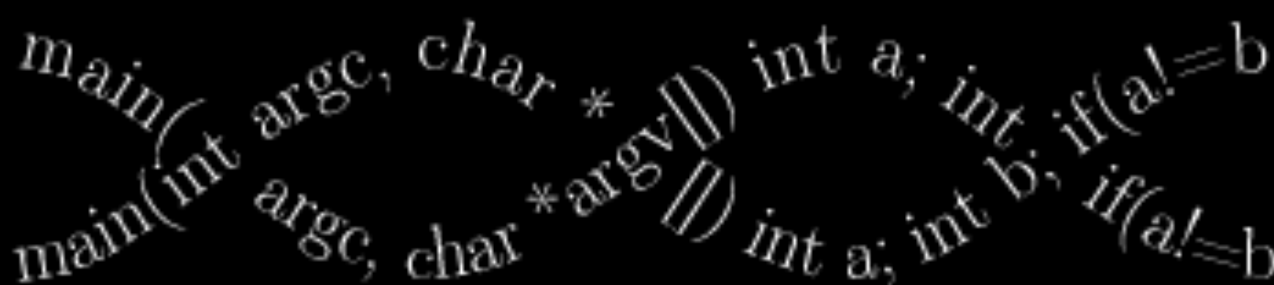
The Biology of Software
Evolution, Robustness, Diversity

Stephanie Forrest
University of New Mexico
and
Santa Fe Institute
July, 2016

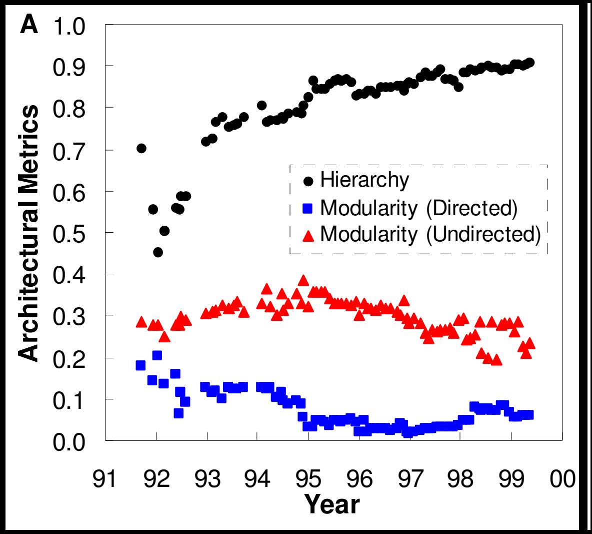
The Biology of Software

- Thesis: Software today is the result of many generations of inadvertent evolution
  - Successful genes (libraries, packages, modules, code snippets) are copied and mutated
  - Recombination of successful genes
- The perspective of evolutionary biology provides insight
  - Science (identifying and measuring patterns)
  - Engineering (improving software)



What are the Patterns?
Hallmarks of Evolution

- Emergence of hierarchy
- Increasing complexity
- Neutral fitness landscapes
- Fundamental distributions
  - Species abundance



"Detecting Evolving Patterns of Self-Organizing Networks by Flow Hierarchy Measurement" Luo and MaGee (2010)

Overview of Talk

Engineering, Science, Engineering

- Evolving software automatically with GenProg
  – Repairing bugs
  – Energy optimization
- Mutational robustness and neutral landscapes
- Proactive diversity for resilience to unknown bugs / attacks

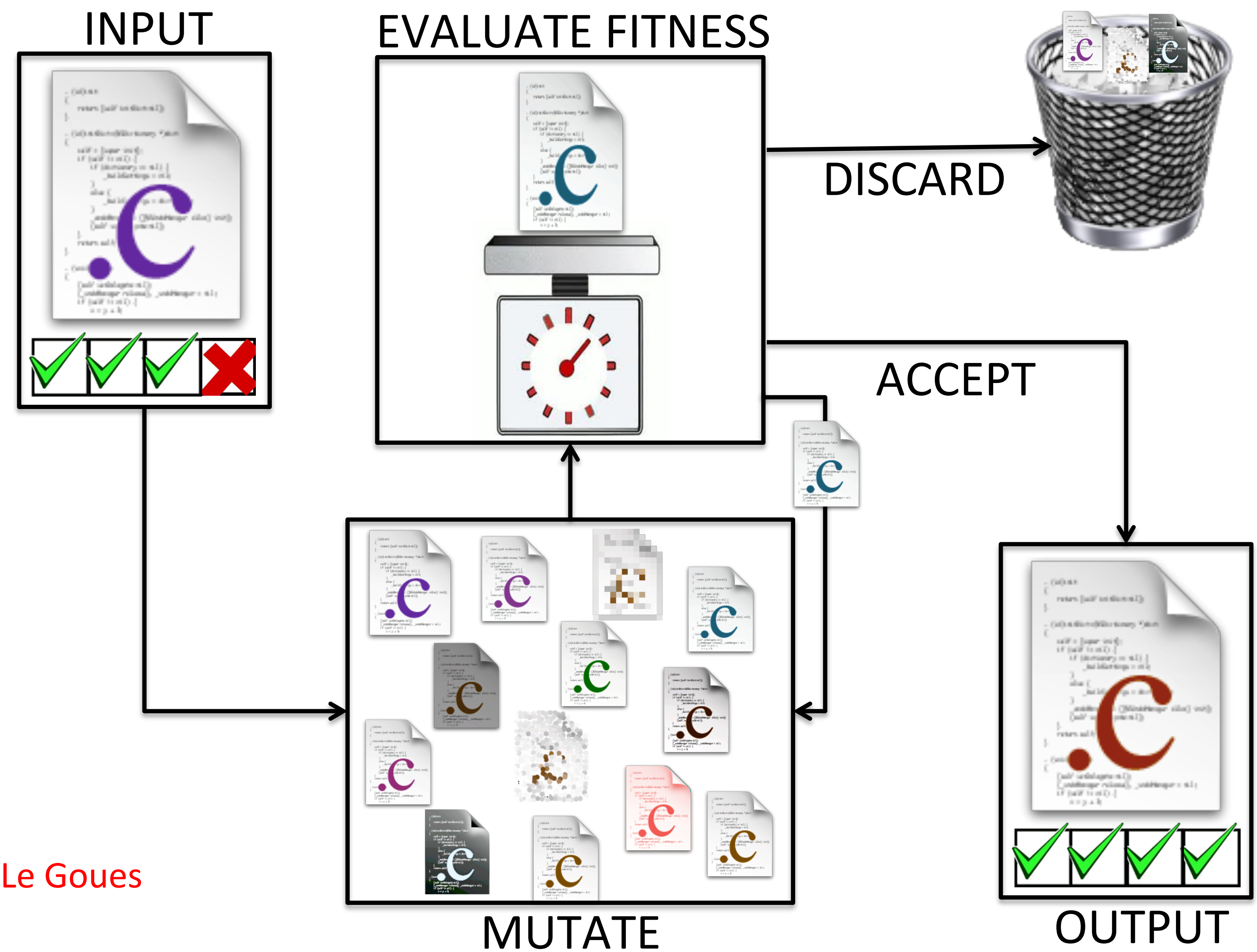
   

Evolution for Program Repair with Westley Weimer (UVA/UM)

Goal: A generic method for automated software repair

Legacy code
Do not assume a formal specification

GenProg



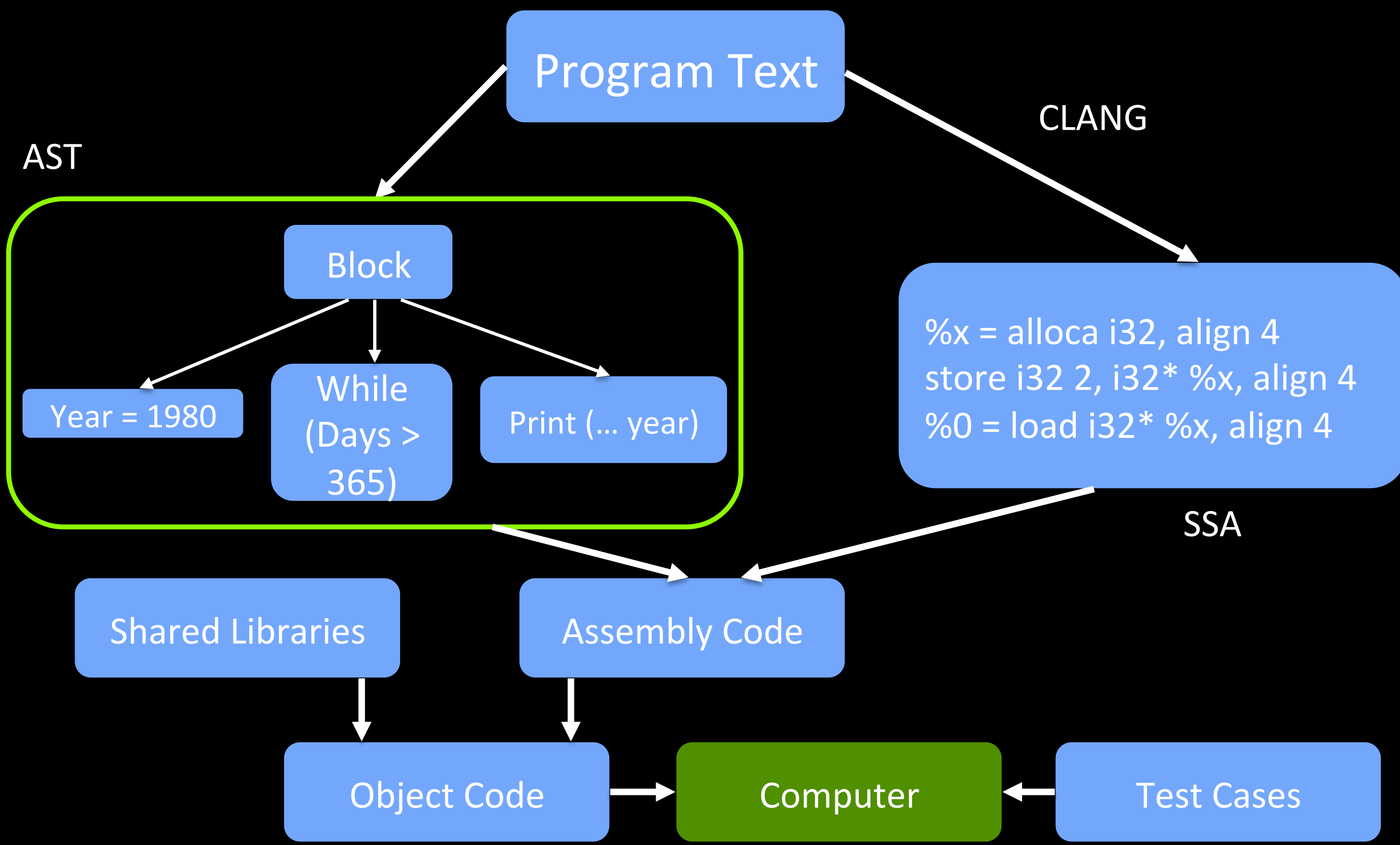
So: C. Le Goues

Design Decisions

- What to repair?
  - Program representation
- How to repair?
  - Genetic operators
- Where to repair?
  - Fault localization
- Fitness function

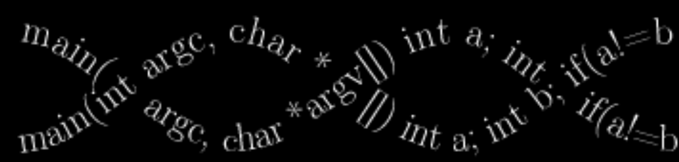
What to repair?

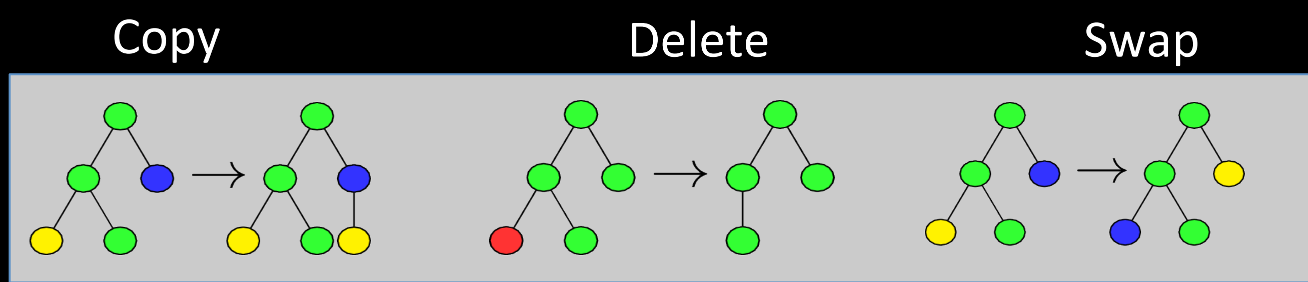
Program Representation

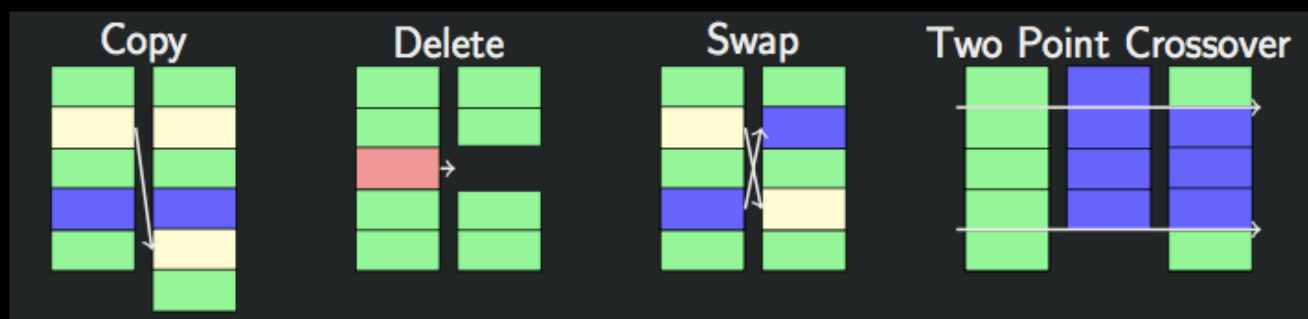


How to repair?

Genetic Operators

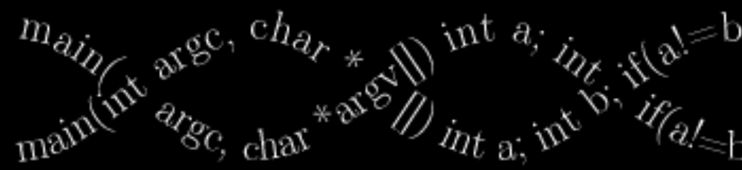




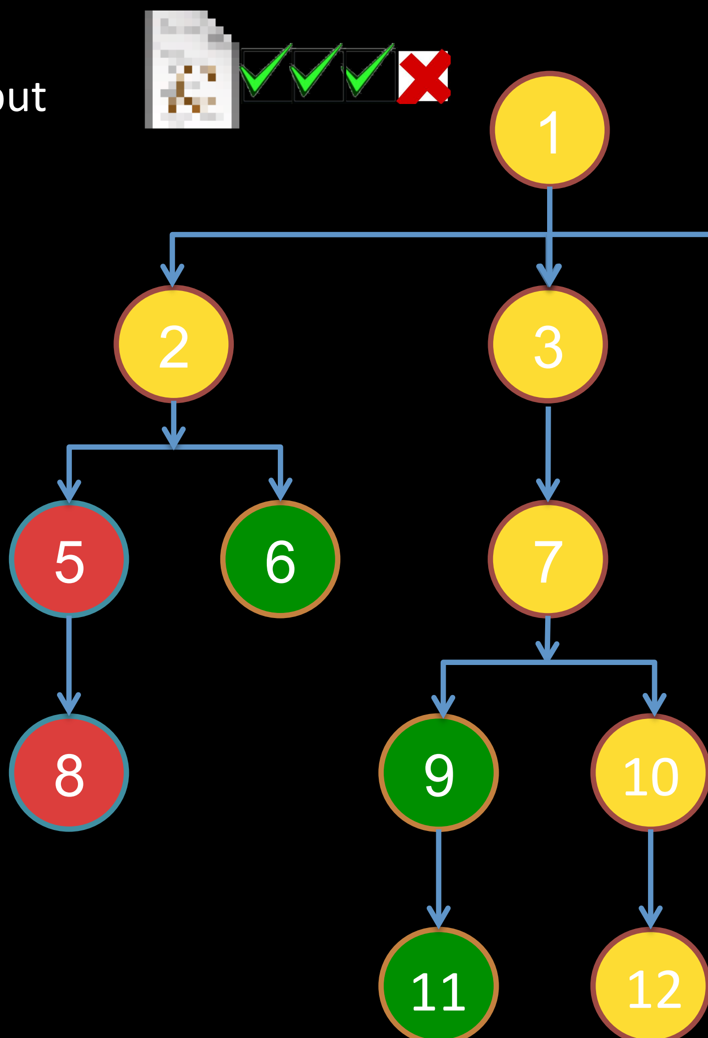


- Don’t invent new code
- Statement-level operations

Where to repair?



Input



Legend:

Likely faulty.

Maybe faulty.

Not faulty.

Weighted Path (stmts)

1.0: Nodes visited only by negative test case

0.01: Nodes visited by neg and pos

0.0: All other nodes

Parameters

- Fitness: Weighted sum of test cases that the program passes
- Std. run
  - Population size: 40
  - Run for 10 generations
  - 1 mutation per indiv. per gen.
  - Each individual participates in 1 crossover per gen.
- Test suite sampling and parallelism

Example Repair: Infinite loop

```c
void zunebug_repair(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear(year)){
      if (days > 366) {
        // days -= 366; // repair deletes
        year += 1;
      }
      else {
      }
      days -= 366;       // repair inserts
    } else {
      days -= 365;
      year += 1;
    }
  }
  printf("current year is %d\n", year);
}
```

Minimized repair produced in 42 seconds

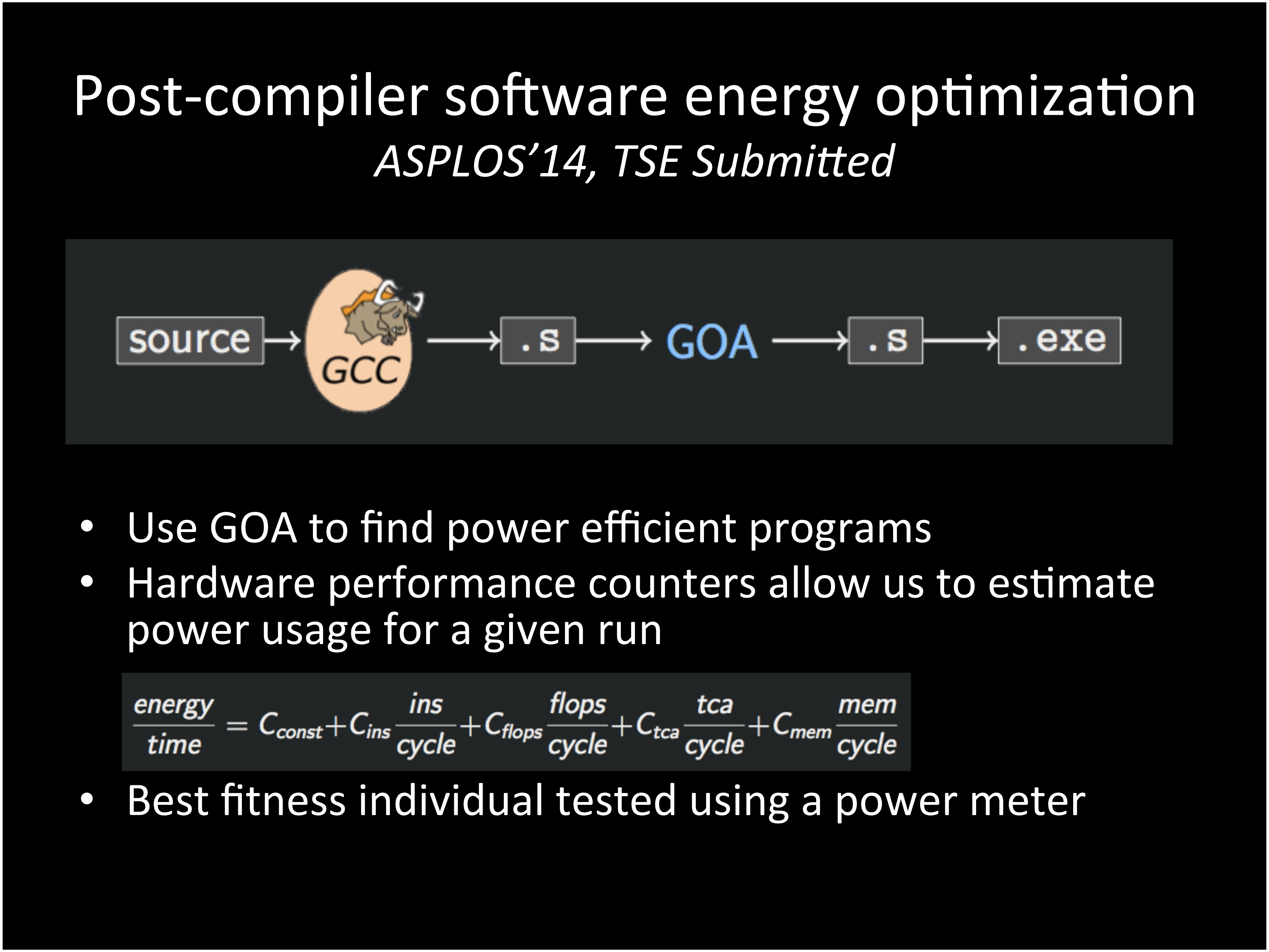
Minimizing the Repair

- Use tree-structured differencing (Al-Ekram et al. 2005)
  - View primary repair as a set of tree-structured operations
- One-minimal subset of repairs
  - Let Cp = {c1, c2, …, cn} be the set of changes in a primary repair
  - One-minimal subset is the minimal subset of Cp that passes all test cases.
- Delta debugging: Search for one-minimal subset using binary search (Zeller, 1999)
  - n^2 time in worst case, often linear

How well does GenProg work in practice? *(ICSE'12, TSE'16)*

Program	Description	LOC	Tests	Bugs	
				Fixed	Total
fbc	Language (legacy)	97K	773	1	3
gmp	Multiple precision math	145K	146	1	2
gzip	Data compression	491K	12	1	5
libtiff	Image manipulation	77K	78	17	24
lighttpd	Web server	62K	295	5	9
php	Language (web)	1,046K	8,471	28	44
python	Language (general)	407K	355	1	11
wireshark	Network packet analyzer	2,814K	63	1	7
Total		**5.14M**	**10,193**	**55**	**105**

Repaired 52% at a cost of $7.32 each
With algorithm tuneups: 5 additional bugs (57%)
With additional CPU resources (69%)


Post-compiler software energy optimization
ASPLOS'14, TSE Submitted
source
GCC
.s
GOA
.s
.exe
• Use GOA to find power efficient programs
• Hardware performance counters allow us to estimate power usage for a given run
$\frac{energy}{time} = C_{const} + C_{ins}\frac{ins}{cycle} + C_{flops}\frac{flops}{cycle} + C_{tca}\frac{tca}{cycle} + C_{mem}\frac{mem}{cycle}$
• Best fitness individual tested using a power meter

GOA Parameters

- Population size: 2^{10}
- 2^{18} fitness evaluations
- ~16 hour run time per optimization

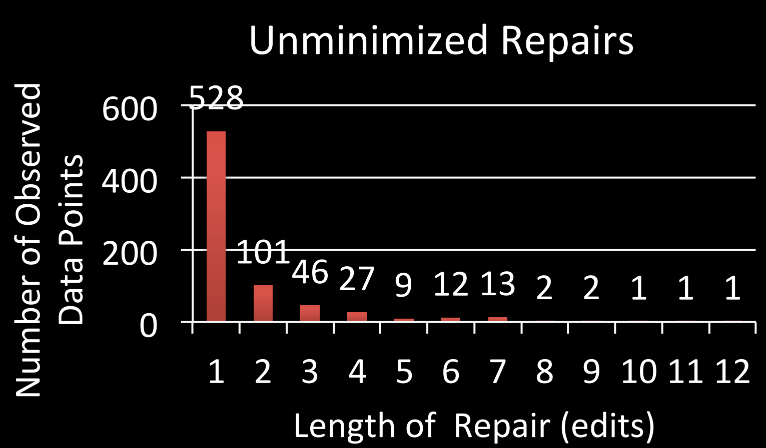
Example Run

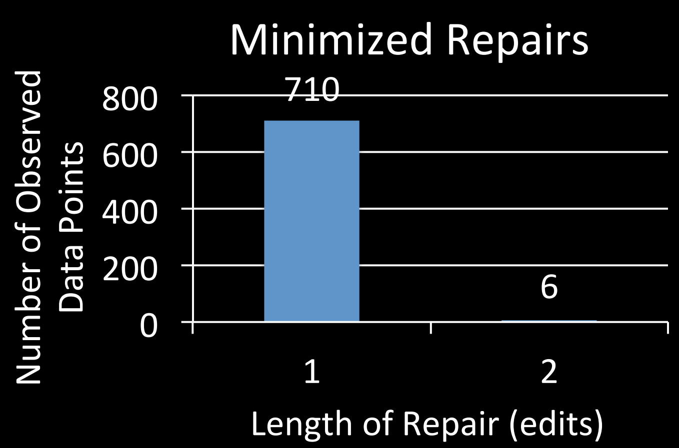

Swaptions Fitness Over Time
Joules
900
850
800
750
700
650
600
550
500
0
5e4
10e4
15e4
20e4
25e4
30e4
Iterations

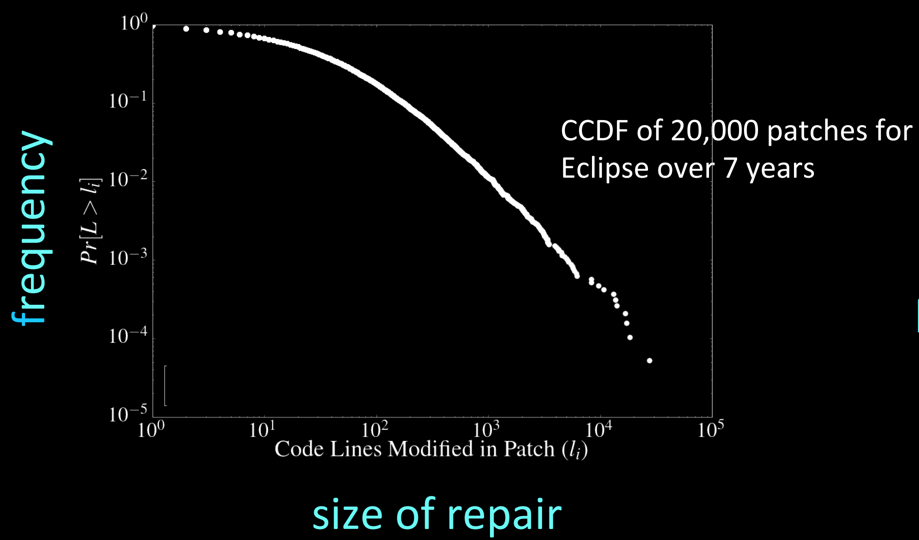
War Stories

- Large open source programs have buggy tests:
  - Test of a sort: “the output of sort is in sorted order”
    - GenProg's fix: “always output the empty set”
  - Typos: Generate a random ID with prefix “999”, check to see if result starts with “9996”
  - Pass if today is less than December 31, 2012
- Binary/assembly programs
  - Test: “compare your-output.txt to trusted output.txt”
  - GenProg's fix: “delete trusted-output.txt, output nothing”
- Sandboxing
  - Programs that kill the parent shell
  - Programs that sleep forever to avoid CPU usage tests for infinite loops

GenProg is excellent at finding single edit repairs







Most bugs are small

Why does GenProg succeed?

- Algorithmic innovations 
- Exploits holes in test cases
- Most bugs are small
- Neutrality
  - Many biological mutations leave fitness unchanged
  - 30% of GenProg's mutations are neutral!

Eric Schulte

Software Mutational Robustness

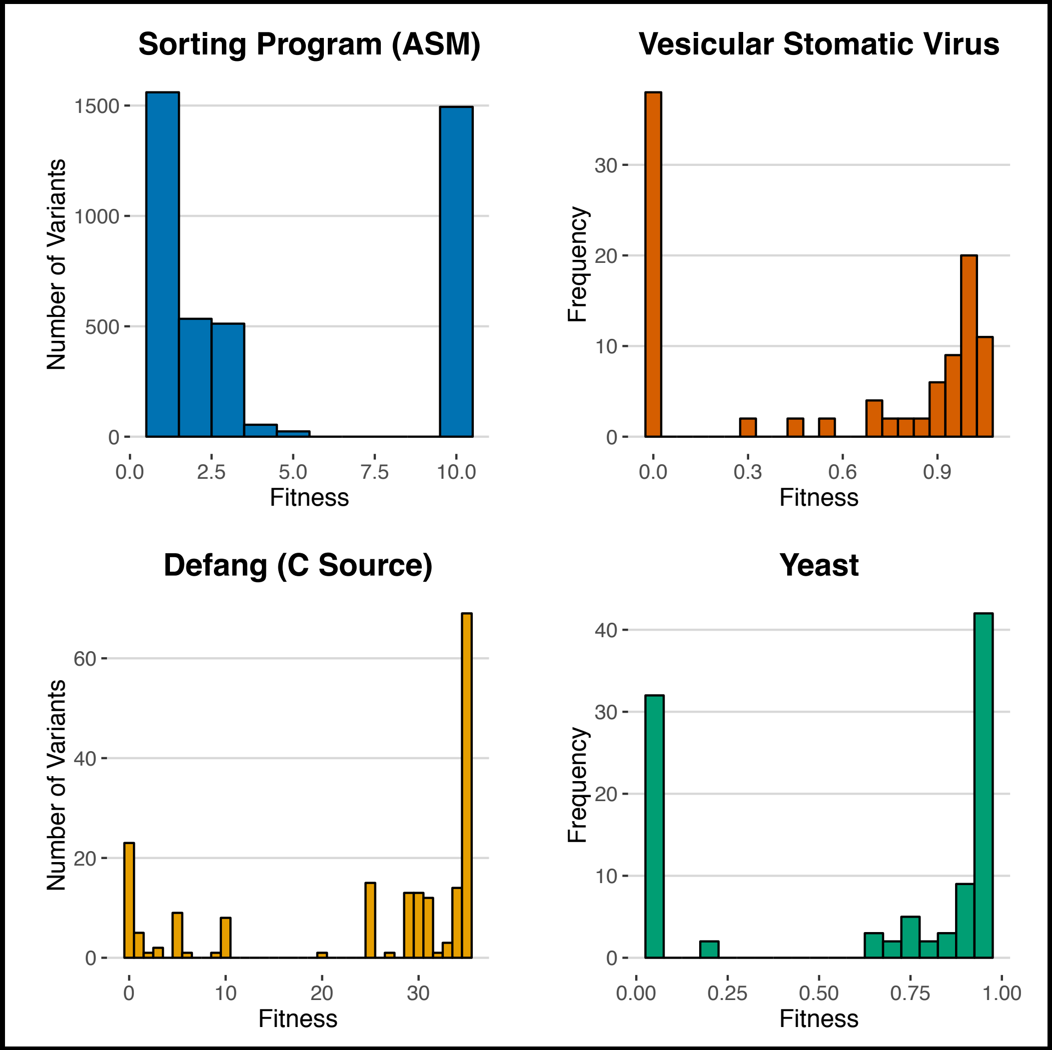
Experimental Results, *GPEM 2014,*

- Metric:
  - % of 1-step mutations that are neutral
  - Mutate only statements visited by at least 1 test case
  - Does-not-compile: non-neutral
- Benchmarks: 22 programs
  - 23,151 total tests
  - Test suite coverage ranges from (0.8 - 100%)
- Results:
  - 33.9% of 1-step AST mutations are neutral
  - 39.6% of 1-step ASM mutations are neutral
  - At least 20% are neutral

Program	Lines of Code		Test Suite		Mut. Robustness	
	ASM	C	# Tests	% Stmt.	AST	ASM
Sorting Algorithms						
bubble-sort	184	34	10	100	27.3	25.7
insertion-sort	170	29	10	100	29.4	26.0
merge-sort	233	38	10	100	29.8	21.2
quick-sort	219	38	10	100	28.9	25.5
Siemens [20]†						
printtokens	2419	536	4130	81.7	21.2	25.8
schedule	922	412	2650	94.4	34.4	29.1
space	18098	9126	13494	91.1	37.7	32.1
tcas	544	173	1608	96.2	33.5	25.9
Systems Programs						
bzip2 1.0.2	18756	7000	6	35.9	33.0	26.1
— *(alt. test suite)*			22	71.0	46.4	23.6
ccrypt 1.2	15261	4249	6	29.5	33.0	69.7
— *(alt. test suite)*			16	40.4	34.6	69.7
grep	28776	10929	119	24.9	50.0	36.7
imagemagick 6.5.2	6128	147	145	0.8	33.3	66.3
jansson 1.3	6830	2975	30	28.8	33.3	28.0
leukocyte	40226	7970	5	45.4	33.3	39.9
lighttpd 1.4.15	34165	3829	11	40.1	61.5	56.9
nullhttpd 0.5.0	5951	5575	6	64.5	41.5	37.8
oggenc 1.0.1	299959	59094	10	38.4	33.4	22.1
— *(alt. test suite)*			40	58.8	40.5	72.3
potion 40b5f03	80406	15033	204	48.4	33.3	48.9
redis 1.3.4	44802	17203	234	9.2	33.3	34.0
sed	17026	8059	360	42.0	33.0	25.6
tiff 3.8.2	22458	1732	10	15.4	33.3	90.4
vyquon 335426d	20567	4390	5	50.6	33.3	69.0
total or average	**664100**	**158571**	**23151**	**40.9**	**33.9** $\pm$10	**39.6** $\pm$22

Test suite coverage does not explain mutational robustness

Bimodal Fitness Distributions of Mutations

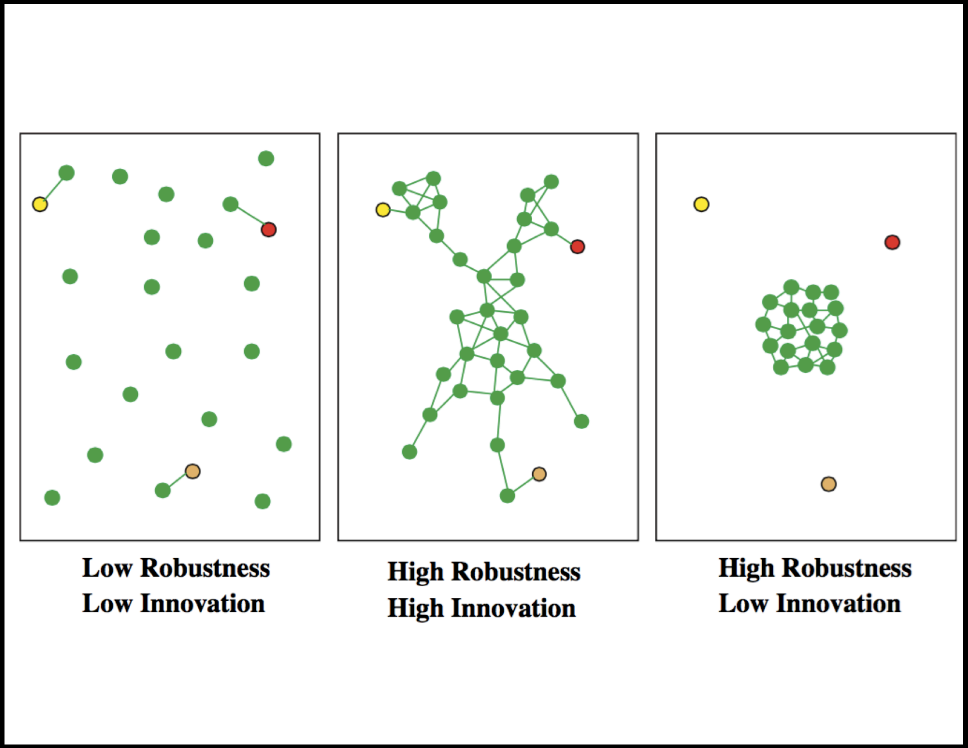


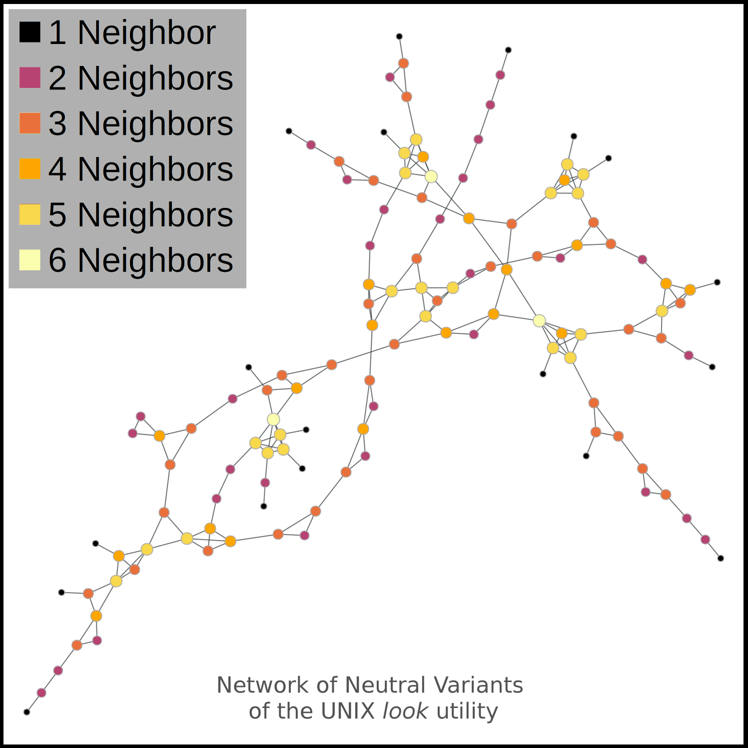
So. E. Schulte, A. Milligan

So. Eyre-Walker & Keightley, 2007; courtesy of J. Masel

Neutral Networks

High robustness and the ability to innovate

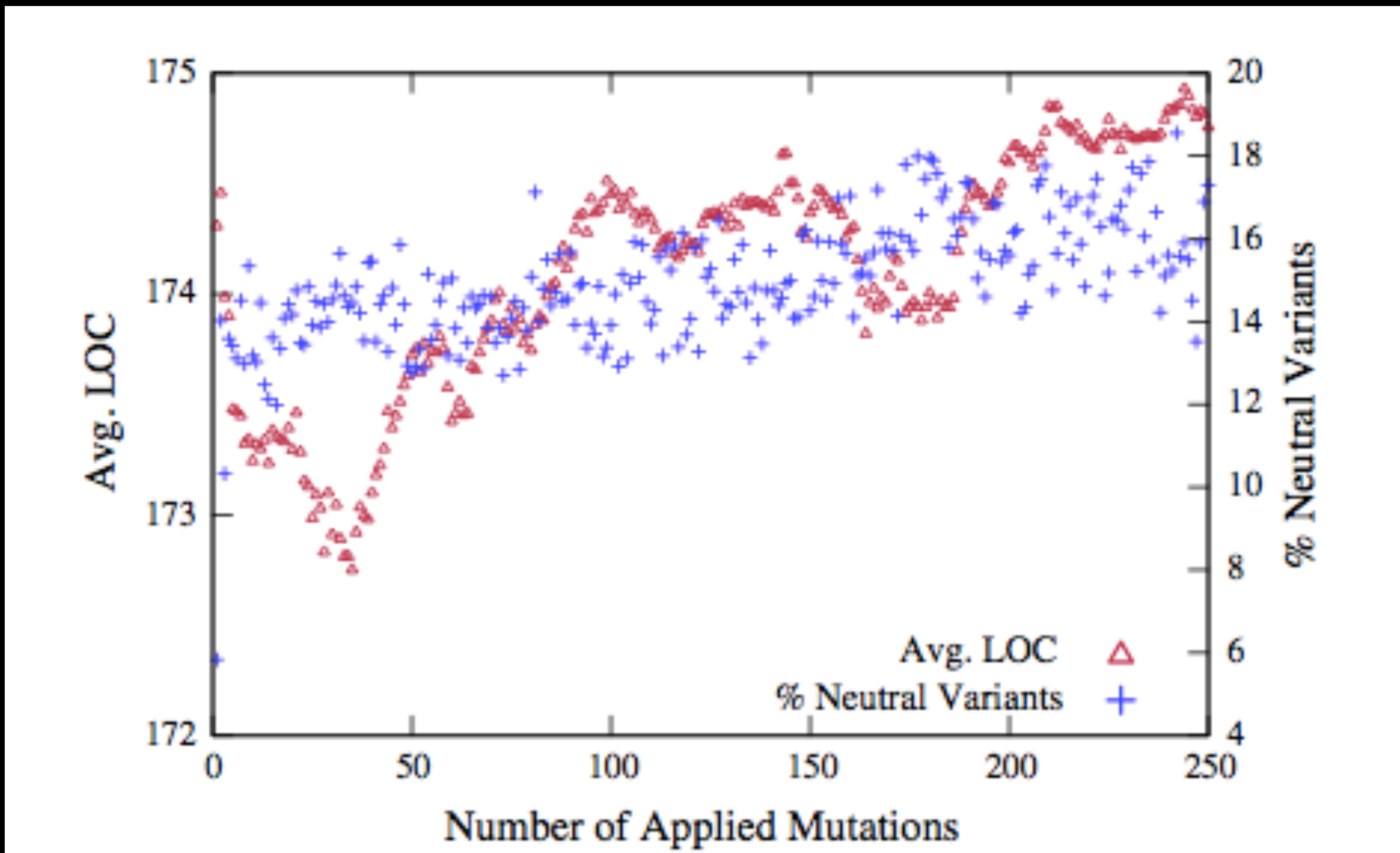




So: S. Ciliberti, O. Martin, and A. Wagner. Innovation and robustness in complex regulatory gene networks. *PNAS*104(34):13591, 2007

Work in progress, Renzullo 2017

Random Walks in Assembly Code
(GPEM, 2014)



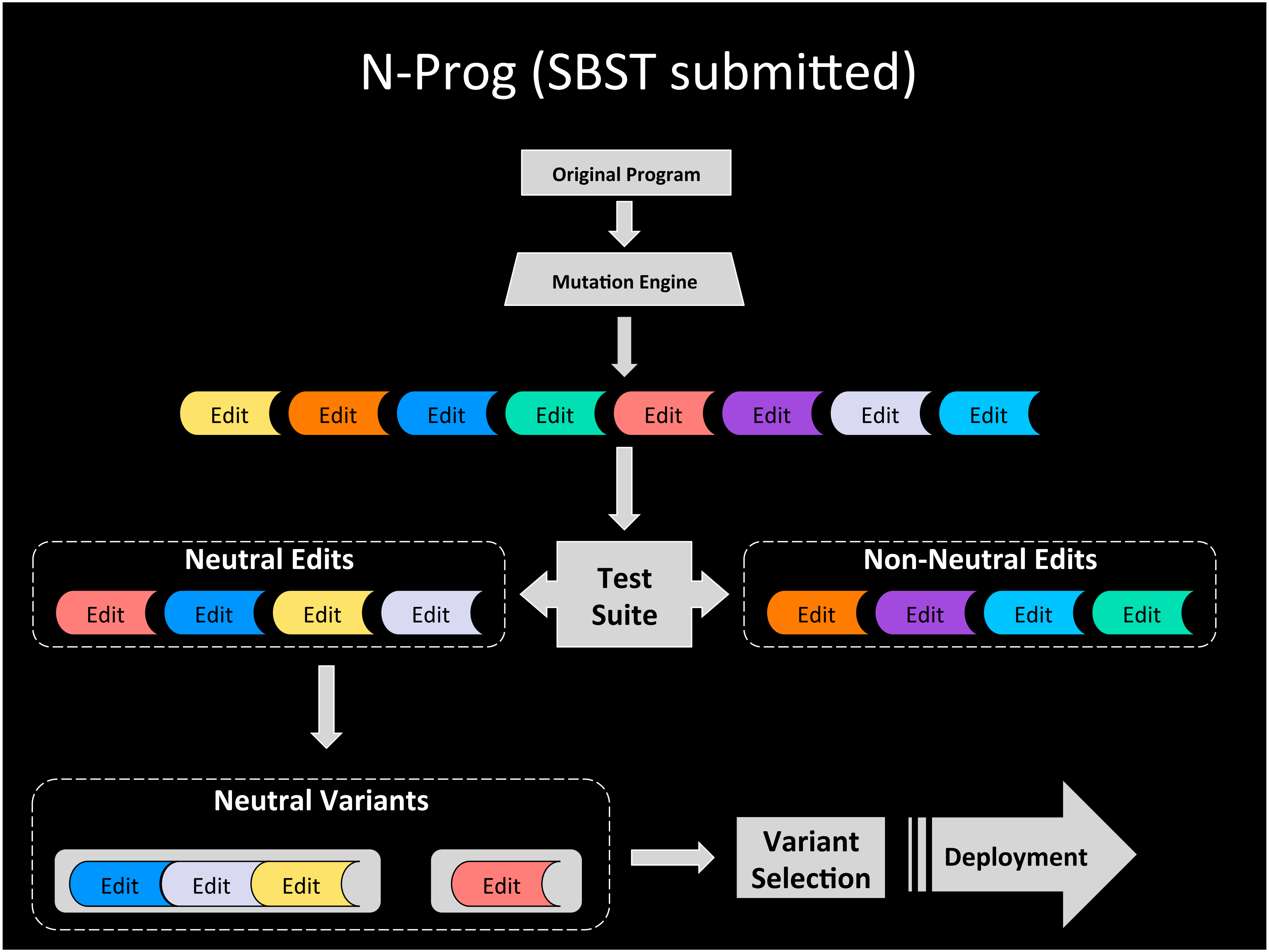
Significance of Software Neutrality

- Contradicts idea that "programs are fragile"
- Supports *strong biology hypothesis* of computing
  - More than just "bio-inspired"
  - Software has acquired biological properties through inadvertent evolution
- Path to more powerful automated repairs?
  - Multi-edit repairs, other learning methods, etc.

DIVERSITY

Evolution produces diversity

- The problem with monoculture
  - ISR, ASR, BSD anti-ROP mechanism
- Coarse-grained diversity (N-Prog)
  - Generate populations of semantically distinct programs
  - Automatically repair latent bugs and avoid security flaws


N-Prog (SBST submitted)
Original Program
Mutation Engine
Edit
Edit
Edit
Edit
Edit
Edit
Edit
Edit
Neutral Edits
Edit
Edit
Edit
Edit
Test Suite
Non-Neutral Edits
Edit
Edit
Edit
Edit
Neutral Variants
Edit
Edit
Edit
Edit
Variant Selection
Deployment

Example: defang (from thttpd)

Original program:

```
for ( cp1 = str, cp2 = dfstr;
      *cp1 != '\0' && (cp2 - dfstr < dfsize - 1)
      ++cp1, ++cp2 )
   {
   switch ( *cp1 )
      {
      case '<':
      *cp2++ = '&';
      *cp2++ = 'l';
      *cp2++ = 't';
      *cp2 = ';';
      break;
      . . .
```

Leaves space for 1 character

Appends 4 chars

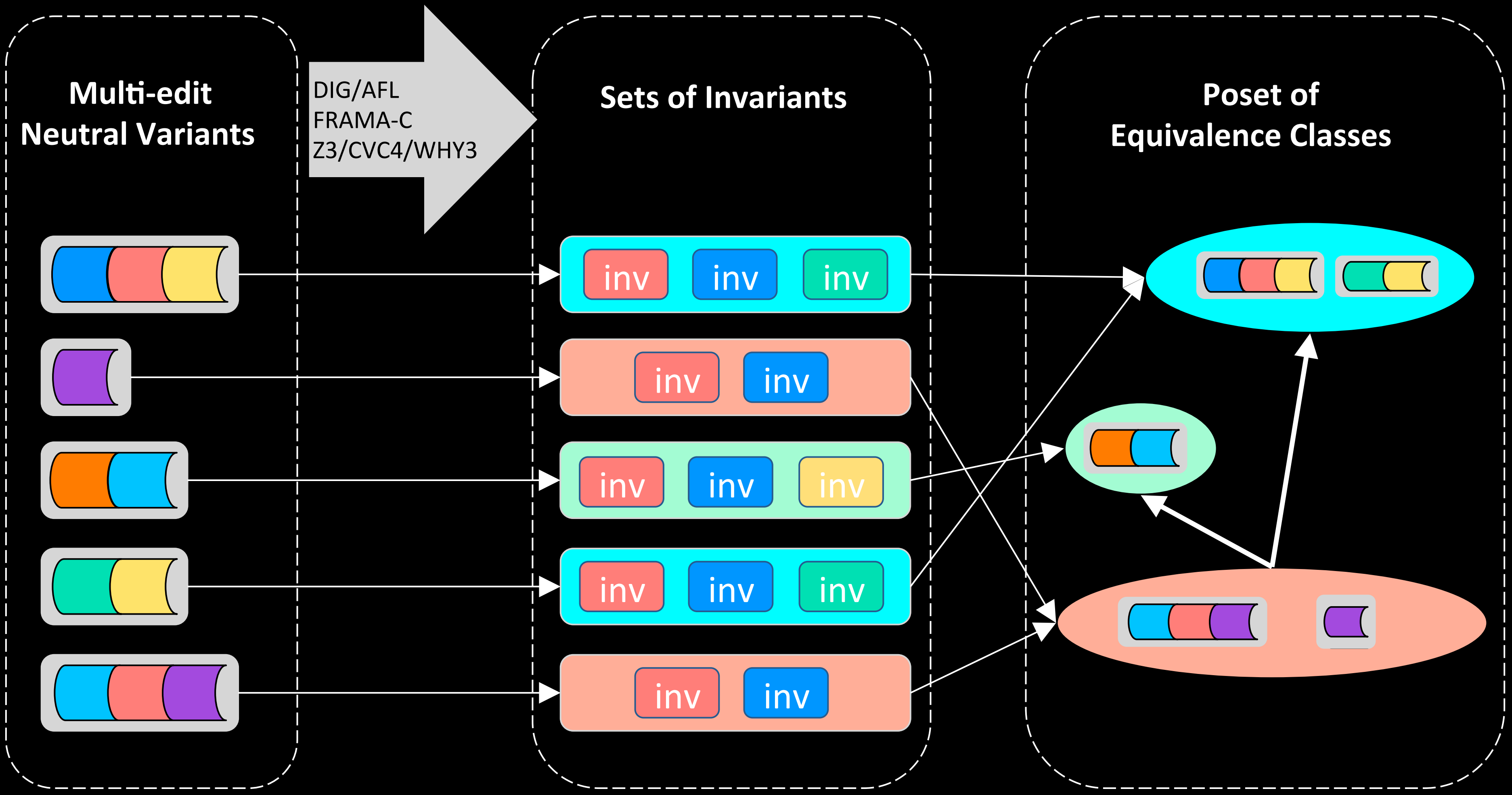
Single-edit neutral mutation:

```
for ( cp1 = str, cp2 = dfstr;
      *cp1 != '\0' && (cp2 - dfstr < dfsize - 1);
      ++cp1, ++cp2 )
   {
   switch ( *cp1 )
      {
      case '<':
      *cp2++ = '&';
// n-prog adds a check after the first char is
// written to prevent overflow:
if (! (cp2 - dfstr < (long )(dfsize - 1))) {
            break;
}
. . .
```

Why should I trust a program with random mutations?

- Testing alone is probably insufficient
  - Clever mutations, incomplete test suites
- Goals:
  - Show that transformed programs preserve required functionality (repaired, neutral)
  - Maximize diversity among deployed variants
- Approach: Program analysis
  - Combine dynamic invariant generation with theorem proving (DIG + KIP)
  - Work in progress

Equivalence Classes of Neutral Variants



Summing Up

- Generic approach to software repair
  - Does not rely on a formal specification
  - Does not require prior enumeration of vulnerability types or repair approaches
  - Down payment on goal of automated programming
- Software is biological
  - Mutational robustness
  - Malicious behavior
- Tools
  - GenProg: Evolution for software repair
  - N-prog: Coarse-grained diversity for security

“We can't solve problems by using the same kind of thinking we used when we created them”

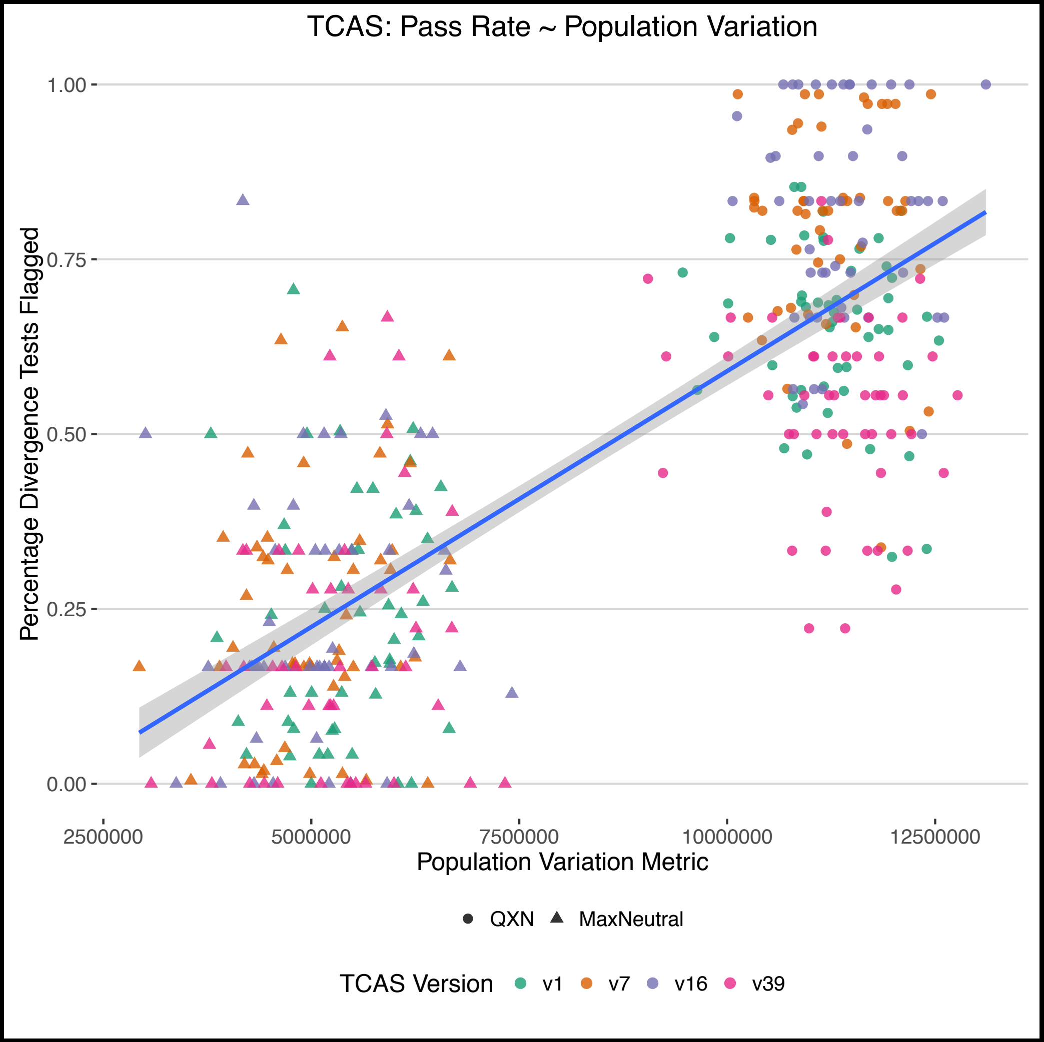
- Why do we need engineering practices based on biology?
  - Software ecosystem is evolving
  - Dynamic, mobile, complex, hostile environments
  - Moore’s Law won’t rescue us
- Hallmarks of biological computation
  - Resilience and adaptation as first-class citizens
  - Robustness, diversity, evolution

THANK YOU!

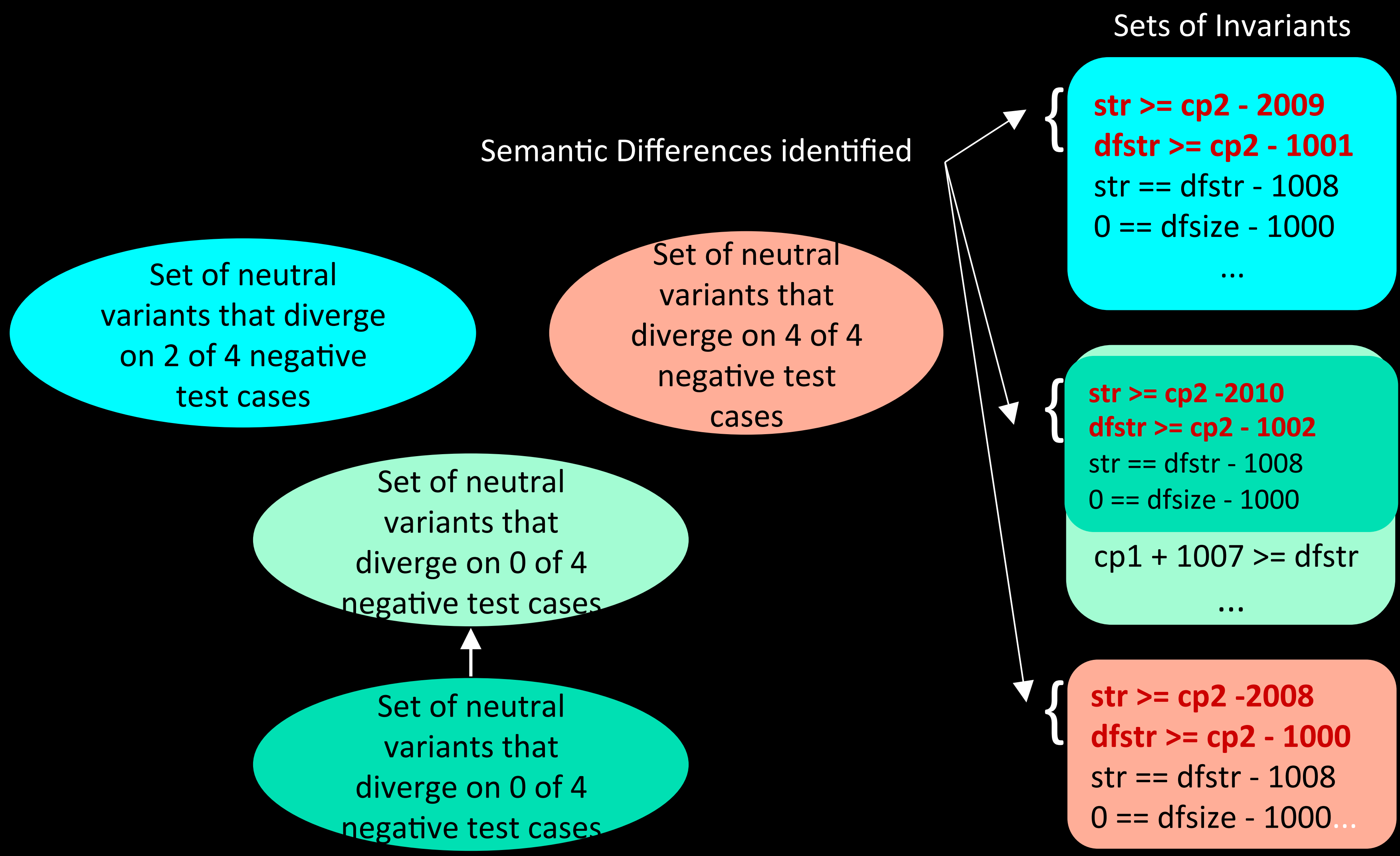
References

- https://cs.unm.edu/~forrest
- forrest@cs.unm.edu
- https://dijkstra.cs.virginia.edu/genprog/

Measuring Diversity



Example: defang

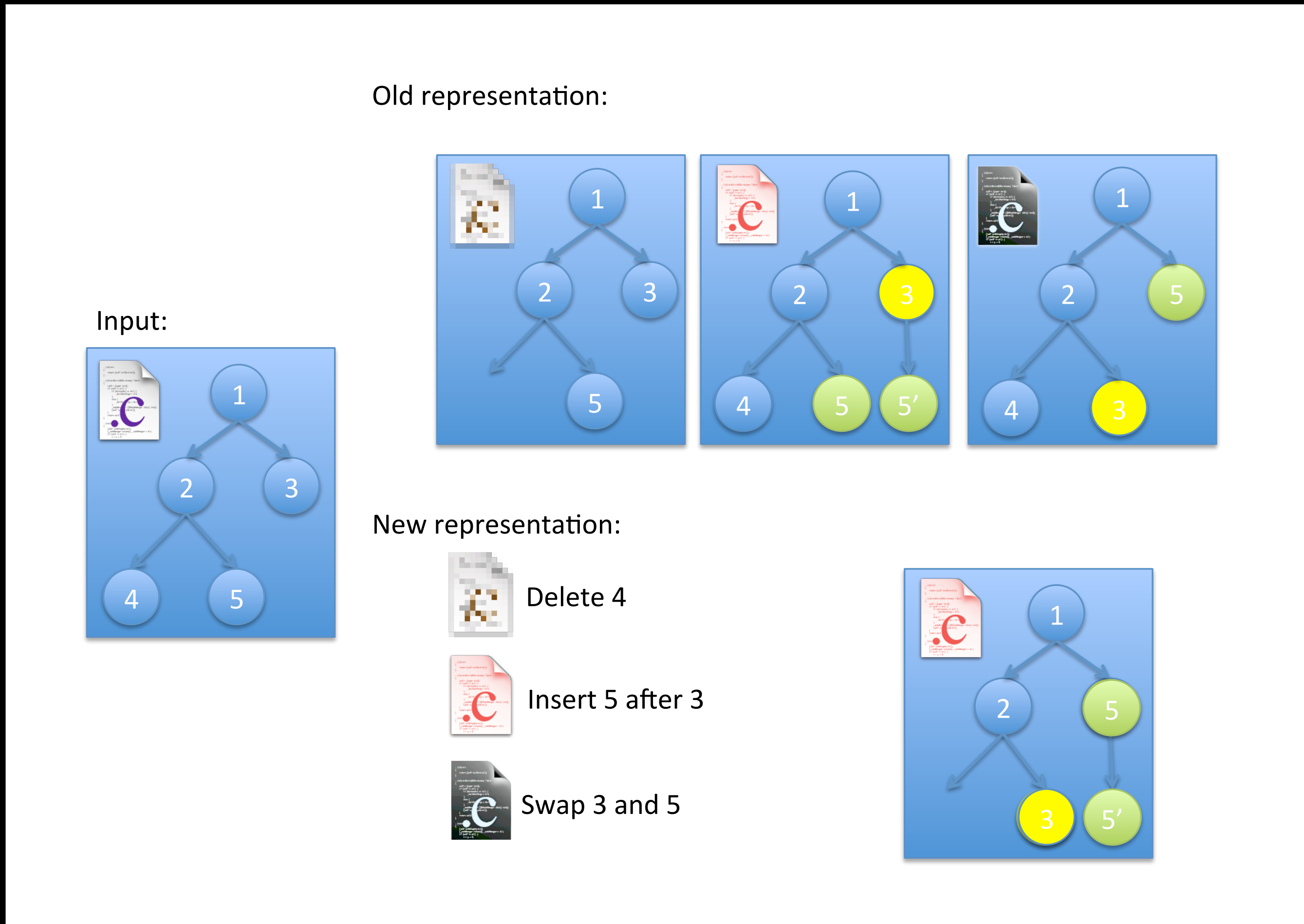


NOTE: Relies on manually generated test inputs to provide DIG enough coverage to find accurate postconditions. We will explore fuzzy test input generation options (AFL) in the future.

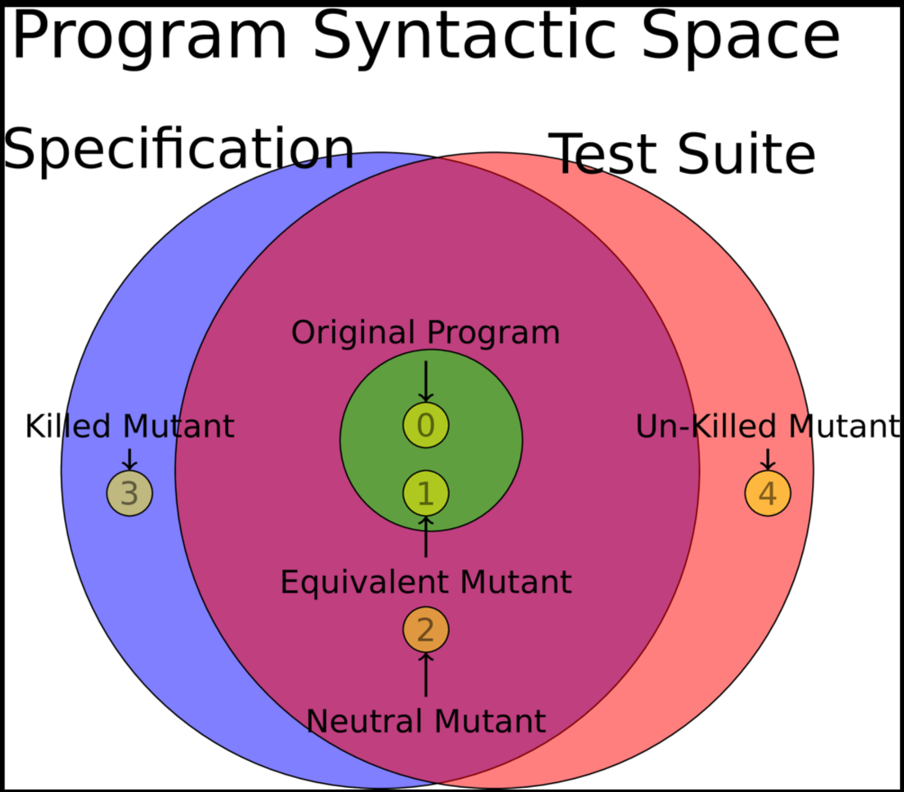
Increasing Trust
Invariants for coarse-grained diversity

- Use dynamic analysis to
  - Show that candidate variants preserve functionality
  - Find the most divergent (sensitive) variants
- UNM Dynamic Invariant Generator (DIG)
  - Generates invariants of neutral variants (and orig. program)
  - Automatically finds nonlinear and array invariants
- Neutral variants retain important functionality (defang)
  - dfsize – 100 == 0 // size of array is preserved
  - str == dfstr – 1008 // ptrs given as input preserve their relative locations
- Combine with fuzzy testing to find diverse candidate variants
  - cp2 – dfstr <= 1000 // predicts defang variants that diverge on all heldout neg tests

Patch Representation
(ICSE'12, GECCO'12)



Resembles Mutation Testing



- MT Goal: Develop test suite to match green circle
- Search for mutants that pass test suite
  - Semantically equivalent
  - OR unkilled
- Mutants are "neutral" if they pass the test suite
  - Semantically equivalent
  - Similar but still in spec

Challenges

- GenProg
  - Does mutational robustness enable GenProg success?
  - How to get beyond single-edit repairs?
- Why should I trust a program constructed/modified by random mutations?
- Mutational robustness
  - *How* is robustness produced?
    - PL design, algorithms, coding practices, etc.
  - *Why* does robustness emerge?
    - Unlike bio, it isn't serving any obvious useful purpose in software
  - *How much* robustness is optimal?
  - How could we answer these questions?
- Evidence of other evolutionary patterns

How do we repair bugs now?

- We ignore them
- We pay expensive programmers to fix them manually
- We develop tools to help the programmers
  - Debuggers, profilers, smart compilers
  - Type checkers
- Mathematical models of program correctness
  - Don't scale up to production software

Categorizing Neutral Mutations

Functional Category	Frequency (/35)
Different whitespace in output	12
Inconsequential state change	10
Extra or redundant computation	6
Equivalent or redundant conditional guard	3
Switched to nonexplicit return	2
Changed code is unreachable	1
Removed optimization	1

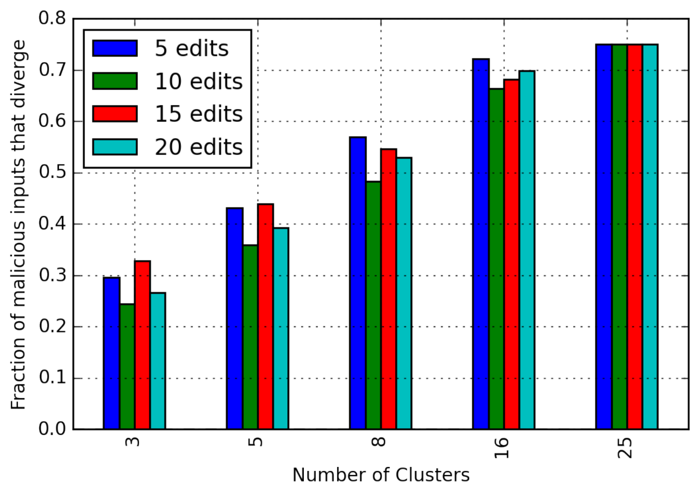
Example Repairs: Security Vulnerabilities

(ICSE'09, TSE'12)

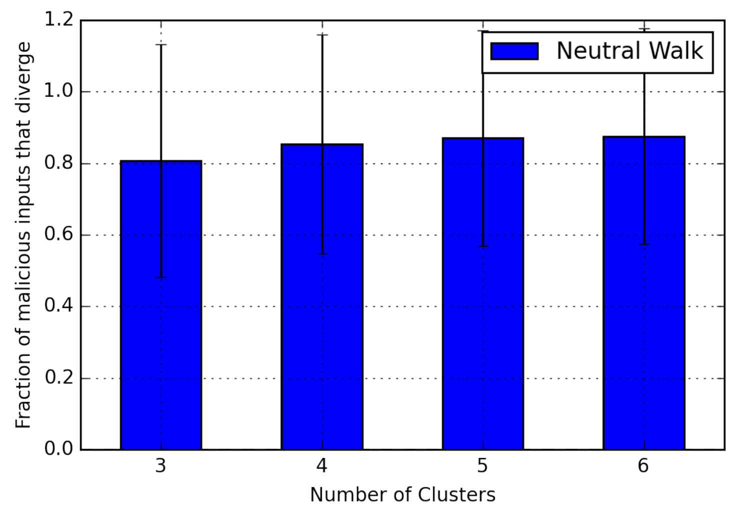
Program	LOC	Path Length	Program Description	Vulnerability	Time to Repair
nullhttp	5575	768	Webserver	Remote heap overflow	578s
openldap	6519	25	Directory protocol	Non-overflow denial-of-service	665s
lighttp	13984	136	Webserver	Remote heap overflow	49s
atris	21553	34	Graphical game	Buffer overflow	80s
php	26044	52	Scripting Language	Integer overflow	6s
wu-ftp	35109	149	FTP server	Format string	2256s
ccrypt	7515	18	Encryption ytility	Seg. fault	47s

Generating neutral variants for defang

- How hard is it to generate multi-edit neutral variants?
- How many variants do we need (on average) to diverge on all buggy inputs?



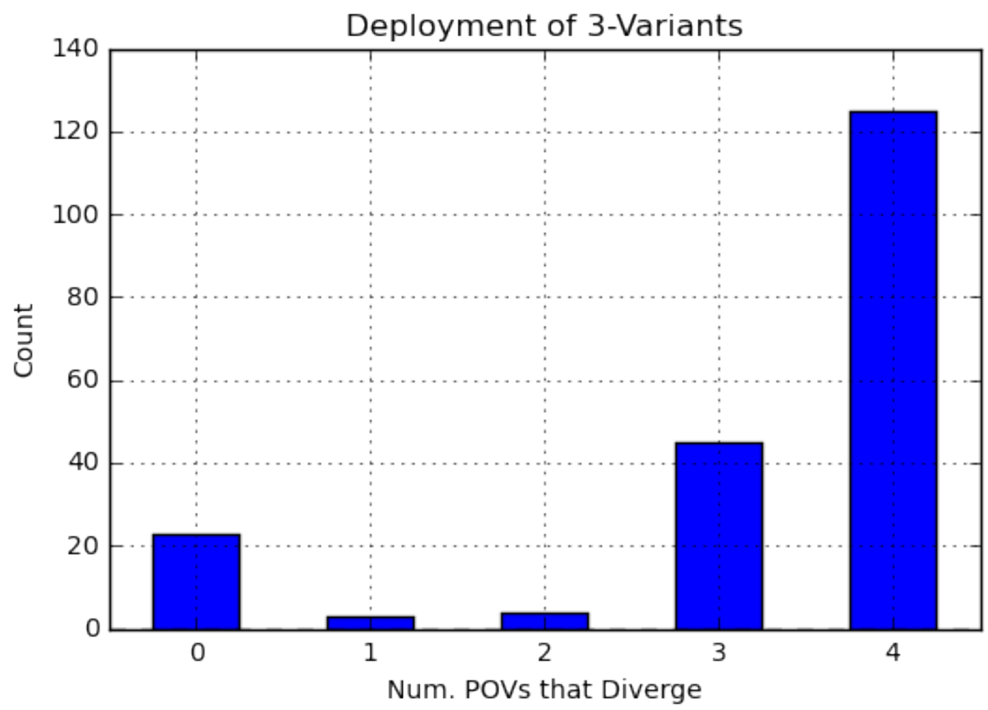
With just 3 clusters, each with 15 edits, we expect to diverge 33% of the time on bug-inducting inputs.



With just 3 clusters, we expect to diverge 80% of the time on bug-inducing inputs

Algorithmic Advance
Generating Candidate Variants

- How many held-out neg. test cases do we diverge on (on average) if we deploy 3 variants?
  - defang experiment (4 held-out neg. test cases)
  - N-prog: We observe divergence 33% of the time on the held-out neg. tests
  - Directed neutral walk (D. Mohr): We observe divergence 80% of the time on the POVs



8/60 variants diverge on all POVs when deployed alone

N-Prog Results (SBST, submitted)

Program	Scenarios	LOC	Tests	Source	Average N-variant System Bug Detection Success				
					$N=3$	$N=5$	$N=8$	$N=16$	$N=\text{All}$
print_tokens	7	472	4,140	Siemens	27.4%	28.4%	28.5%	28.6%	28.6%
print_tokens2	10	399	4,115	Siemens	25.8%	32.0%	36.3%	39.3%	40.0%
replace	31	512	5,542	Siemens	23.1%	26.3%	28.3%	30.1%	32.3%
schedule	9	292	2,650	Siemens	11.0%	11.1%	11.1%	11.1%	11.1%
schedule2	10	301	2,710	Siemens	18.4%	22.9%	26.0%	28.7%	30.0%
tcas	41	141	1,608	Siemens	29.7%	36.8%	41.6%	45.5%	48.7%
tot_info	23	440	1,052	Siemens	19.1%	22.0%	24.8%	28.3%	30.4%
Siemens Total	131	2,557	21,817	Siemens	23.7%	28.1%	31.1%	33.9%	35.9%
checksum	61	13	16	IntroClass	67.7%	70.7%	73.0%	75.8%	78.7%
digits	199	15	16	IntroClass	67.0%	72.5%	76.8%	81.4%	85.4%
grade	252	19	18	IntroClass	75.8%	80.4%	83.6%	85.9%	86.9%
median	170	24	13	IntroClass	68.4%	75.4%	80.3%	84.9%	87.6%
smallest	117	20	16	IntroClass	87.1%	90.6%	92.7%	94.8%	96.6%
syllables	128	23	16	IntroClass	83.4%	85.4%	87.0%	88.8%	89.8%
IntroClass Total	927	114	95	IntroClass	74.5%	79.1%	82.4%	85.6%	87.8%
gzip	5	491K	12	ManyBugs	79.3%	79.9%	80.0%	80.0%	80.0%
php	63	1,046K	8,471	ManyBugs	11.2%	13.3%	15.4%	18.3%	21.0%
ManyBugs Total	68	1,537K	8,483	ManyBugs	16.1%	18.1%	20.1%	22.8%	25.4%
potion	15	15K	220	Schulte et al.	19.1%	22.0%	24.3%	26.2%	26.7%
Overall Total	1,141	1,555K	30,615	-	64.7%	69.1%	72.2%	75.3%	77.5%


.c
Anonymous uploader

N-Prog

- Goals
  - Divergent behavior on heldout/unknown buggy inputs
  - Proactive bug repair
- Based on GenProg
  - Apply coarse-grained (stmt) mutation operators
  - Accept mutations that pass all test cases
  - Combine single mutations (edits) into multi-edit variants
- Early results
  - Proactive repair demonstrated on seed bugs (GPEM, 2014)
  - With sufficient diversity, detects >75% of bugs (in one data set of 16 programs and 1000 bug scenarios), SBST submitted

John was always ahead of his time

- Biology in the era of artificial intelligence
- Statistical learning in the era of expert systems
- Computational thinking in the era of computer engineering
- Interdisciplinarity in the era of specialization
- Agent-based modeling in the era of big data

Scaling up the Evolutionary Process

- Micro-evolution
  - Single bugs
  - Individual programs and packages
- Macro-evolution
  - Evolution over time (multiple edits)
  - Large-scale software systems
  - Human in the loop
- Competitive co-evolution
  - Exploit vs. Repair

Perpetual Novelty

QUESTIONS?

WWW.CS.UNM.EDU/~FORREST

Recombination?

Crossover Operator	Success	Fitness Evals Req'd
No Crossover	54.4%	82.43
Patch Subset	61.1%	163.05
WP One-Point	63.7%	114.12
Patch One-Point	65.2%	118.20

GECCO, 2012

Conclude: Usually, if GenProg succeeds, mutation is sufficient

ROBUSTNESS