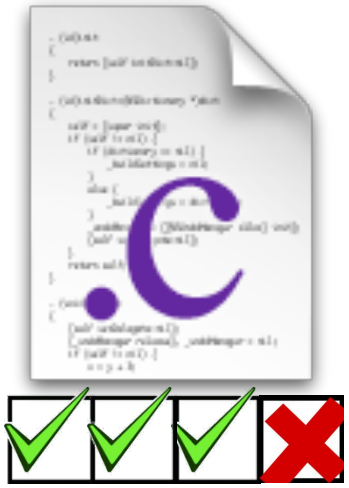
ONCE UPON A TIME...

Young Claire
(developer)



Bug report from a customer…

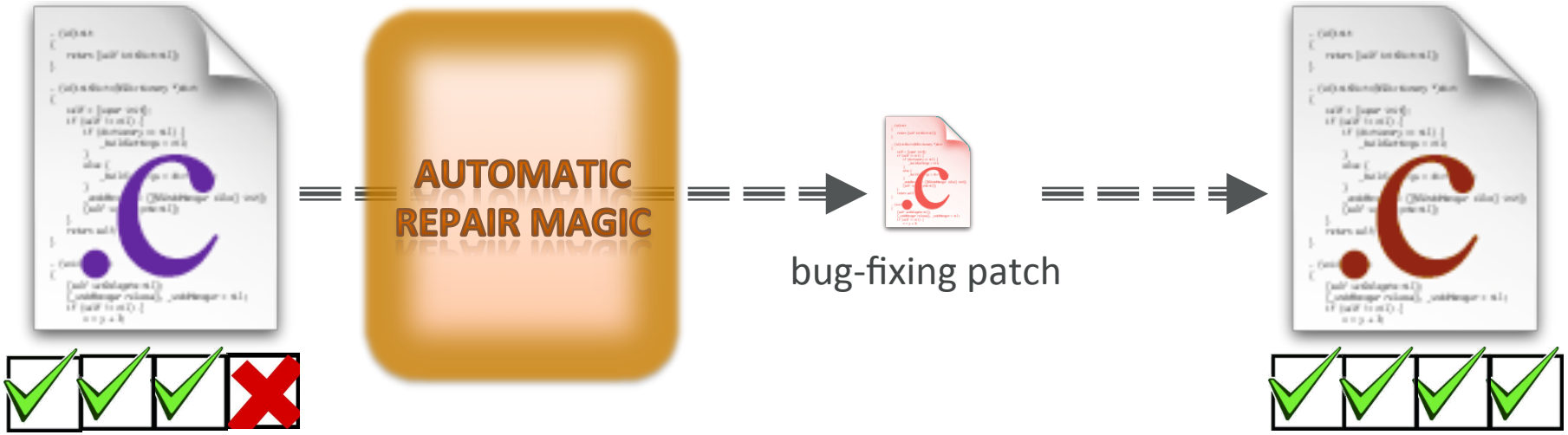
https://upload.wikimedia.org/wikipedia/commons/6/6e/Український_алфавіт.jpg



? ? !

32-64 bit transition
for Unicode encoding
of Ukrainian.

Problem: source-level defect repair

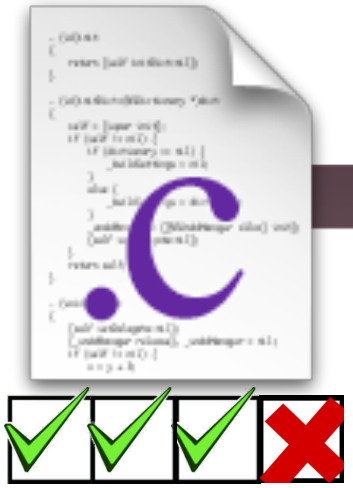


Bug fixing: the 30000-foot view

1. Localize the bug.
   - And possibly analyze it a little bit…
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patches.

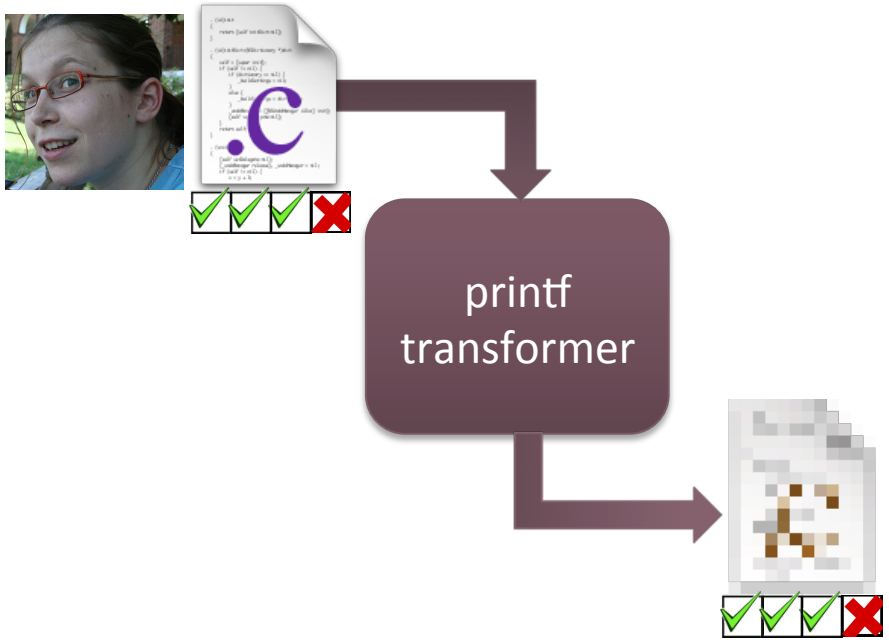
Bug fixing: the 30000-foot view

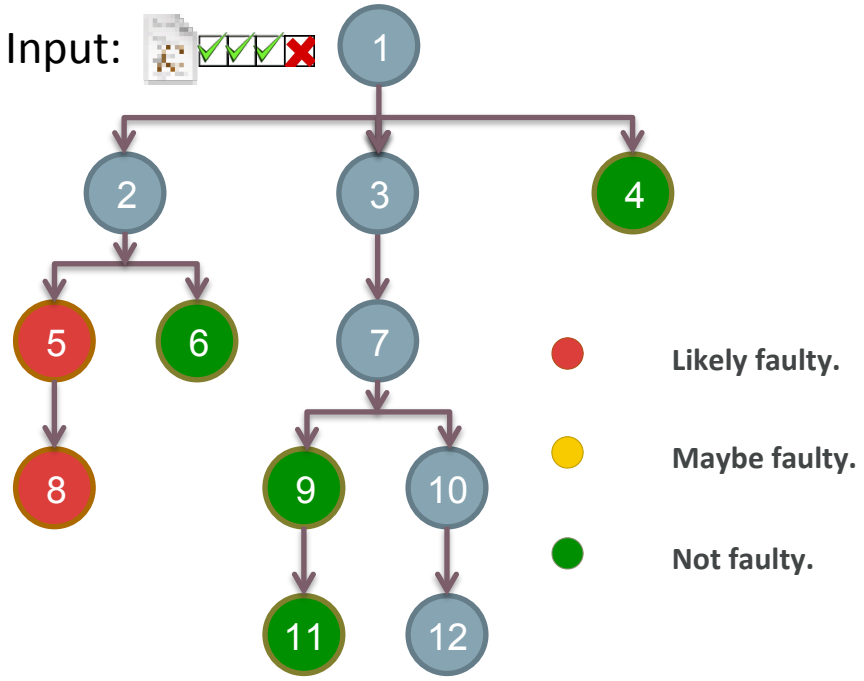
1. Localize the bug
   - And possibly analyze it a little bit…
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patches.



printf
transformer

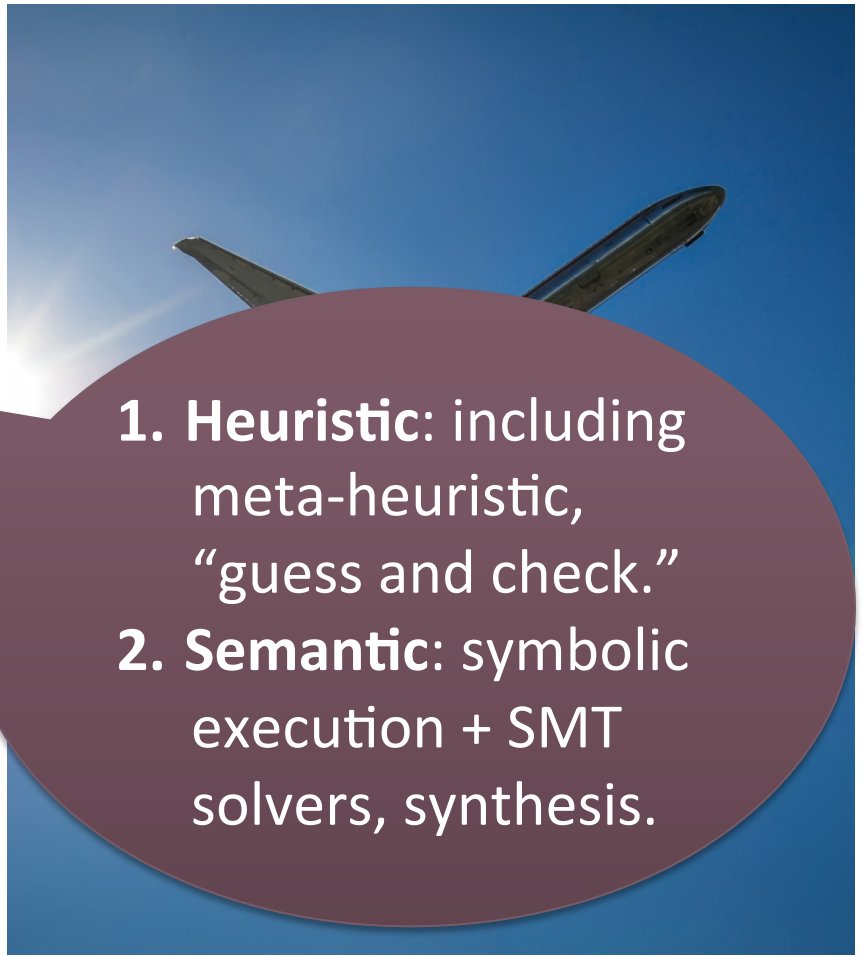
Spectrum-based fault localization automatically ranks potentially buggy program pieces based on test case behavior.





Bug fixing: the 30000-foot view

1. Localize the bug.
   - And possibly analyze it a little bit…
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patch.

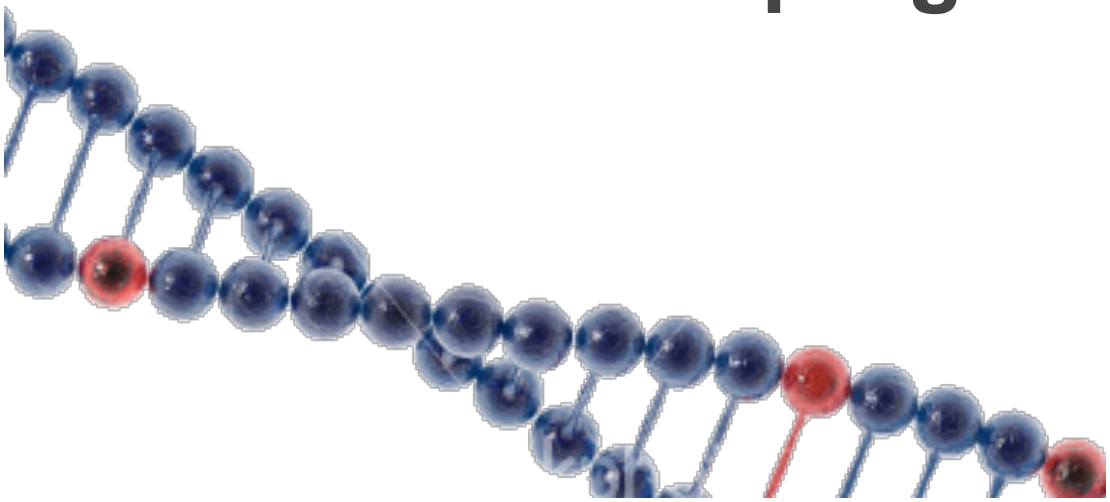


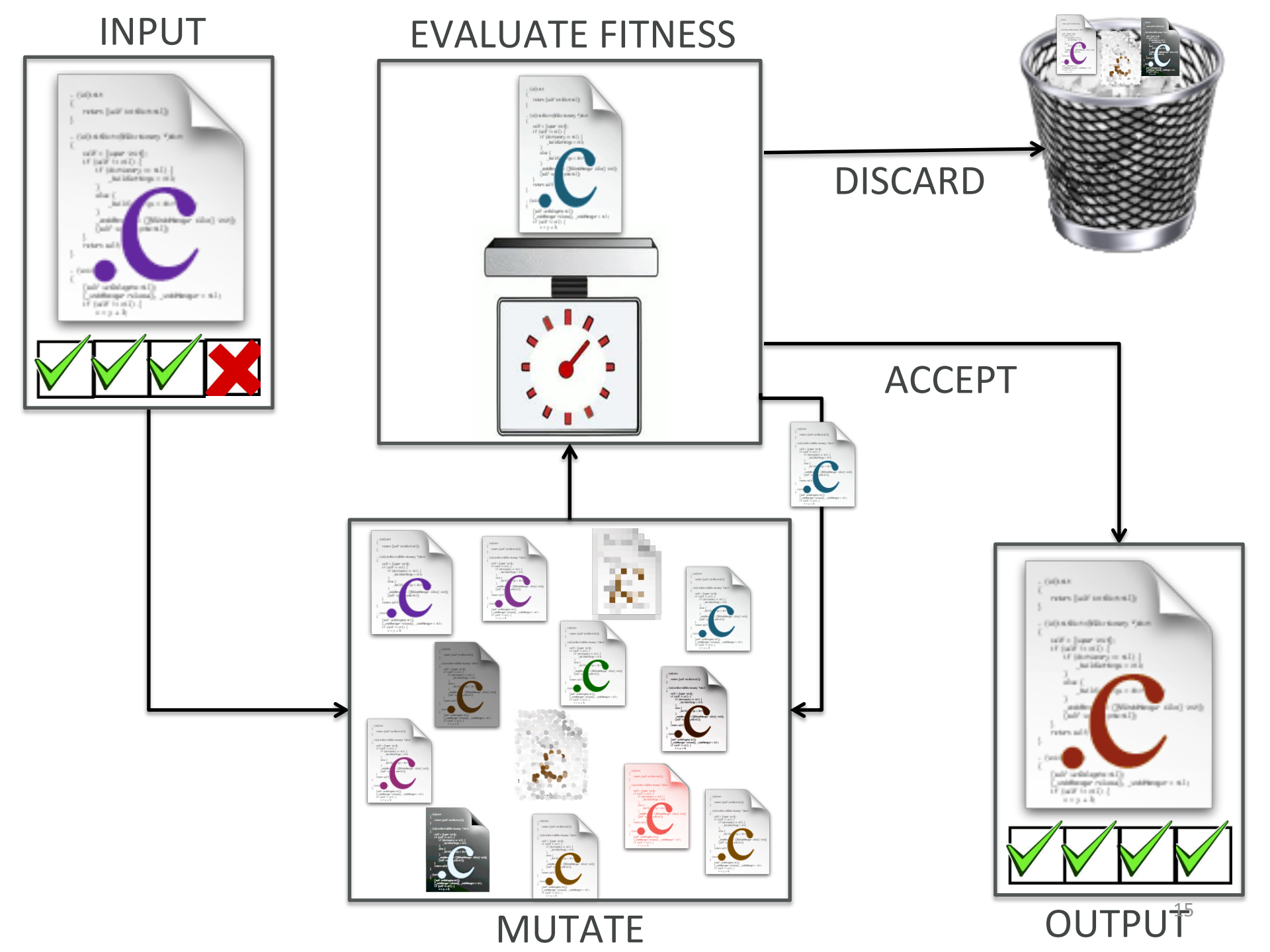
GenProg: automatic program repair using genetic programming.

Biased, random search for a AST-level edits to a program that fixes a given bug without breaking any previously-passing tests.

https://upload.wikimedia.org/wikipedia/commons/a/a4/13-02-27-spielbank-wiesbaden-by-RalfR-093.jpg

Genetic programming: the application of evolutionary or **genetic algorithms** to program source code.

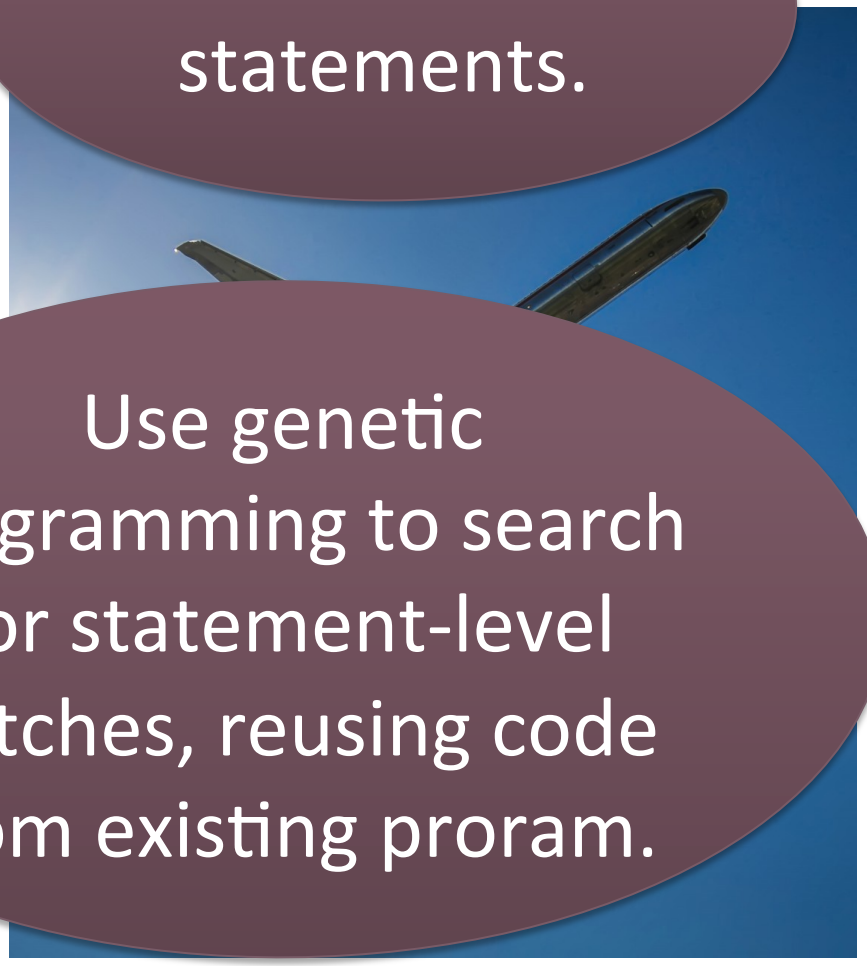



INPUT
EVALUATE FITNESS
DISCARD
ACCEPT
MUTATE
OUTPUT

GenProg: meta-heuri

1. Localize the bug.
   - And possibly analyze it a little bit…
2. Create/combine fix possibilities i possible patch
3. Validate candi patch.

Localize to C statements.



Use genetic programming to search for statement-level patches, reusing code from existing proram.

>

```
1  void gcd(int a, int b) {
2    if (a == 0) {
3      printf("%d", b);
4    }
5    while (b > 0) {
6      if (a > b)
7        a = a – b;
8      else
9        b = b – a;
10   }
11   printf("%d", a);
12   return;
13 }
```

```
> gcd(4,2)
> 2
>
> gcd(1071,1029)
> 21
>
> gcd(0,55)
> 55
```

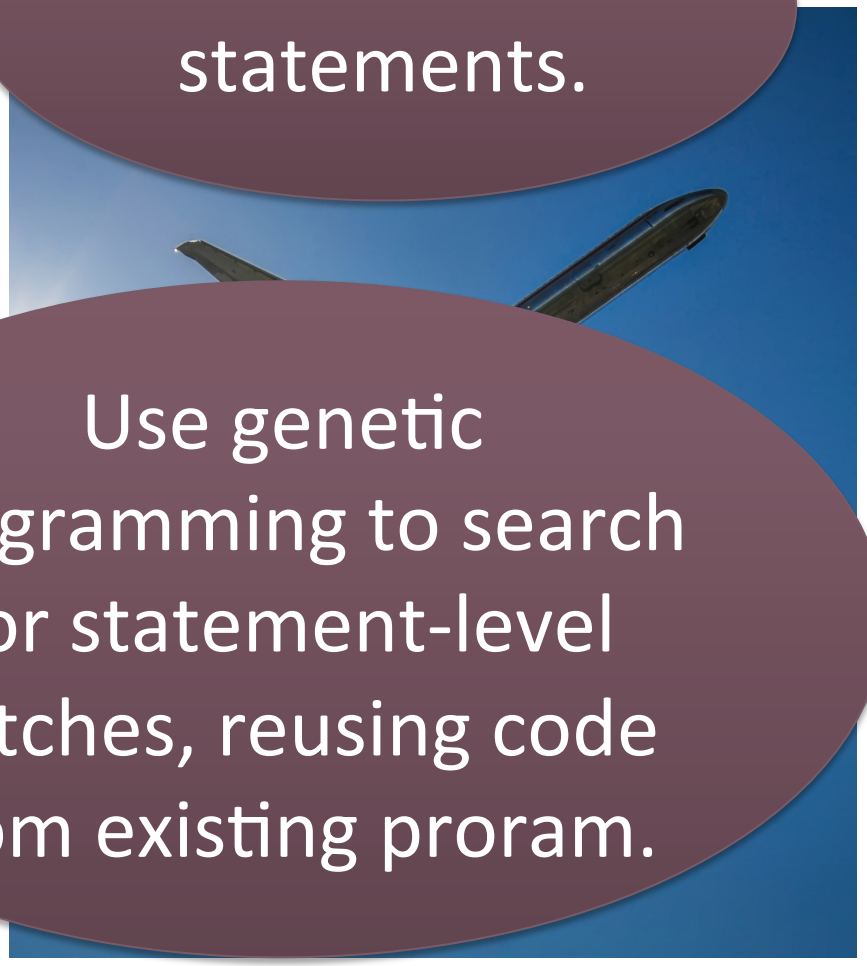
(looping forever)

```
1  void gcd(int a, int b) {
2    if (a == 0) {
3      printf("%d", b);
4    }
5    while (b > 0) {
6      if (a > b)
7        a = a – b;
8      else
9        b = b – a;
10   }
11   printf("%d", a);
12   return;
13 }
```

GenProg: meta-heuri

1. Localize the bug.
   - And possibly analyze it a little bit…
2. Create/combine fix possibilities i possible patch
3. Validate candi patch.

Localize to C statements.



Use genetic programming to search for statement-level patches, reusing code from existing proram.

Input:

{block}
if(a==0)
while (b>0)
printf(a)
return
{block}
{block}
{block}
printf(b)
if(a>b)
{block}
{block}
a = a - b
b = b - a

Input: 

Legend:

- High change probability.
- Low change probability.
- Not changed.

An individual is a candidate patch/set of changes to the input program.

- A patch is a series of *statement-level* edits:
  - delete X
  - replace X with Y
  - insert Y after X.
- Replace/insert: pick Y from somewhere else in the program.
- To mutate an individual, add new random edits to a given (possibly empty) patch.
  - (Where? Right: fault localization!)

Input: 

An edit is:

- Insert statement X after statement Y
- Replace statement X with statement Y
- Delete statement X

Input: 

An edit is:

- **Insert statement X after statement Y**
- Replace statement X with statement Y
- Delete statement X

Input: 

An edit is:

- **Insert statement X after statement Y**
- Replace statement X with statement Y
- Delete statement X

Input: 

An edit is:

- **Insert statement X after statement Y**
- Replace statement X with statement Y
- Delete statement X

What about Angelix?

1. Localize the bug.
   - And possibly analyze it a little bit…
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patch.

Same idea, but localizing to *expressions*.

RHS of assignments, conditionals.

```
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  }
```

Tremendous gratitude to Abhik Roychoudhury for sharing slides with me as starting material for this talk.

```
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  }
```

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	0	100	0	0	pass
1	***11***	***110***	***0***	***1***	***fail***
0	100	50	1	1	pass
1	***-20***	***60***	***0***	***1***	***fail***
0	0	10	0	0	pass

What about Angelix?

1. Localize the bug.
   - And possibly
     a little bit…
2. Create/combine fix
   possibilities into 1+
   possible patch
3. Validate candidate
   patch.

Concolic execution to find expression values that would make the test pass.

Program synthesis to construct replacement code that produces those values.

An expression's *angelic value* is the value that would make a given test case pass.

- This value is set "arbitrarily", by which we mean symbolically.
- You can *solve* for this value if you have:
  - the test case's expected input/output.
  - the path condition controlling its execution.
- Path condition: the set of conditions that controlled a particular execution.
  - Start executing the test concretely, and then switch to symbolic execution when the angelic value starts to matter.

```
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  }
```

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	0	100	0	0	pass
1	***11***	***110***	***0***	***1***	***fail***
0	100	50	1	1	pass
1	***-20***	***60***	***0***	***1***	***fail***
0	0	10	0	0	pass

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

inhibit	up_sep	down_sep	Observed output	Expected Output	Result
1	***11***	***110***	***0***	***1***	***fail***

inhibit = 1, up_sep = 11, down_sep = 110
bias = α, PC = true

Line 4

Line 7

inhibit = 1, up_sep = 11, down_sep = 110
bias = α, PC= α> 110

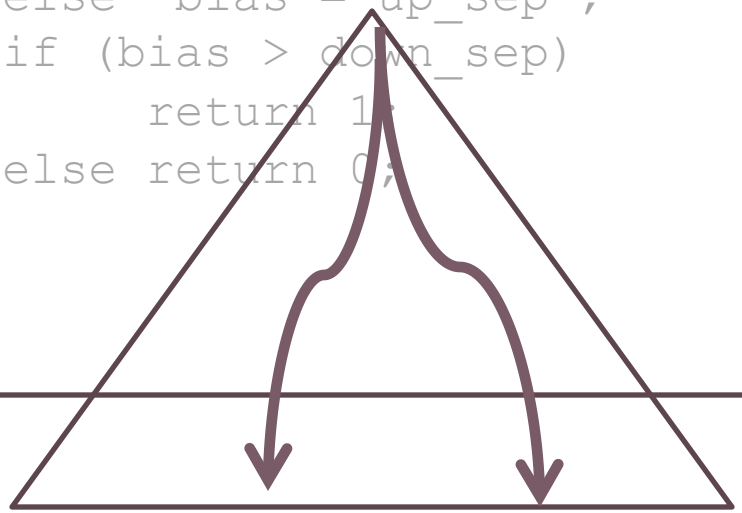
Line 8

inhibit = 1, up_sep = 11, down_sep = 110
bias =α, PC= α > 110

What should it have been?

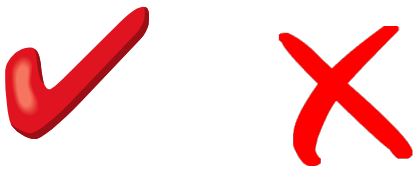
inhibit == 1	up_sep == 11	down_sep == 110

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



Symbolic Execution

f(1,11,110) > 110



Collect all of the constraints!

- Accumulated constraints over all test cases:

$$f(1,11,110) > 110 \wedge f(1,0,100) \leq 100 \wedge f(1,-20,60) > 60$$

- Use oracle guided component-based program synthesis to construct satisfying *f*:
  - Fix a set of of operators (component-based).
  - Synthesize code that only uses those operators and satisfies the constraints (oracle guided).
- Generated fix
  - `f(inhibit,up_sep,down_sep) = up_sep + 100`

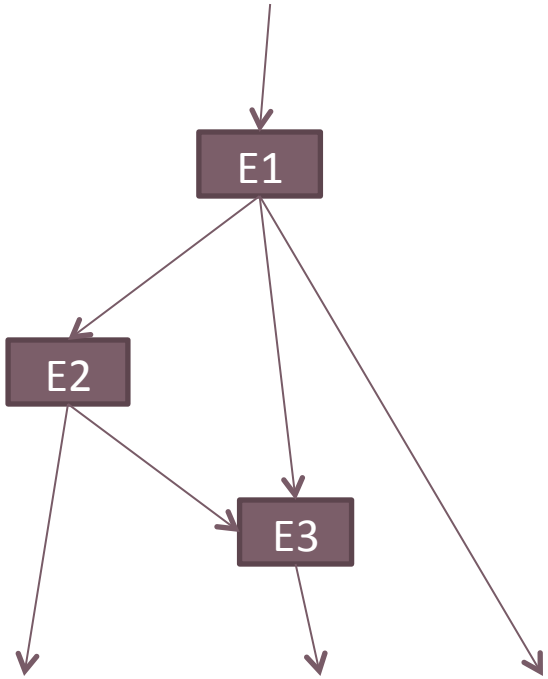
(Legitimately interesting encoding of synthesis problem elided for (dubious) brevity.)

So why all that attention paid to "forests"?

https://commons.wikimedia.org/wiki/File:Michael_Spiller_-_twisty_forest_paths_(by-sa).jpg

Angelic Forest

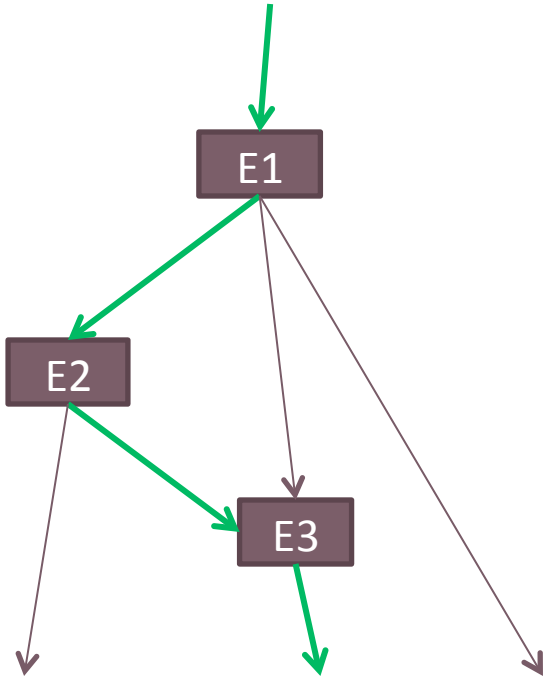
Program



Angelic Paths

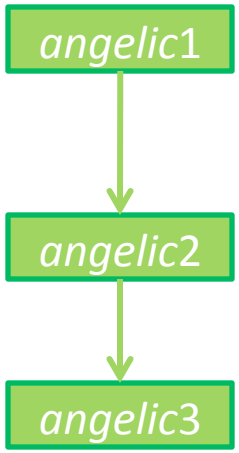
Angelic Forest

Program



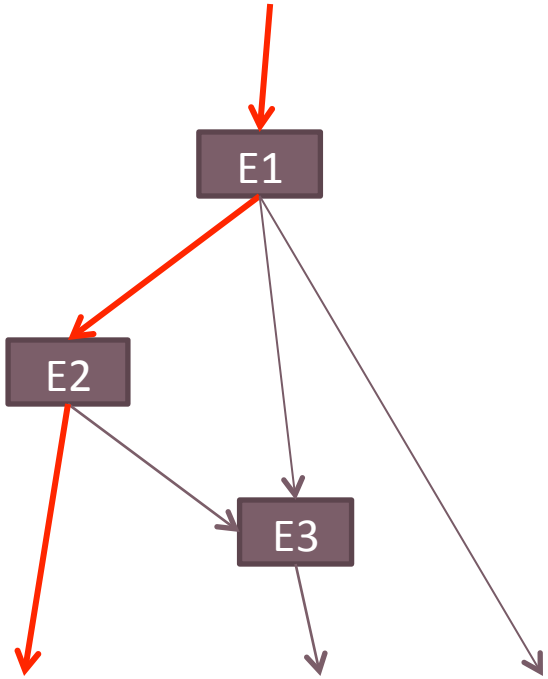
SAT

Angelic Paths

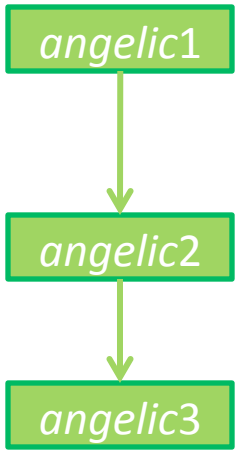


Angelic Forest

Program



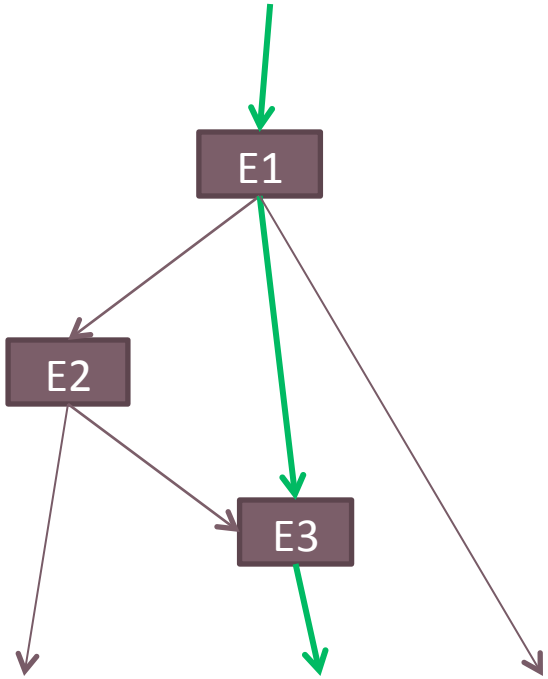
Angelic Paths



UNSAT

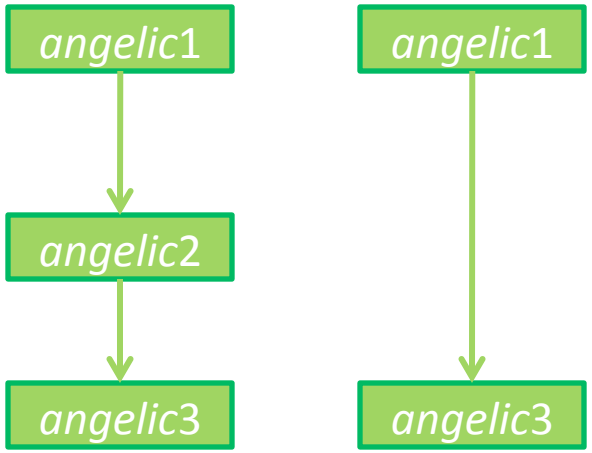
Angelic Forest

Program



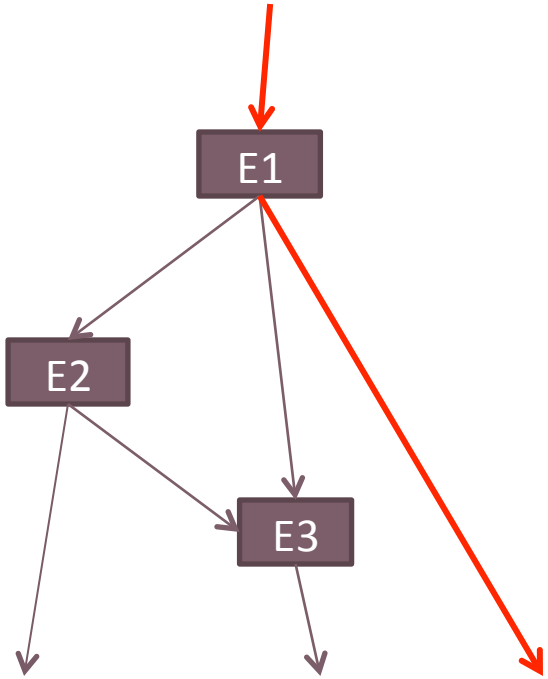
SAT

Angelic Paths

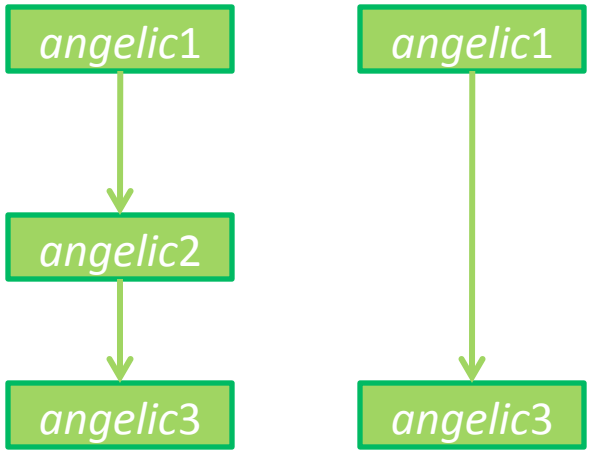


Angelic Forest

Program



Angelic Paths



UNSAT

Tradeoffs and Challenges

Scalability

Output quality

Expressive power

https://www.flickr.com/photos/86530412@N02/7935377706

: https://pixabay.com/en/approved-control-quality-stamp-147677/

https://www.flickr.com/photos/cimmyt/5219256862

Program	Description	LOC	Bug Type	Time (s)

Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578
zune	example	28	infinite loop	42
uniq	text processing	1146	segmentation fault	34
look-u	dictionary lookup	1169	segmentation fault	45
look-s	dictionary lookup	1363	infinite loop	55
units	metric conversion	1504	segmentation fault	109
deroff	document processing	2236	segmentation fault	131
indent	code processing	9906	infinite loop	546
flex	lexical analyzer generator	18774	segmentation fault	230
openldap	directory protocol	292598	non-overflow denial of service	665
ccrypt	encryption utility	7515	segmentation fault	330
lighttpd	webserver	51895	heap buffer overflow (vars)	394
atris	graphical game	21553	local stack buffer exploit	80
php	scripting language	764489	integer overflow	56
wu-ftpd	FTP server	67029	format string vulnerability	2256
leukocyte	computational biology	6718	segmentation fault	360
tiff	image processing	84067	segmentation fault	108
imagemagick	image processing	450516	wrong output	2160

Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578
zune	example	28	infinite loop	42
uniq	text processing	1146	segmentation fault	34
look-u	dictionary lookup	1169	segmentation fault	45
look-s	dictionary lookup	1363	infinite loop	55
units	metric conversion	1504	segmentation fault	109
deroff	document processing	2236	segmentation fault	131
indent	code processing	9906	infinite loop	546
flex	lexical analyzer generator	18774	segmentation fault	230
openldap	directory protocol	292598	non-overflow denial of service	665
ccrypt	encryption utility	7515	segmentation fault	330
lighttpd	webserver	51895	heap buffer overflow (vars)	394
atris	graphical game	21553	local stack buffer exploit	80
php	scripting language	764489	integer overflow	56
wu-ftpd	FTP server	67029	format string vulnerability	2256
leukocyte	computational biology	6718	segmentation fault	360
tiff	image processing	84067	segmentation fault	108
imagemagick	image processing	450516	wrong output	2160

Program	Description	LOC	Bug Type	Time (s)
gcd	example	2	infinite loop	153
nullhttpd	webserver	557	heap buffer overflow (code)	578
zune	example	2	infinite loop	42
uniq	text processing	114	segmentation fault	34
look-u	dictionary lookup	116	segmentation fault	45
look-s	dictionary lookup	136	infinite loop	55
units	metric conversion	150	segmentation fault	109
deroff	document processing	223	segmentation fault	131
indent	code processing	990	infinite loop	546
flex	lexical analyzer generator	1877	segmentation fault	230
openldap	directory protocol	29259	non-overflow denial of service	665
ccrypt	encryption utility	751	segmentation fault	330
lighttpd	webserver	5189	heap buffer overflow (vars)	394
atris	graphical game	2155	local stack buffer exploit	80
php	scripting language	76448	integer overflow	56
wu-ftpd	FTP server	6702	format string vulnerability	2256
leukocyte	computational biology	671	segmentation fault	360
tiff	image processing	8406	segmentation fault	108
imagemagick	image processing	45051	wrong output	2160

Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578
zune	example	28	infinite loop	42
uniq	text processing	1146	segmentation fault	34
look-u	dictionary lookup	1169	segmentation fault	45
look-s	dictionary lookup	1363	infinite loop	55
units	metric conversion	1504	segmentation fault	109
deroff	document processing	2236	segmentation fault	131
indent	code processing	9906	infinite loop	546
flex	lexical analyzer generator	18774	segmentation fault	230
openldap	directory protocol	292598	non-overflow denial of service	665
ccrypt	encryption utility	7515	segmentation fault	330
lighttpd	webserver	51895	heap buffer overflow (vars)	394
atris	graphical game	21553	local stack buffer exploit	80
php	scripting language	764489	integer overflow	56
wu-ftpd	FTP server	67029	format string vulnerability	2256
leukocyte	computational biology	6718	segmentation fault	360
tiff	image processing	84067	segmentation fault	108
imagemagick	image processing	450516	wrong output	2160

“IF I GAVE YOU THE LAST 100 BUGS FROM <MY PROJECT>, HOW MANY COULD GENPROG *ACTUALLY* FIX?”

– MANY PEOPLE

Challenge: Indicative Bug Set

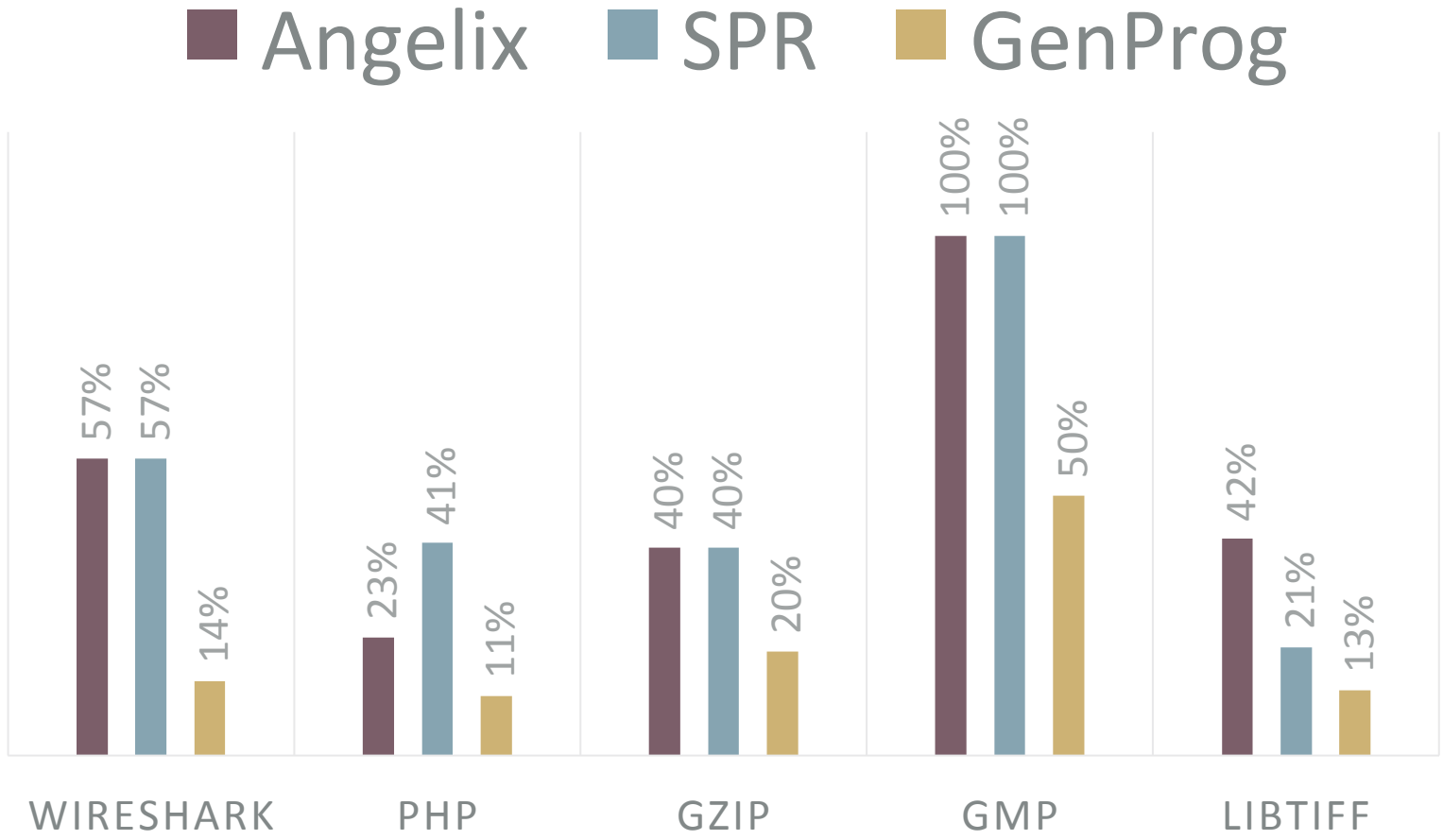
- Goal: a large set of important, reproducible bugs in non-trivial programs.
- Approach: use historical data (source control!) to approximate discovery and repair of bugs in the wild.

Success/Cost

Program	Defects Repaired	Cost per non-repair		Cost per repair	
		Hours	US$	Hours	US$
fbc	1/3	8.52	5.56	6.52	4.08
gmp	1/2	9.93	6.61	1.60	0.44
gzip	1/5	5.11	3.04	1.41	0.30
libtiff	17/24	7.81	5.04	1.05	0.04
lighttpd	5/9	10.79	7.25	1.34	0.25
php	28/44	13.00	8.80	1.84	0.62
python	1/11	13.00	8.80	1.22	0.16
wireshark	1/7	13.00	8.80	1.23	0.17
Total	**55/105**	**11.22h**		**1.60h**	

- $403 for all 105 trials, leading to 55 repairs; $7.32 per bug repaired.

Comparison (Repair-ability)



Heartbleed patch

Generated patch

```
if (hbtype == TLS1_HB_REQUEST
    && (payload + 18) < s->s3->rrec.length) {
    ...
} else if (hbtype == TLS1_HB_RESPONSE) {
    ...
}
return 0;
```

Developer patch

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
  return 0;
    ...
if (hbtype == TLS1_HB_REQUEST) {
    ...
} else if (hbtype == TLS1_HB_RESPONSE) {
    ...
}
return 0;
```

Scalability

Tradeoffs and Challenges

Output quality

Expressive power

← nullhttpd: a webserver with basic GET + POST functionality.
Version 0.5.0: remote-exploitable heap-based buffer overflow in handling of POST.

Failing test case: run exploit, see if webserver is still running

+

Easy passing test cases:
1. "GET index.html"
2. "GET image.jpg"
3. "GET notfound.html"

=

"delete handling of POST requests"

Flashback to 2008…

When we added a non-crashing test case for POST, proto-GenProg found a much better patch.

- When the test suite is your objective function, test suite quality matters.
  - …how much is a trickier issue.
- But we're begging the question...

https://en.wikipedia.org/wiki/Basket-hilted_sword#/media/File:Schiavona-Morges.jpg

https://en.wikipedia.org/wiki/Basket-hilted_sword#/media/File:Schiavona-Morges.jpg

When we added a non-crashing test case for POST, proto-GenProg found a much better patch.

- When the test suite is your objective function, test suite quality matters.
  - …how much is a trickier issue.
- But we're begging the question...

https://en.wikipedia.org/wiki/Basket-hilted_sword#/media/File:Schiavona-Morges.jpg

What is a high quality patch, anyway?

- Understandable?
  - Well, I had no problem understanding the POST-deleting patch…
  - (non-functional properties are important and being studied by others!)
- Doesn't delete?
  - But what about goto fail?
- Does the same thing the human did/would do?
  - But humans are often wrong! And how close does it have to be?
- Doesn't introduce new bugs?
  - How to tell?
- Addresses the cause, not the symptom…

Proposal: measure quality based on degree to which results *generalize*.

- In machine learning, techniques are trained and evaluated on disjoint datasets to assess overfitting.
- In program repair:
  - Tests used to build a repair are *training* tests
  - Tests used to assess correctness are *evaluation* tests

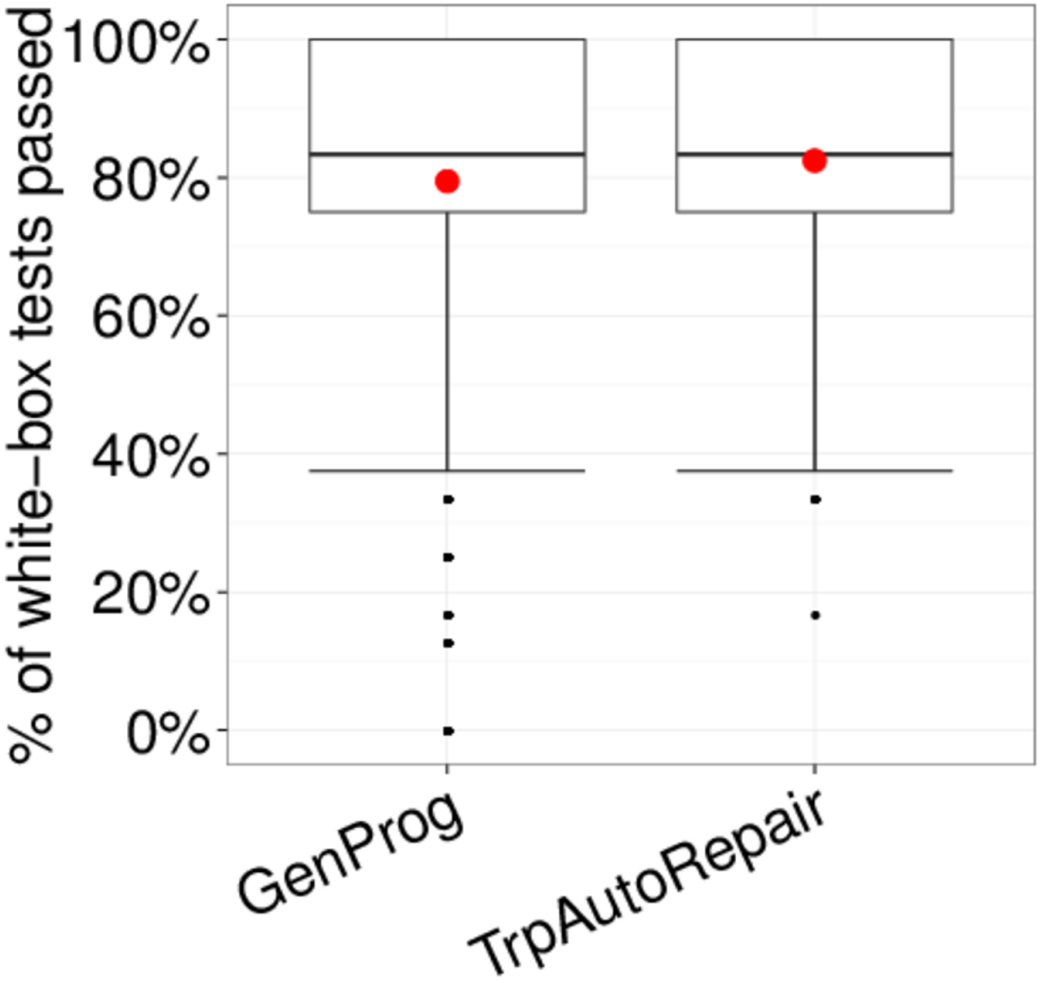
PROBLEM: THE DESIRED STUDY IS IMPOSSIBLE.

[Dataset + Tools]

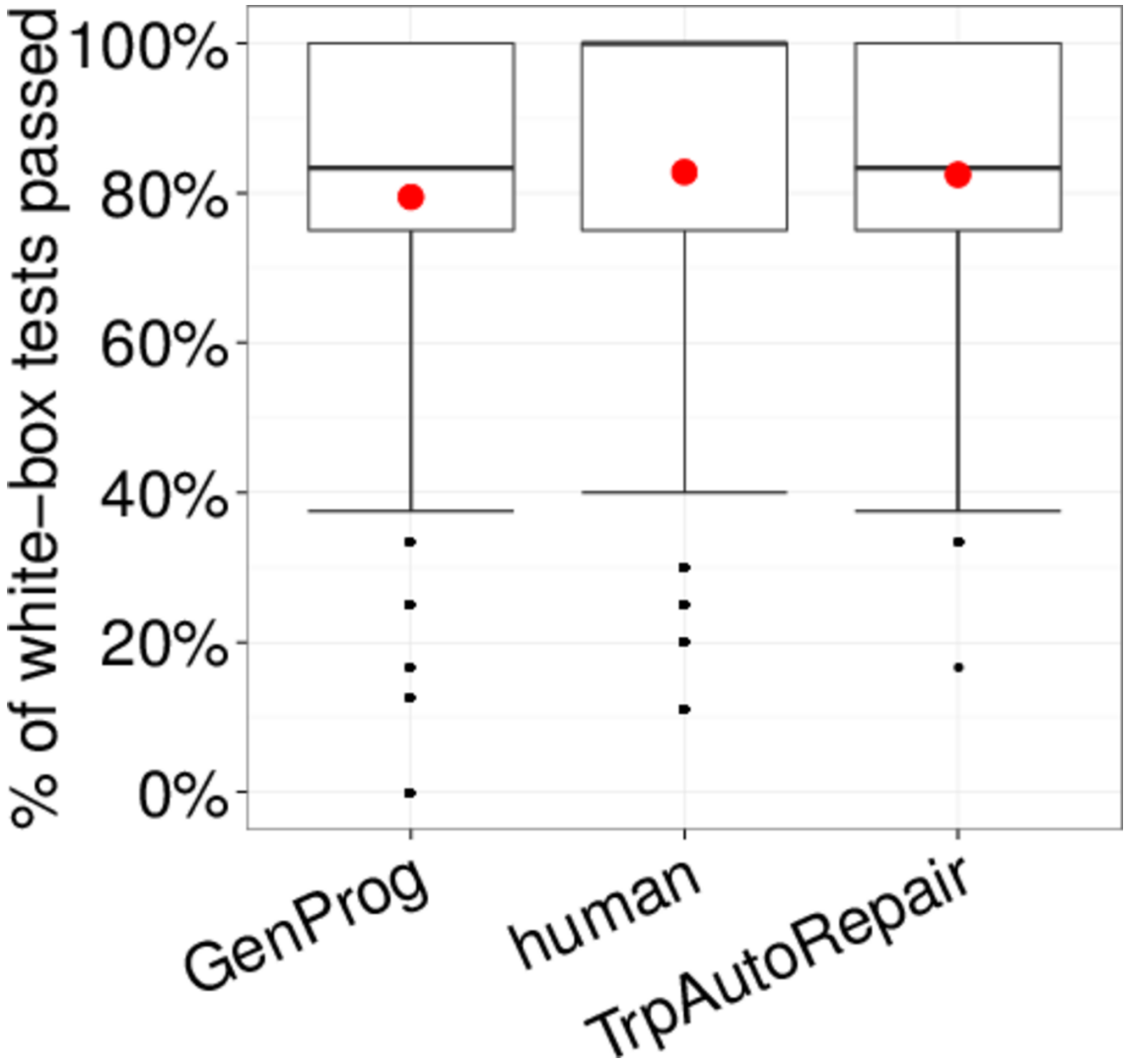
- Student homework submissions from six UC Davis Introduction to Programming assignments
- Two full-coverage test suites:
  - White-box suite generated by Klee from reference implementation.
  - Black-box suite written by course instructor.
  - *Feature: Assess patch quality as distinct from test suite quality.*
- Goal: Compare GenProg and TrpAutoRepair/ RSRepair, G&V techniques with different search strategies.

Full dataset available at repairbenchmarks.cs.umass.edu

Both tools produced patches that overfit to the training set.



But: the tools do as well as the students!



Overfitting is not unique toheuristic techniques.

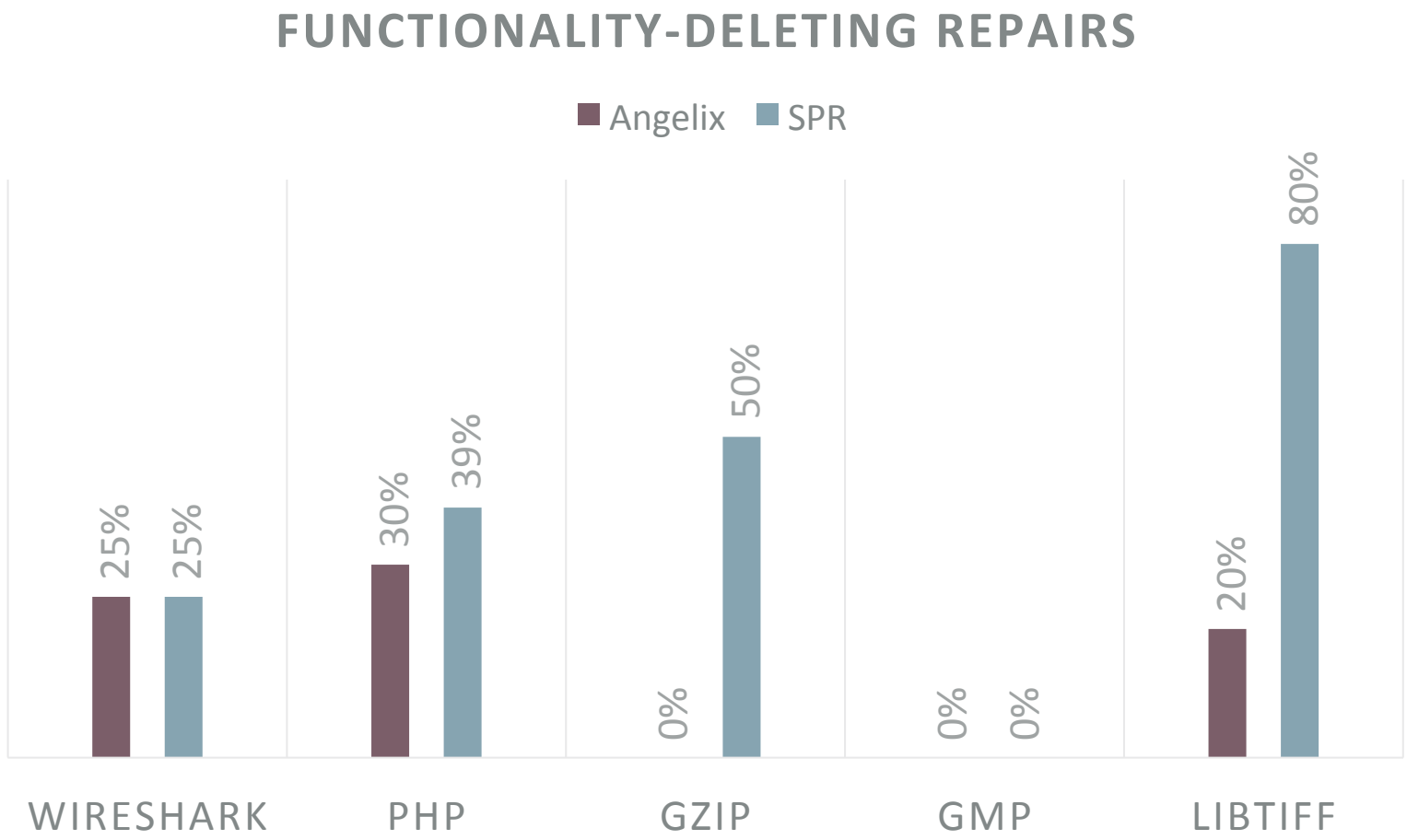
- Angelix: 120/233 of patches produced on a subset to IntroClass overfit.
- ~40% of SPR patches studied in Angelix paper delete functionality by generating tautological if conditions.

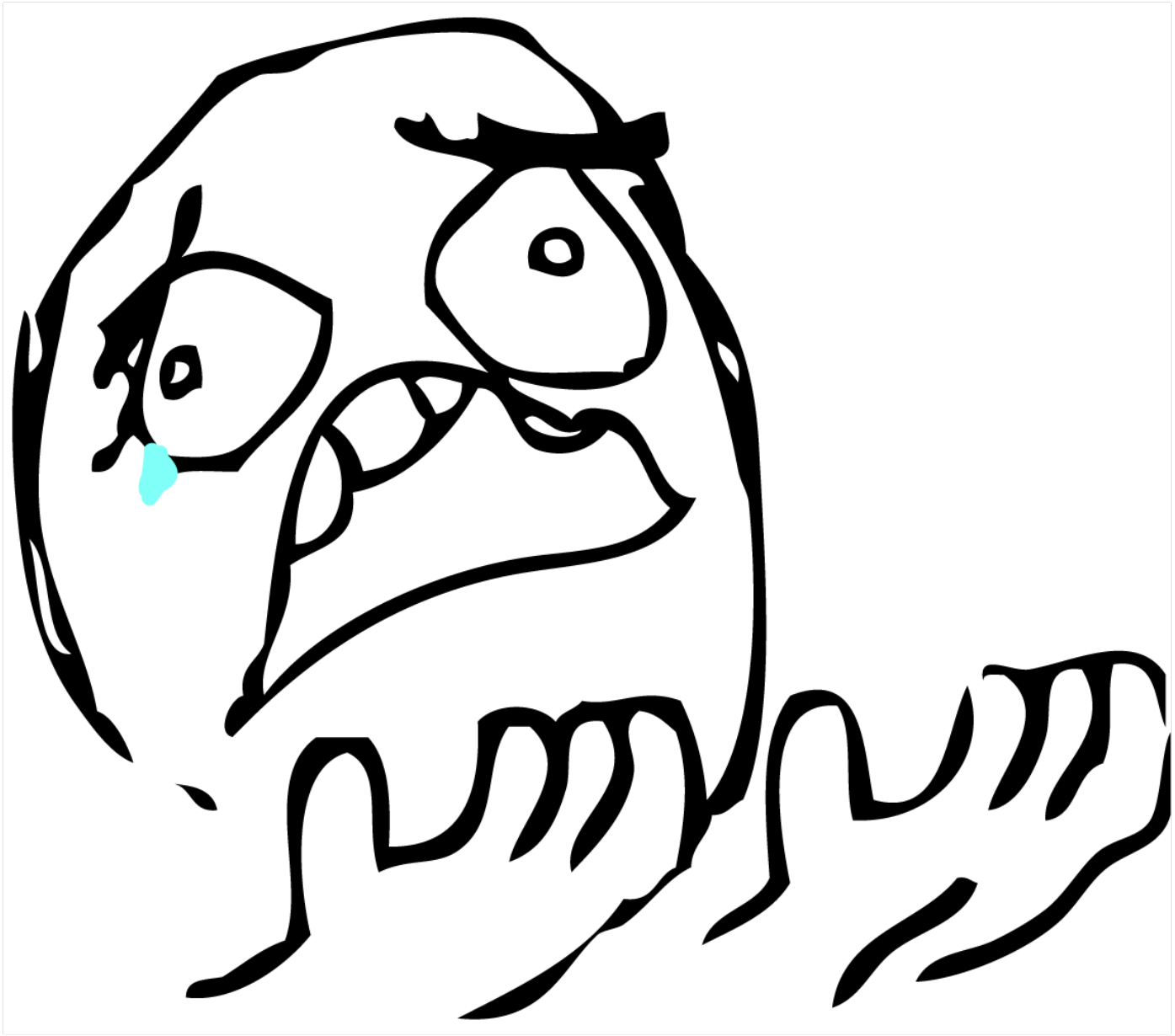
Overfitting is not unique toheuristic techniques.

- Angelix: 120/233 of patches produced on a subset [illegible]IntroClass overfit.
- ~40% of [illegible]lix paper d[illegible]g tautologica[illegible]

PhD students observe that much of the problem is in the synthesis of overly constrained if conditions.

Quality Comparison with SPR





Context matters!

OPTION 1: UNDERSTAND AND REASON ABOUT THE CIRCUMSTANCES UNDER WHICH PERFECTION IS NOT REQUIRED.

← Scenario: Long-running servers + IDS + generate repairs for detected anomalies.

Workloads: a day of unfiltered requests to the UVA CS webserv

THIS PATCH WAS BAD

LIGHTTPD
fly light.

NullHttpd
(logo-less)

php

2012 flashback…

Even a functionality-reducing repair had little practical impact.

Program	Post-patch requests lost	Fuzz Tests Failed	
		General	Exploit
nullhttpd	0.00 % ± 0.25%	0 → 0	10 → 0
lighttpd	0.03% ± 1.53%	1410 → 1410	9 → 0
php-BAD	**0.02% ± 0.02%**	**3 → 3**	**5 → 0**

How?

OPTION 2: DEVELOP TECHNIQUES THAT ARE MORE LIKELY TO GENERALIZE.

Challenge your assumptions!

EXAMPLE ASSUMPTION:

bug-fixing patches are like kittens: smaller is better!

"Retouched Kitty" by Ozan Kilic, CC2.0 http://www.freestockphotos.biz/stockphoto/9343

What if…

- Instead of trying to make small changes, we replaced buggy regions with code that correctly captures the overall desired logic?
- Principle: using human-written code to fix code at a higher granularity level leads to better quality repairs.

SEARCHREPAIR: HIGH-QUALITY AUTOMATED BUG REPAIR USING SEMANTIC SEARCH

Semantic code search looks for code based on what it should *do.*

- Keyword: “C median three numbers”
- Semantic:

Input	Expected
2,6,8	6
2,8,6	6
6,2,8	6
6,8,2	6
8,6,2	6
9,9,9	9

…Generate and validate + Semantic reasoning!

SearchRepair patches were of much higher quality than those produced by previous techniques.

Technique	Held out tests passed
SearchRepair	**97.2%**
GenProg	68.7%
TRPAP	72.1%
AE	64.2%

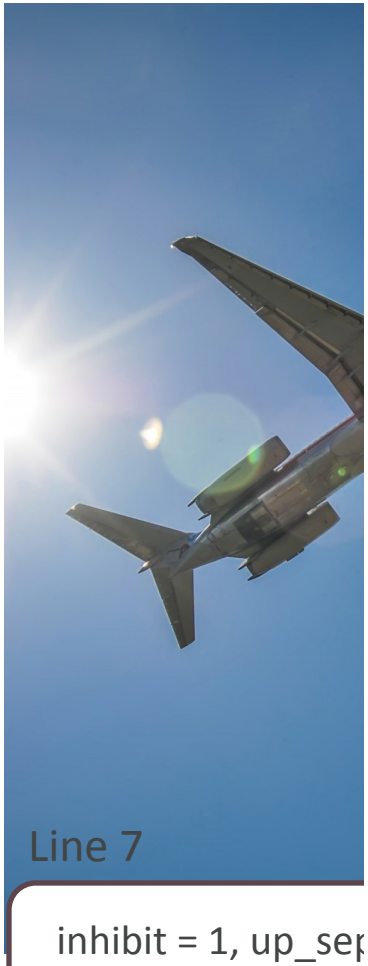
The Three Major Challenges

Scalability

Output quality

Expressive power

Young Claire



Line 7

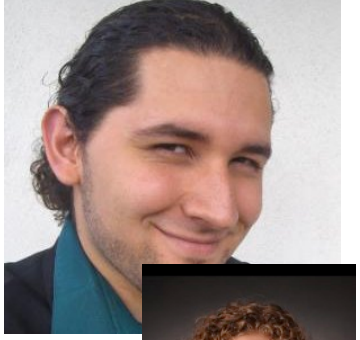
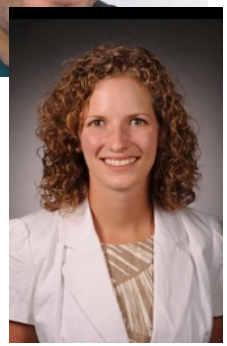
inhibit = 1, up_sep

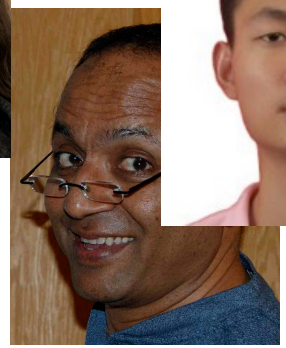
bias = α, PC= α> 110

Line 8

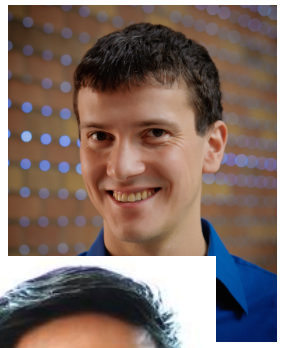
down_sep = 110

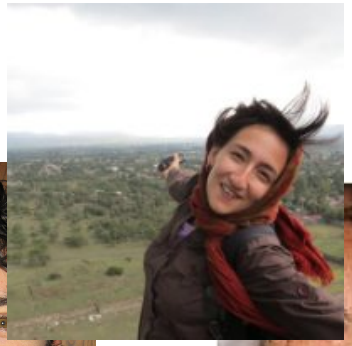
bias =α, PC= $\alpha \leq 110$

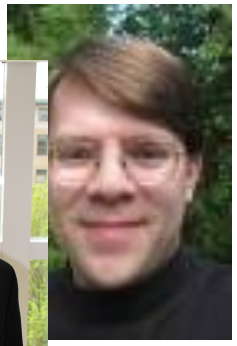



COLLABORATORS MAKE THE WORLD GO ROUND



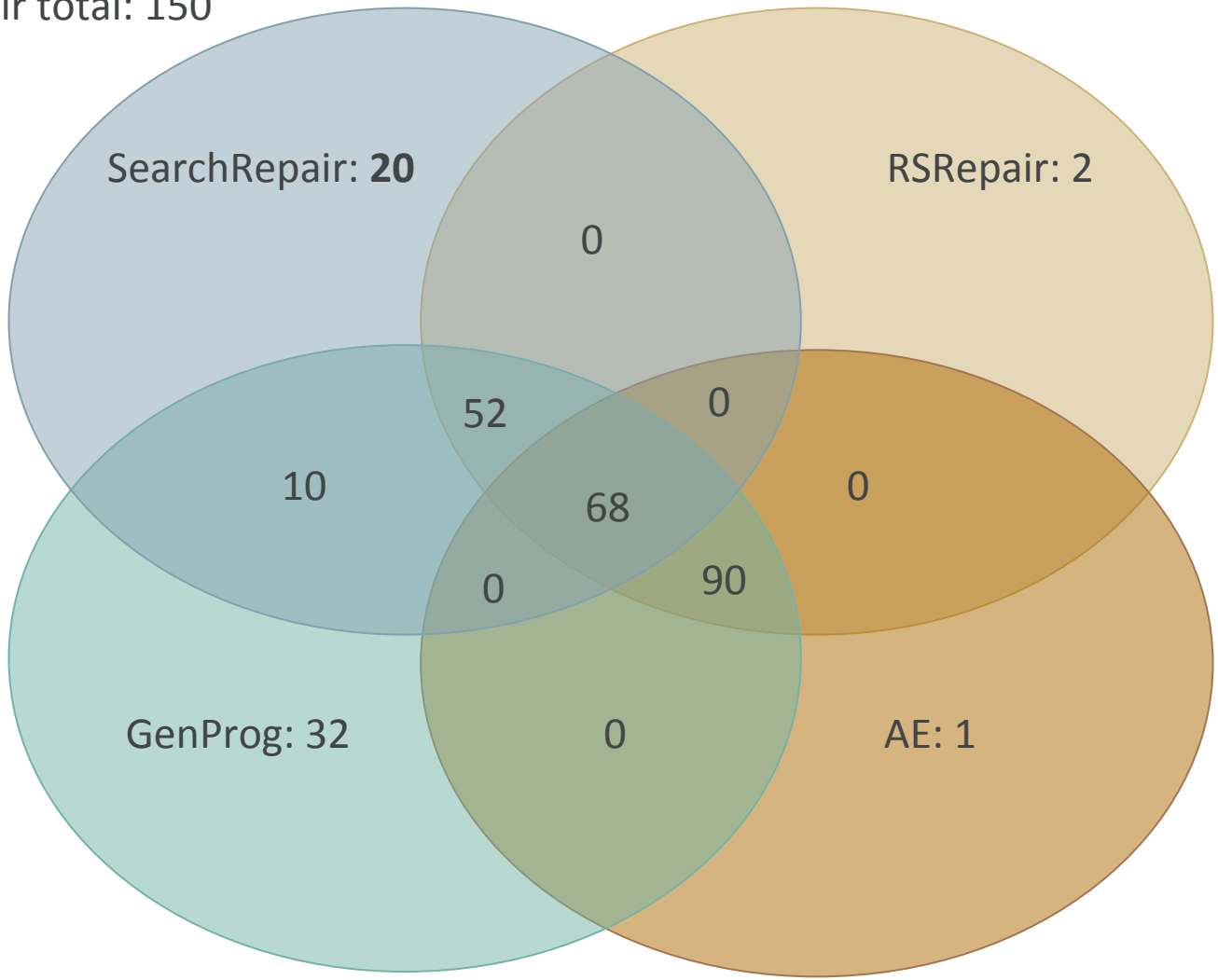







SearchRepair total: 150

RSRepair total: 247



GenProg total: 287

AE total: 159