Neutral Networks of Real-World Programs and their Application to Automated Software Evolution

Eric Schulte

Department of Computer Science
University of New Mexico
Albuquerque, NM

July 2014

Software Engineering

Motivation

Bugs cost estimated $312 billion per year
[Britton, University of Cambridge 2013]

Software Engineering

Motivation

Bugs cost estimated $312 billion per year
[Britton, University of Cambridge 2013]

Security millions of dollars per exploit
[Van, OECD Publishing 2008]

Software Engineering

Motivation

Bugs cost estimated $312 billion per year
[Britton, University of Cambridge 2013]

Security millions of dollars per exploit
[Van, OECD Publishing 2008]

Resources 1% of global energy to data centers in 2010
[Koomey, Analytics Press 2011]

Software Engineering

Motivation

Bugs cost estimated $312 billion per year
[Britton, University of Cambridge 2013]

Security millions of dollars per exploit
[Van, OECD Publishing 2008]

Resources 1% of global energy to data centers in 2010
[Koomey, Analytics Press 2011]

Developers projected 1.5 million in US by 2018
[U.S Department of Labor, 2011]

Software Engineering

Evolution and Automation

Evolved Ecosystem

- applications
- libraries
- compilers
- operating systems
- architectures

Software Engineering

Evolution and Automation

Evolved Ecosystem

- applications
- libraries
- compilers
- operating systems
- architectures

Evolvable Software

- automated diversity
- assembly and binary repair
- patch closed source executable
- optimization

Genprog

Automatically Fix Bugs in C Software

Genprog

Automatically Fix Bugs in C Software

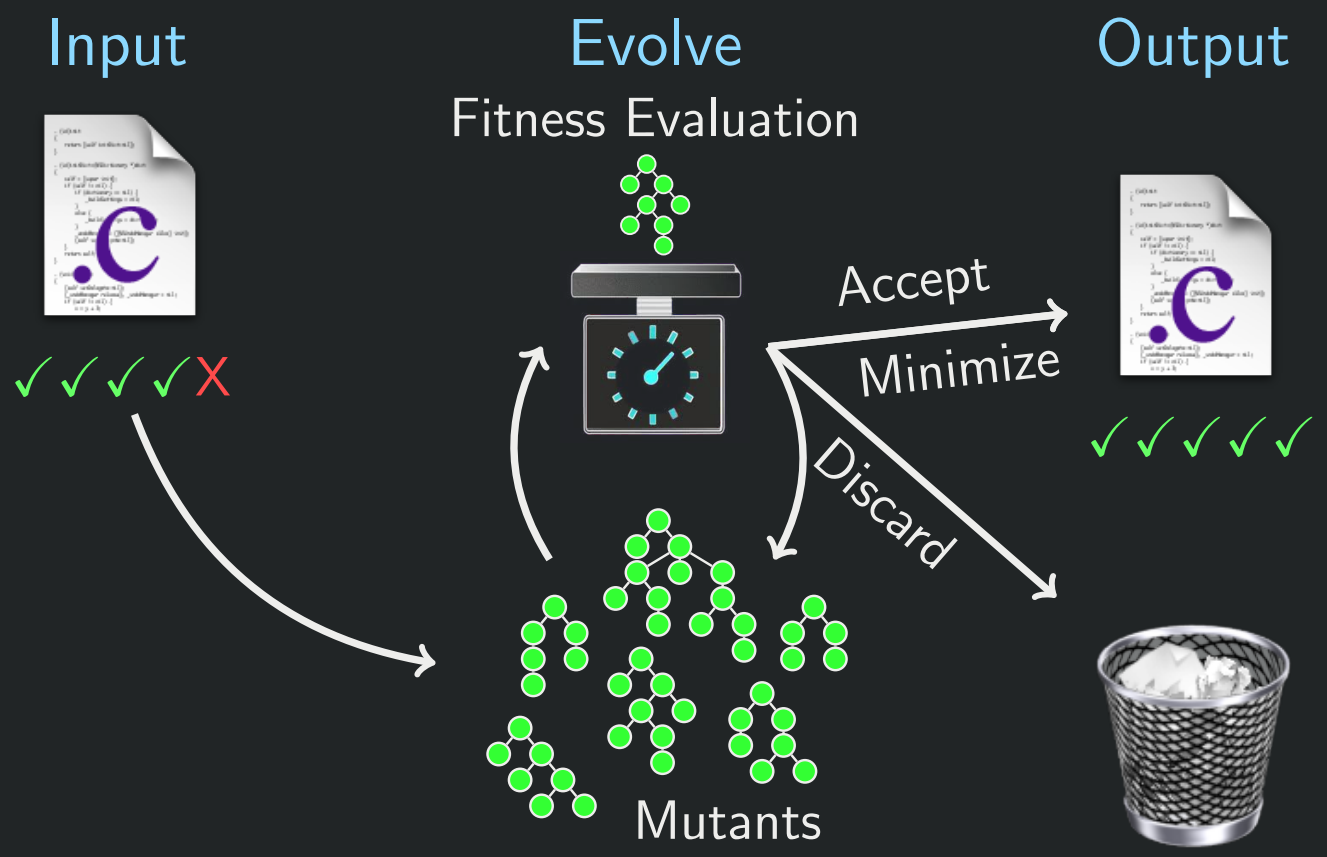
Collaboration between UNM and UVA

Dr. Stephanie Forrest

Dr. Westley Weimer

Genprog

Automatically Fix Bugs in C Software



[Le Goues, 2013, Figure 3.2]

Genprog

Automatically Fix Bugs in C Software

Strengths

Effective	Repaired 55/105 bugs for $8 each
General	Multiple classes of bugs and security defects
Best Papers	ICSE 2009, GECCO 2009, SBST 2009
Humies	Gold 2009, Bronze 2012

Genprog

Automatically Fix Bugs in C Software

But

Why doesn't this break my software?

Outline

Program Representation

Source

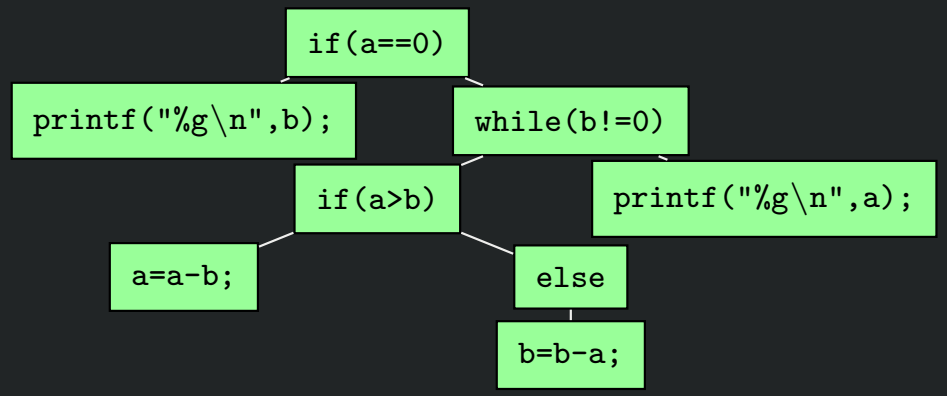
```
if(a==0){
  printf("%g\n", b); }
  else {
    while(b!=0){
      if(a>b){ a=a-b; }
      else   { b=b-a; }}}
  printf("%g\n", a);
```

Program Representation

Source

```
if(a==0){
  printf("%g\n", b); }
  else {
    while(b!=0){
      if(a>b){ a=a-b; }
      else   { b=b-a; }}}
  printf("%g\n", a);
```

Tree

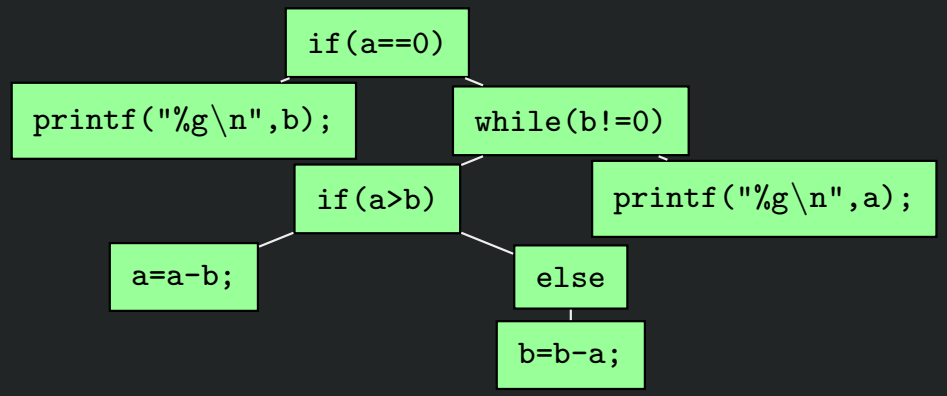


Program Representation

Source

```
if(a==0){
  printf("%g\n", b); }
  else {
    while(b!=0){
      if(a>b){ a=a-b; }
      else   { b=b-a; }}}
  printf("%g\n", a);
```

Tree



Vector

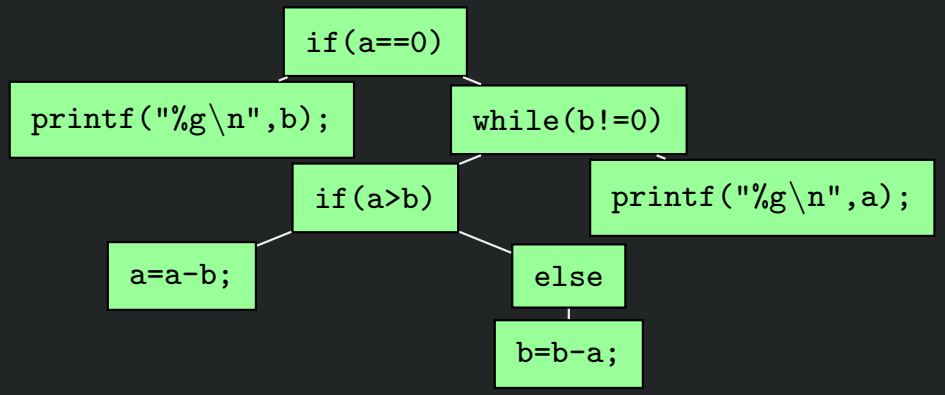
```
main:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $48, %rsp
```

Program Representation

Representations

Rep.	Type	Languages
CLang	tree	C-family source
CIL	tree	C Int. Lang.
LLVM	vector	LLVM SSA IR
ASM	vector	x86 & ARM
ELF	vector	x86 & MIPS

Tree: CIL, CLang

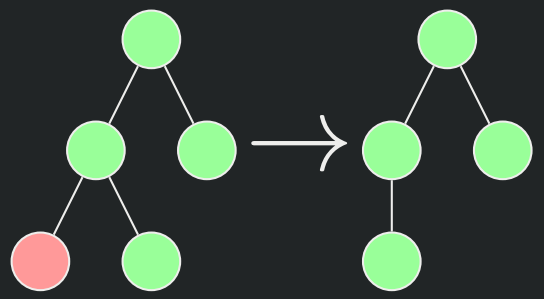


Vector: ASM, LLVM, ELF

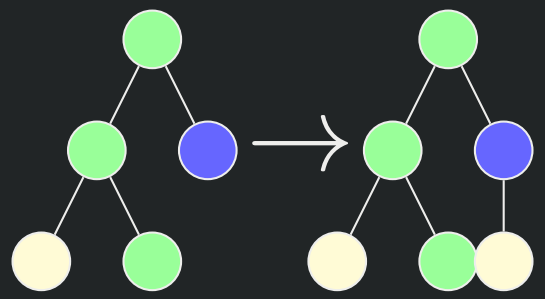
```
main:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $48, %rsp
```

Program Mutation Operations

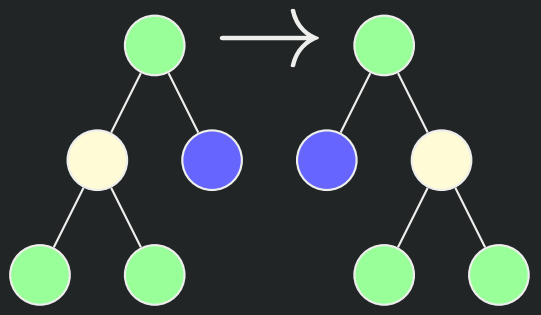
Tree Delete



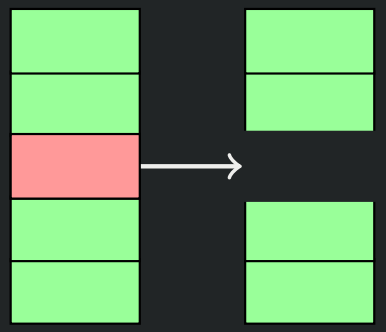
Tree Insert



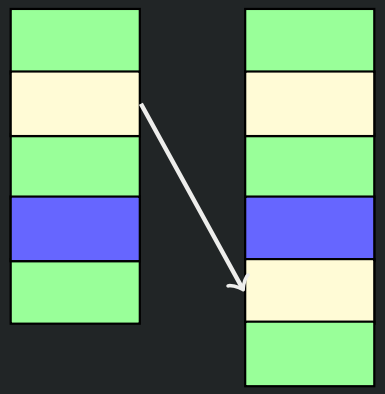
Tree Swap



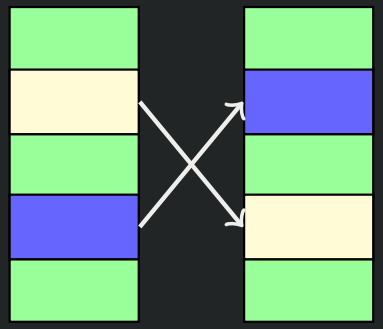
Vector Delete



Vector Insert

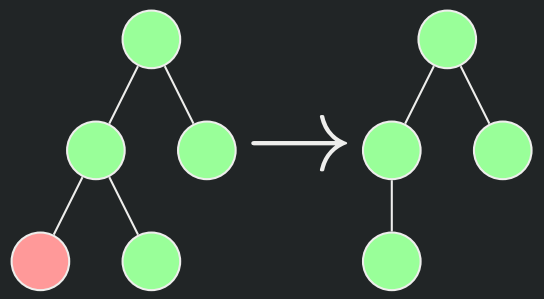


Vector Swap

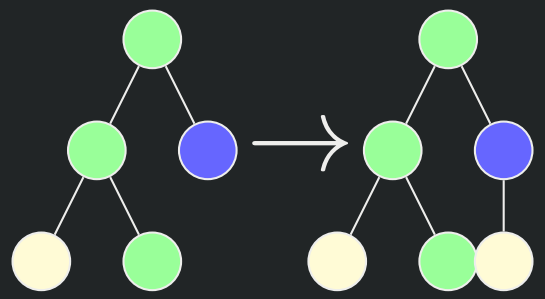


Program Mutation Operations

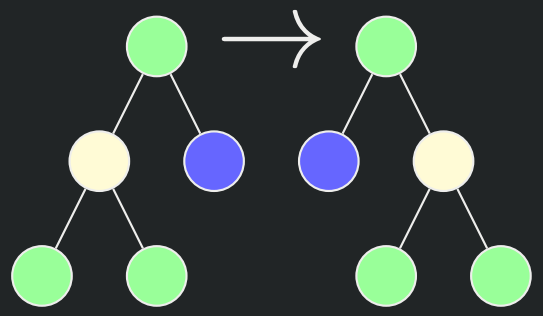
Tree Delete



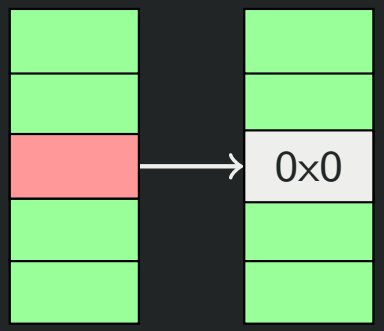
Tree Insert



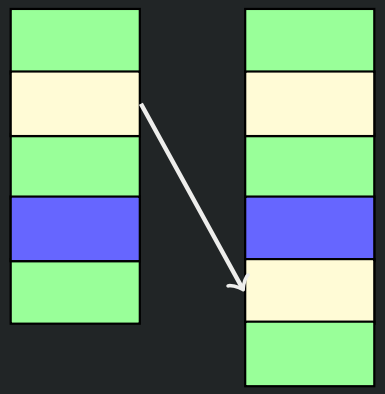
Tree Swap



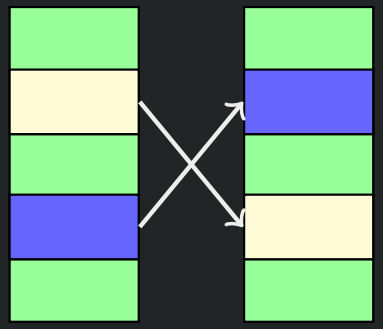
ELF Delete



Vector Insert

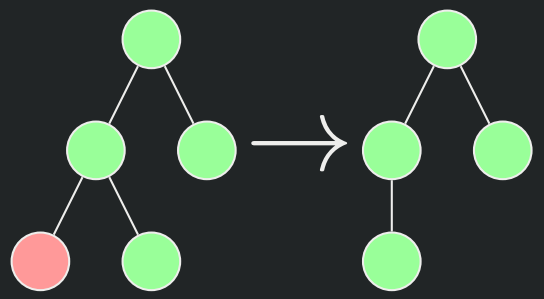


Vector Swap

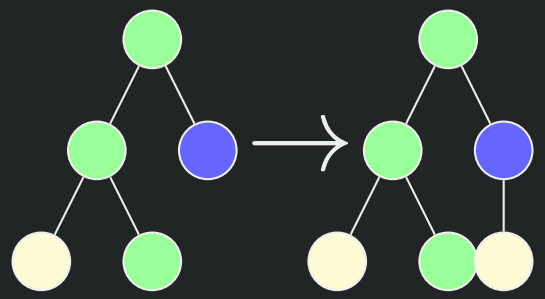


Program Mutation Operations

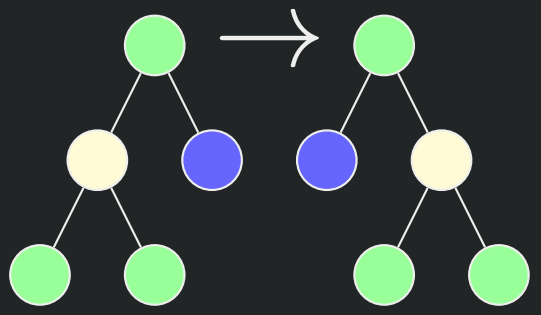
Tree Delete



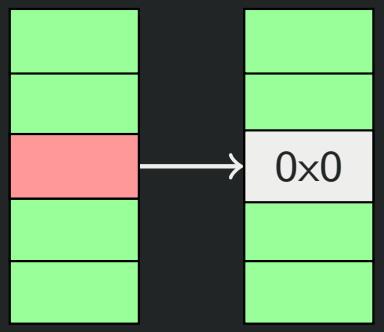
Tree Insert



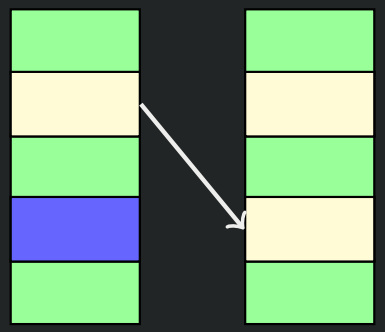
Tree Swap



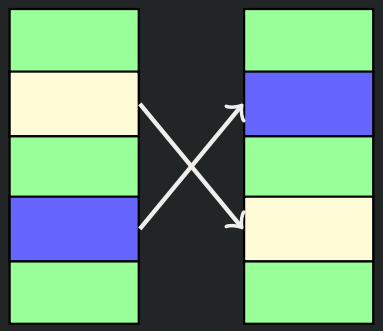
ELF Delete



ELF Replace



Vector Swap



LLVM Mutation Illustration

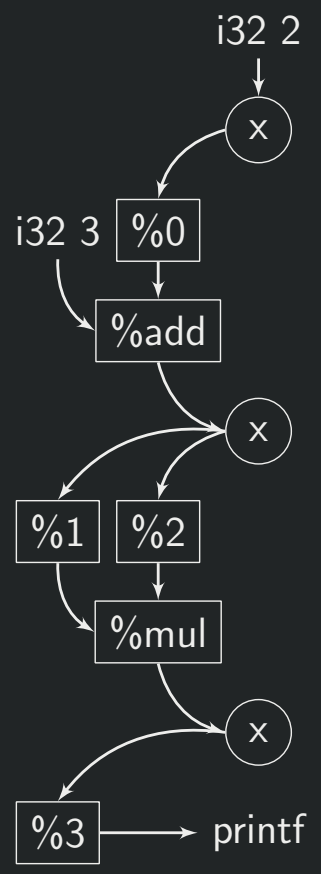
C source

```
int main(int argc, char *argv[]){
  int x=2;
  x+=3;
  x=x*x;
  printf("%d\n", x);
  return 0;
}
```

LLVM Mutation Illustration

LLVM

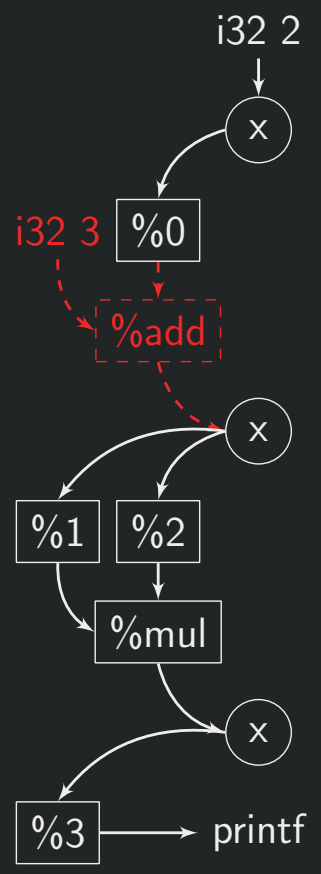
```
%x = alloca i32, align 4
store i32 2, i32* %x, align 4
%0 = load i32* %x, align 4
%add = add nsw i32 %0, 3
store i32 %0, i32* %x, align 4
%1 = load i32* %x, align 4
%2 = load i32* %x, align 4
%mul = mul nsw i32 %1, %2
store i32 %mul, i32* %x, align 4
%3 = load i32* %x, align 4
%call = call ... @printf(@.str %3)
```



LLVM Mutation Illustration

LLVM: Delete 4

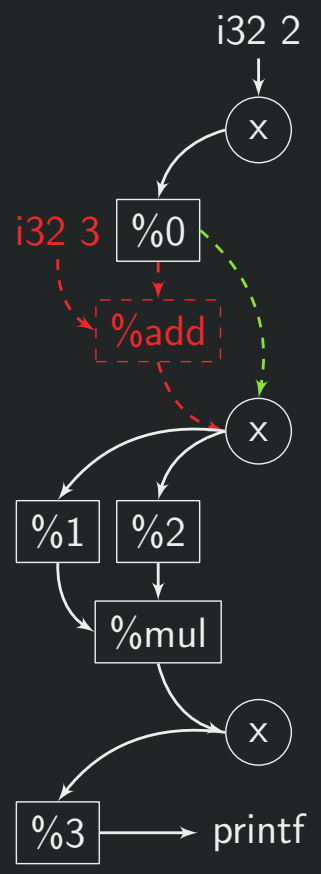
```
%x = alloca i32, align 4
store i32 2, i32* %x, align 4
%0 = load i32* %x, align 4
%add = add nsw i32 %0, 3
store i32 %0, i32* %x, align 4
%1 = load i32* %x, align 4
%2 = load i32* %x, align 4
%mul = mul nsw i32 %1, %2
store i32 %mul, i32* %x, align 4
%3 = load i32* %x, align 4
%call = call ... @printf(@.str %3)
```



LLVM Mutation Illustration

LLVM: Delete 4

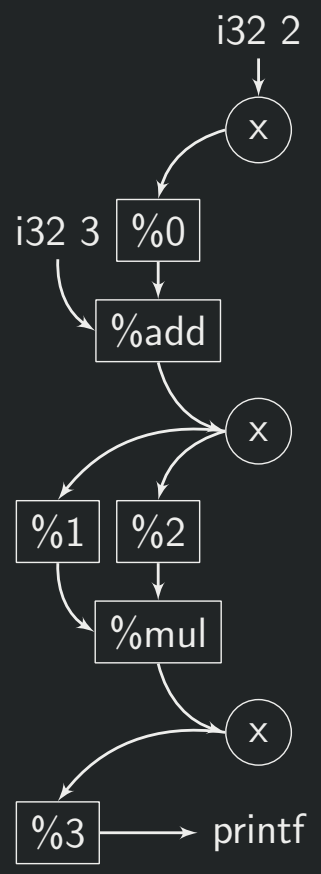
```
%x = alloca i32, align 4
store i32 2, i32* %x, align 4
%0 = load i32* %x, align 4
~~%add = add nsw i32 %0, 3~~
store i32 ~~%0~~, i32* %x, align 4
%1 = load i32* %x, align 4
%2 = load i32* %x, align 4
%mul = mul nsw i32 %1, %2
store i32 %mul, i32* %x, align 4
%3 = load i32* %x, align 4
%call = call ... @printf(@.str %3)
```



LLVM Mutation Illustration

LLVM: Insert 9 4

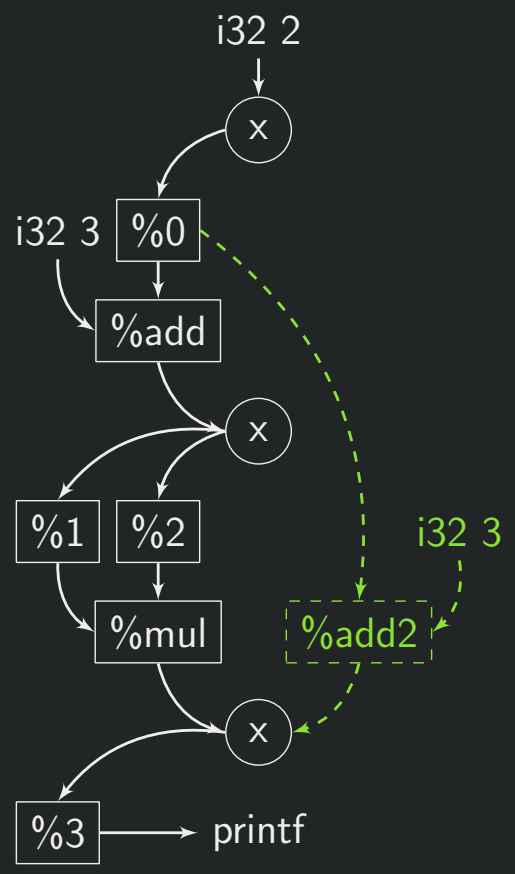
```
%x = alloca i32, align 4
store i32 2, i32* %x, align 4
%0 = load i32* %x, align 4
%add = add nsw i32 %0, 3
store i32 %add, i32* %x, align 4
%1 = load i32* %x, align 4
%2 = load i32* %x, align 4
%mul = mul nsw i32 %1, %2

store i32 %mul, i32* %x, align 4
%3 = load i32* %x, align 4
%call = call ... @printf(@.str %3)
```



LLVM Mutation Illustration

LLVM: Insert 9 4

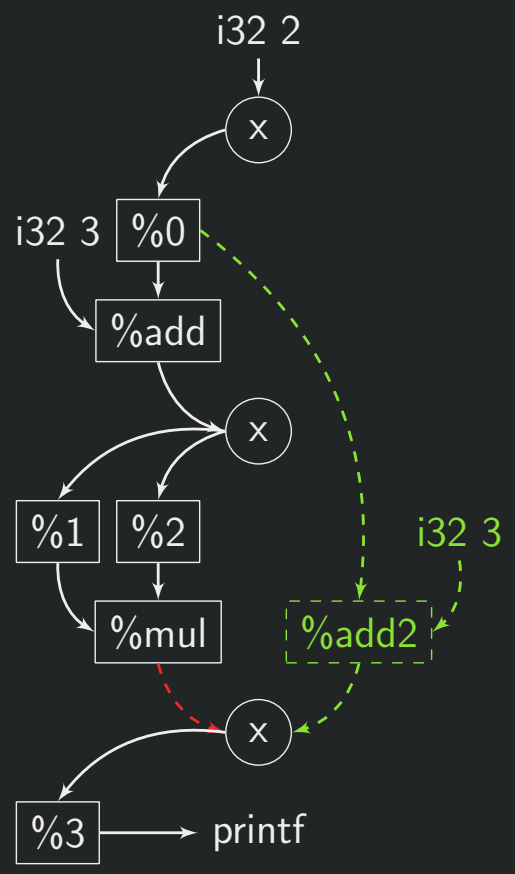
```
%x = alloca i32, align 4
store i32 2, i32* %x, align 4
%0 = load i32* %x, align 4
%add = add nsw i32 %0, 3
store i32 %add, i32* %x, align 4
%1 = load i32* %x, align 4
%2 = load i32* %x, align 4
%mul = mul nsw i32 %1, %2
%add.insert = add nsw i32 %0, 3
store i32 %mul, i32* %x, align 4
%3 = load i32* %x, align 4
%call = call ... @printf(@.str %3)
```



LLVM Mutation Illustration

LLVM: Insert 9 4

```
%x = alloca i32, align 4
store i32 2, i32* %x, align 4
%0 = load i32* %x, align 4
%add = add nsw i32 %0, 3
store i32 %add, i32* %x, align 4
%1 = load i32* %x, align 4
%2 = load i32* %x, align 4
%mul = mul nsw i32 %1, %2
%add.insert = add nsw i32 %0, 3
store i32 %mul, i32* %x, align 4
%3 = load i32* %x, align 4
%call = call ... @printf(@.str %3)
```



Conclusion

Representations

Rep.	Type	Languages
CLang	tree	C-family source
CIL	tree	C Int. Lang.
LLVM	vector	LLVM SSA IR
ASM	vector	x86 & ARM
ELF	vector	x86 & MIPS

Differences

- Mutation operations and Search Space
- Languages
- Requirements
- Expression as executables
- Communication of mutations

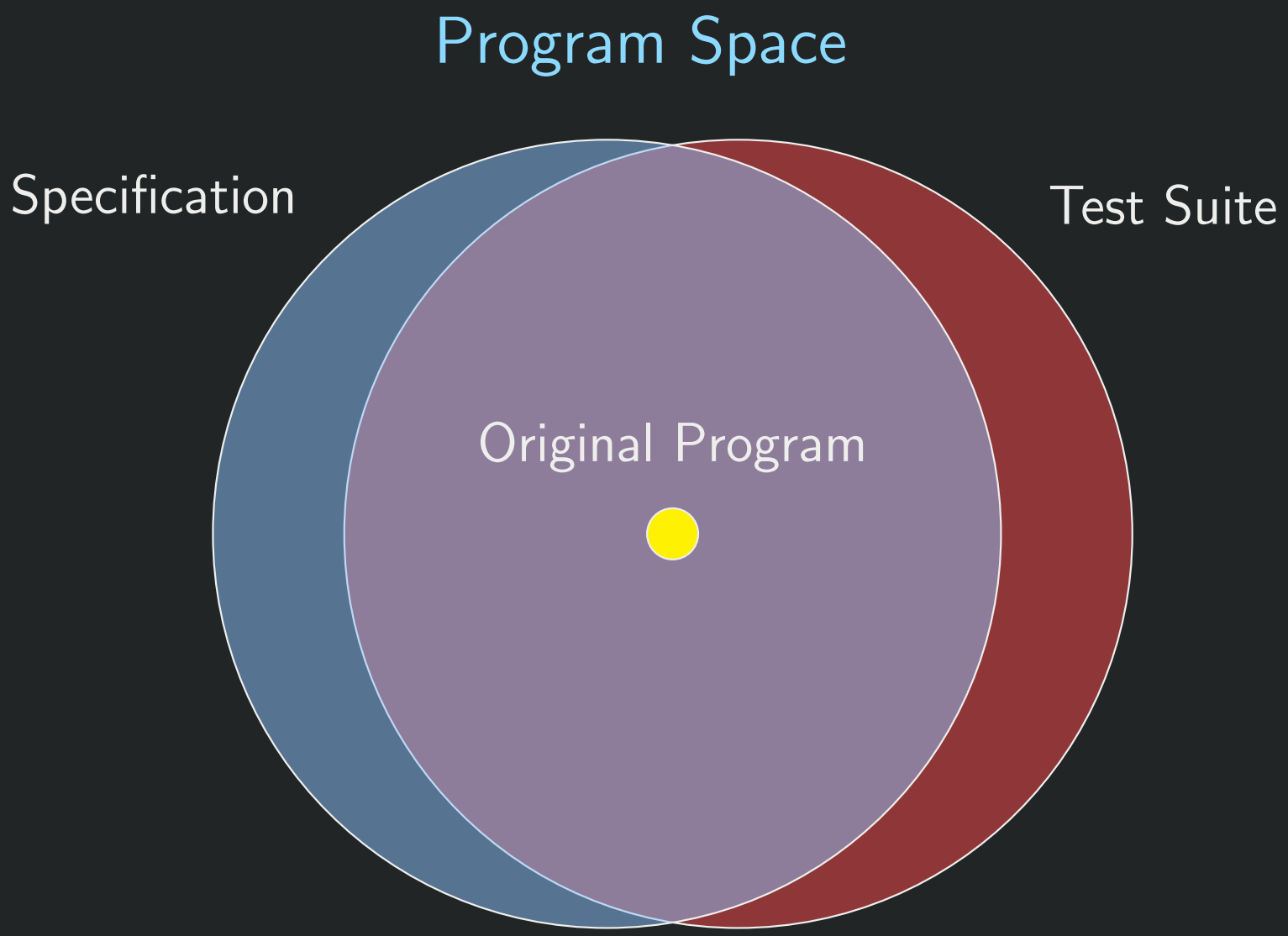
Outline

Software Mutational Robustness

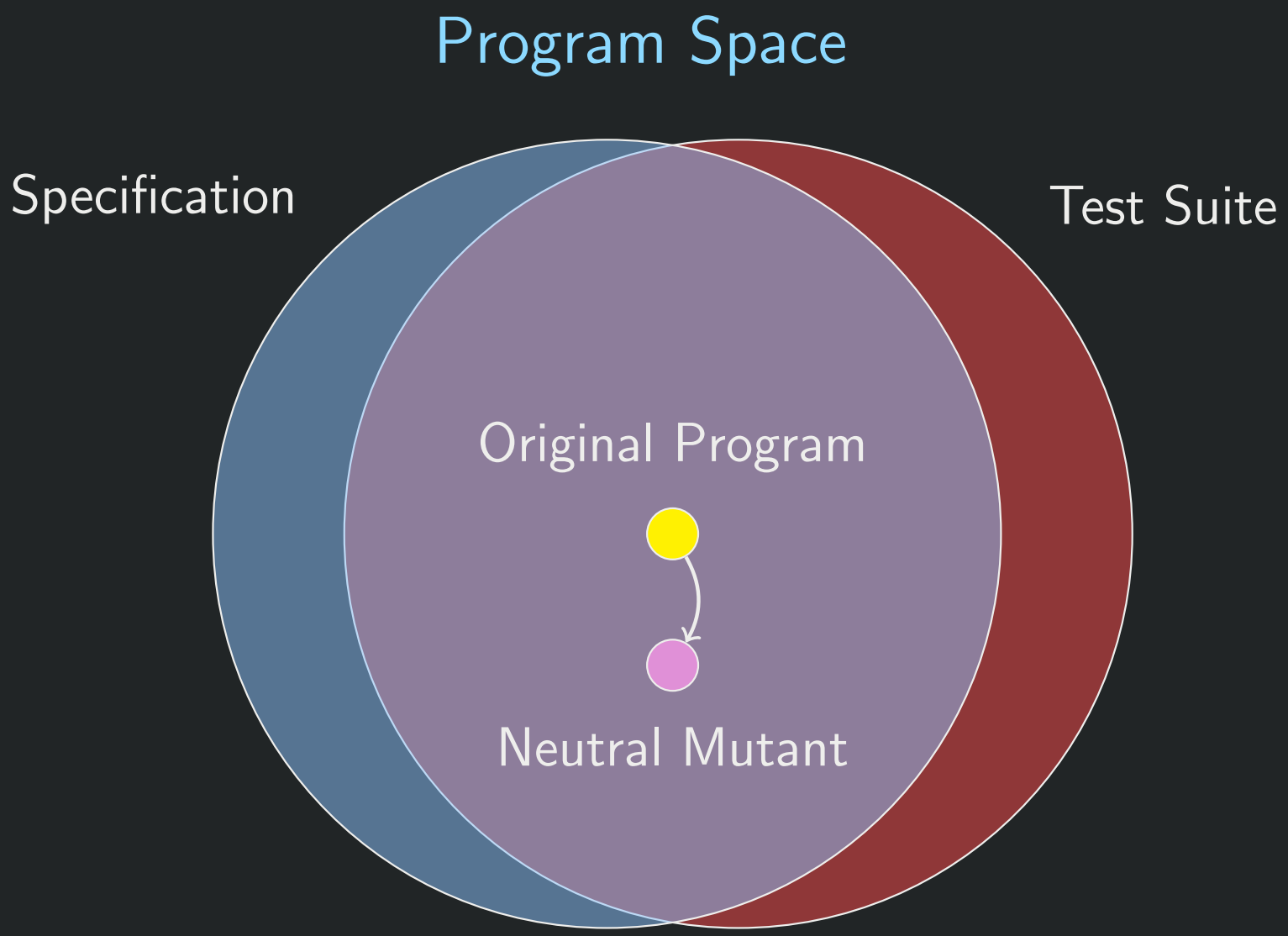
[Schulte *et al.*, GPEM 2013]

percentage of mutants which are functional

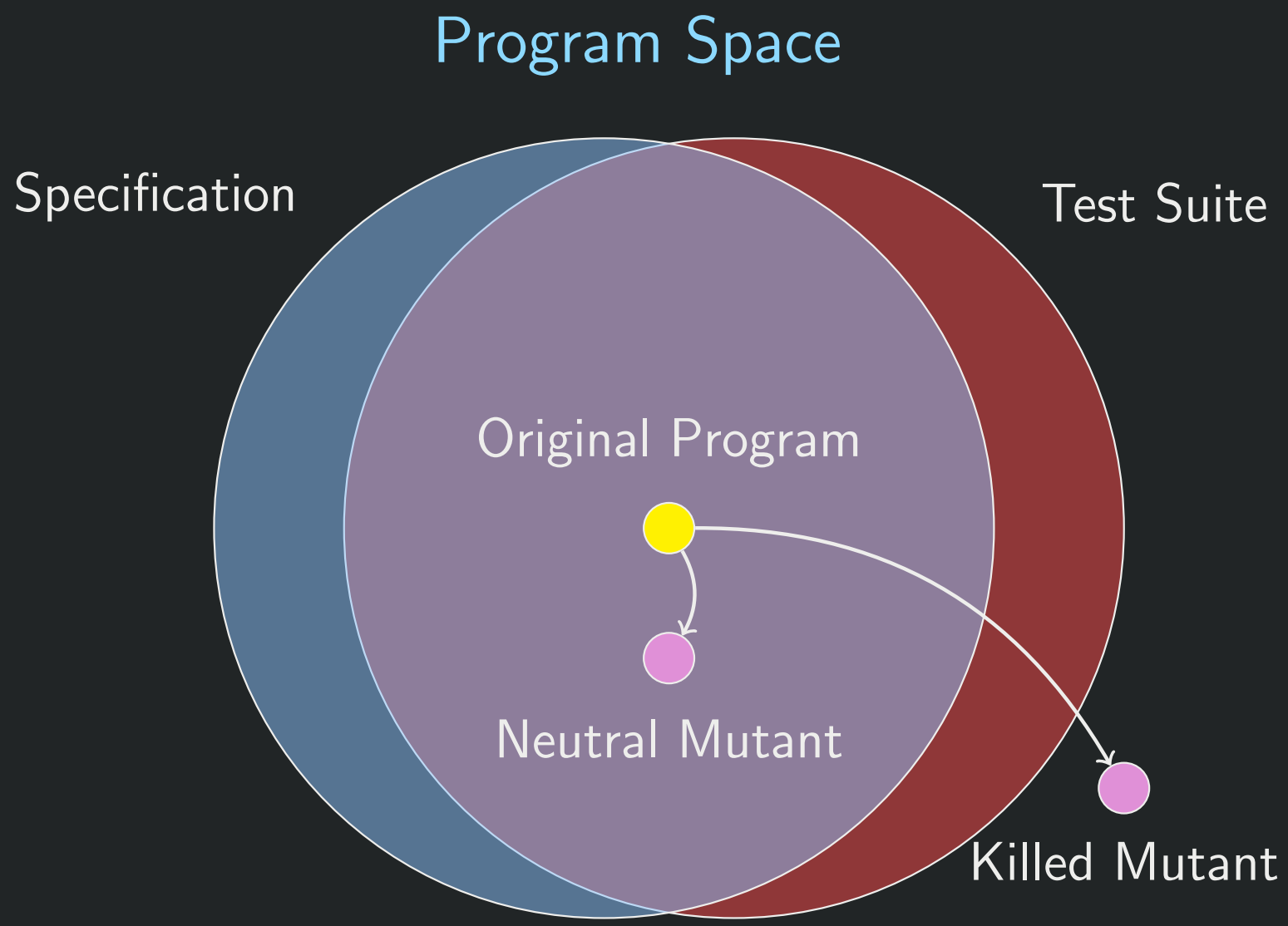
Software Mutational Robustness



Software Mutational Robustness



Software Mutational Robustness



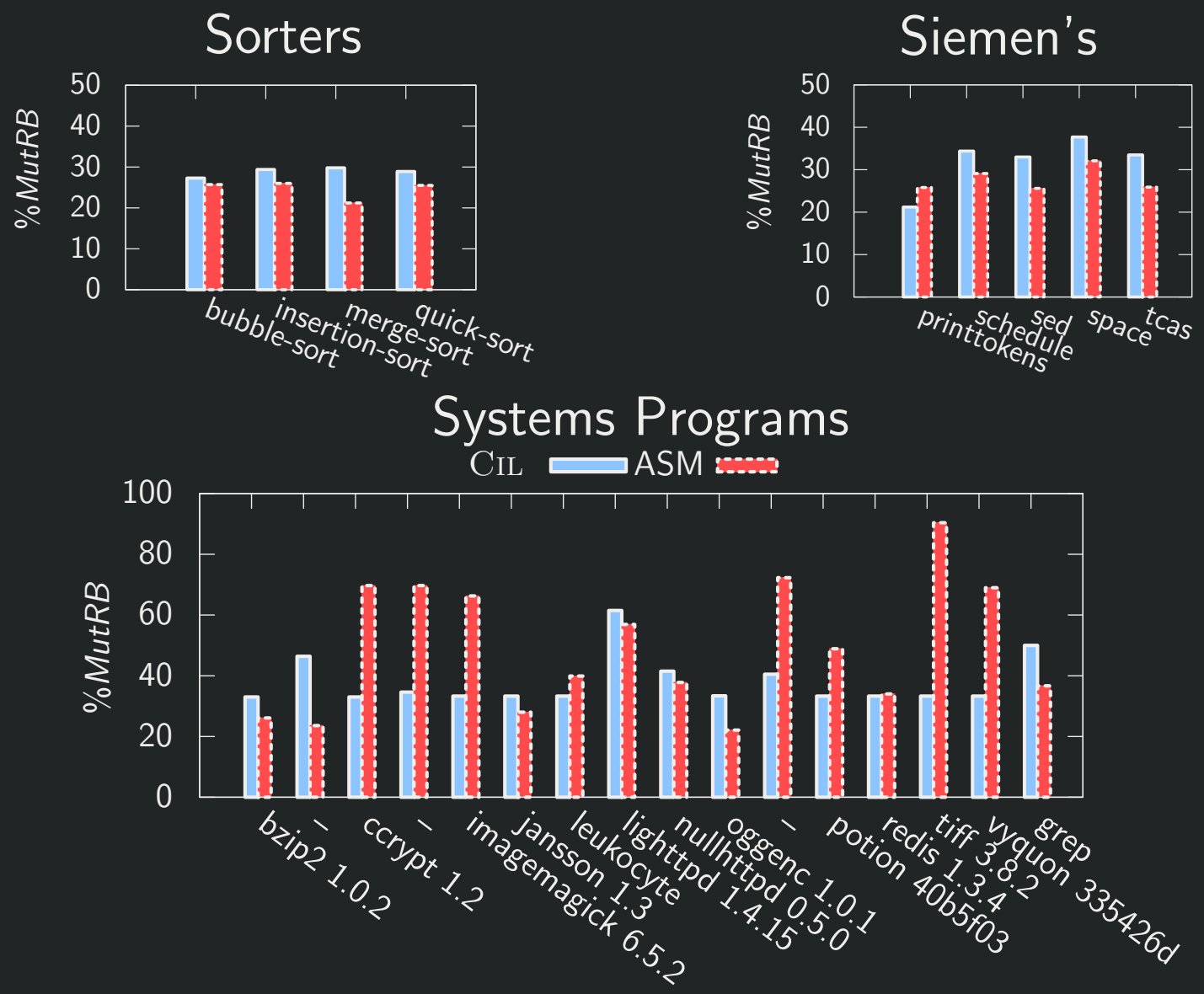
Definition

$MutRB(P, T, M)$

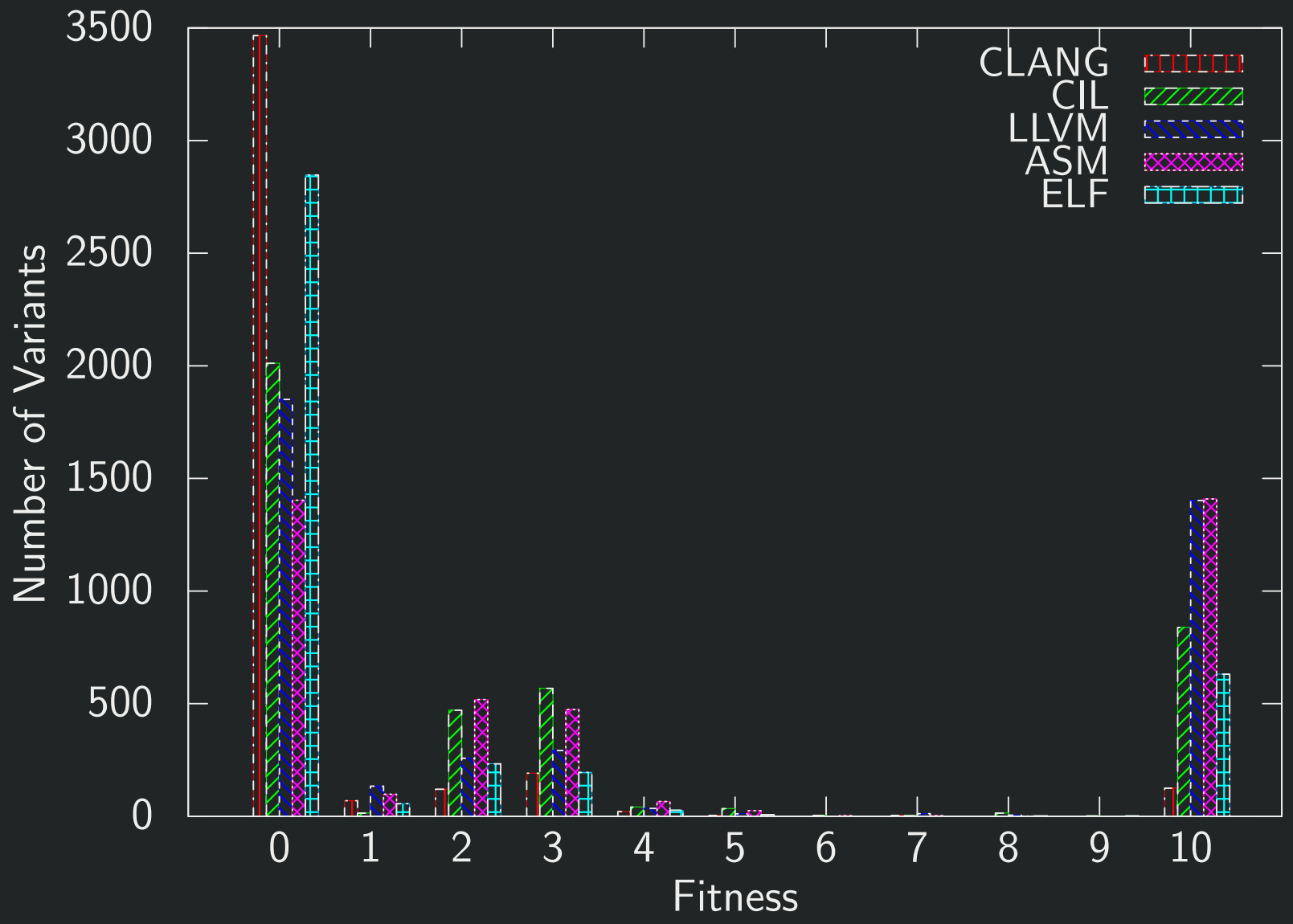
P	program
T	test suite
M	mutation operators

$$MutRB(P, T, M) = \frac{|\{P' \mid m \in M.\ P' \leftarrow m(P) \ \wedge \ T(P')\}|}{|\{P' \mid m \in M.\ P' \leftarrow m(P)\}|}$$

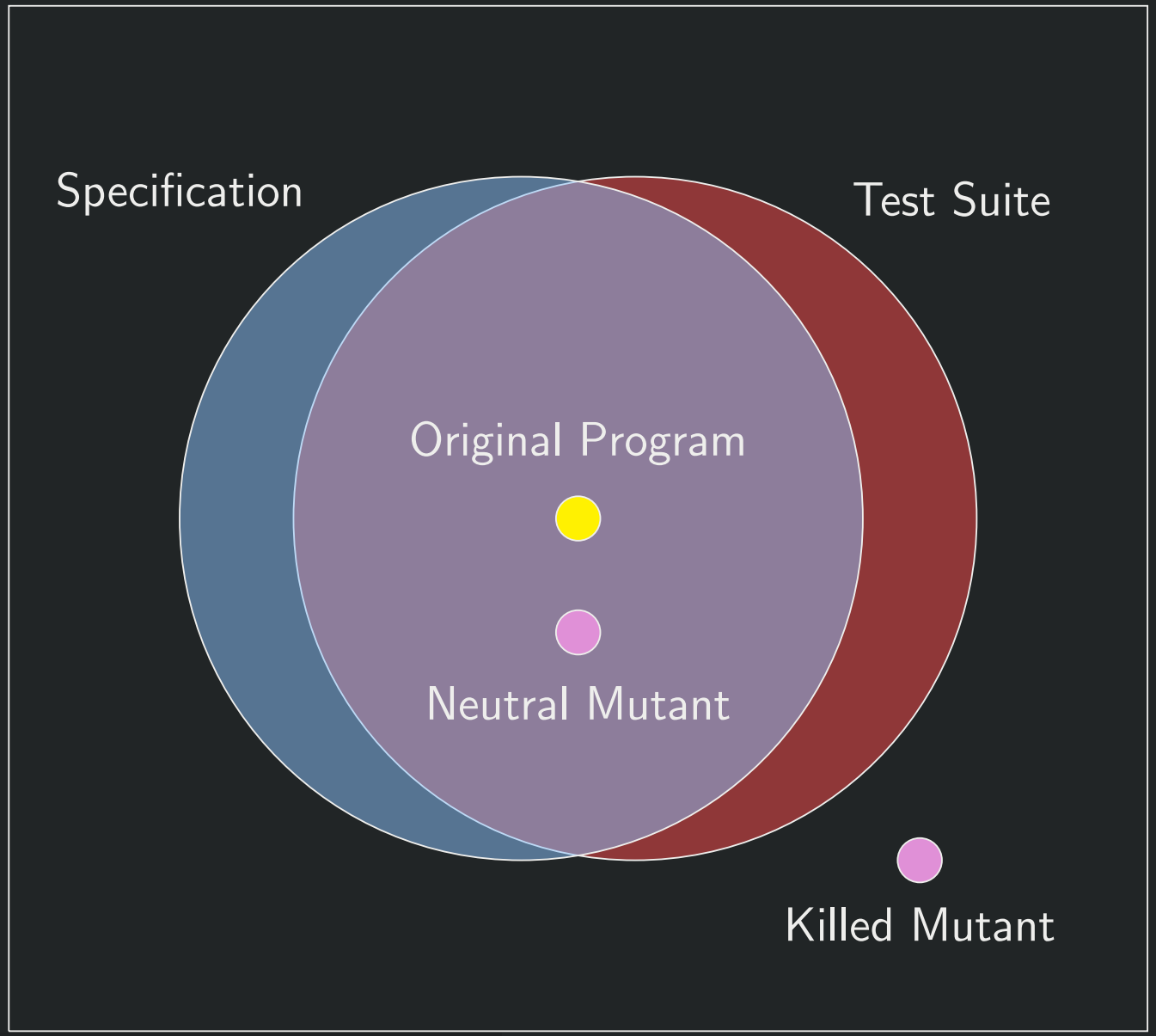
Mutational Robustness by Test Suite



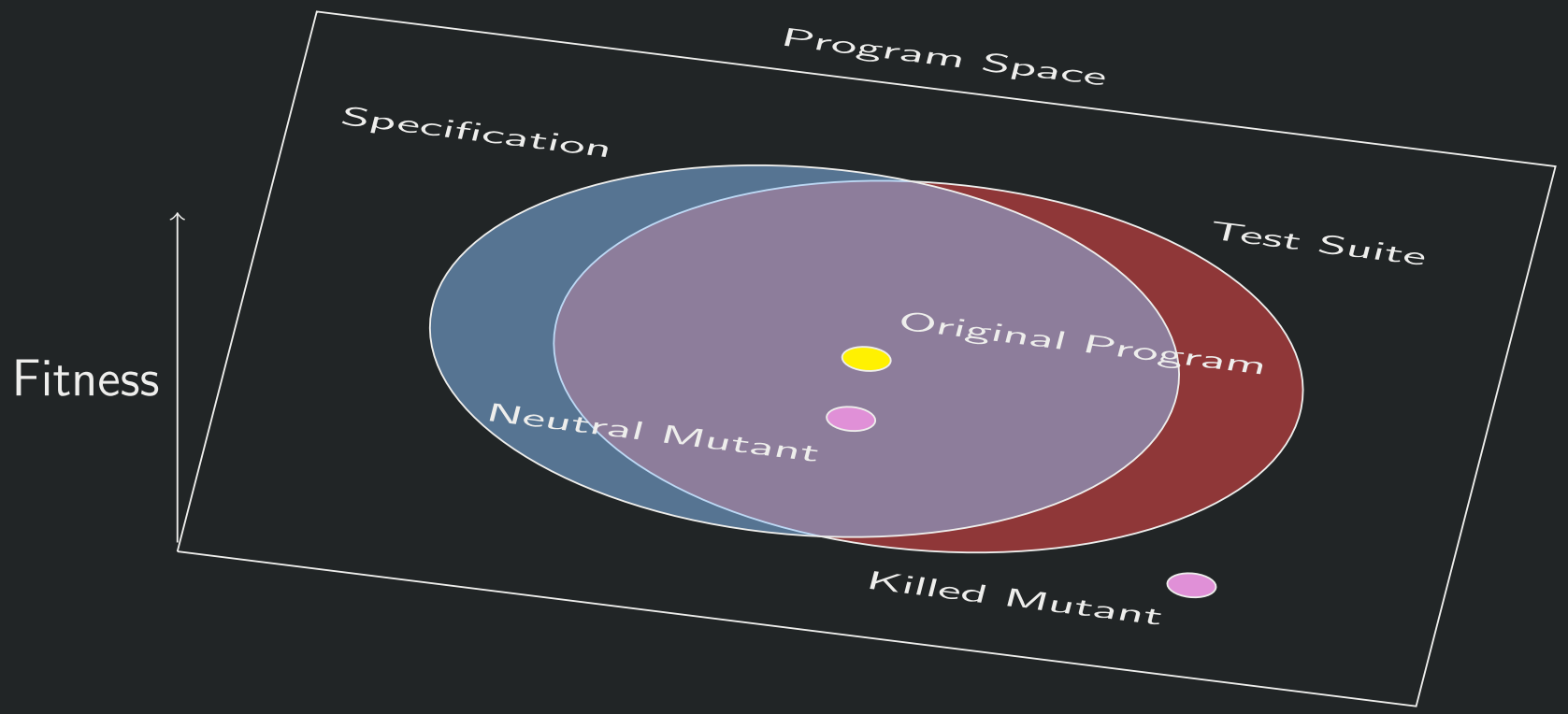
Fitness Distribution by Representation



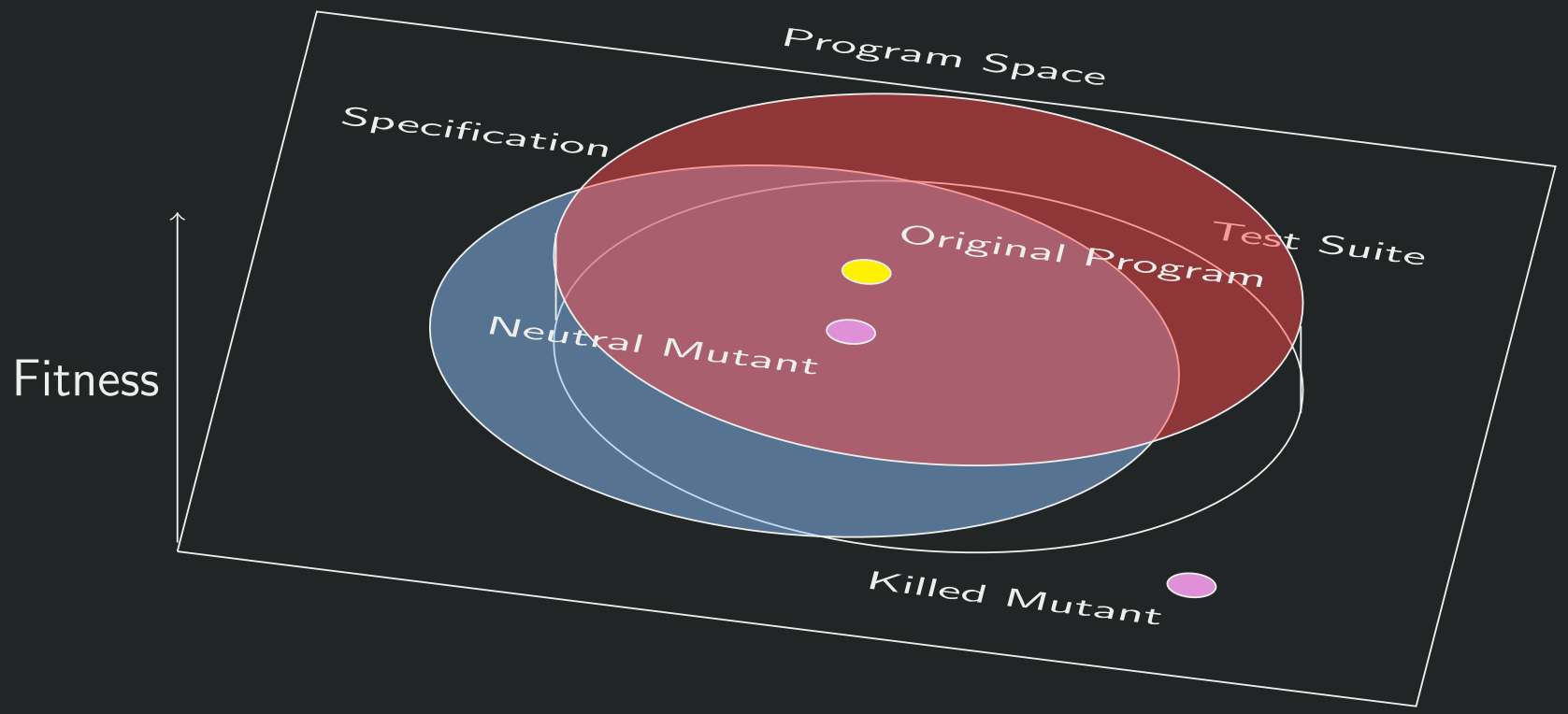
Fitness Landscapes and Neutral Networks



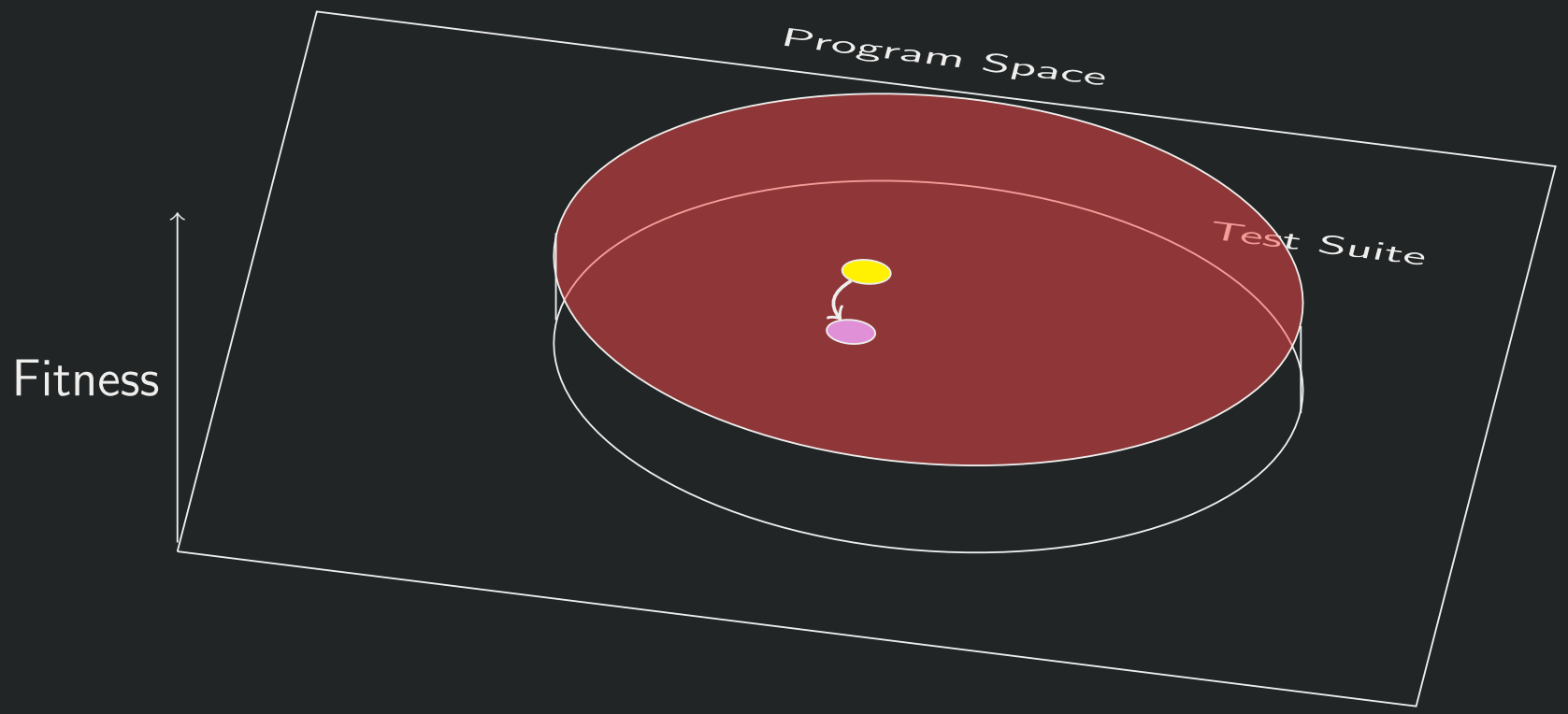
Fitness Landscapes and Neutral Networks



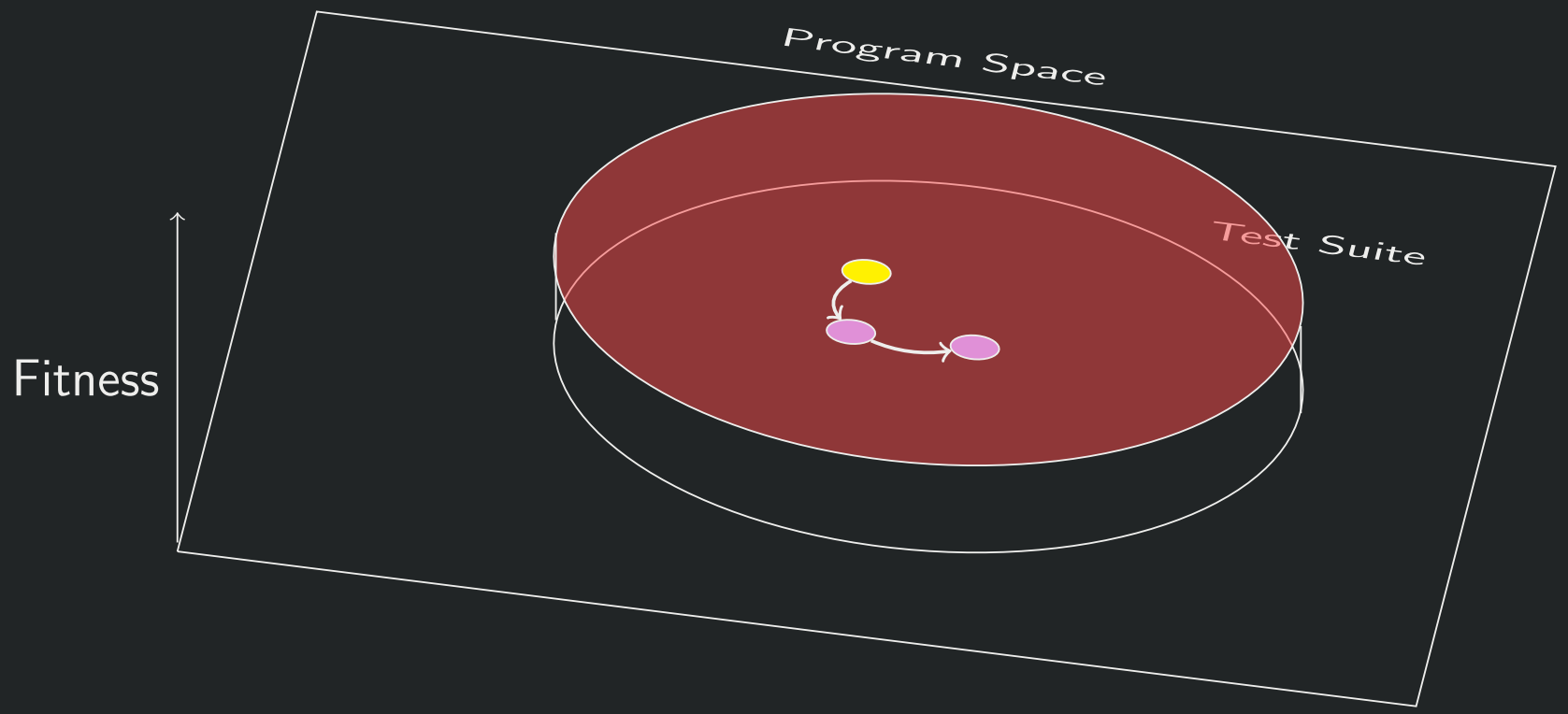
Fitness Landscapes and Neutral Networks



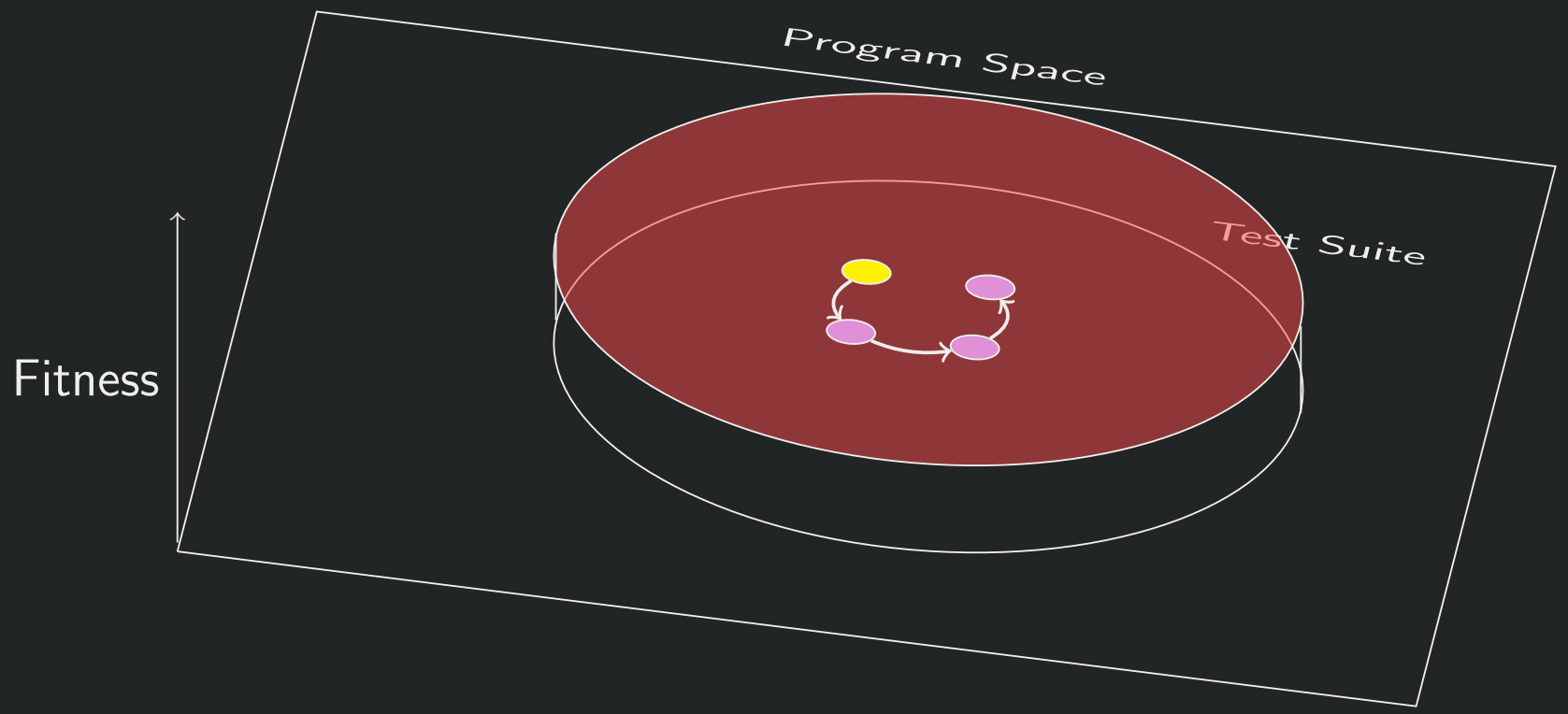
Fitness Landscapes and Neutral Networks



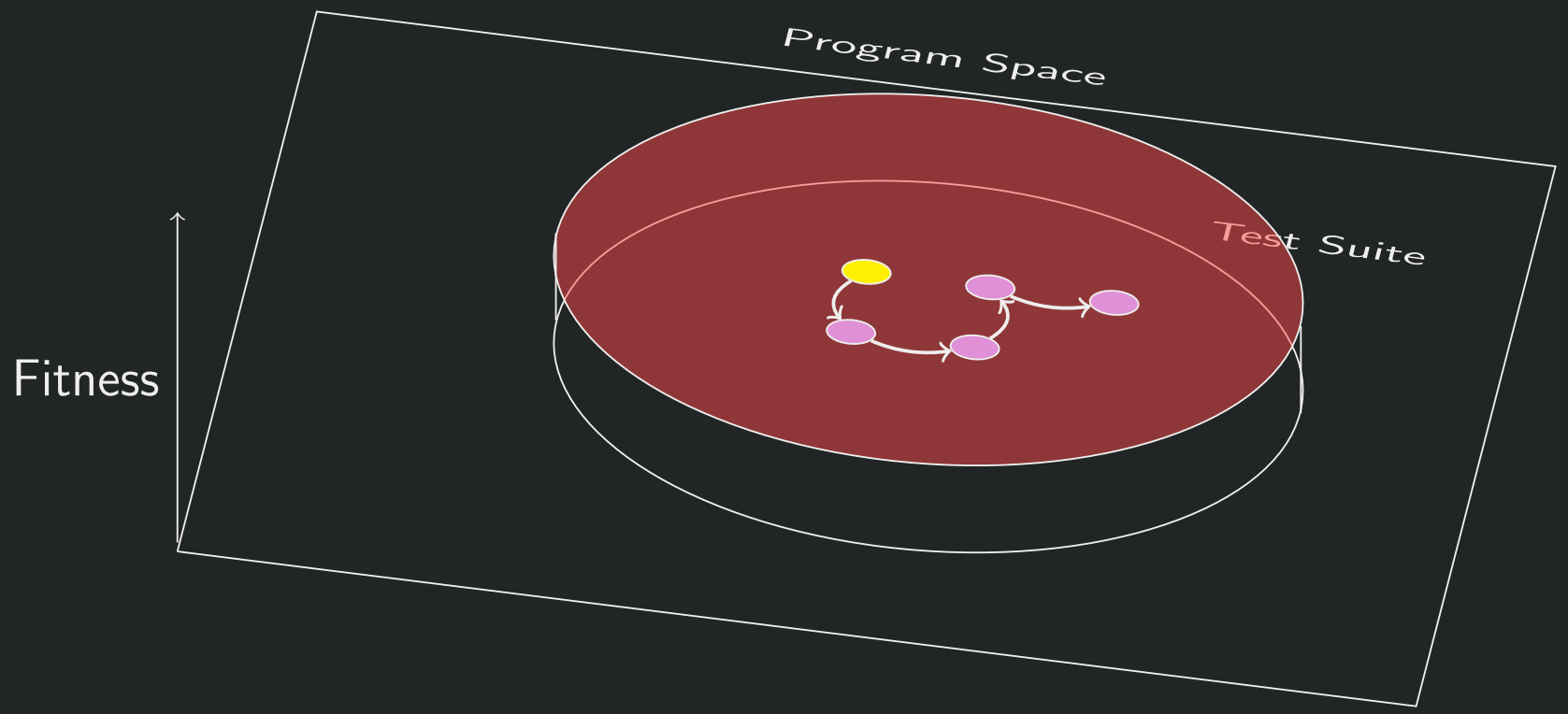
Fitness Landscapes and Neutral Networks



Fitness Landscapes and Neutral Networks

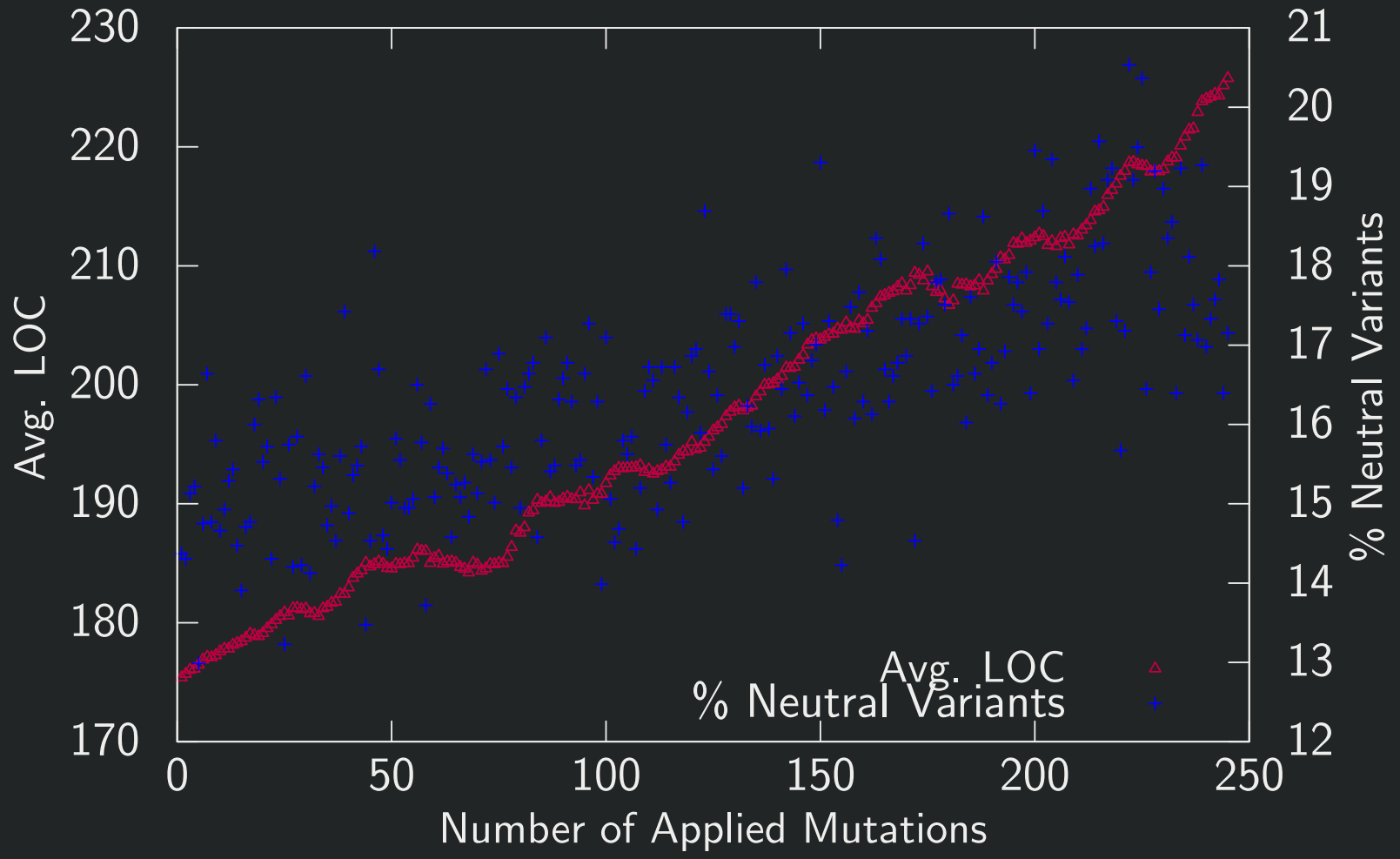


Fitness Landscapes and Neutral Networks



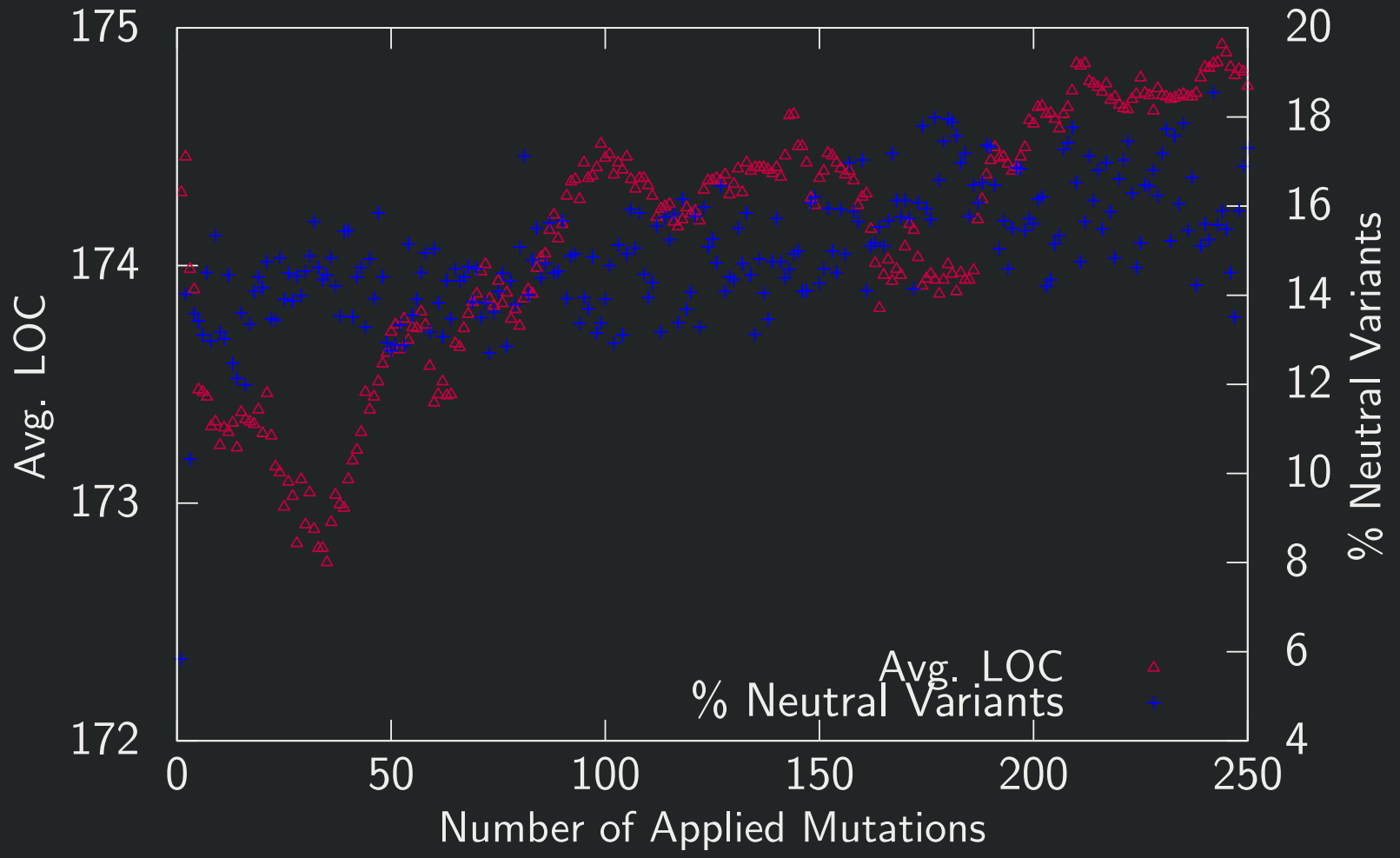
Span of Neutral Networks

Insertion Sort Neutral Variants



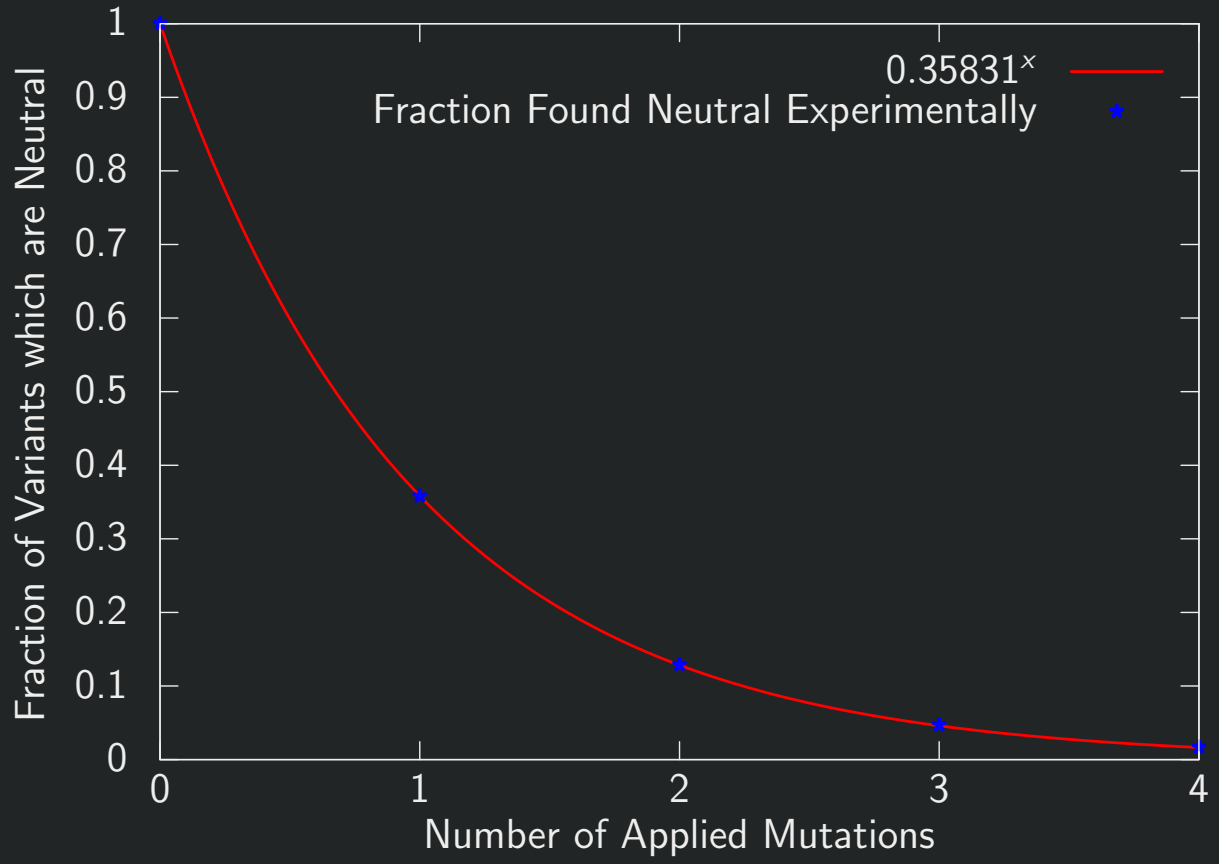
Span of Neutral Networks

Insertion Sort Neutral Variants: size controlled



Random Walks in Program Space

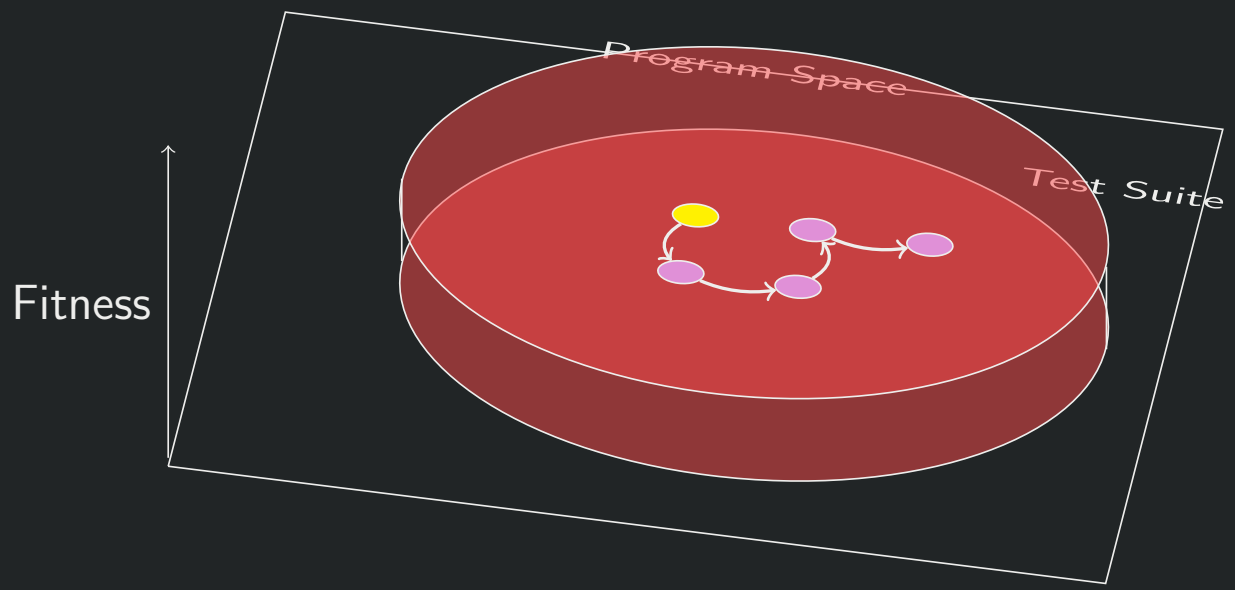
Quick Sort Random Variants



Conclusion

Software Neutral Networks are

- **wide** span large distances in program space
- **thick** consistent mutational robustness
- **accessible** automatically explorable



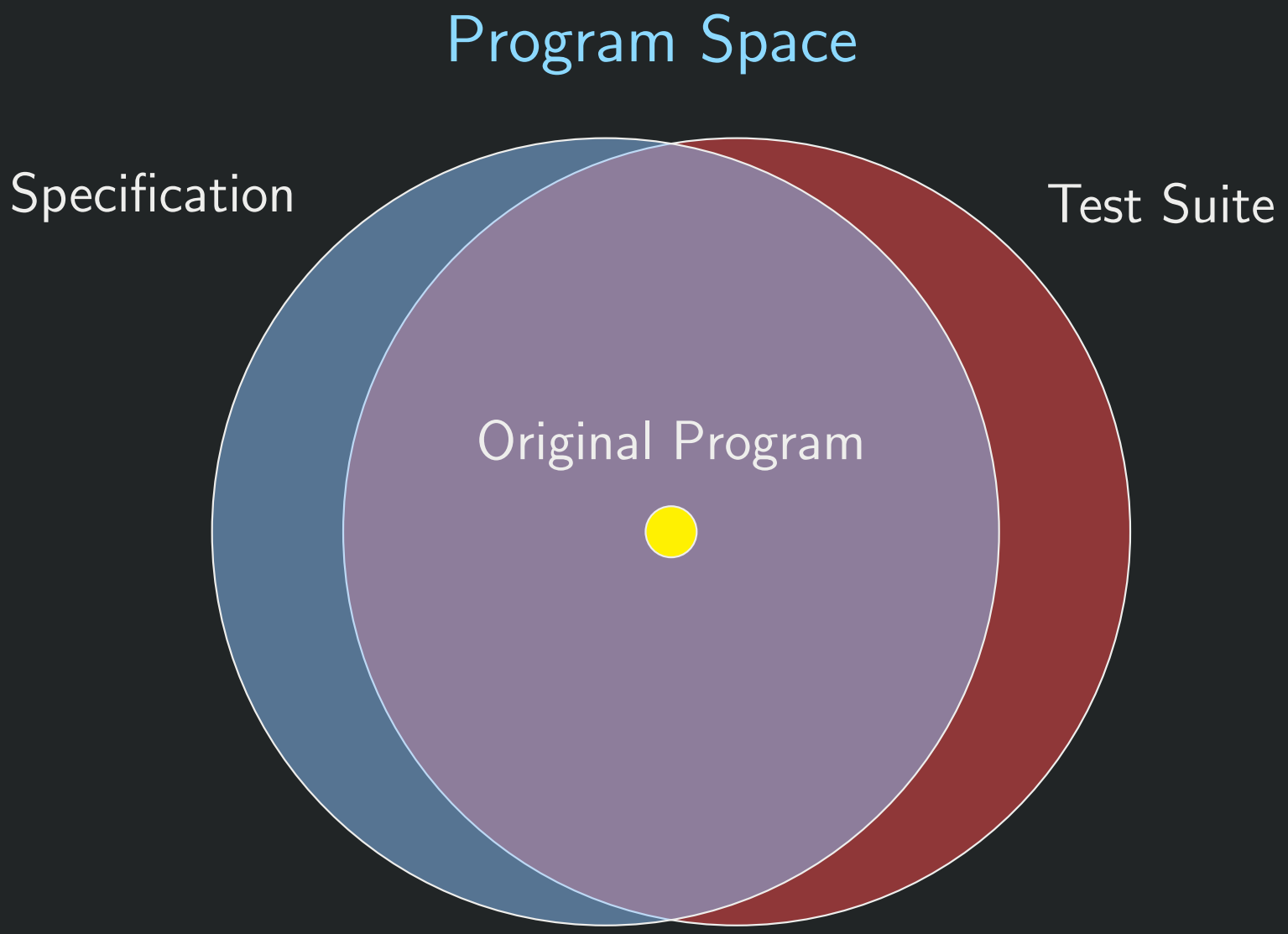
Conclusion

Software Neutral Networks are

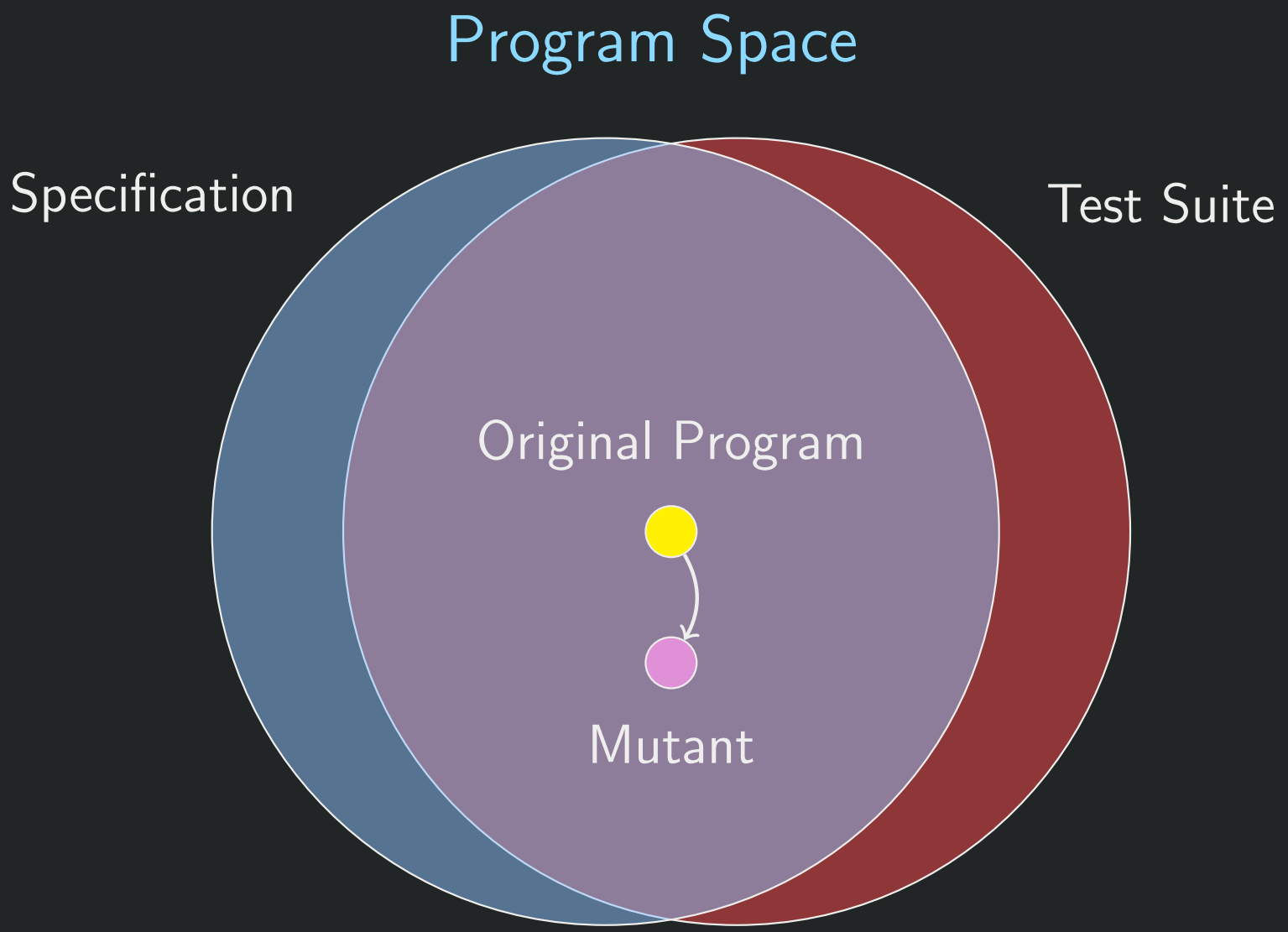
wide	span large distances in program space
thick	consistent mutational robustness
accessible	automatically explorable

Outline

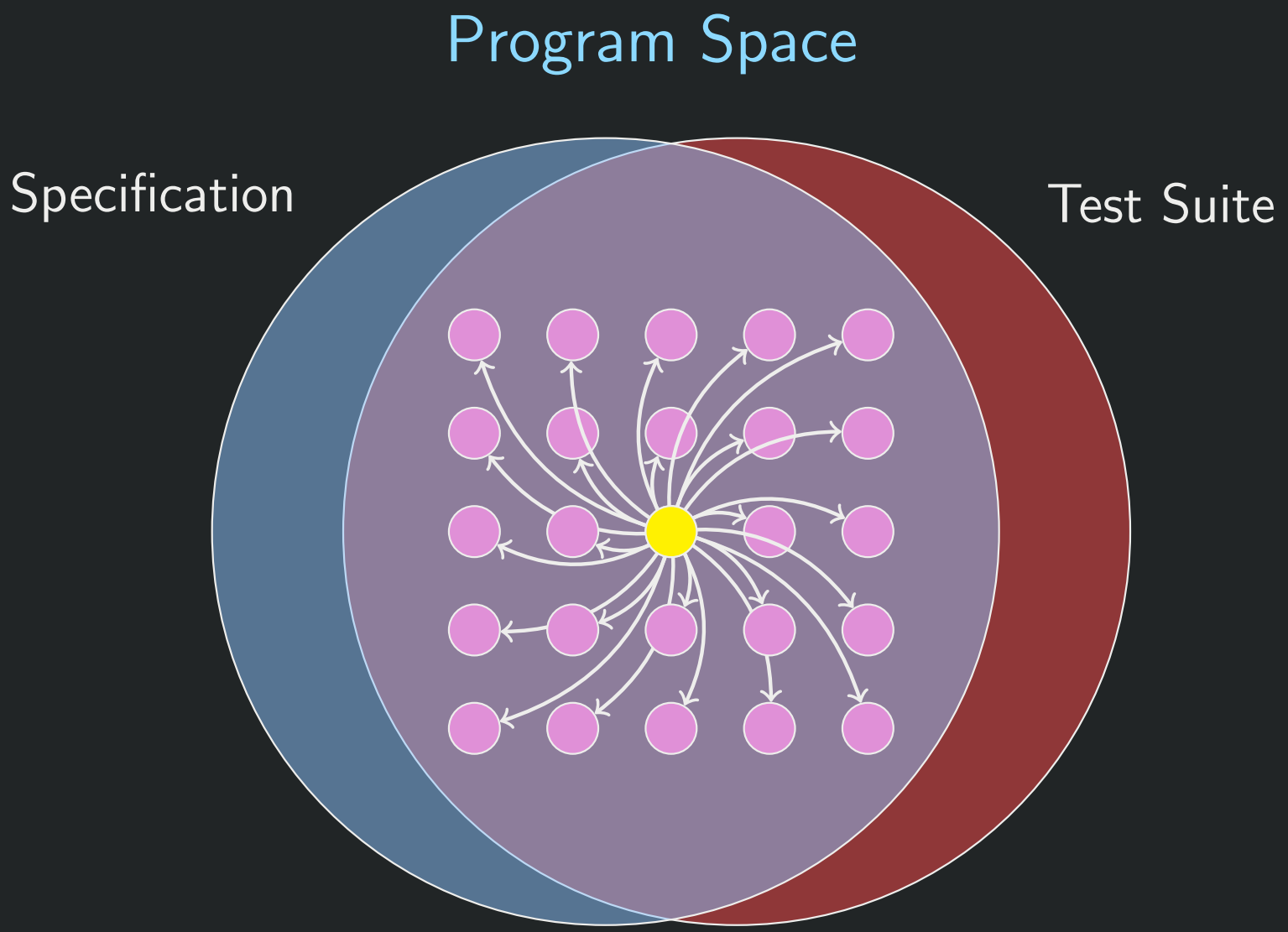
Automated Diversity



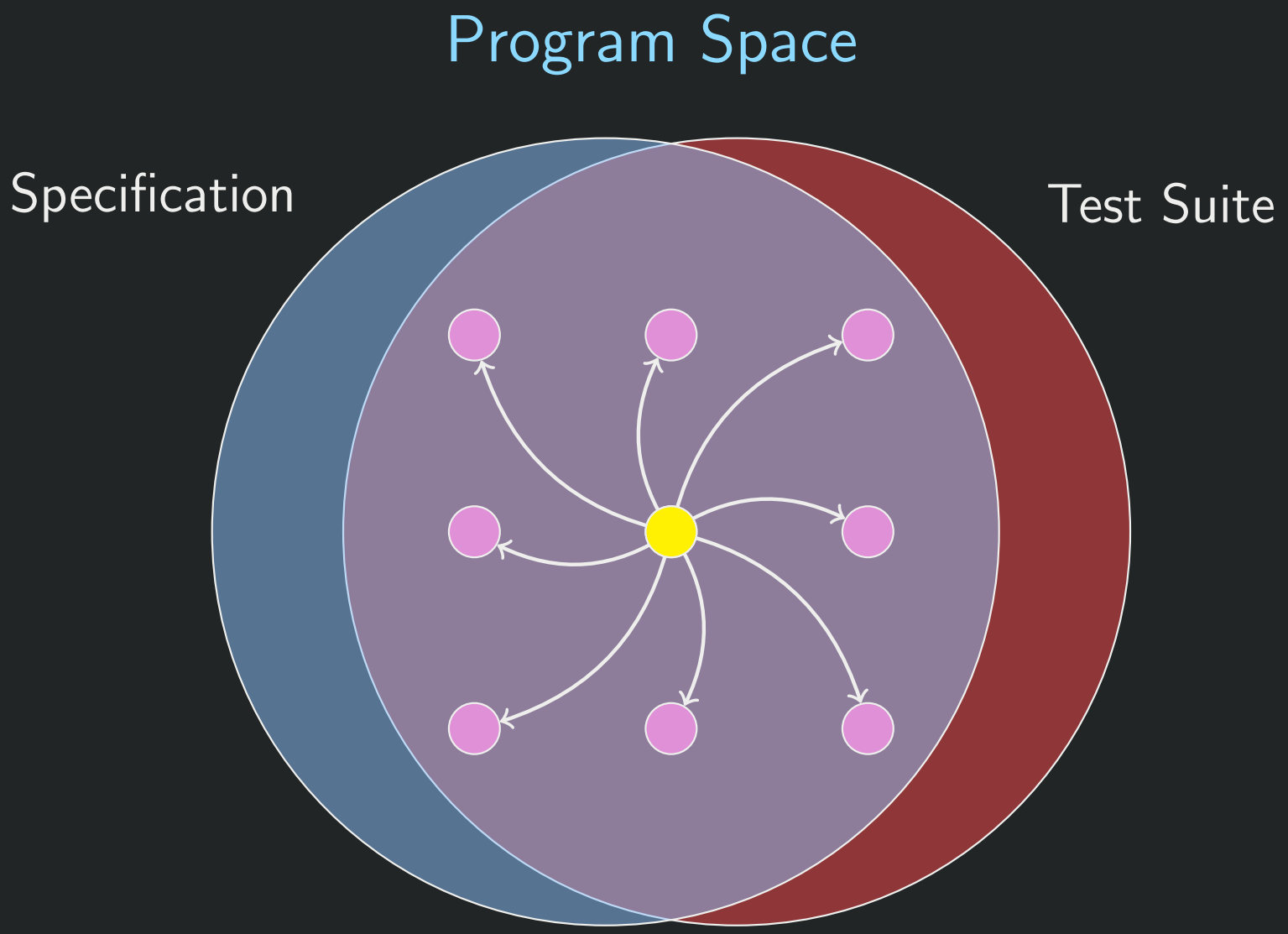
Automated Diversity



Automated Diversity

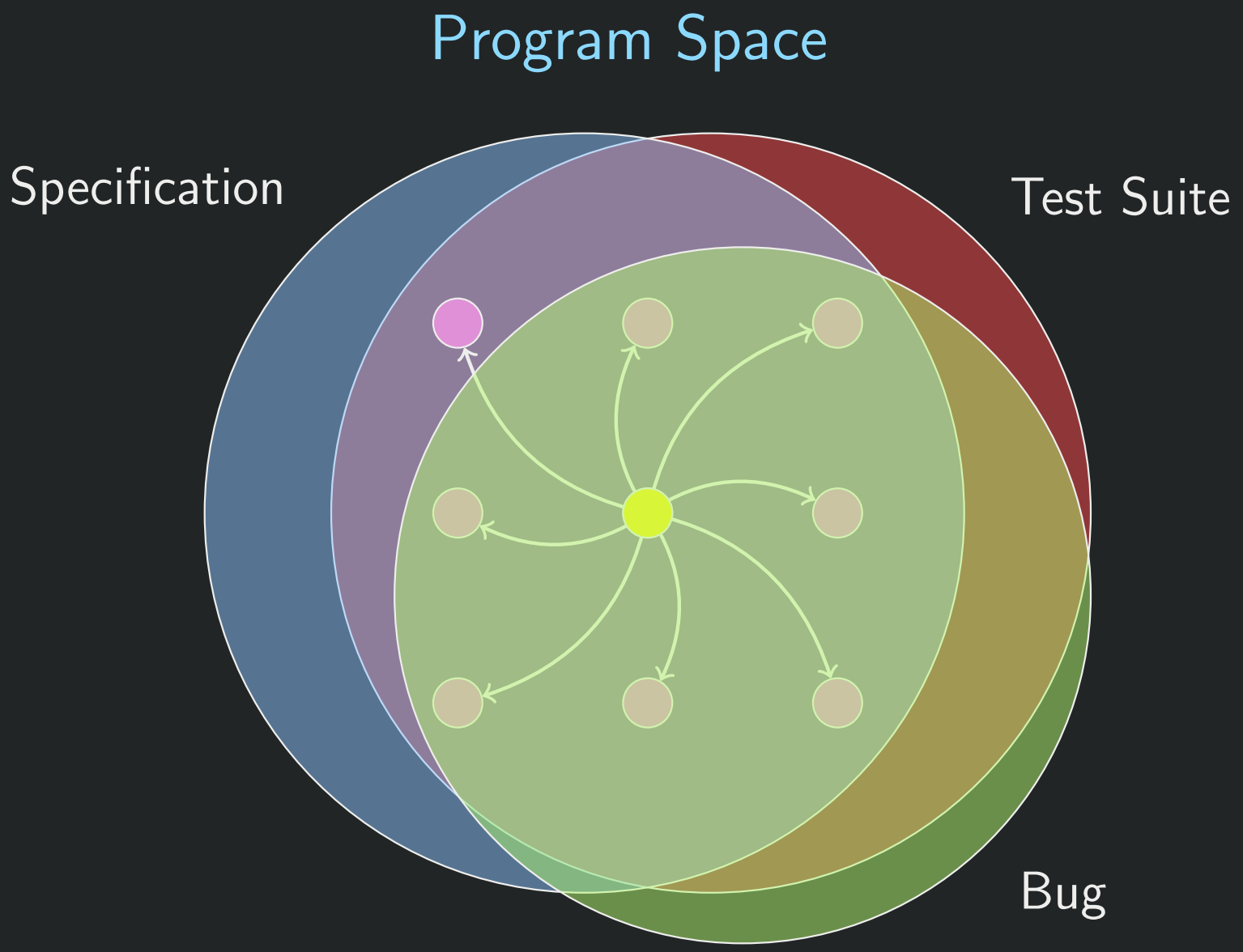


Automated Diversity



Automated Diversity

Proactive Bug Repair



Contribution

Automated Diversity

- Neutral networks expose meaningful diversity [Baudry, ISSTA 2014]
- Diverse variants proactively fix defects
- Select small mutually diverse populations

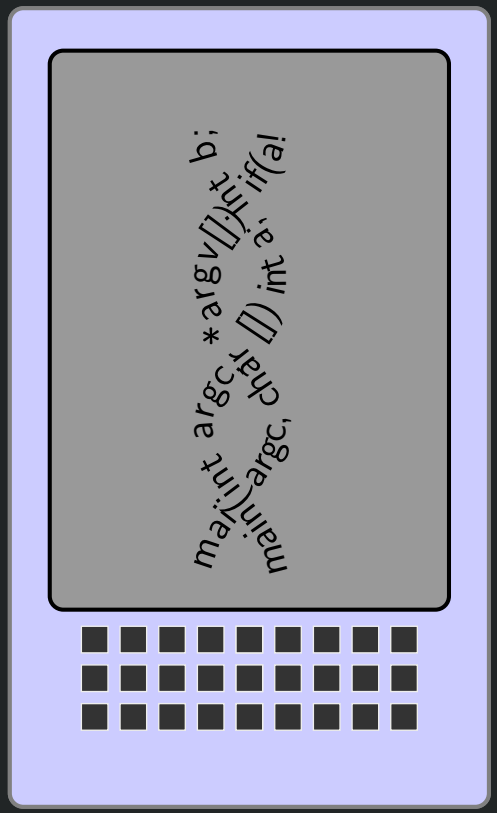
Outline

Program Repair in Embedded Devices

[Schulte *et al.*, ASPLOS 2013]

Resource Constraints

- Small disks
- Less memory
- Slow processors
- Slow, costly comm.



Genprog

Automatically Fix Bugs in C Software

Strengths

Effective	Repaired 55/105 bugs for $8 each
General	Multiple classes of bugs and security defects
Best Papers	ICSE 2009, GECCO 2009, SBST 2009
Humies	Gold 2009, Bronze 2012

Genprog

Automatically Fix Bugs in C Software

Strengths

Effective	Repaired 55/105 bugs for $8 each
General	Multiple classes of bugs and security defects
Best Papers	ICSE 2009, GECCO 2009, SBST 2009
Humies	Gold 2009, Bronze 2012

Requirements

- Source code
- Build tool chain
- Expensive fault localization
- Expensive fitness function (compilation, test execution)

Genprog: Assembler- and Binary-Level

Automatically Fix Bugs in C Software

Strengths

Effective	Repaired 55/105 bugs for $8 each
General	Multiple classes of bugs and security defects
Best Papers	ICSE 2009, GECCO 2009, SBST 2009
Humies	Gold 2009, Bronze 2012

Requirements

- ~~Source code~~
- ~~Build tool chain~~
- ~~Expensive~~ fault localization
- Expensive fitness function (~~compilation~~, test execution)

Embedded Repair Benchmark Programs

Program	Program Description	Bug
`atris`	graphical tetris game	stack buffer exploit
`ccrypt`	encryption utility	segfault
`deroff`	document processing	segfault
`flex`	lexical analyzer generator	segfault
`indent`	source code processing	infinite loop
`look svr4`	dictionary lookup	infinite loop
`look ultrix`	dictionary lookup	infinite loop
`merge`	merge sort	duplicate inputs
`merge-cpp`	merge sort (in C++)	duplicate inputs
`s3`	sendmail utility	buffer overflow
`uniq`	duplicate text processing	segfault
`units`	metric conversion	segfault
`zune`	embedded media player	infinite loop

Contribution

ASM and ELF representation

- Reduce requirements of automated repair
  - Effective: fixing 12 (ASM) and 11 (ELF) of 13
  - 62% faster runtime
  - 95% smaller disk footprint
  - 86% less memory
- Reduce resources needed to perform repair
- New program repair search spaces

Outline

NETGEAR Exploit

[Schulte *et al.*, unpublished]


www.pcworld.com/article/2057260/vulnerabilities-in-some-netgear-router-and-nas-products-open-door-to-remo

PCWorld Macworld TechHive Subscribe

26 Like 47 Tweet 3 g+1

SECURITY netgear, routers

Vulnerabilities in some Netgear routers open door to remote attacks

Lucian Constantin, IDG News Service Oct 23, 2013 10:30 AM

1. URL starting with `BRS` bypasses authentication
2. URL including `unauth.cgi` or `securityquestions.cgi` bypass authentication
3. unprotected page removes authentication for every page `http://router/BRS_02_genieHelp.html`

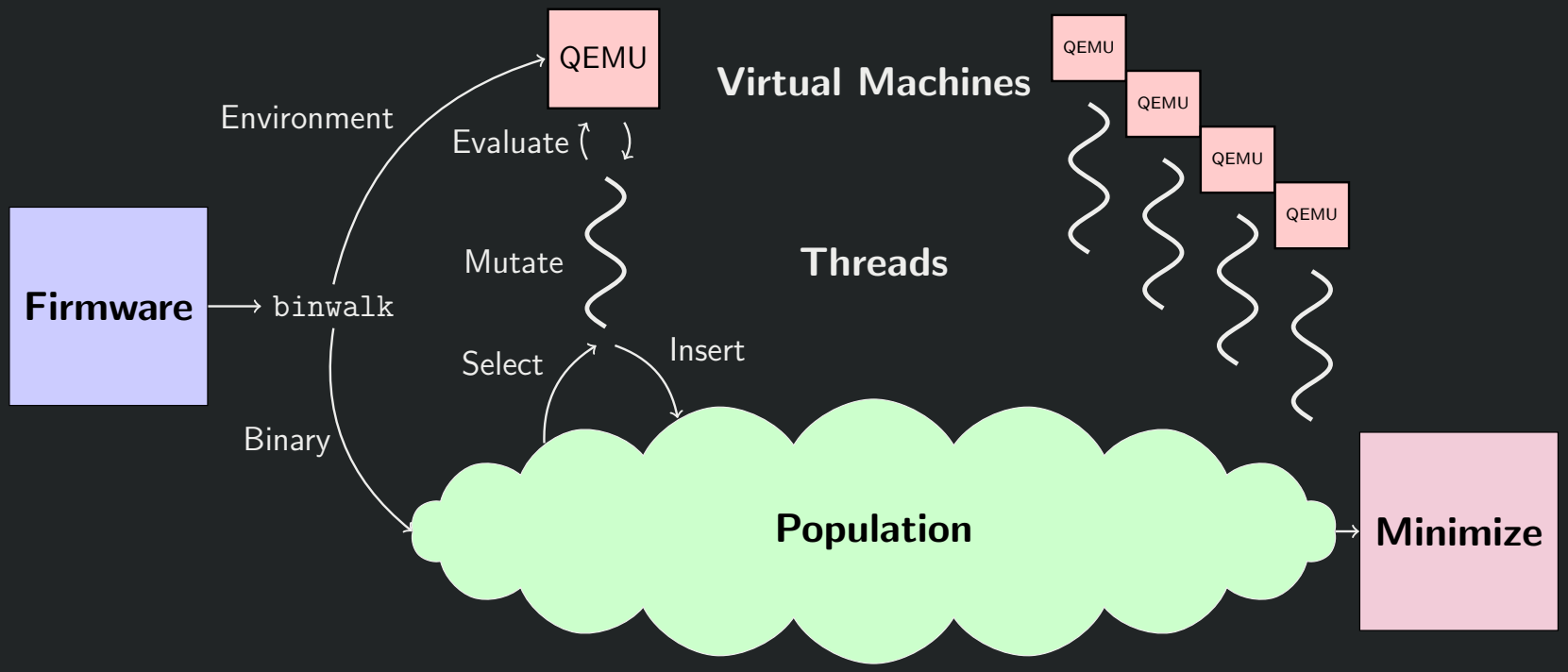
Common Vulnerability

NETGEAR WNDR4700

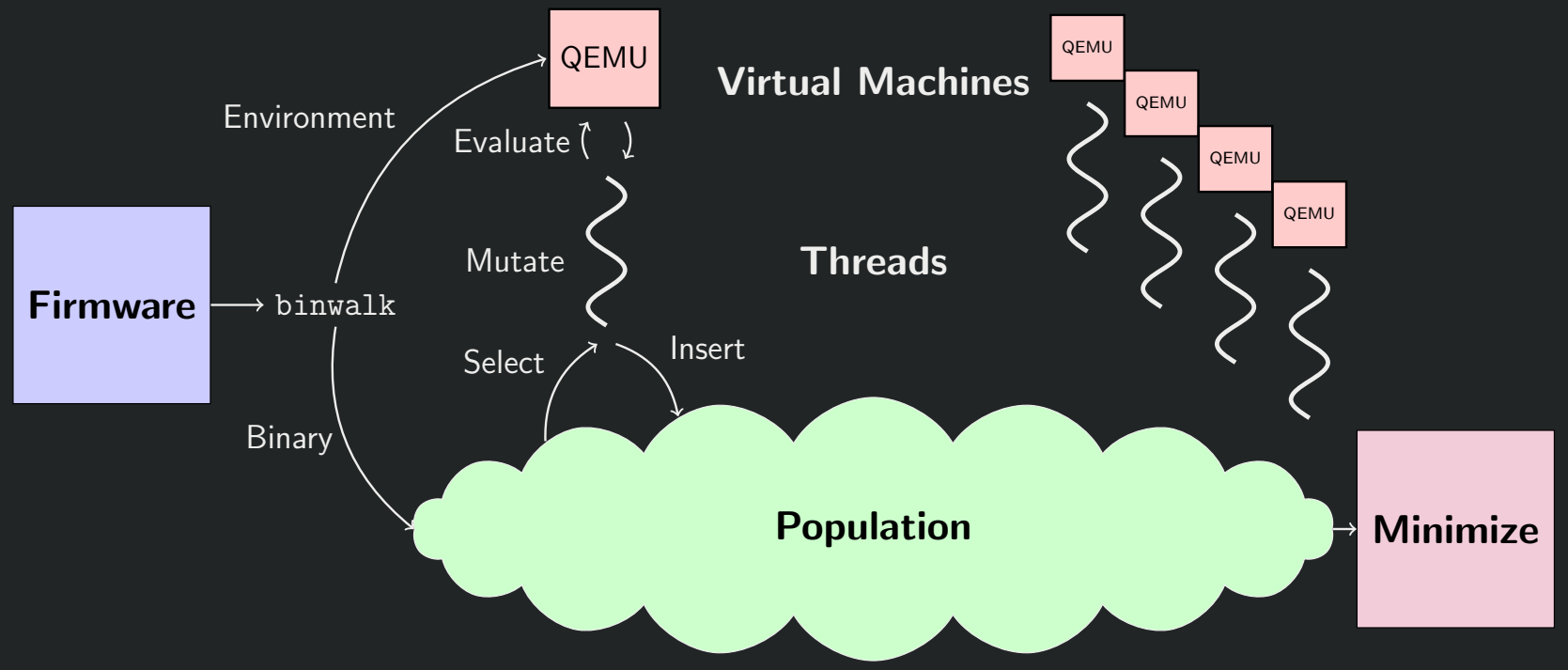
D-Link DIR-100

Linksys WAG200G

NETGEAR Repair Overview



NETGEAR Repair Overview



Repair Runtime

evaluations	$\sim 36,000$
runtime	86.6 min.
threads	32

Contribution

Genprog Requirements

- Non-stripped ELF file
- Regression test suite
- Fault localization

Contribution

Genprog Requirements

- ~~Non-stripped ELF file~~
- ~~Regression test suite~~
- ~~Fault localization~~

Contribution

Genprog Requirements

- ~~Non-stripped ELF file~~
- ~~Regression test suite~~
- ~~Fault localization~~

Enables

- repair of black box executables
- repair in data sections
- patching proprietary software

Outline

Optimization Problem

[Schulte, Dorn *et al.*, ASPLOS 2014]

Optimizing complex non-functional properties

$$\textit{properties} \times \textit{hardware} \times \textit{environment}$$

properties memory, network, energy, etc. . .

hardware architectures, processors, memory stack, etc. . .

environment variables, load, etc. . .

Every program transformation requires

- a-priori reasoning
- manual implementation
- guaranteed correctness

Genetic Optimization Algorithm (GOA)

Post-compiler, test-driven, Genetic Optimization Algorithm

Post-compiler

Genetic Optimization Algorithm (GOA)

Post-compiler, test-driven, Genetic Optimization Algorithm

Test driven

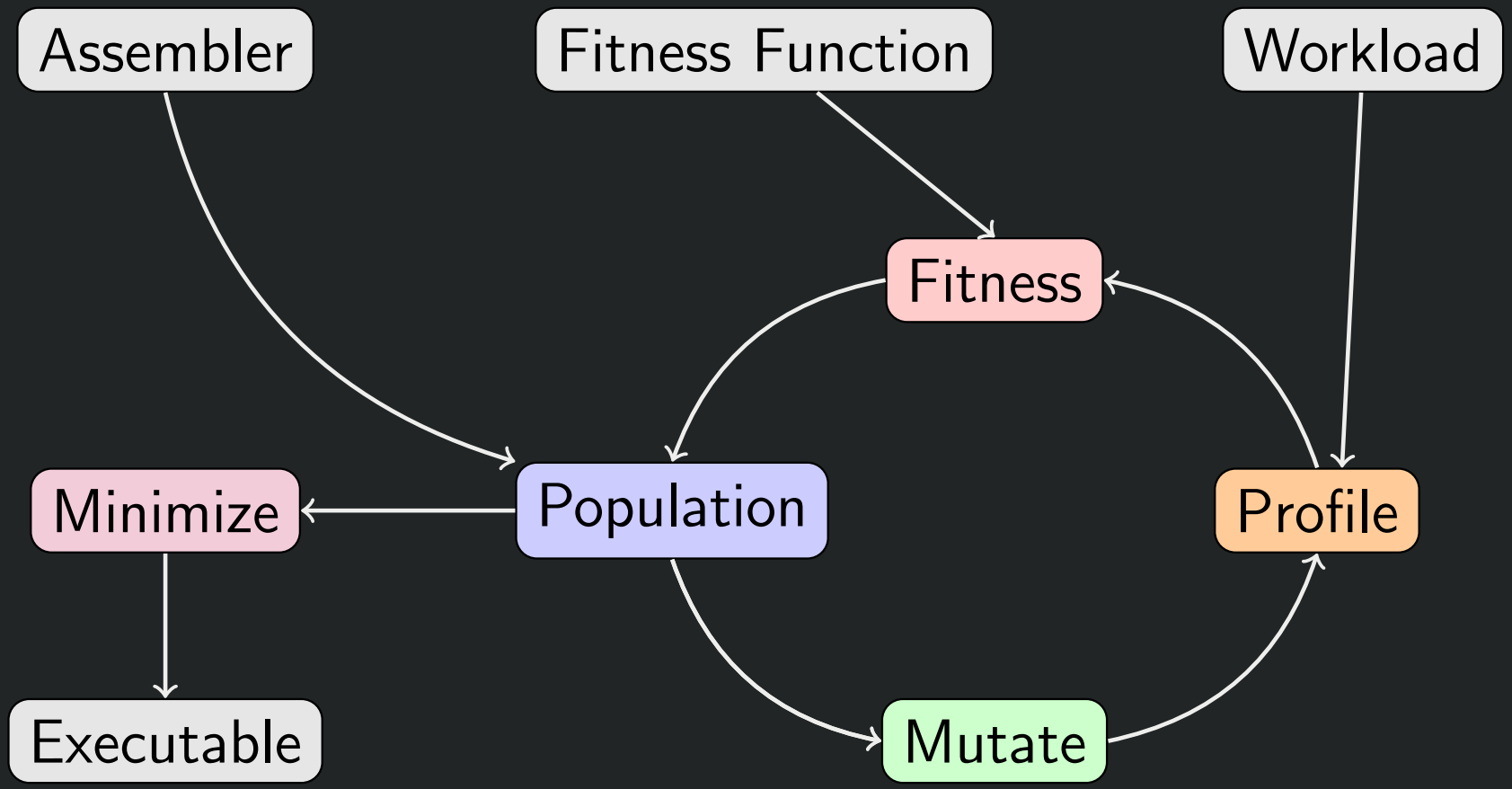
Use test cases to exercise program

- evaluate functionality
- measure runtime properties

Genetic Optimization Algorithm (GOA)

Post-compiler, test-driven, Genetic Optimization Algorithm

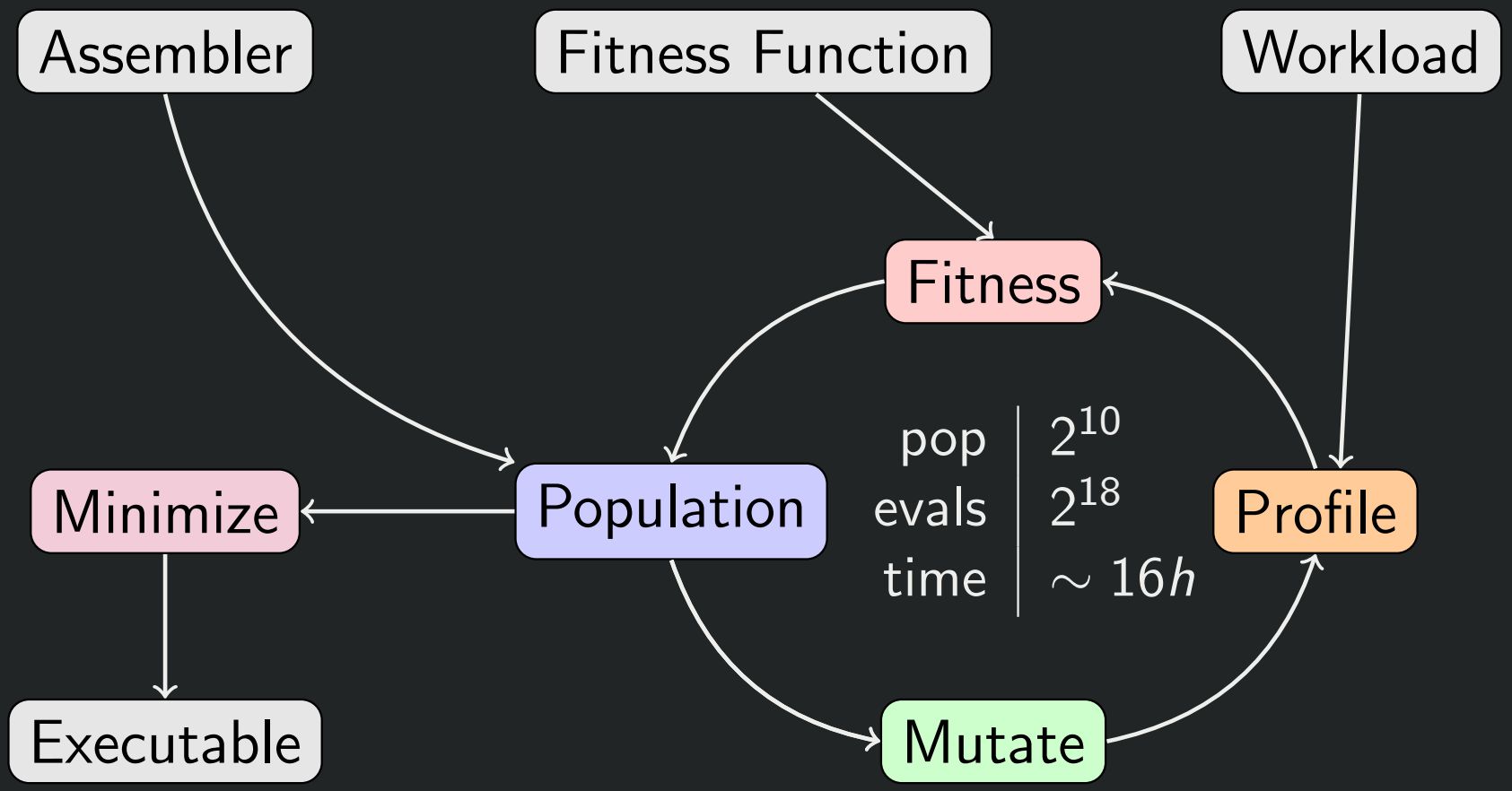
Algorithm



Genetic Optimization Algorithm (GOA)

Post-compiler, test-driven, Genetic Optimization Algorithm

Algorithm



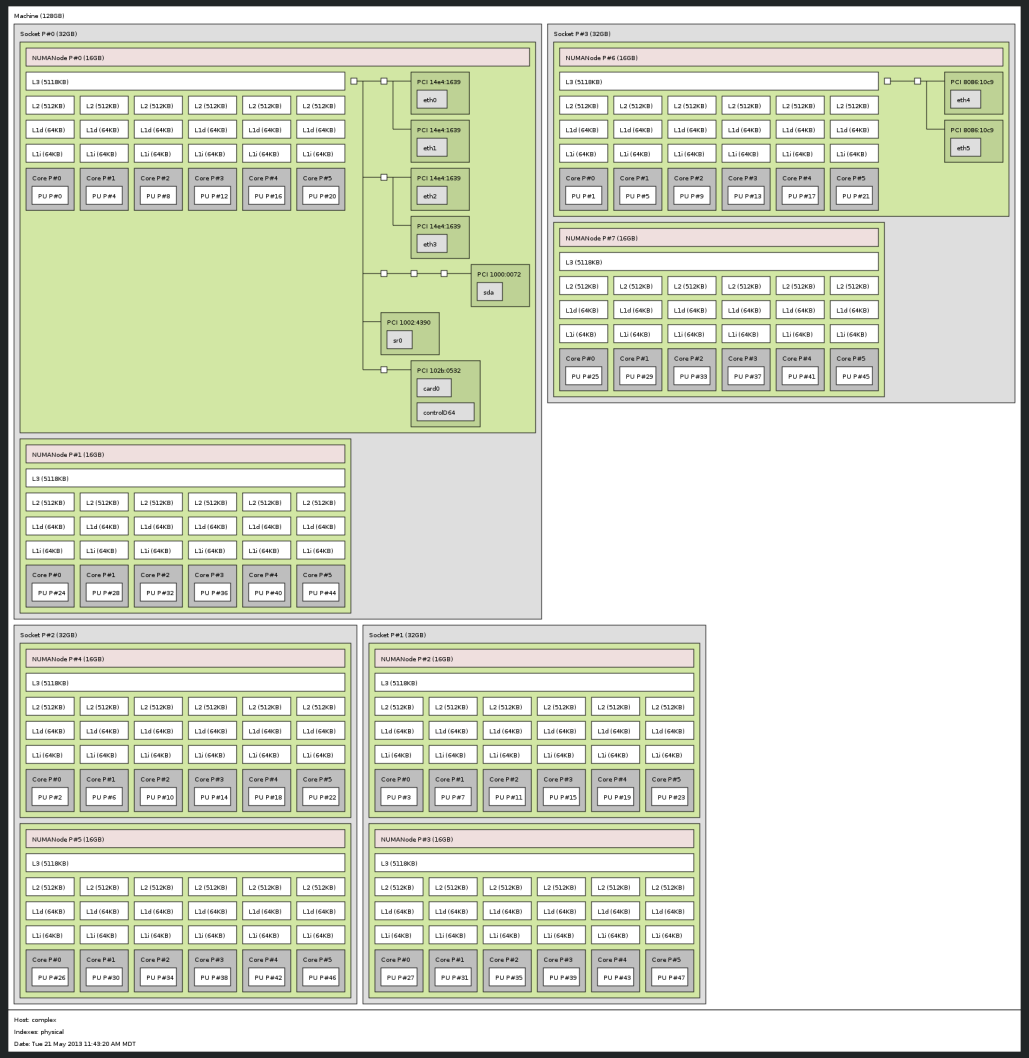
Benchmark Applications

Program	C/C++ Lines of Code	ASM Lines of Code	Description
blackscholes	510	7,932	Finance modeling
bodytrack	14,513	955,888	Human video tracking
facesim			no alternate inputs
ferret	15,188	288,981	Image search engine
fluidanimate	11,424	44,681	Fluid dynamics animation
freqmine	2,710	104,722	Frequent itemset mining
raytrace			no testable output
swaptions	1,649	61,134	Portfolio pricing
vips	142,019	132,012	Image transformation
x264	37,454	111,718	MPEG-4 video encoder
total	225,467	1,707,068	

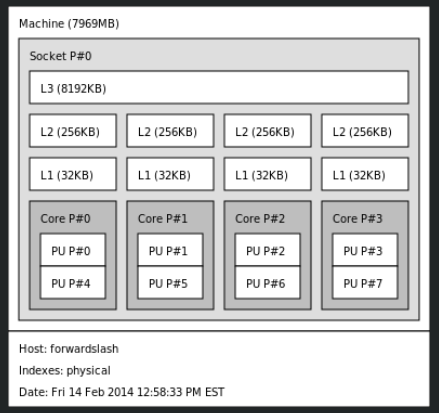
Table: PARSEC benchmark applications.

Hardware Platforms

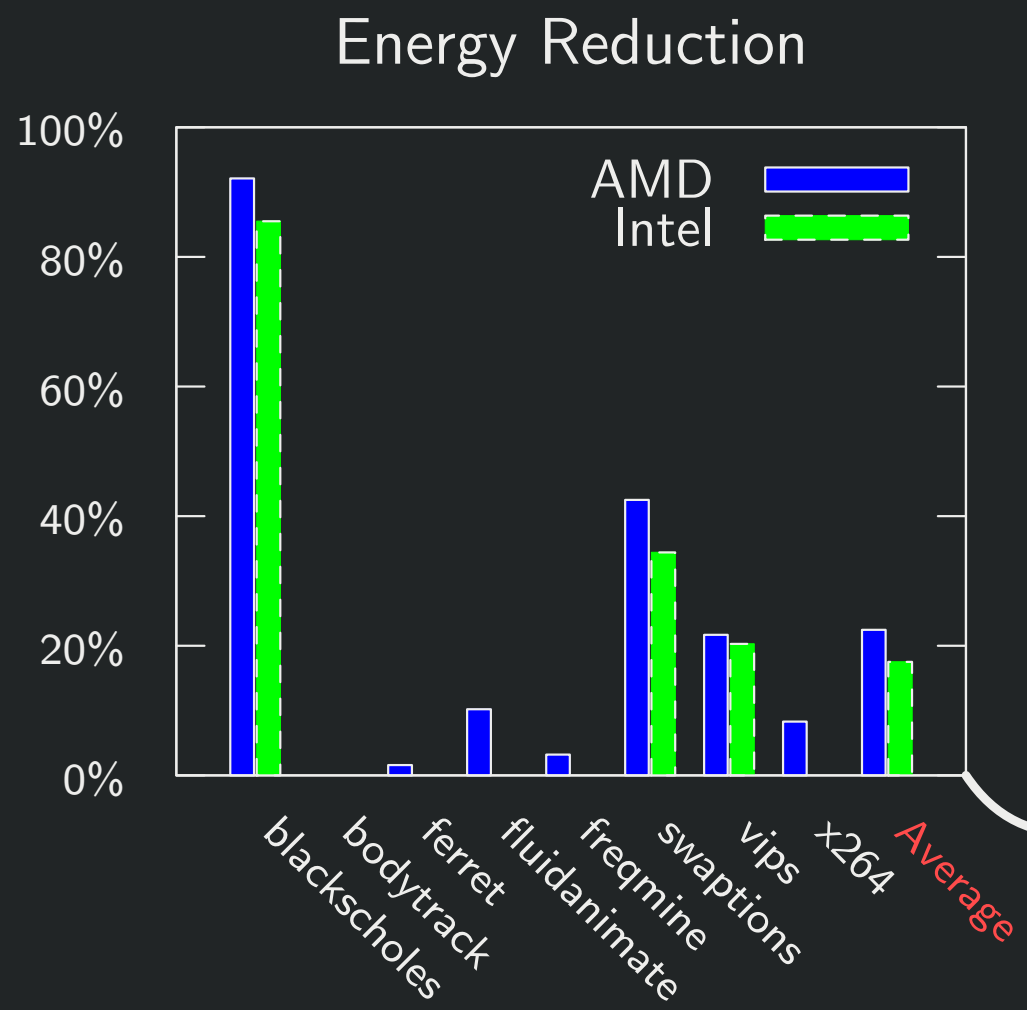
AMD Server

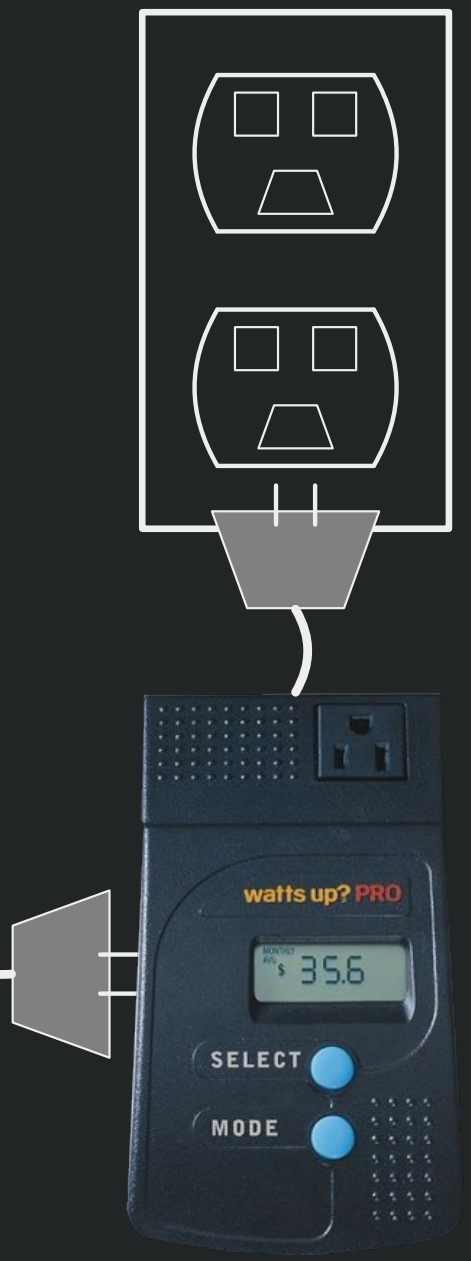


Intel Desktop



Results: Energy Reduction





Functionality on Withheld Tests

Program	AMD	Intel
blackscholes	100%	100%
bodytrack	92%	100%
ferret	100%	100%
fluidanimate	6%	31%
freqmine	100%	100%
swaptions	100%	100%
vips	100%	100%
x264	27%	100%

Anecdotes

Blackscholes

- 90% less energy
- removed redundant outer loop
- modified semantics

Anecdotes

Blackscholes

- 90% less energy
- removed redundant outer loop
- modified semantics

Swaptions

- 42% less energy
- improved branch prediction
- hardware specific

Anecdotes

Blackscholes

- 90% less energy
- removed redundant outer loop
- modified semantics

Swaptions

- 42% less energy
- improved branch prediction
- hardware specific

Vips

- 20% less energy
- substitution of memory access for calculation
- resource trade-off

Contribution

Genetic Optimization Algorithm

1. optimize complex runtime properties (energy)
2. leverages particulars of hardware, and environment
3. reveal compiler inefficiencies
4. find efficiencies, e.g., loop elimination
5. transformations presented as ASM diff

Outline

Challenges

Interface

- Guarantee semantics preservation
- Communicate goals (fitness functions)
- Incorporate into development cycle

Power

- Novel functionality
- Combine components

Future Work

Verification, Mutation, Crossover, Hardening, Fitness-Distance Corr.

- Assembler diff chunks formally equivalent
- Static analysis to,
  - find bugs
  - measure code quality
- Invariant preservation
  e.g., Daikon

Future Work

Verification, Mutation, Crossover, Hardening, Fitness-Distance Corr.

"Smarter" mutation operations

- Repair templates and external code libraries
- Type-aware operations
- Learned mutation heuristics

Future Work

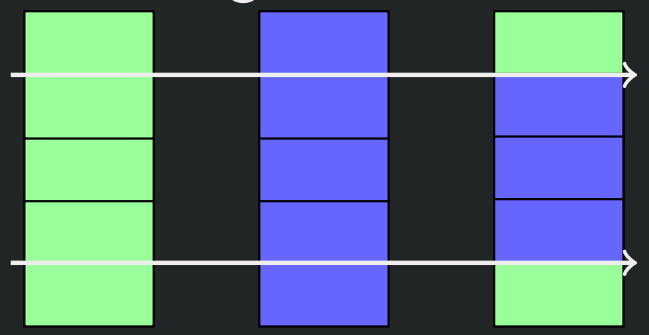
Verification, Mutation, Crossover, Hardening, Fitness-Distance Corr.

Heterologous Crossover

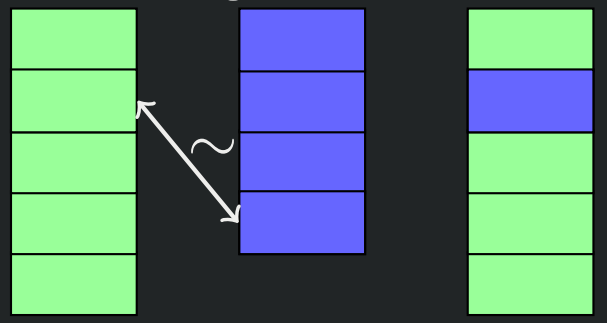
between different

- compilation flags
- versions
- implementations
- programs

Homologous Crossover

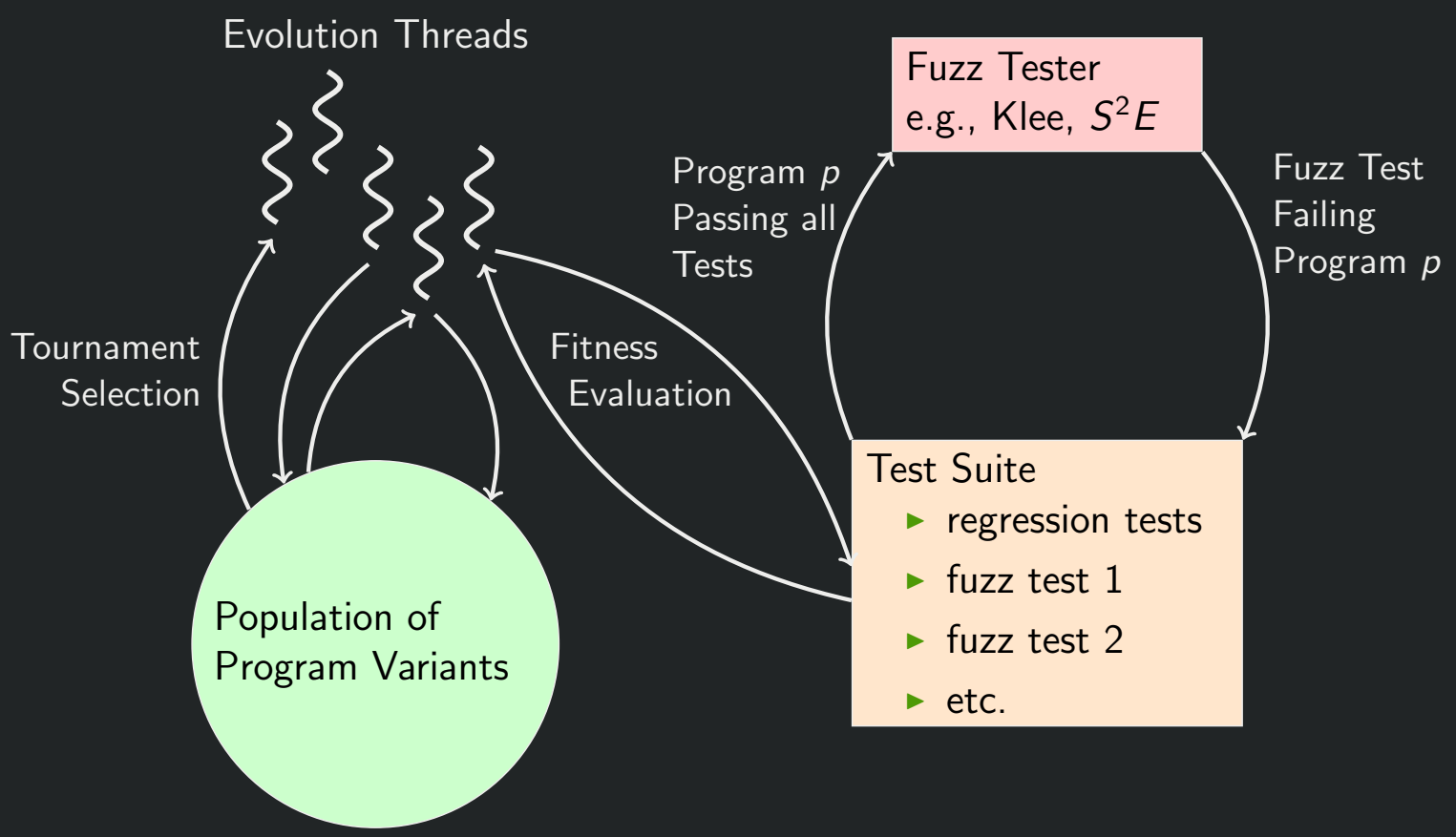


Heterologous Crossover



Future Work

Verification, Mutation, Crossover, Hardening, Fitness-Distance Corr.



Future Work

Verification, Mutation, Crossover, Hardening, Fitness-Distance Corr.

Augment tests to increase granularity

- Count unique values of program counter
- Real-valued oracle comparison (e.g., `numdiff`)

Outline

Conclusion

Extant software is

1. Mutationally robust with large neutral networks

Conclusion

Extant software is

1. Mutationally robust with large neutral networks
2. A product of evolution

Conclusion

Extant software is

1. Mutationally robust with large neutral networks
2. A product of evolution
3. Amenable to automated evolutionary improvement

Conclusion

Extant software is

1. Mutationally robust with large neutral networks
2. A product of evolution
3. Amenable to automated evolutionary improvement

Software evolution might

facilitate the automation of software engineering

Thank You

Eric Schulte

email `eschulte@cs.unm.edu`
homepage `https://cs.unm.edu/~eschulte`
dissertation `https://cs.unm.edu/~eschulte/dissertation`

References

Forrest, GECCO 2009 Stephanie Forrest, Westley Weimer, ThanhVu Nguyen and Claire Le Goues. A Genetic Programming Approach to Automatic Program Repair, in *G*ECCO, 2009

Baudry, ISSTA 2014 Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *I*SSTA, 2014.

Draghi, Nature 2010 J.A. Draghi, T.L. Parsons, G.P. Wagner, and J.B. Plotkin. Mutational robustness can facilitate adaptation. *N*ature, 463(7279):353–355, 2010.

Le Goues, ICSE 2012 Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *I*CSE, 2012.

Nguyen, SBST 2009 ThanhVu Nguyen, Westley Weimer, Claire Le Goues and Stephanie Forrest. Extended Abstract: Using Execution Paths to Evolve Software Patches, in *S*BST, 2009.

Schulte, GPEM 2013 Eric Schulte, Zachary. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *G*enetic Programming and Evolvable Machines, pages 1–32, 2013.

Schulte, ASPLOS 2013 Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *A*SPLOS, 2013.

Schulte, ASPLOS 2014 Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *A*SPLOS, 2014.

Weimer, ICSE 2009 Westley Weimer, ThanhVu Nguyen, Claire Le Goues and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming, in *I*CSE, 2009

Backup Slides

- Program Space
- Program Expression
- Many Bugs
- LLVM SSA
- Robustness and Adaptation Time
- Proactive Bug Repair
- Fault Localization
- Embedded Repair Results
- Distributed Repair
- Benefits of ASM
- Embedded Repair Space
- Profiling (GOA)
- Minimization (GOA)
- Energy Model (GOA)
- Runtime and Energy Reduction (GOA)
- Physical Evolutionary Computation

Program Space

Back

Constants: d=3, e=5

Total

$$n_{i+1} = \frac{n_i(d-i)(e-1)}{(i+1)}$$

Unique

$$n_{z,i+1} = \frac{n_{z-1,i}(d-i)1 + n_{z,i}(d-i)(e-2)}{i+1}$$

$$A_i = \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1}$$

Neutral

$$R = \sum_{i \leq d} r^i \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1}$$

Step	Total
0	1
1	
2	
3	

Program Space

Back

Constants: d=3, e=5

Total

$$n_{i+1} = \frac{n_i(d-i)(e-1)}{(i+1)}$$

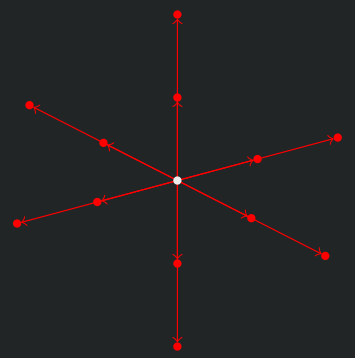
Unique

$$n_{z,i+1} = \frac{n_{z-1,i}(d-i)1 + n_{z,i}(d-i)(e-2)}{i+1}$$

$$A_i = \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1}$$

Neutral

$$R = \sum_{i \leq d} r^i \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1}$$



Step	Total
0	1
1	12
2	
3	

Program Space

Back

Total

$$n_{i+1} = \frac{n_i(d-i)(e-1)}{(i+1)}$$

Unique

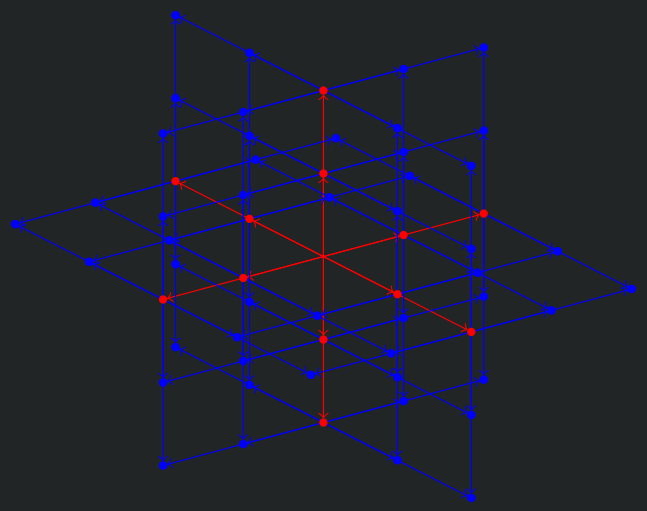
$$n_{z,i+1} = \frac{n_{z-1,i}(d-i)1 + n_{z,i}(d-i)(e-2)}{i+1}$$

$$A_i = \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1}$$

Neutral

$$R = \sum_{i \leq d} r^i \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1}$$

Constants: d=3, e=5



Step	Total
0	1
1	12
2	48
3	

Program Space

Back

Total

$$n_{i+1} = \frac{n_i(d-i)(e-1)}{(i+1)}$$

Unique

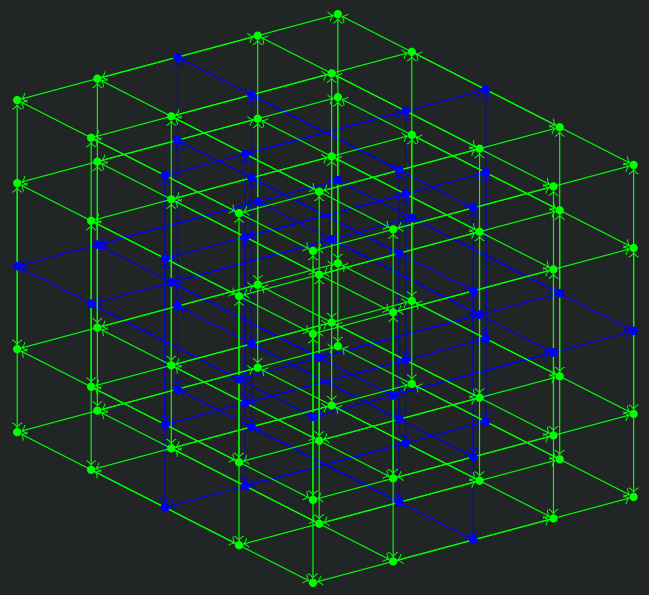
$$n_{z,i+1} = \frac{n_{z-1,i}(d-i)1 + n_{z,i}(d-i)(e-2)}{i+1}$$

$$A_i = \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1}$$

Neutral

$$R = \sum_{i \leq d} r^i \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1}$$

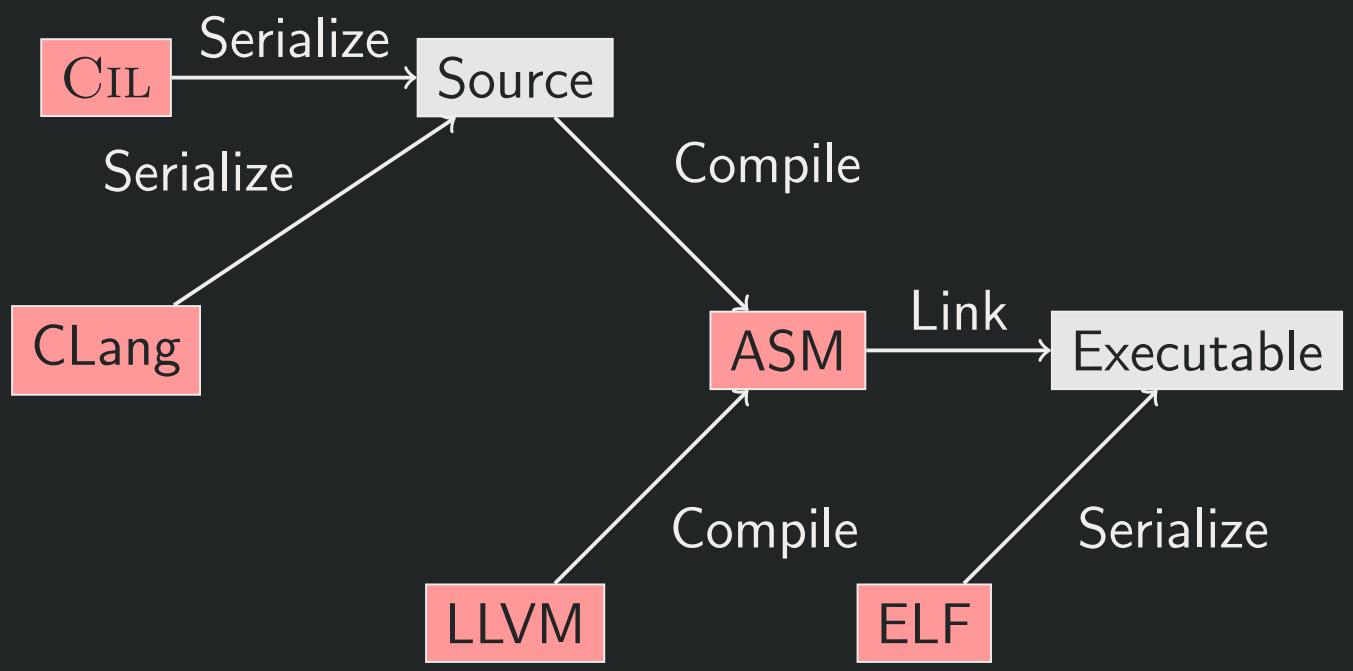
Constants: d=3, e=5



Step	Total
0	1
1	12
2	48
3	64

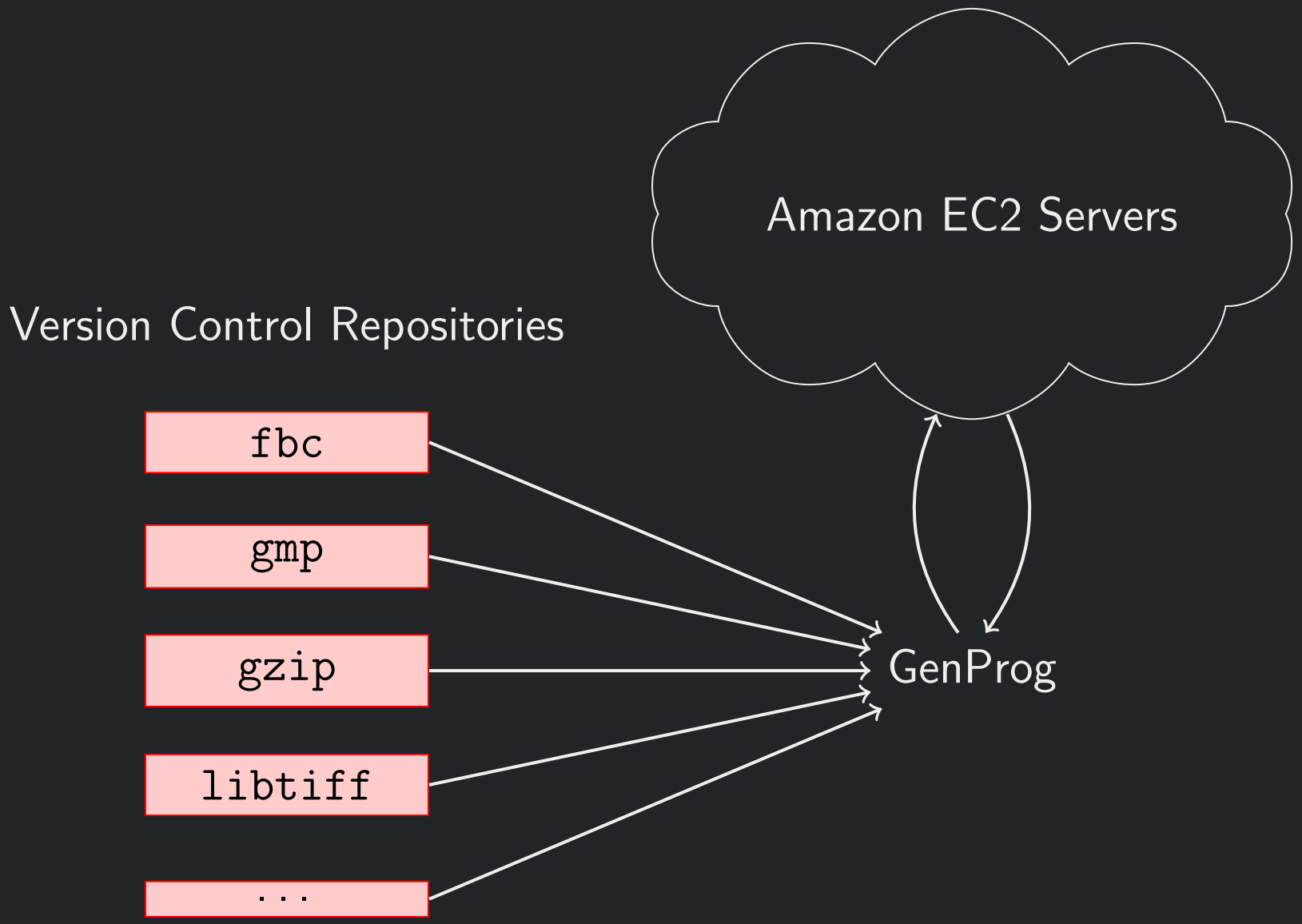
Program Expression

Back



Systematic Evaluation

[Le Goues *et al.*, ICSE 12] Back



Fixing Bugs for $8 a Bug

Back

Program	Defects Repaired	Avg. Cost per Non-Repair Hours	Avg. Cost per Non-Repair US$	Avg. Cost Per Repair Hours	Avg. Cost Per Repair 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**	

LLVM SSA

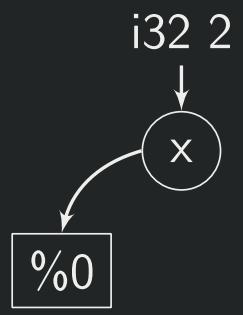
Back


```
%x = alloca i32, align 4
store i32 2, i32* %x, align 4
%0 = load i32* %x, align 4
%add = add nsw i32 %0, 3
store i32 %add, i32* %x, align 4
%1 = load i32* %x, align 4
%2 = load i32* %x, align 4
%mul = mul nsw i32 %1, %2
store i32 %mul, i32* %x, align 4
%3 = load i32* %x, align 4
%call = call ... @printf(@.str %3)
```

LLVM SSA

Back

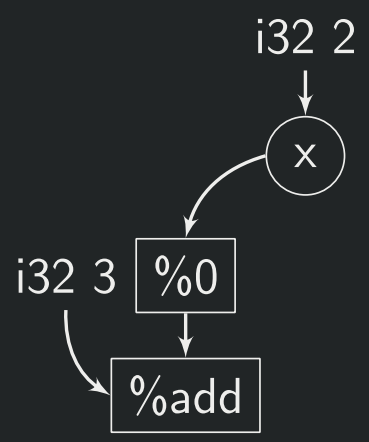
- `%x = alloca i32, align 4`
- `store i32 2, i32* %x, align 4`
- `%0 = load i32* %x, align 4`
- `%add = add nsw i32 %0, 3`
- `store i32 %add, i32* %x, align 4`
- `%1 = load i32* %x, align 4`
- `%2 = load i32* %x, align 4`
- `%mul = mul nsw i32 %1, %2`
- `store i32 %mul, i32* %x, align 4`
- `%3 = load i32* %x, align 4`
- `%call = call ... @printf(@.str %3)`



LLVM SSA

Back

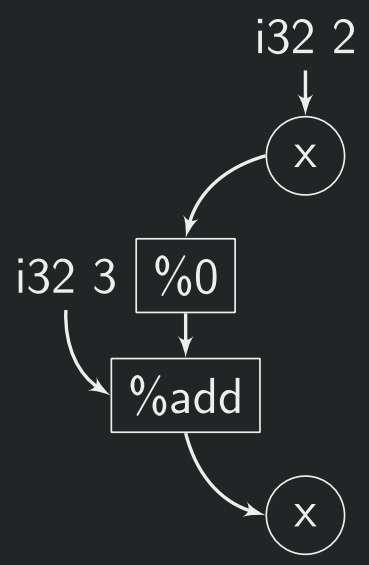
- `%x = alloca i32, align 4`
- `store i32 2, i32* %x, align 4`
- `%0 = load i32* %x, align 4`
- `%add = add nsw i32 %0, 3`
- `store i32 %add, i32* %x, align 4`
- `%1 = load i32* %x, align 4`
- `%2 = load i32* %x, align 4`
- `%mul = mul nsw i32 %1, %2`
- `store i32 %mul, i32* %x, align 4`
- `%3 = load i32* %x, align 4`
- `%call = call ... @printf(@.str %3)`



LLVM SSA

Back

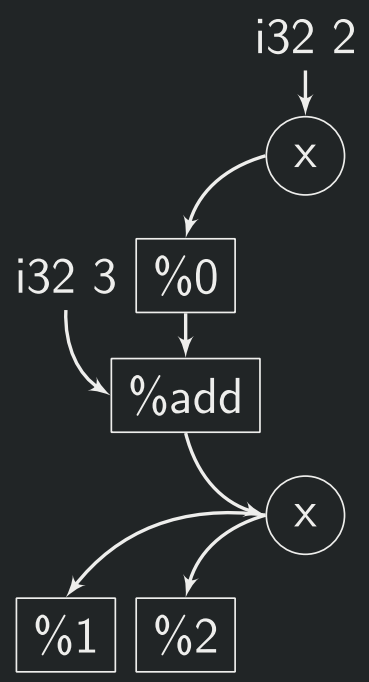
- `%x = alloca i32, align 4`
- `store i32 2, i32* %x, align 4`
- `%0 = load i32* %x, align 4`
- `%add = add nsw i32 %0, 3`
- `store i32 %add, i32* %x, align 4`
- `%1 = load i32* %x, align 4`
- `%2 = load i32* %x, align 4`
- `%mul = mul nsw i32 %1, %2`
- `store i32 %mul, i32* %x, align 4`
- `%3 = load i32* %x, align 4`
- `%call = call ... @printf(@.str %3)`



LLVM SSA

Back

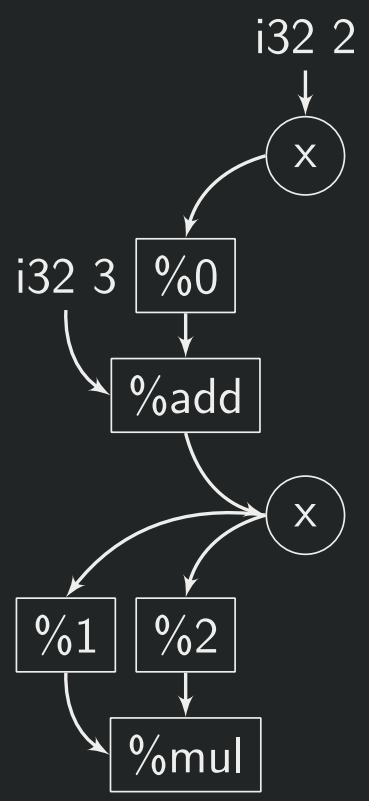
- `%x = alloca i32, align 4`
- `store i32 2, i32* %x, align 4`
- `%0 = load i32* %x, align 4`
- `%add = add nsw i32 %0, 3`
- `store i32 %add, i32* %x, align 4`
- `%1 = load i32* %x, align 4`
- `%2 = load i32* %x, align 4`
- `%mul = mul nsw i32 %1, %2`
- `store i32 %mul, i32* %x, align 4`
- `%3 = load i32* %x, align 4`
- `%call = call ... @printf(@.str %3)`



LLVM SSA

Back

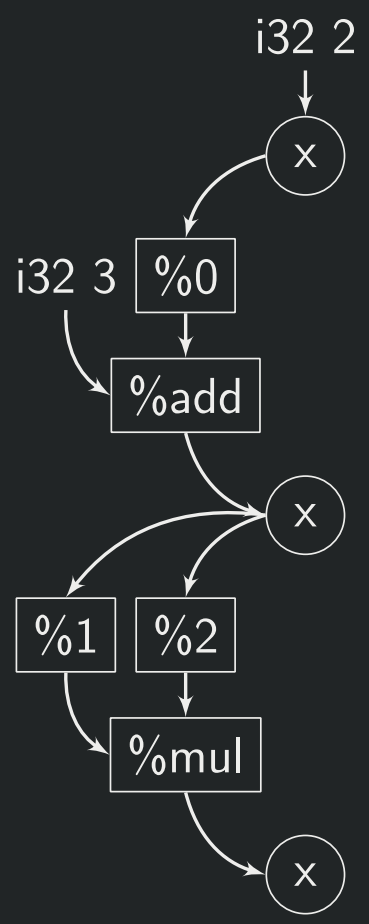
- `%x = alloca i32, align 4`
- `store i32 2, i32* %x, align 4`
- `%0 = load i32* %x, align 4`
- `%add = add nsw i32 %0, 3`
- `store i32 %add, i32* %x, align 4`
- `%1 = load i32* %x, align 4`
- `%2 = load i32* %x, align 4`
- `%mul = mul nsw i32 %1, %2`
- `store i32 %mul, i32* %x, align 4`
- `%3 = load i32* %x, align 4`
- `%call = call ... @printf(@.str %3)`



LLVM SSA

Back

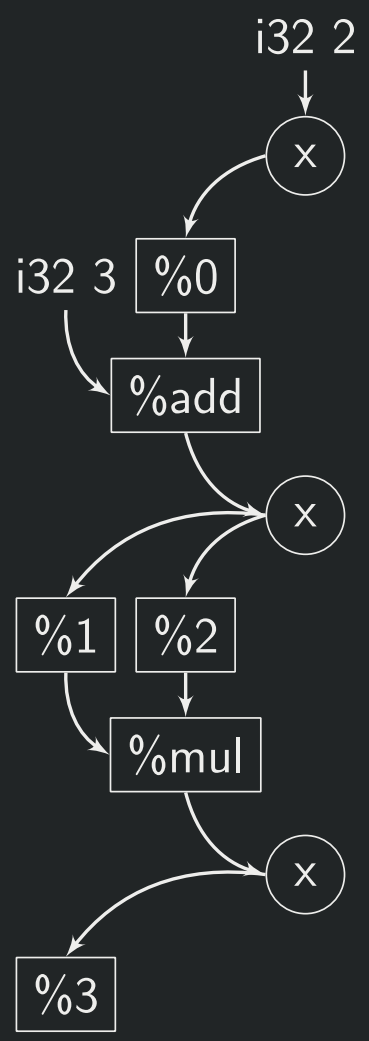
- `%x = alloca i32, align 4`
- `store i32 2, i32* %x, align 4`
- `%0 = load i32* %x, align 4`
- `%add = add nsw i32 %0, 3`
- `store i32 %add, i32* %x, align 4`
- `%1 = load i32* %x, align 4`
- `%2 = load i32* %x, align 4`
- `%mul = mul nsw i32 %1, %2`
- `store i32 %mul, i32* %x, align 4`
- `%3 = load i32* %x, align 4`
- `%call = call ... @printf(@.str %3)`



LLVM SSA

Back

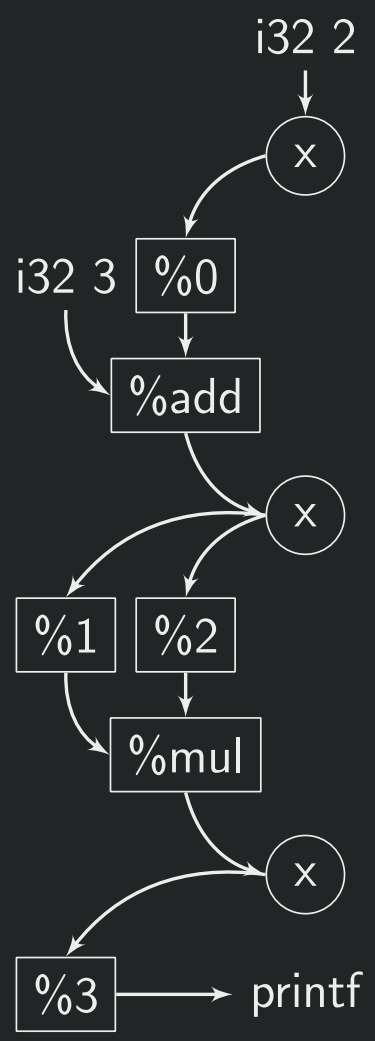
- `%x = alloca i32, align 4`
- `store i32 2, i32* %x, align 4`
- `%0 = load i32* %x, align 4`
- `%add = add nsw i32 %0, 3`
- `store i32 %add, i32* %x, align 4`
- `%1 = load i32* %x, align 4`
- `%2 = load i32* %x, align 4`
- `%mul = mul nsw i32 %1, %2`
- `store i32 %mul, i32* %x, align 4`
- `%3 = load i32* %x, align 4`
- `%call = call ... @printf(@.str %3)`



LLVM SSA

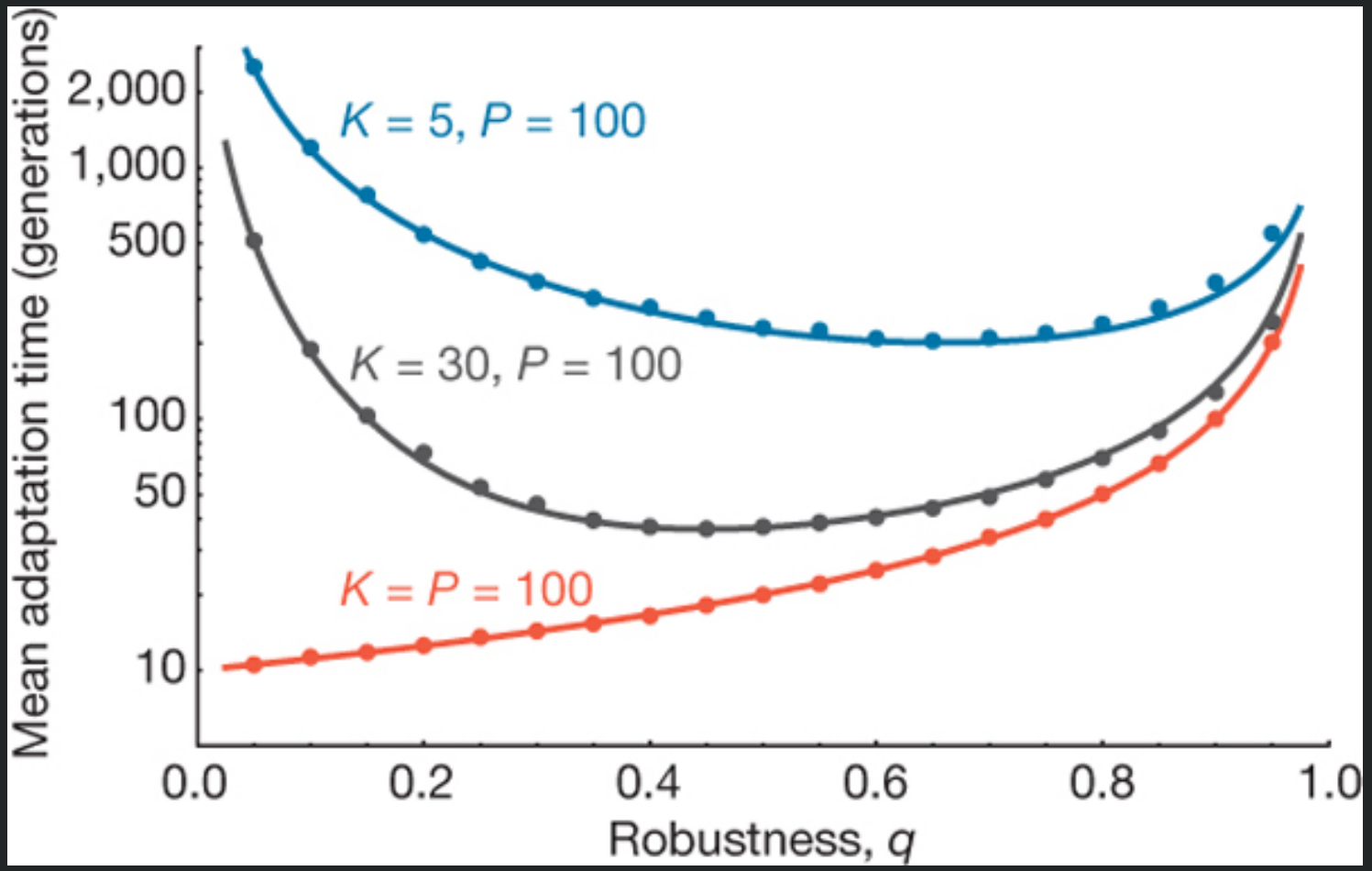
Back

- `%x = alloca i32, align 4`
- `store i32 2, i32* %x, align 4`
- `%0 = load i32* %x, align 4`
- `%add = add nsw i32 %0, 3`
- `store i32 %add, i32* %x, align 4`
- `%1 = load i32* %x, align 4`
- `%2 = load i32* %x, align 4`
- `%mul = mul nsw i32 %1, %2`
- `store i32 %mul, i32* %x, align 4`
- `%3 = load i32* %x, align 4`
- `%call = call ... @printf(@.str %3)`



Robustness and Adaptation Time

Back



[Draghi *et al.*, Nature 2010]

Proactive Bug Repair

Back

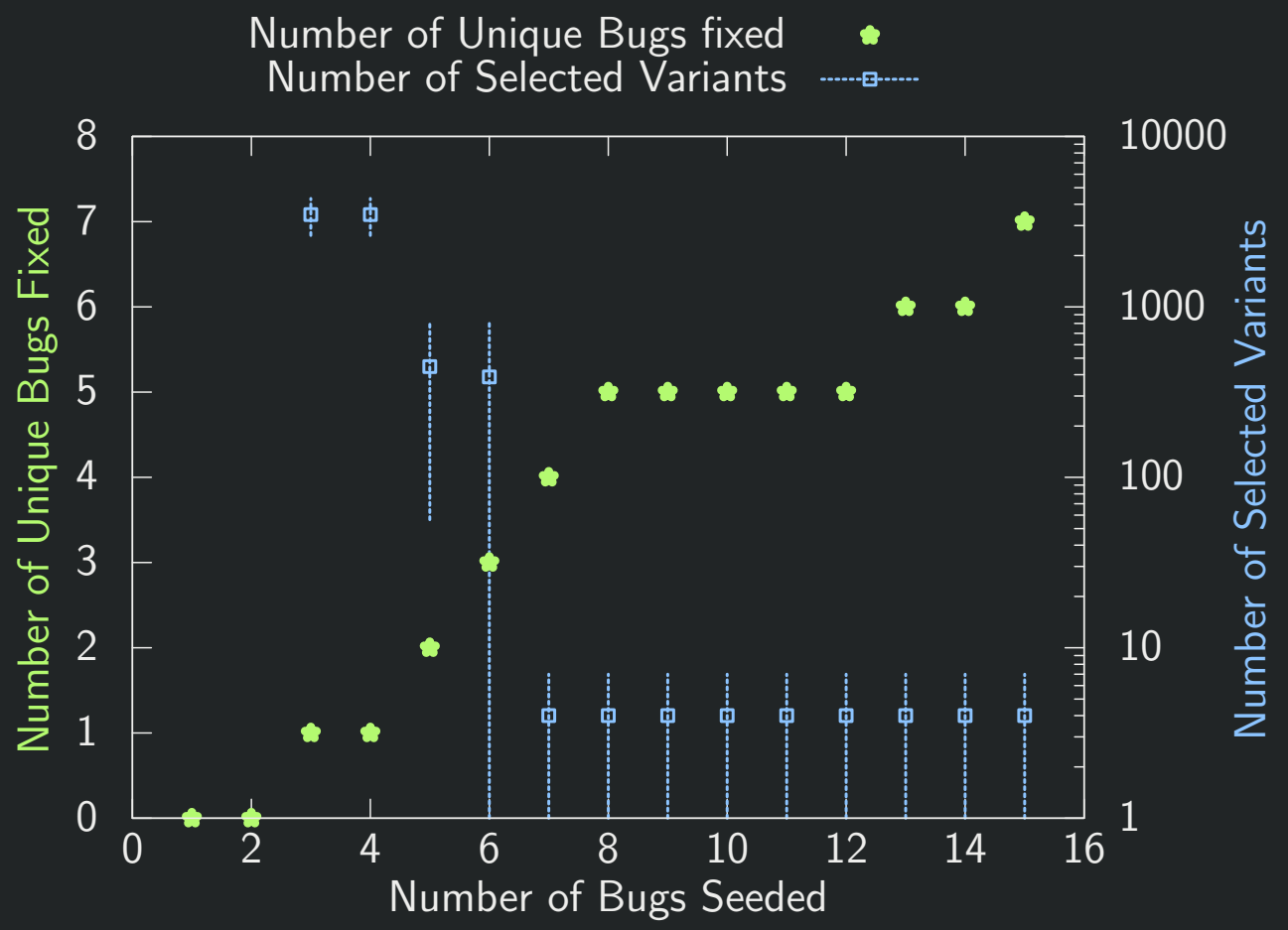
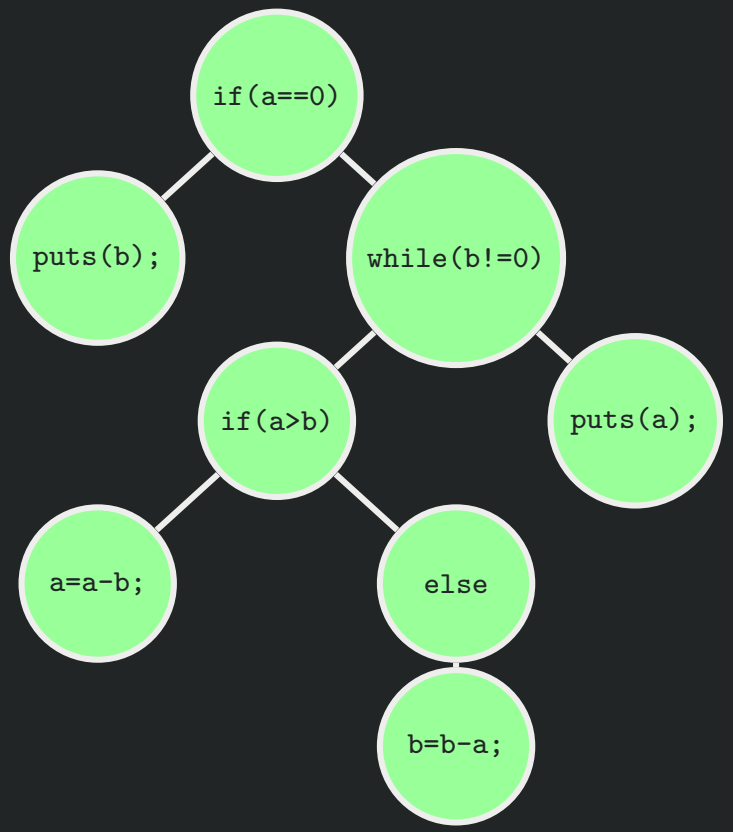


Figure: Bugs Fixed by Number of Variants in `Potion`

Fault Localization

Back

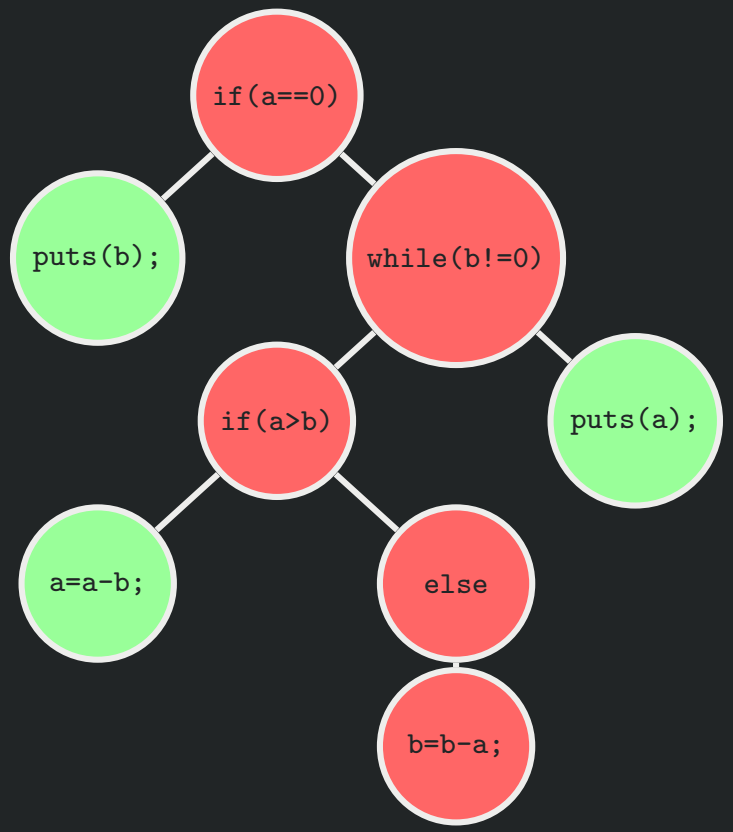
- Focus Mutation
- Fault Localization
  1. instrument
  2. compile
  3. evaluate



Fault Localization

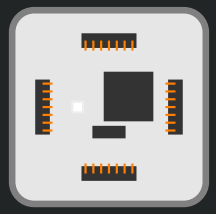
Back

- Focus Mutation
- Fault Localization
  1. instrument
  2. compile
  3. evaluate



Light Weight Fault Localization

1. Sample program counter.
2. Translate memory addresses to program offsets.
3. Smooth sample with Gaussian convolution.



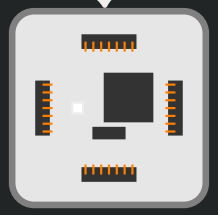
CPU

```
movq 8(%rdx), %rdi
xorl %eax, %eax
movl %eax, (%r15)
addl $1, %r14d
call atoi
movq -80(%rbp), %rdx
movq %rdx, -80(%rbp)
addq $4, %r15
movq 8(%rdx), %rdi
xorl %eax, %eax
movl %eax, (%r15)
```

Machine-code Instructions

Light Weight Fault Localization

1. Sample program counter.
2. Translate memory addresses to program offsets.
3. Smooth sample with Gaussian convolution.



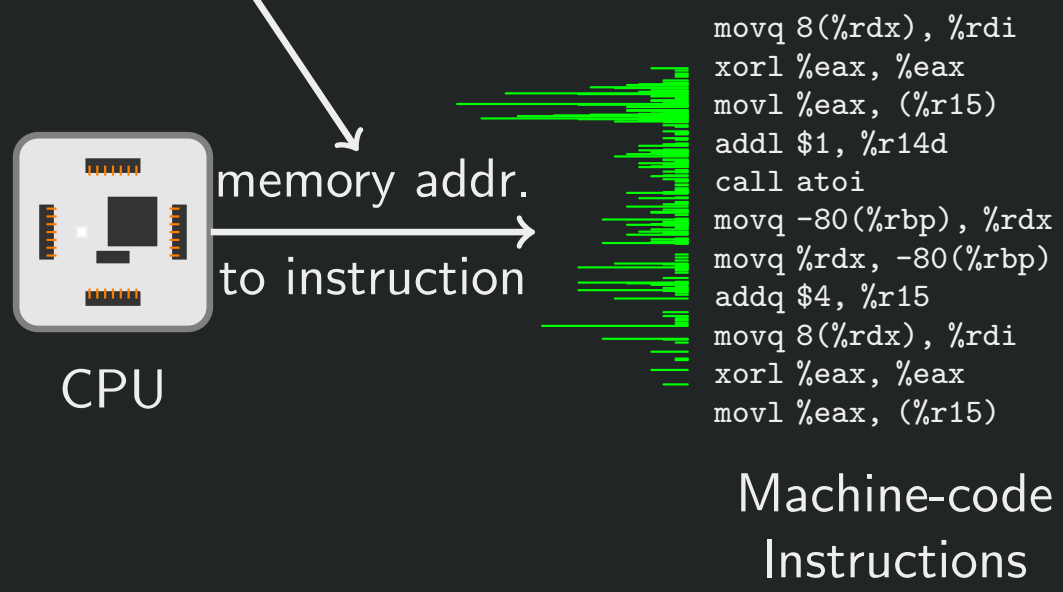
CPU

```
movq 8(%rdx), %rdi
xorl %eax, %eax
movl %eax, (%r15)
addl $1, %r14d
call atoi
movq -80(%rbp), %rdx
movq %rdx, -80(%rbp)
addq $4, %r15
movq 8(%rdx), %rdi
xorl %eax, %eax
movl %eax, (%r15)
```

Machine-code Instructions

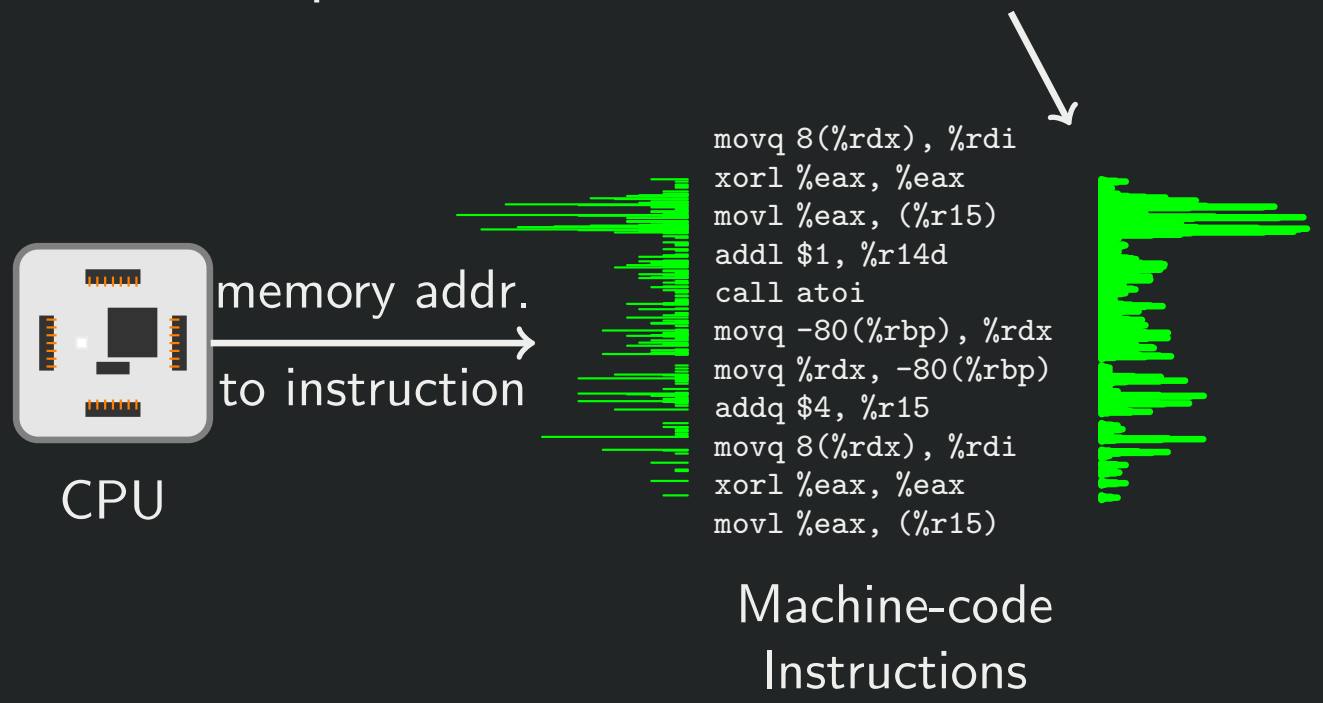
Light Weight Fault Localization

1. Sample program counter.
2. Translate memory addresses to program offsets.
3. Smooth sample with Gaussian convolution.



Light Weight Fault Localization

1. Sample program counter.
2. Translate memory addresses to program offsets.
3. Smooth sample with Gaussian convolution.



Embedded Repair Results

Back

- Effective
- 62% faster runtime
- 95% smaller disk footprint
- 86% less memory

Total bugs repaired

Rep.	Num. Bugs
AST	13
ASM	12
ELF	11

Average success rate

100 runs per bug

Rep.	Success Rate
AST	78.17%
ASM	70.75%
ELF	65.83%

Embedded Repair Results

Back

- Effective
- 62% faster runtime
- 95% smaller disk footprint
- 86% less memory

Expected fitness evaluations

Rep.	Evaluations
AST	583.98
ASM	188.38
ELF	207.15

Total runtime

Rep.	Sec.
AST	229.50
ASM	278.30
ELF	74.20

Embedded Repair Results

Back

- Effective
- 62% faster runtime
- 95% smaller disk footprint
- 86% less memory

Example: Merge Sort

Repair by representation

- AST, 2 of 4900 Swaps
- ASM, 1 of 280 Deletes

merge.c

```
if(left[l-mid-1]<=right[0]){
  result=list; }
else{/* fix:swap branches */
  result=merge(left,l-mid,
                 right,mid); }
```

merge.s

```
cmpl %eax, %edx ; fix: del.
jg   .L12
movq -72(%rbp), %rax
```

Embedded Repair Results

Back

- Effective
- 62% faster runtime
- 95% smaller disk footprint
- 86% less memory

Disk size

Rep.	Requirements
AST	Source code & build toolchain
ASM	Assembly code & linker
ELF	Compiled executable

Embedded Repair Results

Back

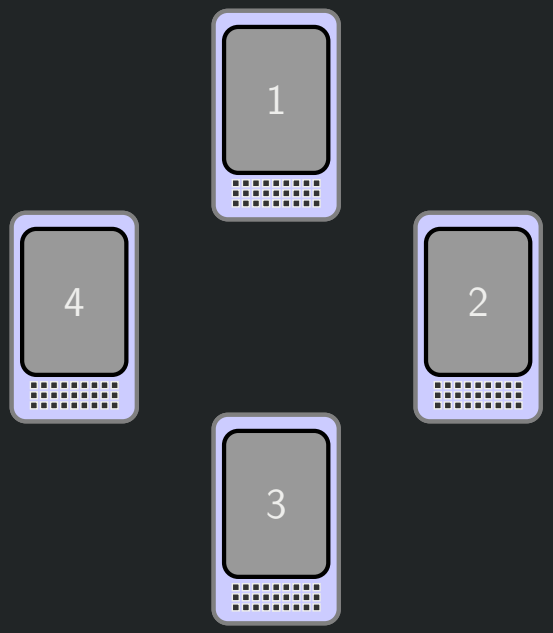
- Effective
- 62% faster runtime
- 95% smaller disk footprint
- 86% less memory

Working memory

Rep.	MB
AST	1402
ASM	756
ELF	200

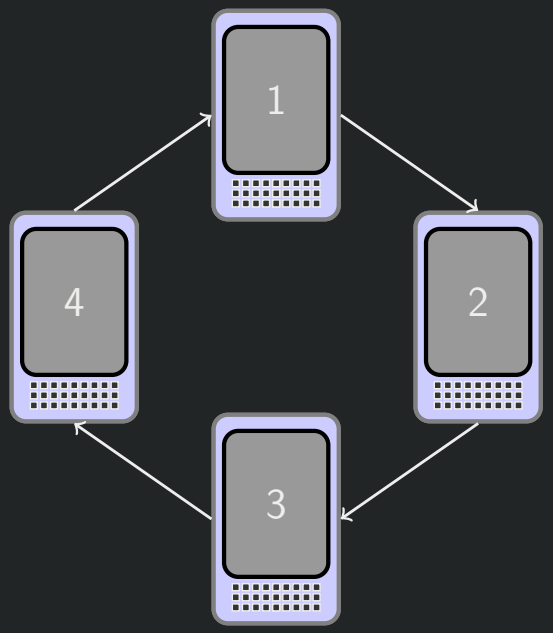
Distributed Embedded Repair

Back



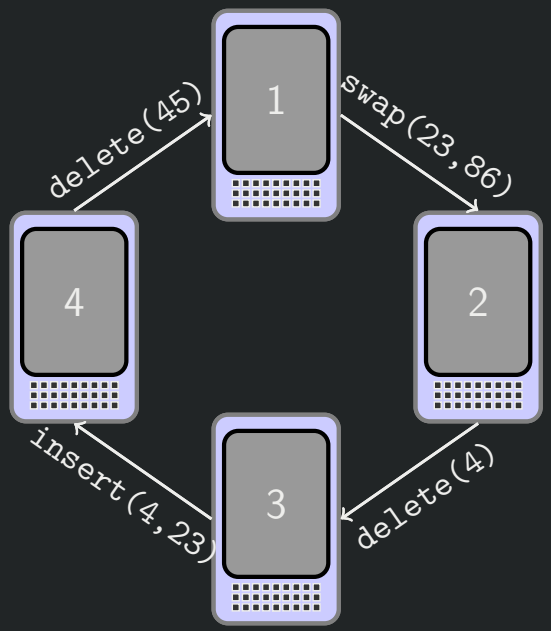
Distributed Embedded Repair

Back



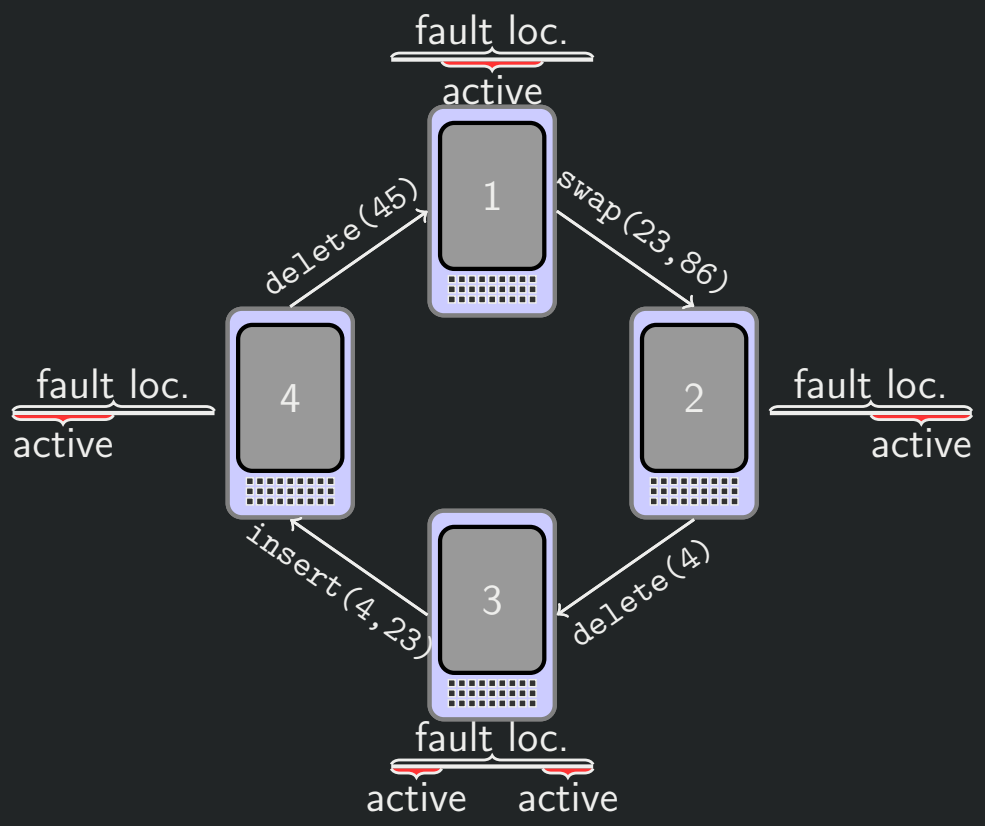
Distributed Embedded Repair

Back



Distributed Embedded Repair

Back



Distributed Genetic Repair Evaluation

Relative performance of Distributed Algorithm

# Nodes	Expected Fitness Evaluations	Wall Clock Seconds	Wall Clock w/SMS
1	1	1	1
2	0.94	0.89	1.07
3	0.84	0.67	0.81
4	0.80	0.55	0.63

Distributed Genetic Repair Evaluation

Relative performance of Distributed Algorithm

# Nodes	Expected Fitness Evaluations	Wall Clock Seconds	w/SMS
1	1	1	1
2	0.94	0.89	1.07
3	0.84	0.67	0.81
4	0.80	0.55	0.63

Distributed Genetic Repair Evaluation

Relative performance of Distributed Algorithm

# Nodes	Expected Fitness Evaluations	Wall Clock Seconds	Wall Clock w/SMS
1	1	1	1
2	0.94	0.89	1.07
3	0.84	0.67	0.81
4	0.80	0.55	0.63

Benefits of ASM

Back

The ASM representation performs well.

Avoids

- direct addresses
- argumented instructions

Similarities to DNA

- Linear vector genome
- read sequentially
- reading frames
- start and stop codes
- padding
- bootstrapped compilers

Embedded Repair Search Space

Back

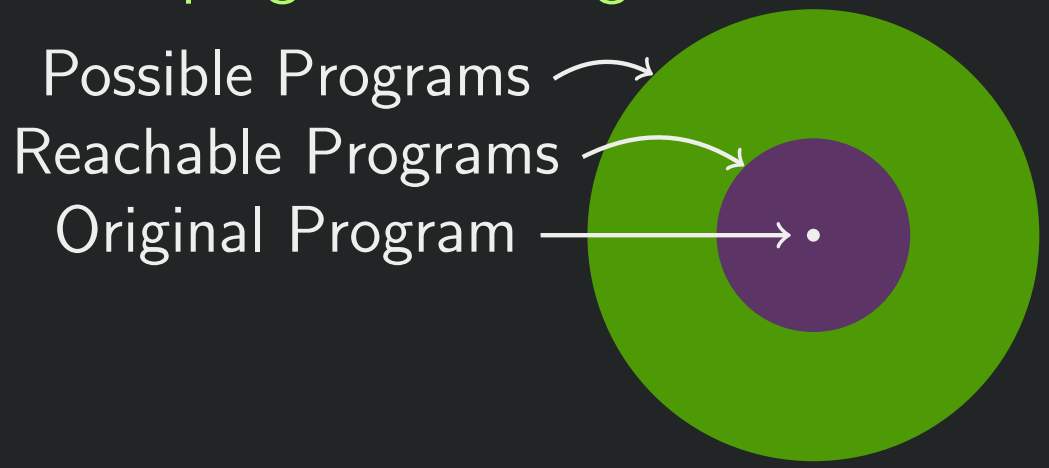
Program size

$\approx 3\times$ more assembly instructions than C statements

Search space size

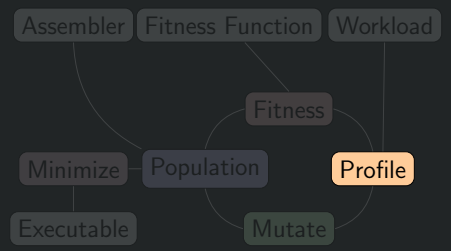
$= |\text{alphabet}|^{\text{program size}}$

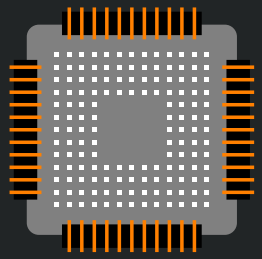
Possible program coverage



Profiling

Back

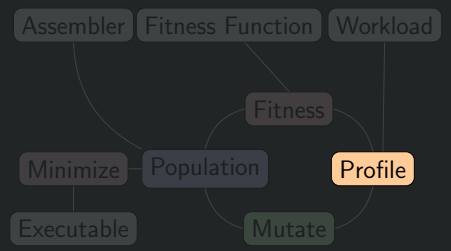




Hardware Performance Counters

Profiling

Back

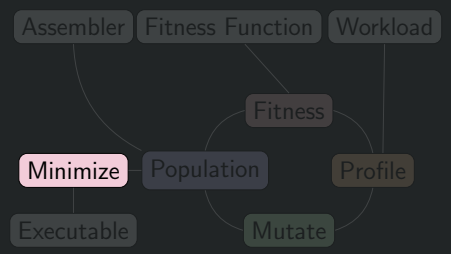



```
$ perf stat -- ./blackscholes 1 input /tmp/output

  6,864,315,342 cycles
  5,062,293,918 instructions
  2,944,060,039 r533f00
  1,113,084,780 cache-references
      1,122,960 cache-misses

    3.227585368 seconds time elapsed
```

Minimization

Back

Delta Debugging




```
5358c5358
< .L808:
---
>  addl %ebx, %ecx
```

```
5416c5416
<  addl %ebx, %ecx
---
> .L808:
```

```
5463c5463
< .L970:
---
>  .byte 0x33
```

```
5651d5650
<  .loc 1 457 0 is_stmt 0 discriminator 2


5841d5839
<  addq %rdx, %r14
```

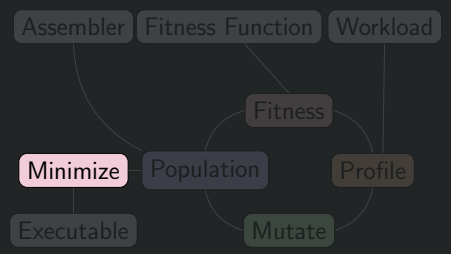
```
6309c6307
<  xorpd %xmm1, %xmm7
---
>  cmpq %r13, %rdi
```

```
6413a6412
>  cmpl %ecx, %esi
```

Minimization

Back

Delta Debugging




```
5358c5358
< .L808:
---
>  addl %ebx, %ecx
```

```
5416c5416
<  addl %ebx, %ecx
---
> .L808:
```

```
5463c5463
< .L970:
---
>  .byte 0x33
```

```
5651d5650
<  .loc 1 457 0 is_stmt 0 discriminator 2


5841d5839
<  addq %rdx, %r14
```

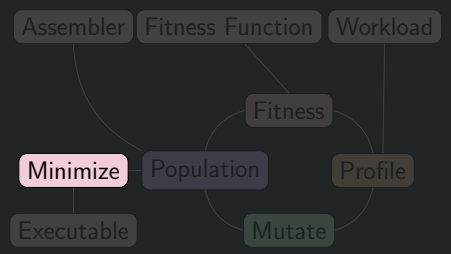
```
6309c6307
<  xorpd %xmm1, %xmm7
---
>  cmpq %r13, %rdi
```

```
6413a6412
>  cmpl %ecx, %esi
```

Minimization

Back

Delta Debugging




```
5358c5358
< .L808:
---
>  addl %ebx, %ecx
```

```
5416c5416
<  addl %ebx, %ecx
---
> .L808:
```

```
5463c5463
< .L970:
---
>  .byte 0x33
```

```
5651d5650
<  .loc 1 457 0 is_stmt 0 discriminator 2


5841d5839
<  addq %rdx, %r14
```

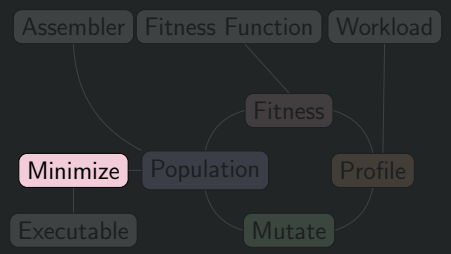
```
6309c6307
<  xorpd %xmm1, %xmm7
---
>  cmpq %r13, %rdi
```

```
6413a6412
>  cmpl %ecx, %esi
```

Minimization

Back

Delta Debugging




```
5358c5358
< .L808:
---
>  addl %ebx, %ecx
```

```
5416c5416
<  addl %ebx, %ecx
---
> .L808:
```

```
5463c5463
< .L970:
---
>  .byte 0x33
```

```
5651d5650
<  .loc 1 457 0 is_stmt 0 discriminator 2
```

```
5841d5839
<  addq %rdx, %r14
```

```
6309c6307
<  xorpd %xmm1, %xmm7
---
>  cmpq %r13, %rdi
```

```
6413a6412
>  cmpl %ecx, %esi
```

Energy Model

Back

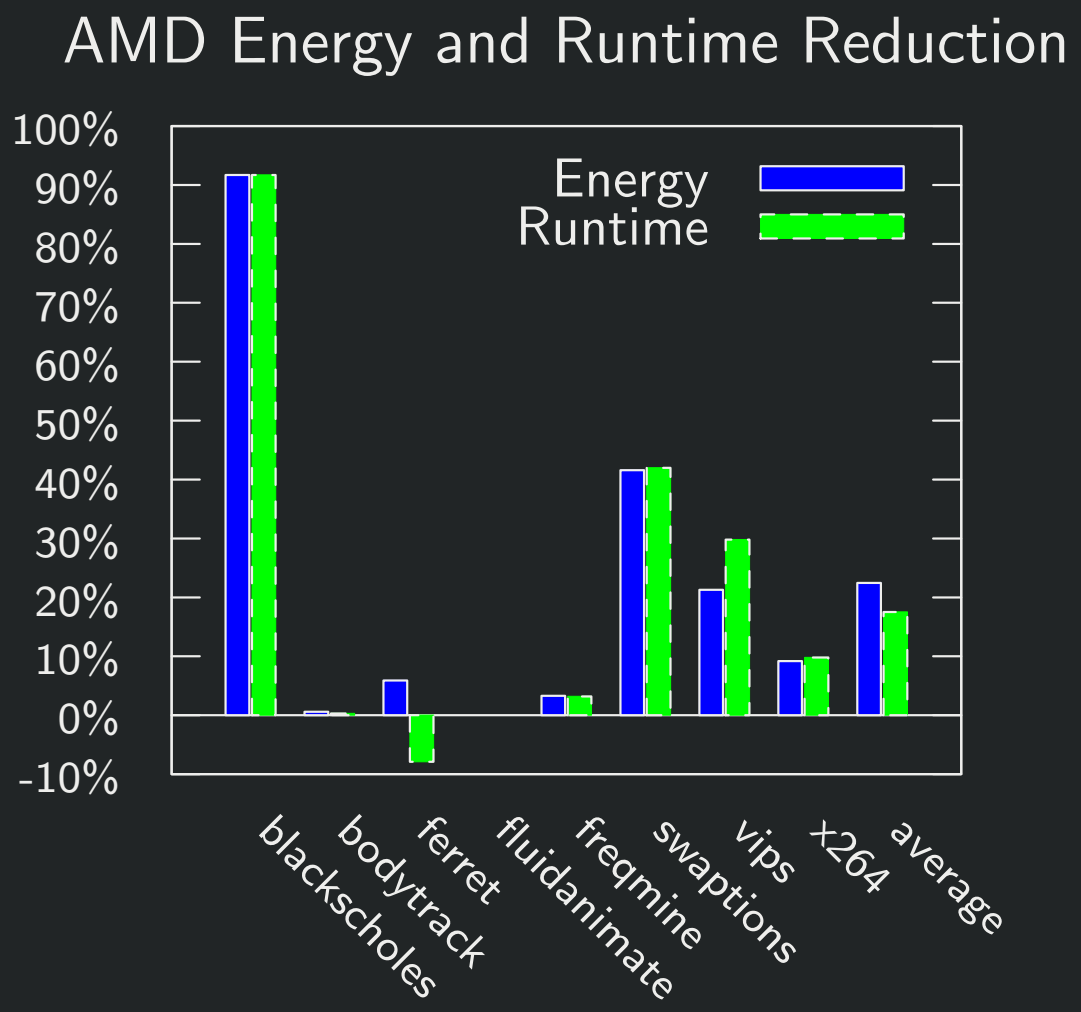
$$\frac{energy}{time} = C_{const} + C_{ins}\frac{ins}{cycle} + C_{flops}\frac{flops}{cycle} + C_{tca}\frac{tca}{cycle} + C_{mem}\frac{mem}{cycle}$$

Coefficient	Description	Intel (4-core)	AMD (48-core)
C_{const}	constant power draw	31.530	394.74
C_{ins}	instructions	20.490	-83.68
C_{flops}	floating point ops.	9.838	60.23
C_{tca}	cache accesses	-4.102	-16.38
C_{mem}	cache misses	2962.678	-4209.09

Table: Energy model coefficients.

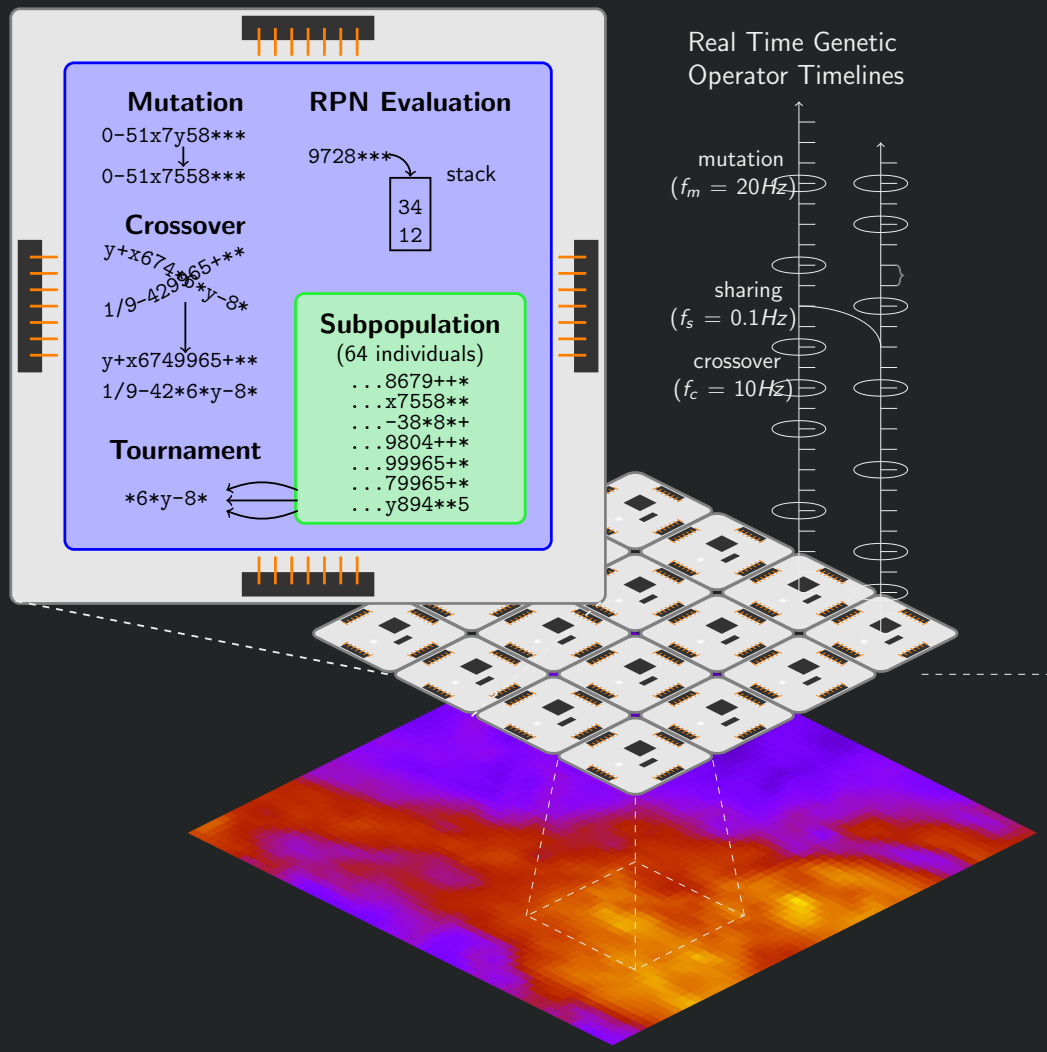
GOA: Runtime and Energy Reduction

Back



Physical Evolutionary Computation

Back



Real Time Genetic Operator Timelines

Hardware Tile

- 70MHz ARM7TDMI MCU
- 32KB RAM, 16KB EEPROM
- 128KB problem data storage
- hardware timers
- power sharing, 4 serial ports

Software Process

$\forall$ evolutionary tile
randomly create an initial population
for ever, concurrently
with frequency f_m
select one individual and mutate
with frequency f_c
select two individuals and crossover
with frequency f_s
select one individual and share
with frequency f_d **if** collecting_data
report statistics to central collector

Computational Space

Sample geometry: Sixteen $4.75cm \times 4.75cm$ tiles arranged in a 4×4 square.

Problem Space

Sample fitness function: Gravity data from the state of Montana. Each hardware tile covers $25'$ *latitude* $\times$ $25'$ *longitude* mapped using the 'local disjoint' spatial mapping. The size and shape of the total covered area is determined by the number and configuration of the hardware tiles, and by the scale and scheme used to map the hardware space to the problem space.

The End

Back

Eric Schulte

email eschulte@cs.unm.edu

homepage https://cs.unm.edu/~eschulte

dissertation https://cs.unm.edu/~eschulte/dissertation