AUTOMATIC PROGRAM REPAIR USING GENETIC PROGRAMMING

CLAIRE LE GOUES

APRIL 22, 2013

GENPROG

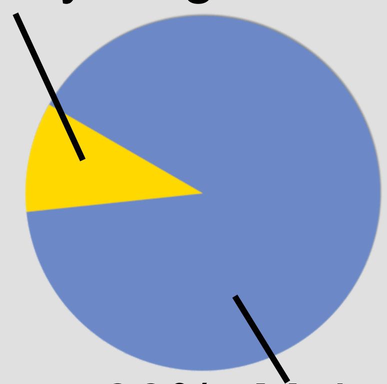
STOCHASTIC SEARCH
+
TEST CASE GUIDANCE
=
AUTOMATIC, EXPRESSIVE, SCALABLE PATCH GENERATION

"Everyday, almost 300 bugs appear […] far too many for only the Mozilla programmers to handle."
– *Mozilla Developer, 2005*

Annual cost of software errors in the US: $59.5 billion (0.6% of GDP).

PROBLEM: BUGGY SOFTWARE

Average time to fix a security-critical error: 28 days.



SOLUTION: AUTOMATE

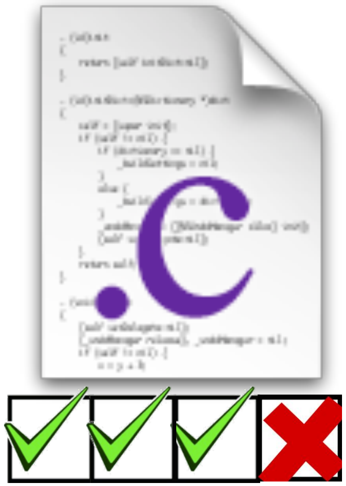
PRIOR ART

Self-healing systems, security research: runtime monitors, repair strategies, error preemption.

- Designed to address particular types of bugs, (e.g., buffer overruns).
- Very successful in that domain (e.g., data execution prevention shipping with Windows 7).

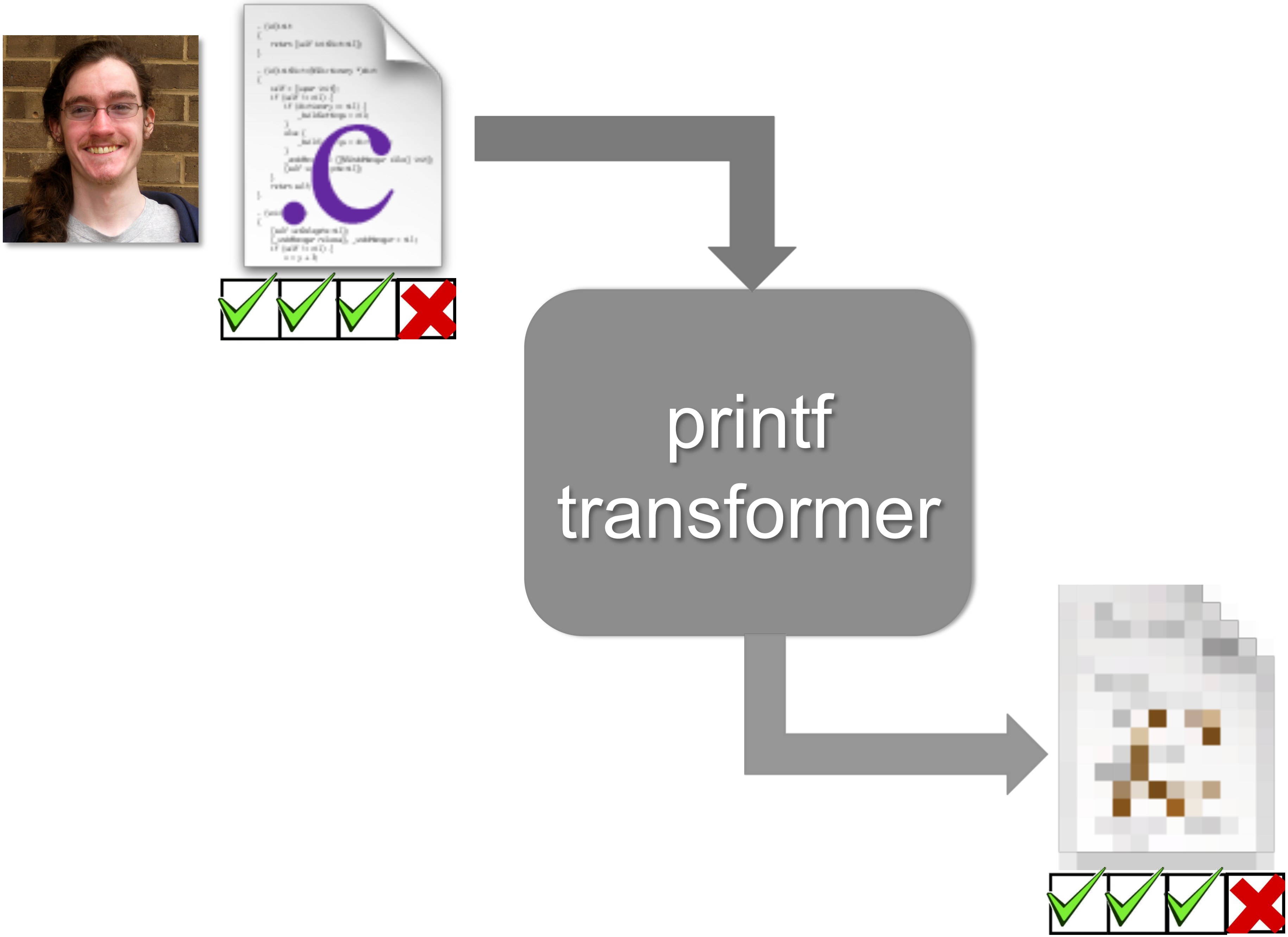
But what about generic repair of new real-world bugs as they come in?

HOW DO HUMANS FIX NEW BUGS?

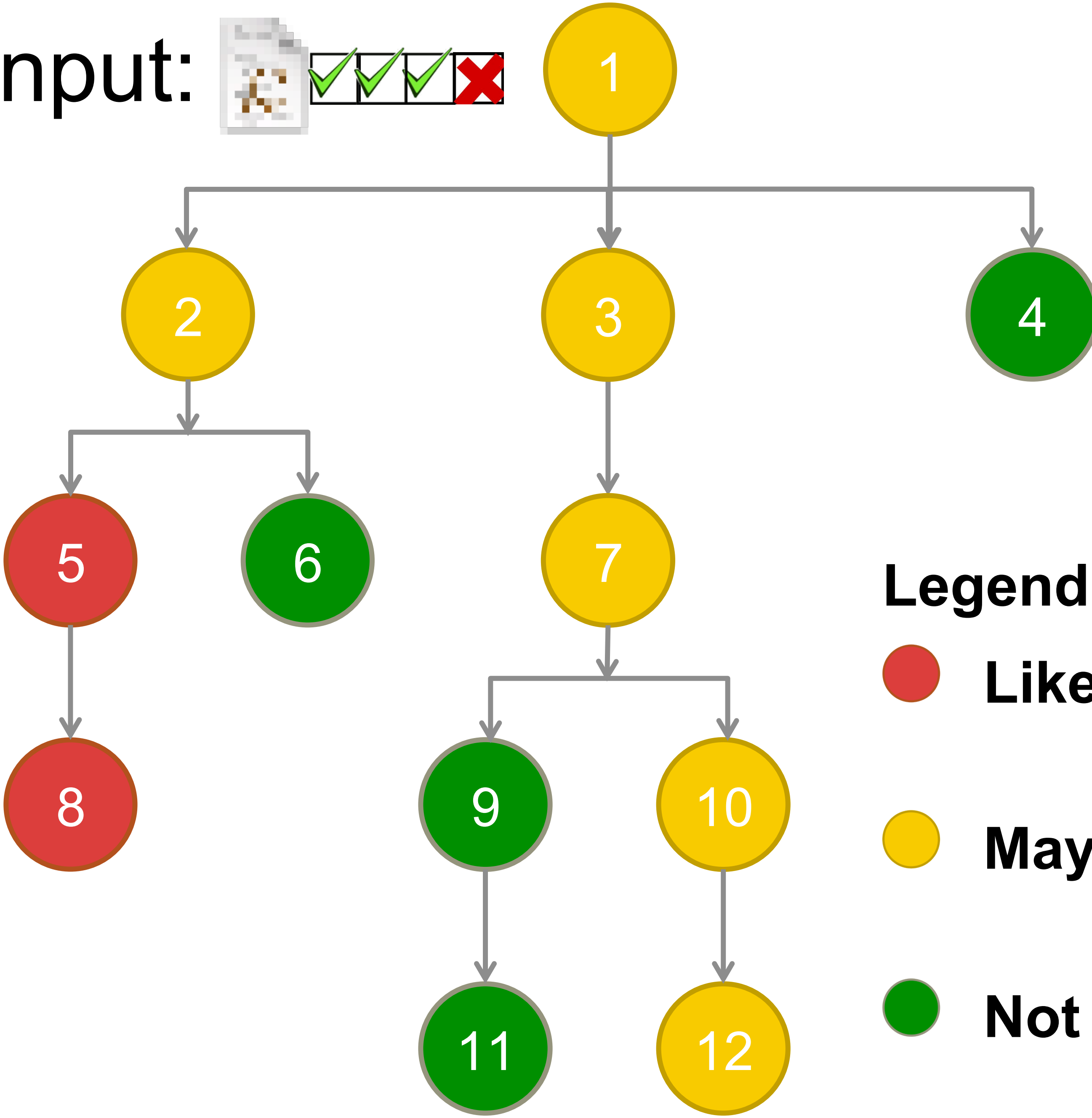


? ? !

NOW WHAT?


printf
transformer

Input:



Legend:

- Likely faulty.
- Maybe faulty.
- Not faulty.

SECRET SAUCES

- Test cases are useful.
- Existing program behavior contains the seeds of many repairs.
- The space of program patches can be **searched**.

THESIS

Stochastic search, guided by existing test cases (GENPROG), can provide a

- scalable
- expressive
- human competitive

…approach for the automated repair of:

- many types of defects
- in many types of real-world programs.

OUTLINE

GenProg: automatic program repair using genetic programming.

Four overarching hypotheses.

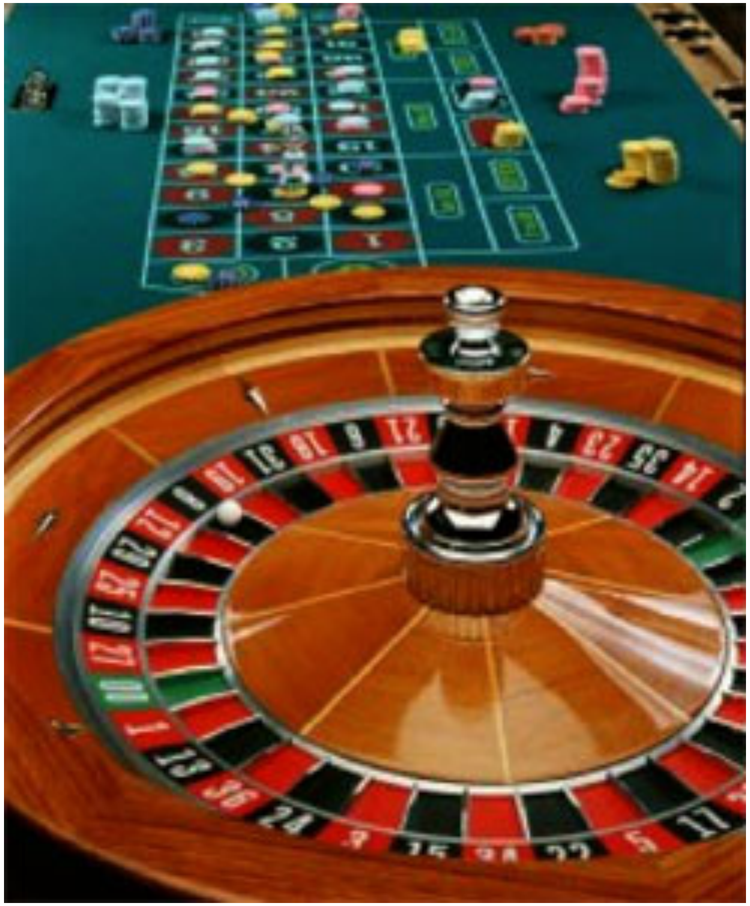
Empirical evaluations of:

- Expressive power.
- Scalability.

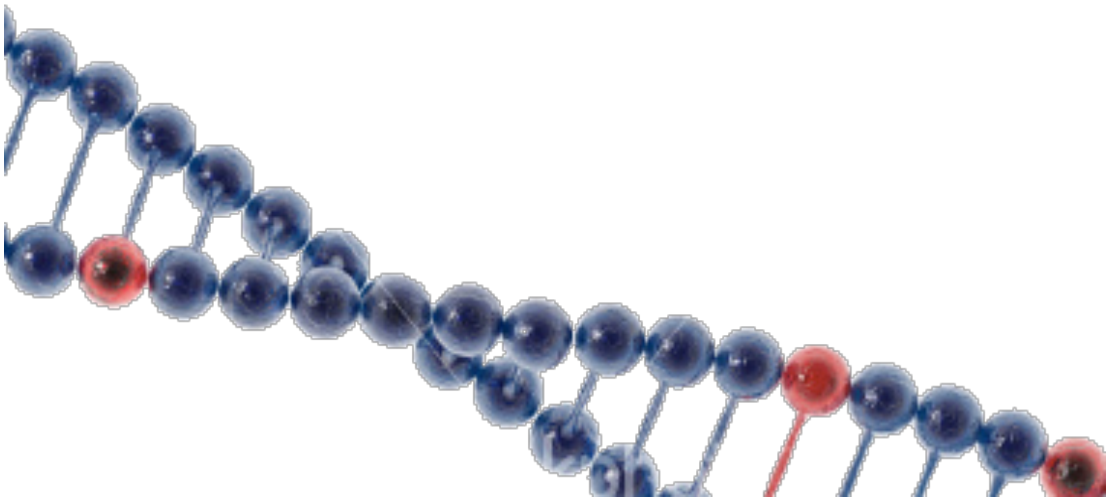
Contributions/concluding thoughts.

APPROACH

Given a program and a set of test cases, conduct a biased, random search for a set of edits to a program that fixes a given bug.



GENETIC PROGRAMMING: the application of evolutionary or genetic algorithms to program source code.



GENETIC PROGRAMMING

Population of variants.

Fitness function evaluates desirability.

Desirable individuals are more likely to be selected for iteration and reproduction.

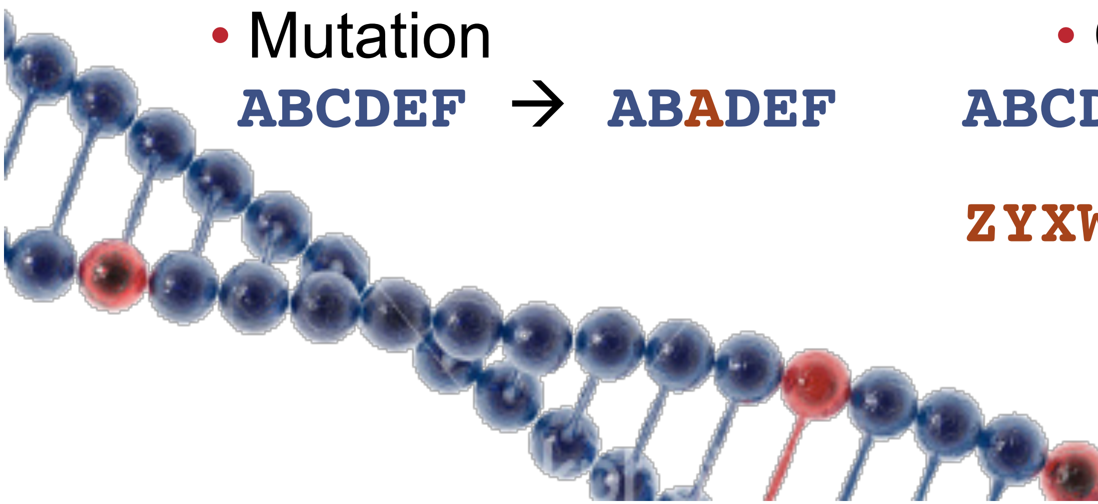
New variants created via:

- Mutation

  ABCDEF → ABADEF

- Crossover

  ABCDEF, ZYXWVU → ABCWVU, ZYXDEF



CHALLENGES

The search is through the space of candidate patches or sets of changes to the input program.

Two concerns:

1. **Scalability** – management, reduction, and traversal of the search space.
2. **Correctness** – proposed repair should fix the bug while maintaining other required functionality.

INSIGHTS

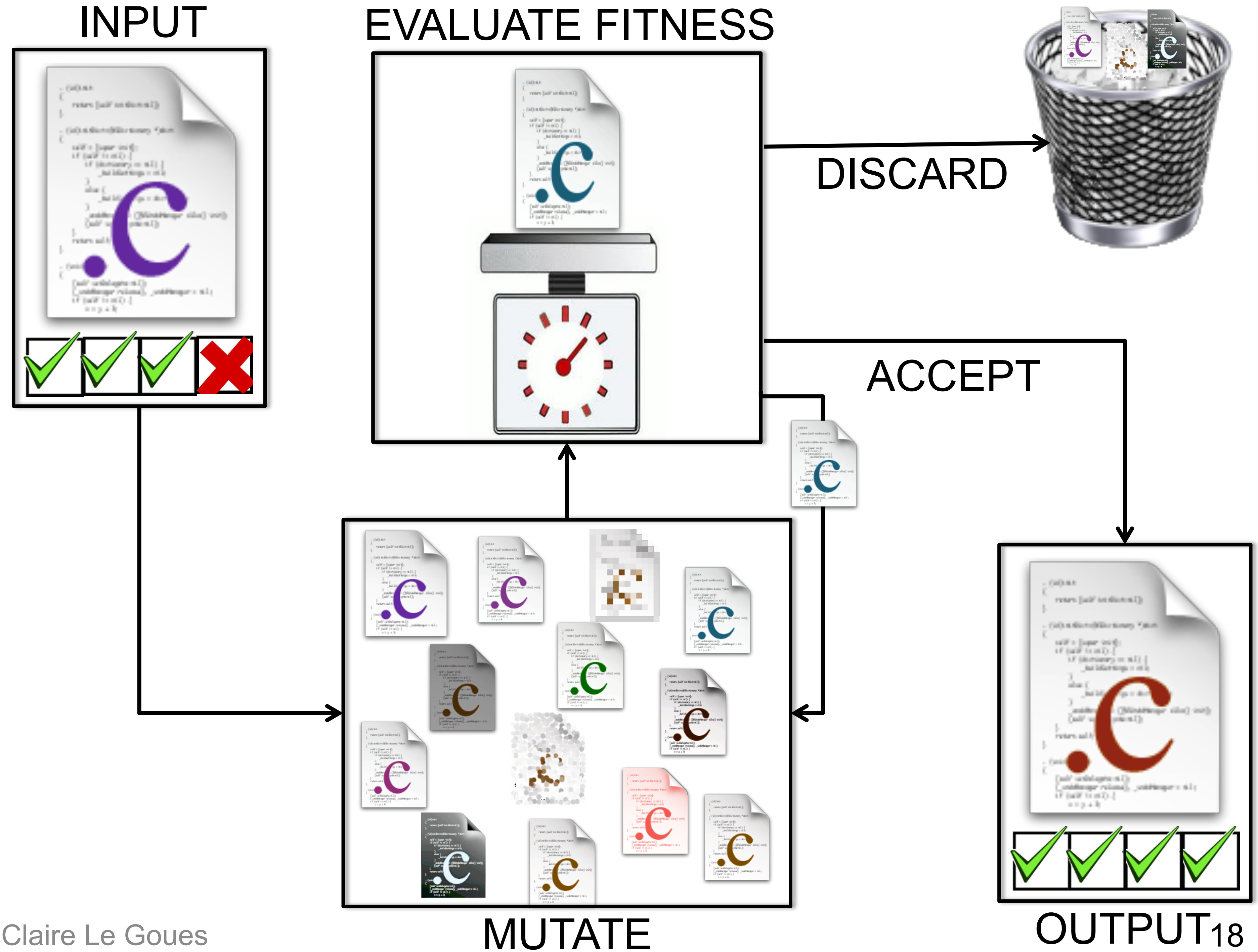
Explore coarse-grained edits at the statement level of the abstract syntax tree ([delete; replace; insert]).

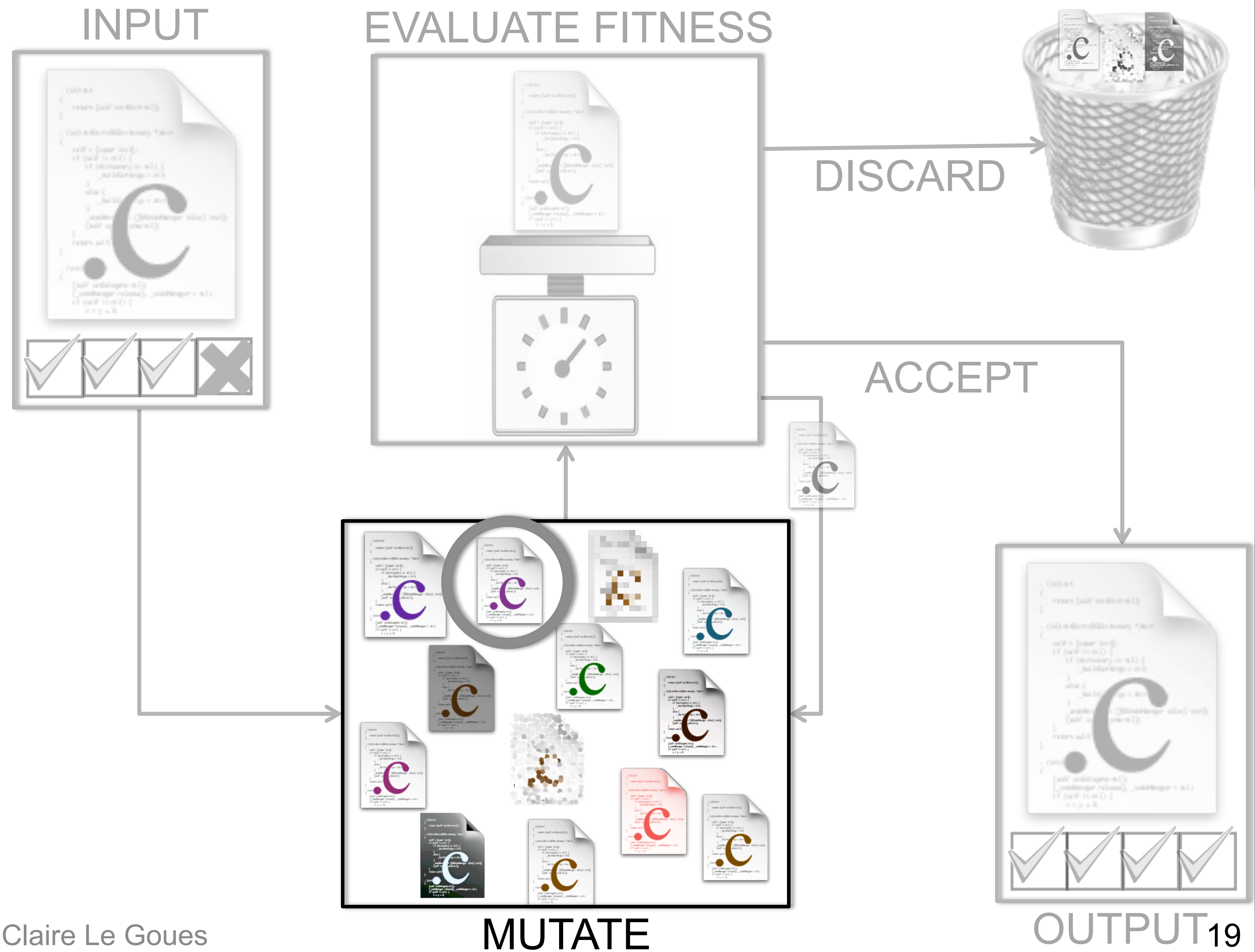
Use existing test suites as proxies for correctness specifications, and to reduce the search space.

- Evaluate intermediate solutions.
- Localize the fault, focusing candidate changes.

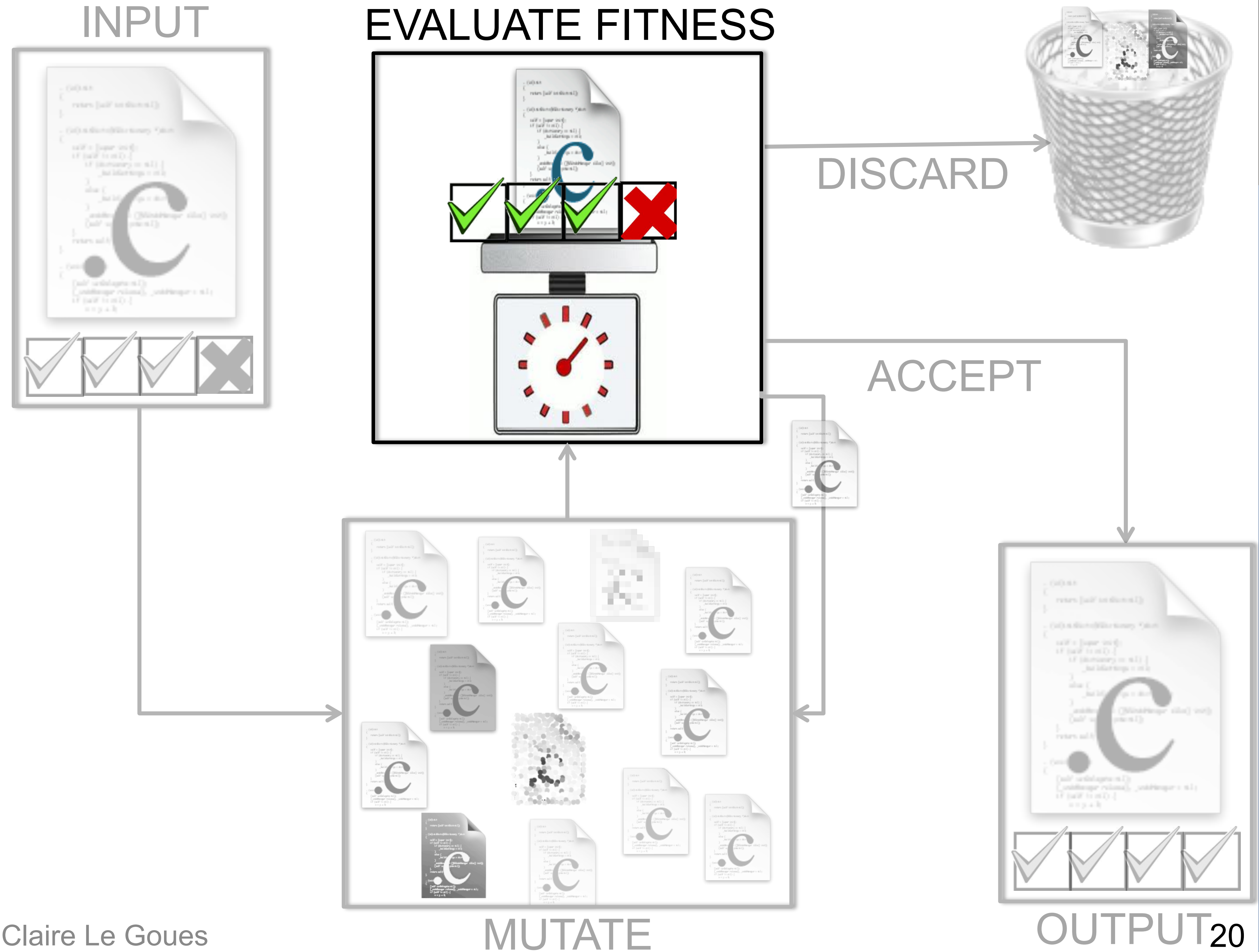
Leverage existing code and behavior.

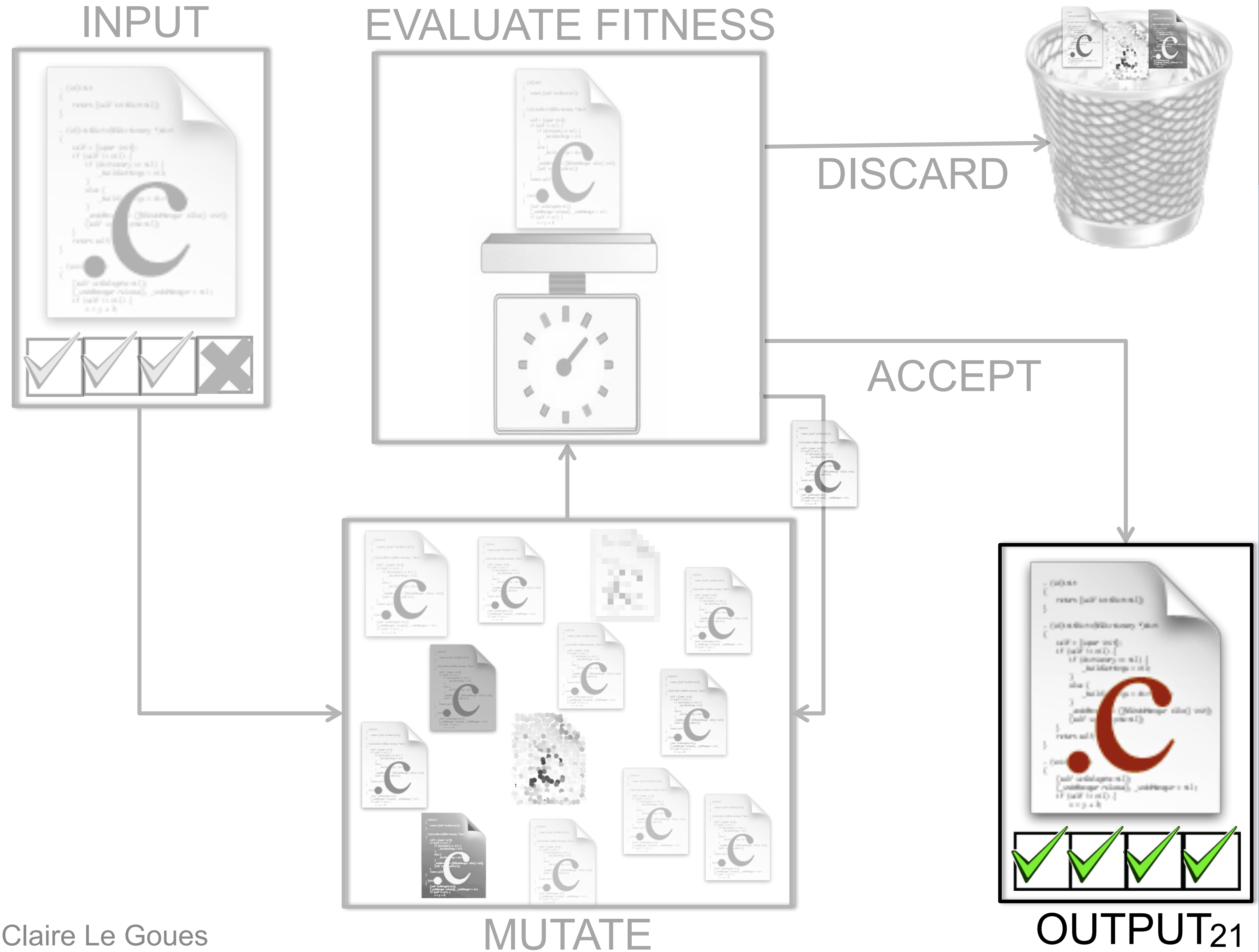
- Do not invent new code; copy code from elsewhere in the same program.


INPUT
EVALUATE FITNESS
DISCARD
ACCEPT
MUTATE
OUTPUT



INPUT
EVALUATE FITNESS
DISCARD
ACCEPT
MUTATE
OUTPUT


INPUT
EVALUATE FITNESS
DISCARD
ACCEPT
MUTATE
OUTPUT


INPUT
EVALUATE FITNESS
DISCARD
ACCEPT
MUTATE
OUTPUT

```
> gcd(4,2)
> 2
> gcd(0,55)
> 55
```

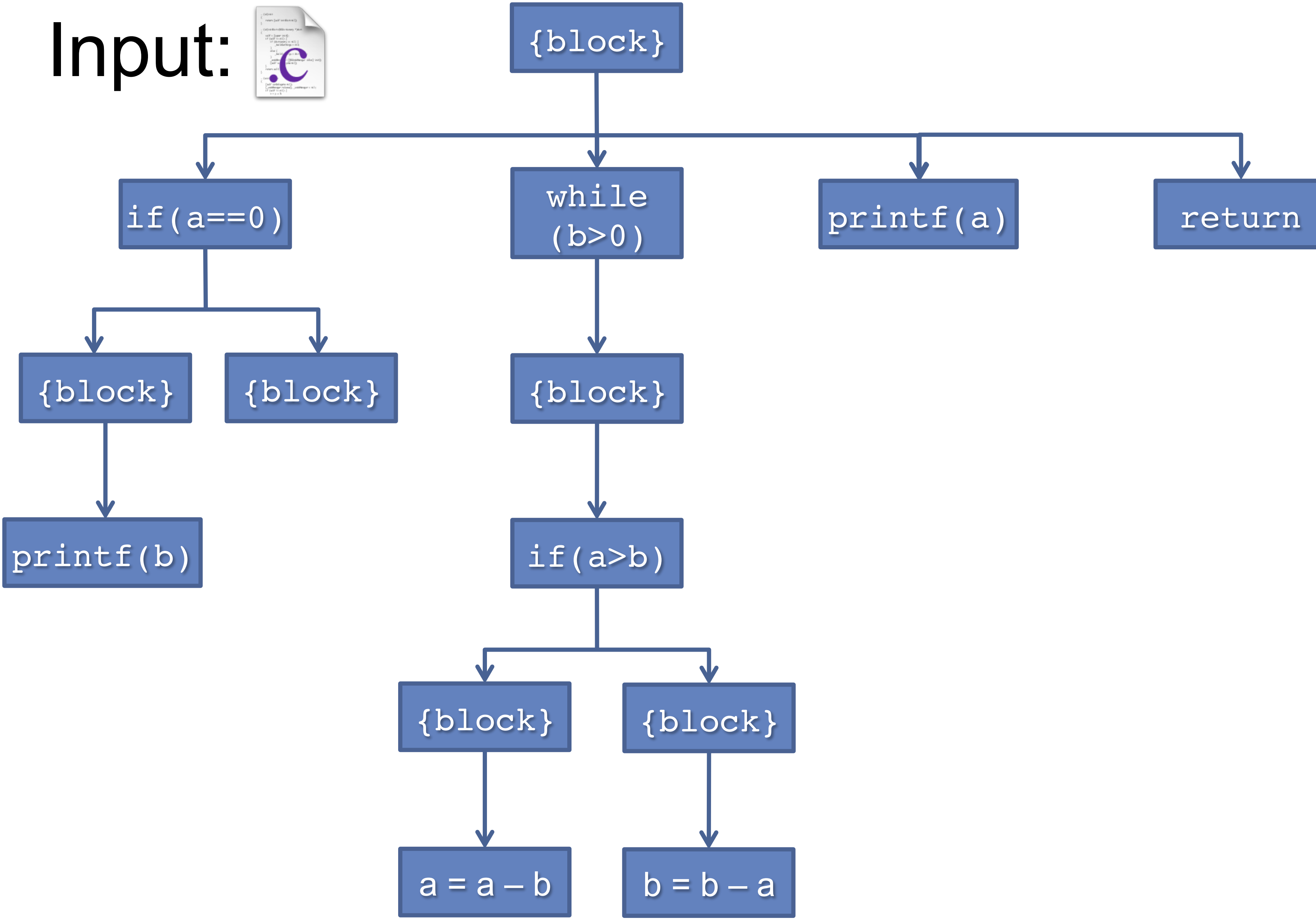
(looping forever)

```c
void gcd(int a, int b) {
  if (a == 0) {
    printf("%d", b);
  }
  while (b > 0) {
    if (a > b)
      a = a – b;
    else
      b = b – a;
  }
  printf("%d", a);
  return;
}
```

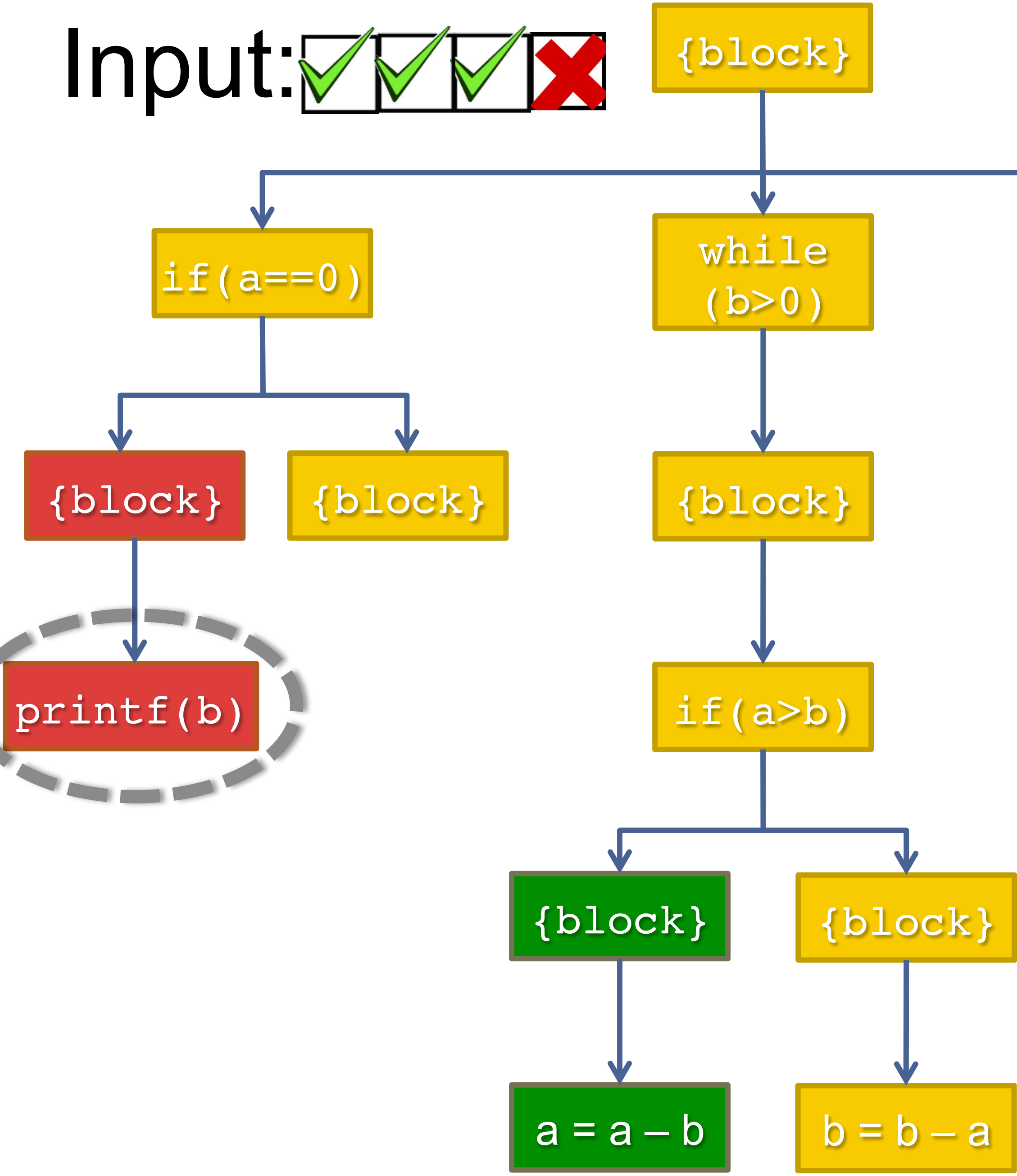
```
(a=0; b=55)            void gcd(int a, int b) {
       true              if (a == 0) {
       > 55                printf("%d", b);
                         }
(a=0; b=55) true         while (b > 0) {
      false                if (a > b)
                             a = a – b;
                           else
b = 55 - 0                   b = b – a;
                         }
                         printf("%d", a);
                         return;
                       }
```

!

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

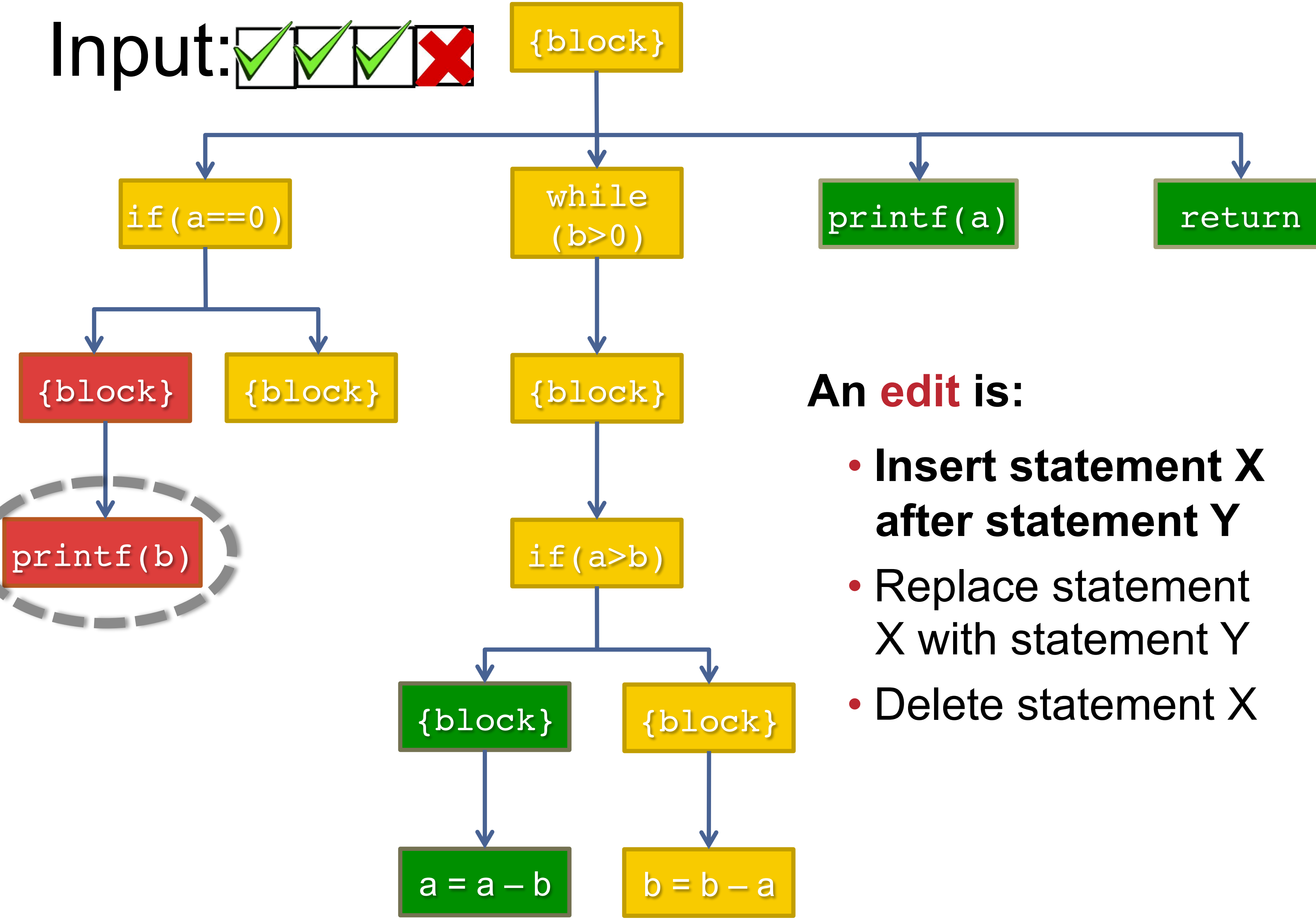
Input:



Legend:

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

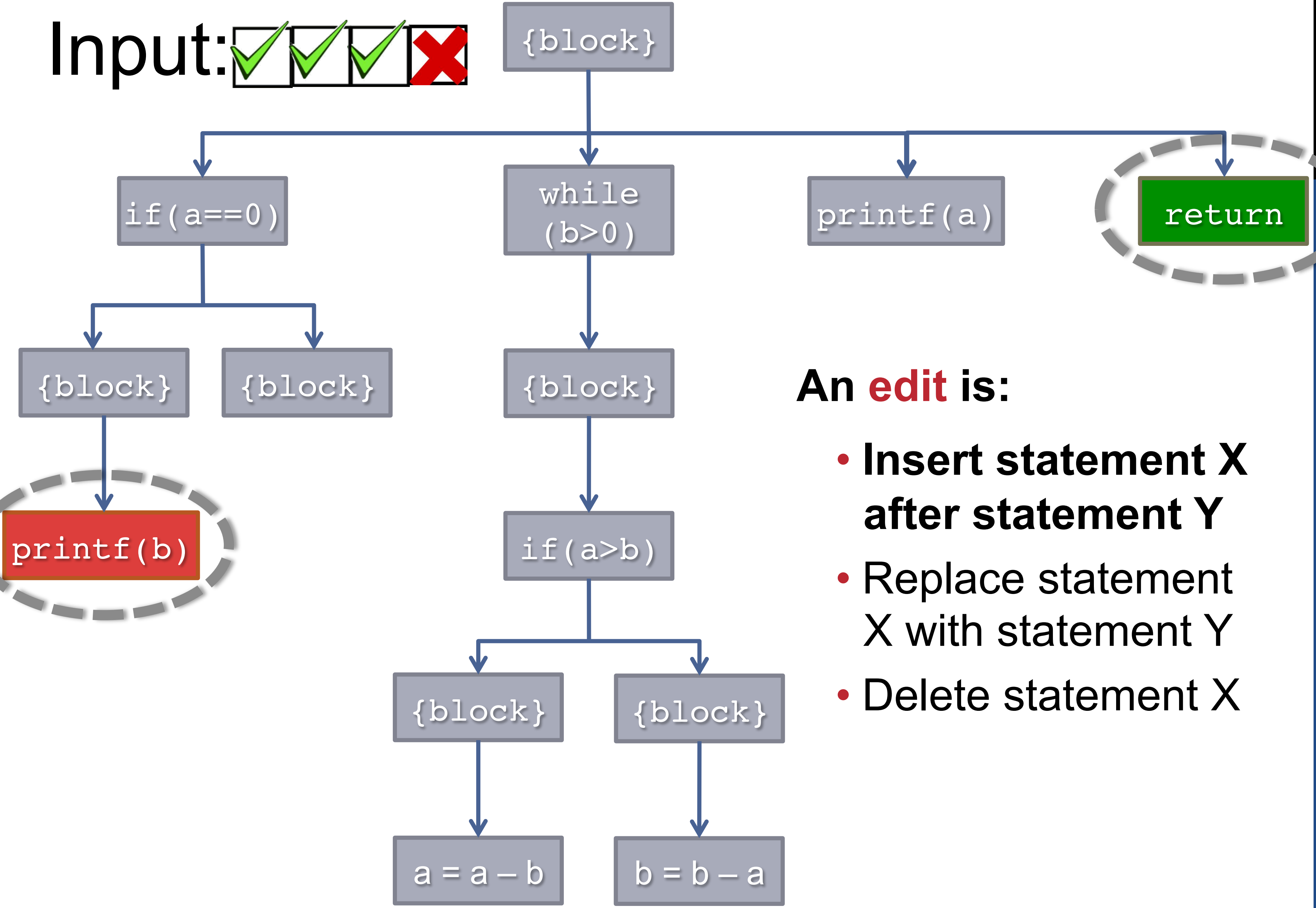
Input:



An edit is:

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

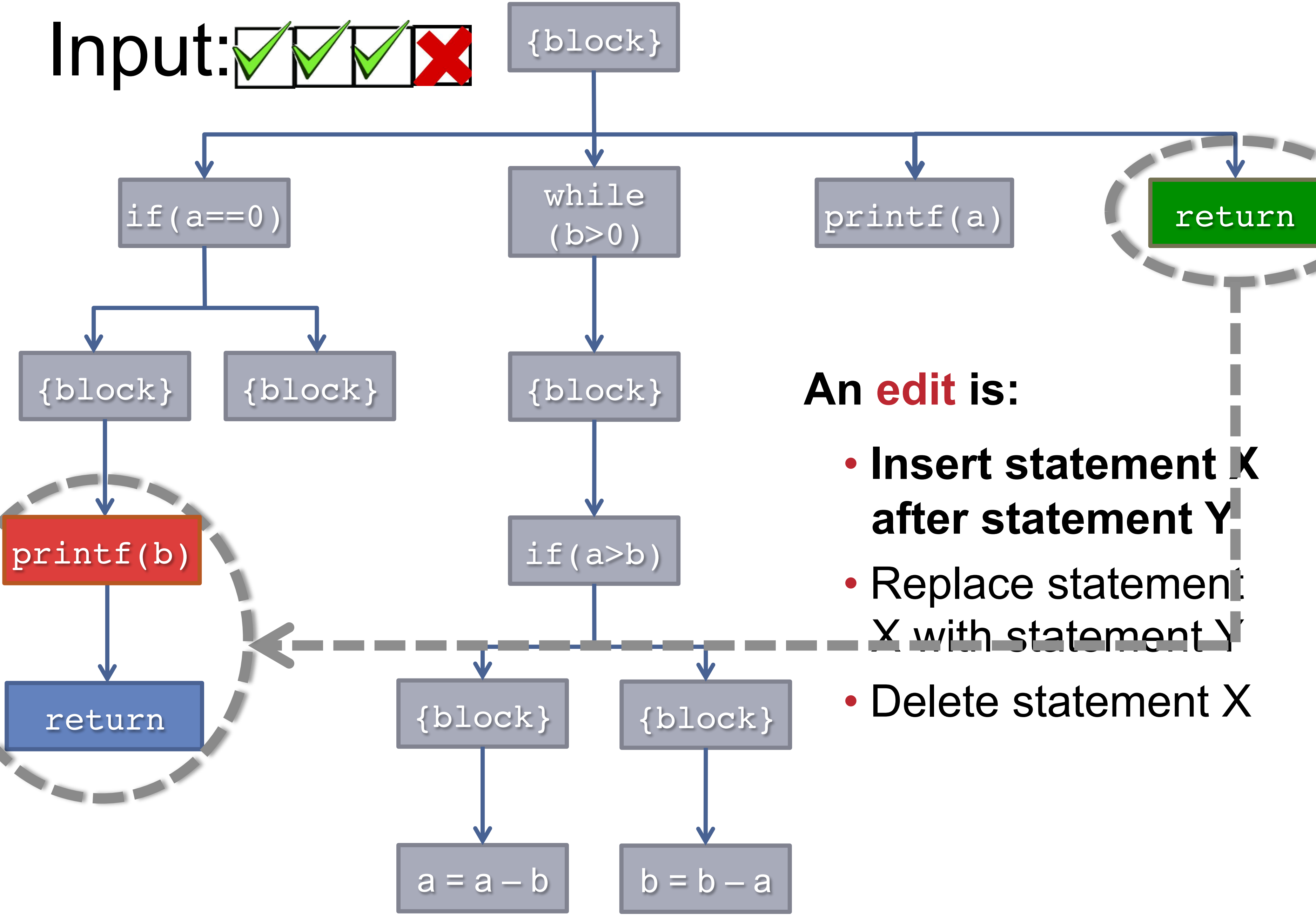
Input:



An edit is:

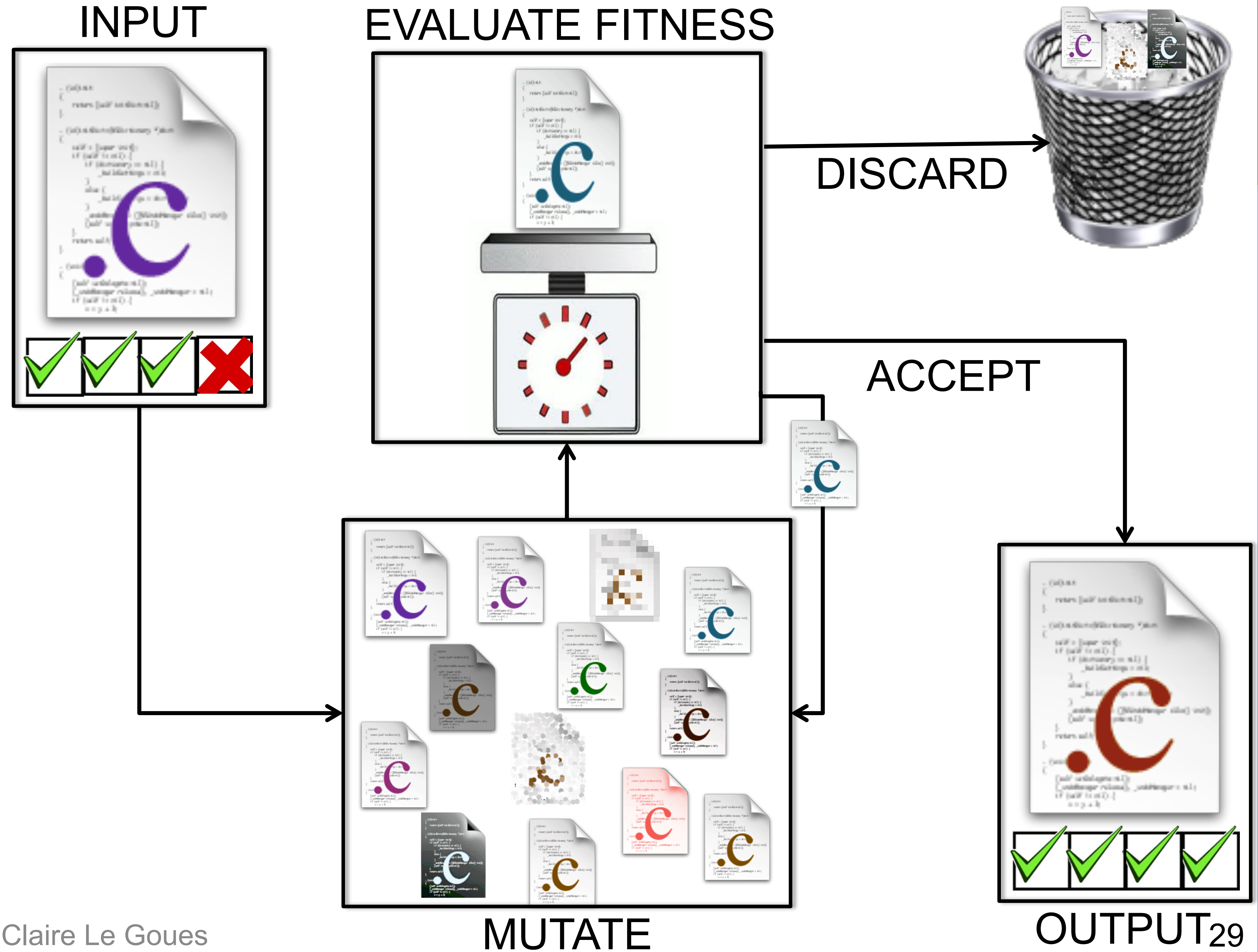
- **Insert statement X after statement Y**
- Replace statement X with statement Y
- Delete statement X

Input:



An edit is:

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


INPUT
EVALUATE FITNESS
DISCARD
ACCEPT
MUTATE
OUTPUT

OUTLINE

GenProg: automatic program repair using genetic programming.

Four overarching hypotheses.

Empirical evaluations of:

- Expressive power.
- Scalability

Contributions/concluding thoughts.

HUMAN-COMPETITIVE REPAIR

Goal: an automatic solution to alleviate a portion of the bug repair burden.

Should be competitive with the humans its designed to help.

Humans can:

- Fix many different kinds of bugs in many different kinds of programs. [expressive power]
- Fix bugs in large systems. [scalability]
- Produce acceptable patches. [repair quality]

HYPOTHESES

Without defect- or program- specific information, GenProg can:

1. repair at least 5 different defect types, and can repair defects in at least least 10 different program types.
2. repair at least 50% of defects that humans developers fix in practice.
3. repair bugs in large programs of up to several million lines of code, and associated with up to several thousand test cases, at a time and economic cost that is human competitive.
4. produce patches that maintain existing program functionality; do not introduce new vulnerabilities; and address the underlying cause of a vulnerability.

Program	Description	LOC	Bug Type
gcd	example	22	infinite loop
nullhttpd	webserver	5575	heap buffer overflow (code)
zune	example	28	infinite loop
uniq	text processing	1146	segmentation fault
look-u	dictionary lookup	1169	segmentation fault
look-s	dictionary lookup	1363	infinite loop
units	metric conversion	1504	segmentation fault
deroff	document processing	2236	segmentation fault
indent	code processing	9906	infinite loop
flex	lexical analyzer generator	18774	segmentation fault
openldap	directory protocol	292598	non-overflow denial of service
ccrypt	encryption utility	7515	segmentation fault
lighttpd	webserver	51895	heap buffer overflow (vars)
atris	graphical game	21553	local stack buffer exploit
php	scripting language	764489	integer overflow
wu-ftpd	FTP server	67029	format string vulnerability

HYPOTHESES

Without defect- or program- specific information, GenProg can:

1. repair at least 5 different defect types, and can repair defects in at least least 10 different program types.
2. repair at least 50% of defects that humans developers fix in practice.
3. repair bugs in large programs of up to several million lines of code, and associated with up to several thousand test cases, at a time and economic cost that is human competitive.
4. produce patches that maintain existing program functionality; do not introduce new vulnerabilities; and address the underlying cause of a vulnerability.

SETUP

Goal: systematically evaluate GenProg on a general, indicative bug set.

General approach:

- Avoid overfitting: fix the algorithm.
- Systematically create a generalizable benchmark set.
- **Try to repair every bug in the benchmark set, establish grounded cost measurements.**

CHALLENGE: INDICATIVE BUG SET

SYSTEMATIC BENCHMARK SELECTION

Goal: a large set of important, reproducible bugs in non-trivial programs.

Approach: use historical source control data to approximate discovery and repair of bugs in the wild.

BENCHMARKS

Program	LOC	Tests	Bugs	Description
fbc	97,000	773	3	Language (legacy)
gmp	145,000	146	2	Multiple precision math
gzip	491,000	12	5	Data compression
libtiff	77,000	78	24	Image manipulation
lighttpd	62,000	295	9	Web server
php	1,046,000	8,471	44	Language (web)
python	407,000	355	11	Language (general)
wireshark	2,814,000	63	7	Network packet analyzer
Total	**5,139,000**	**10,193**	**105**	

CHALLENGE: GROUNDED COST MEASUREMENTS


amazon
web services™

READY:
GO!
13 HOURS LATER

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.

PUBLIC COMPARISON

JBoss issue tracking: median 5.0, mean 15.3 hours.

IBM: $25 per defect during coding, rising at build, Q&A, post-release, etc.

Median programmer salary in the US: $72,630

- $35.40 per hour = $460 for 13 hours

Bug bounty programs:

- Tarsnap.com: $17, 40 hours per non-trivial repair.
- At least $500 for security-critical bugs.
- One of the php bugs that GenProg fixed has an associated NIST security certification.

WHICH BUGS…?

Slightly more likely to fix bugs where the human:

- restricts the repair to statements.
- touched fewer files.

As fault space decreases, success increases, repair time decreases.

As fix space increases, repair time decreases.

Some bugs are clearly more difficult to repair than others (e.g. in terms of random success rate).

HYPOTHESES

Without defect- or program- specific information, GenProg can:

1. repair at least 5 different defect types, and can repair defects in at least least 10 different program types.
2. repair at least 50% of defects that humans developers fix in practice.
3. repair bugs in large programs of up to several million lines of code, and associated with up to several thousand test cases, at a time and economic cost that is human competitive.
4. produce patches that maintain existing program functionality; do not introduce new vulnerabilities; and address the underlying cause of a vulnerability.

REPAIR QUALITY

Any proposed repair must pass all regression test cases.

A post-processing step minimizes the patches.

However, repairs are *not always* what a human would have done.

- Example: Adds a bounds check to a read, rather than refactoring to use a safe abstract string class.

QUANTITATIVE REPAIR QUALITY

What makes a high-quality repair?

- Retains required functionality.
- Does not introduce new bugs.
- Addresses the cause, not just the symptom.

Behavior on held-out workloads.

Large-scale black-box fuzz testing.

Exploit variant fuzzing.

OUTLINE

GenProg: automatic program repair using genetic programming.

Four overarching hypotheses.

Empirical evaluations of:

- Expressive power.
- Scalability.

Contributions/concluding thoughts.

PUBLICATIONS: GENPROG

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest and Westley Weimer. GenProg: A Generic Method for Automated Software Repair. *Transactions on Software Engineering* 38(1): 54-72 (Jan/Feb 2012). *(featured article)*

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. *International Conference on Software Engineering*, 2012: 3-13. *(Humies 2012, Bronze)*

Westley Weimer, ThanhVu Nguyen, **Claire Le Goues** and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. *International Conference on Software Engineering,* 2009:364-374. (*Distinguished Paper, Manfred Paul Award, Humies 2009, Gold)*

Westley Weimer, Stephanie Forrest, **Claire Le Goues** and ThanhVu Nguyen. Automatic Program Repair with Evolutionary Computation, *Communications of the ACM* Vol. 53 No. 5, May, 2010, pp. 109-116. *(invited)*

Claire Le Goues, Stephanie Forrest and Westley Weimer. Current Challenges in Automatic Software Repair. *Journal on Software Quality (invited, to appear).*

Claire Le Goues, Westley Weimer and Stephanie Forrest. Representations and Operators for Improving Evolutionary Software Repair. *Genetic and Evolutionary Computation Conference* , 2012: 959-966. *(Humies 2012, Bronze)*

Ethan Fast, **Claire Le Goues**, Stephanie Forrest and Westley Weimer. Designing Better Fitness Functions for Automated Program Repair. *Genetic and Evolutionary Computation Conference*, 2010: 965-972.

Stephanie Forrest, Westley Weimer, ThanhVu Nguyen and **Claire Le Goues**. A Genetic Programming Approach to Automatic Program Repair. *Genetic and Evolutionary Computation Conference*, 2009: 947-954*. (Best Paper, Humies 2009, Gold)*

Claire Le Goues, Stephanie Forrest and Westley Weimer. The Case for Software Evolution. Working Conference on the Future of Software Engineering 2010: 205-209.

ThanhVu Nguyen, Westley Weimer, **Claire Le Goues** and Stephanie Forrest. ”Using Execution Paths to Evolve Software Patches." *Search-Based Software Testing*, 2009. (Best Short Paper)

Claire Le Goues, Anh Nguyen-Tuong, Hao Chen, Jack W. Davidson, Stephanie Forrest, Jason D. Hiser, John C. Knight and Matthew Gundy. Moving Target Defenses in the Helix Self-Regenerative Architecture. *Moving Target Defense II, Advances in Information Security* vol. 100: 117-149, 2013.

Stephanie Forrest and **Claire Le Goues**. Evolutionary software repair. *GECCO (Companion)* 2012: 1345-1348.

PUBLICATIONS: OTHER

Claire Le Goues and Westley Weimer. Measuring Code Quality to Improve Specification Mining. *Transactions on Software Engineering* 38(1): 175-190 (Jan/Feb 2012).

Claire Le Goues and Westley Weimer. Specification Mining With Few False Positives. *Tools and Algorithms for the Construction and Analysis of Systems,* 2009: 292-306

Claire Le Goues, K. Rustan M. Leino and Michal Moskal. The Boogie Verification Debugger. *Software Engineering and Formal Methods*, 2011: 407-41

CONTRIBUTIONS

GenProg, a novel algorithm that uses genetic programming to automatically repair legacy, off-the-shelf programs.

Empirical evidence (and novel experimental frameworks) substantiating the claims that GenProg:

- is expressive, in that it can repair many different types of bugs in different types of programs.
- produces high quality repairs.
- is human competitive in expressive power and cost.

The ManyBugs benchmark set, and a system for automatically generating such a benchmark set.

Analysis of the factors that influence repair success and time, including a large-scale study of program repair representation, operators, and search space.

CONCLUSIONS

GenProg: scalable, generic, expressive automatic bug repair.

- Genetic programming search for a patch that addresses a given bug.
- Render the search tractable by restricting the search space intelligently.

It works!

- Fixes a variety of bugs in a variety of programs.
- Repaired 60 of 105 bugs for < $8 each, on average.

Benchmarks/results/source code/VM images available:

- http://genprog.cs.virginia.edu

I LOVE QUESTIONS.

UNDER THE HOOD

Representation:

- Which representation choice gives better results?
- Which representation features contribute most to success?

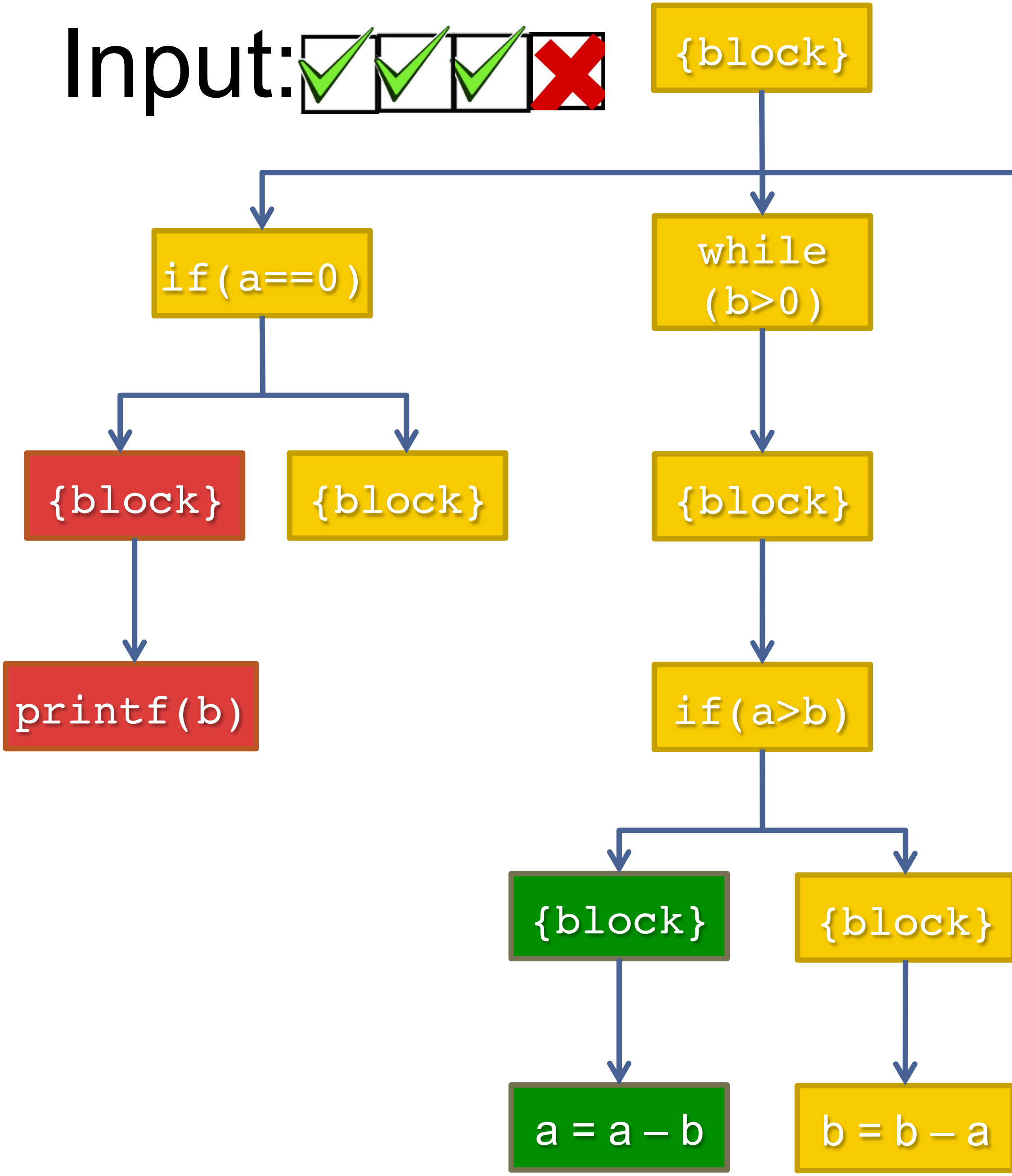
Crossover: Which crossover operator is best?

Operators:

- Which operators contribute the most to success?
- How should they be selected?

Search space: How should the representation weight program statements to best define the search space?

Input:



Legend:

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

SEARCH SPACE: SETUP

Hypothesis: statements executed only by the failing test case(s) should be weighted more heavily than those also executed by the passing test cases.

What is the ratio in actual repairs?

Expected: 10 : 1

vs.

Actual: 1 : 1.85

SEARCH SPACE EXPERIMENT

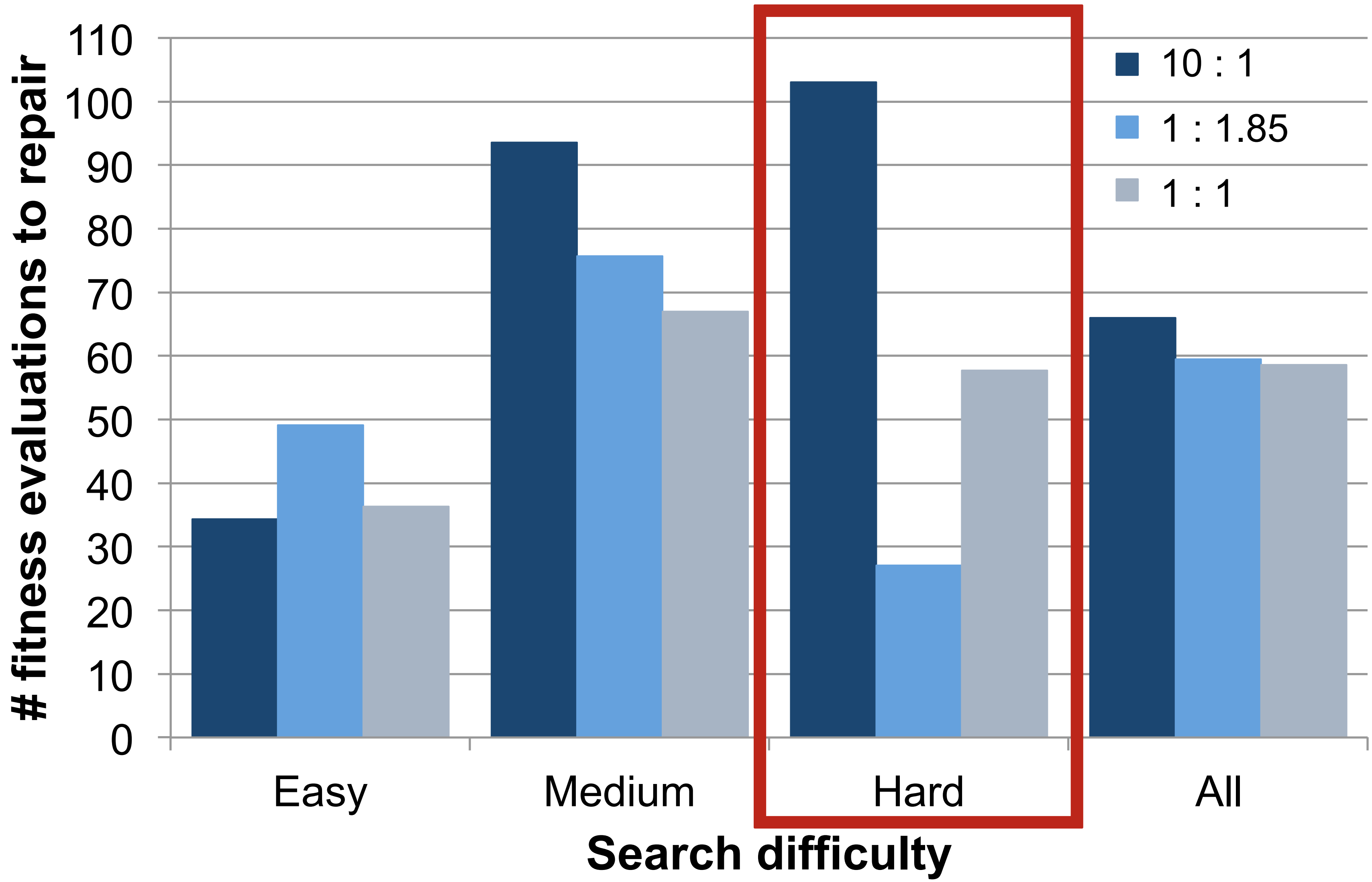
Dataset: the 105 bugs from the earlier dataset.

Rerun that experiment, varying the statement weighting scheme:

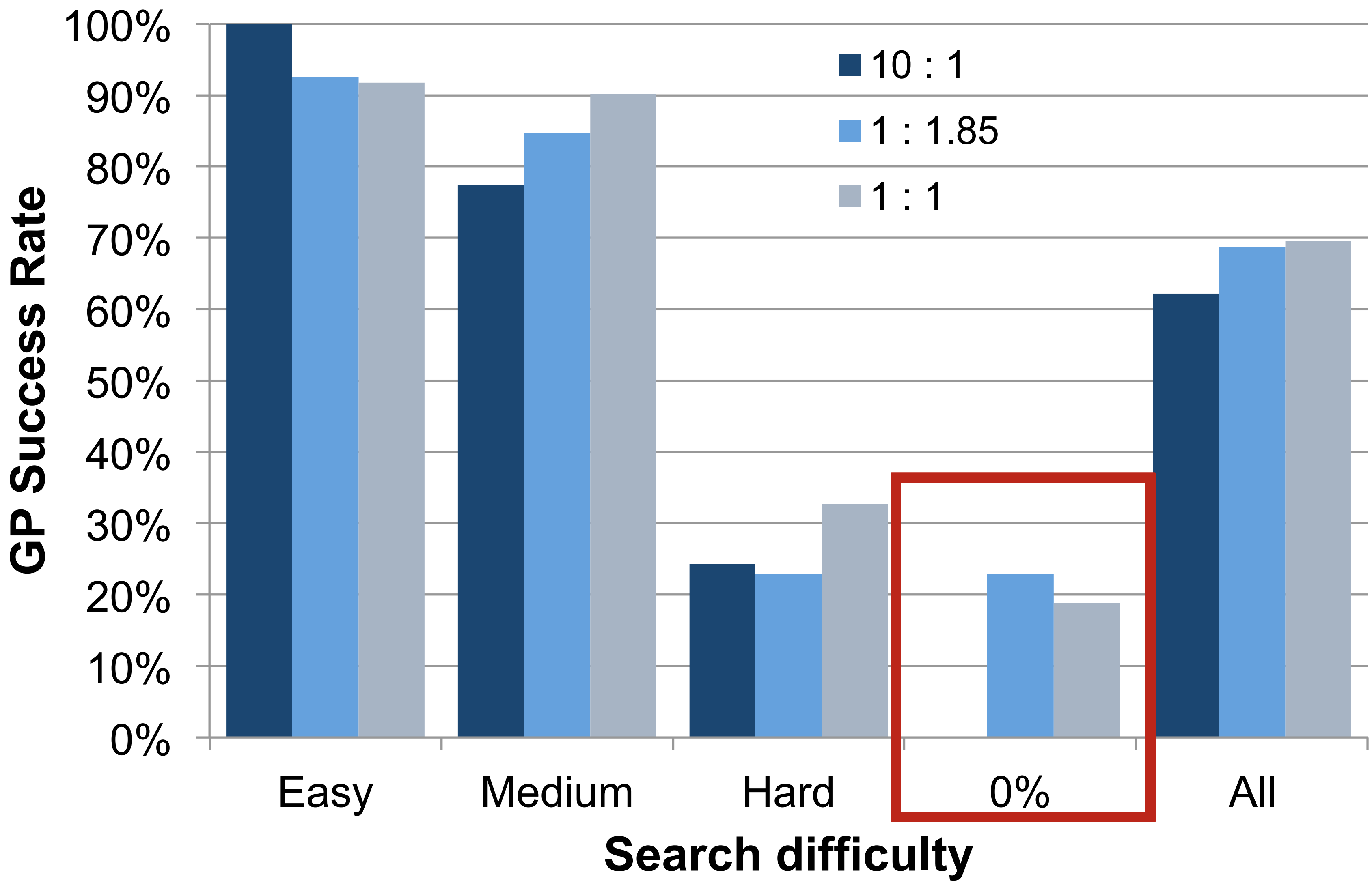
- Default: the original experiment
- Observed: 1 : 1.85
- Uniform: 1 : 1

Metrics: time to repair, success rate.

SEARCH SPACE: REPAIR TIME



SEARCH SPACE: SUCCESS RATE



DISCUSSION

Atypical problems warrant study; some results are counter-intuitive!

Representation and operator choices matter, especially for difficult bugs:

- Repairs an additional 5 bugs: 60 out of 105.
- Reduced time to repair 17 – 43% for difficult bug scenarios.

We have similarly studied fitness function improvements.

MUTATION: HOW

To mutate an individual patch (creating a new one), add a new random edit to it.

- (create new individuals by generating a couple of random edits to make a new patch)

Fault localization guides the mutation process:

1. Instrument program.
2. Record which statements are executed on failing vs. passing test cases.
3. Weight statements accordingly.

SCALABLE: FITNESS

Fitness:

- Subsample test cases.
- Evaluate in parallel.

Random runs:

- Multiple simultaneous runs on different seeds.

CODE BLOAT

Minimization step: try removing each line in the patch, check if the result still passes all tests

Delta Debugging finds a 1-minimal subset of the diff in $O(n^2)$ time

We use a tree-structured diff algorithm (diffX)

- Avoids problems with balanced curly braces, etc.

Takes significantly less time than finding the initial repair repair.

EXAMPLE: PHP BUG #54372

```
1. class test_class {
2.   public function __get($n)
3.     { return $this; %$ }
4.     public function b()
5.      { return; }
6. }
7. global $test3 = new test_class();
8. $test3->a->b();
```

Expected output: nothing

Buggy output: crash on line 8.

EXAMPLE: PHP BUG #54372

`$test3->a->b();`

Note: memory management uses reference counting.

Problem: (in `zend_std_read_property` in `zend_object_handlers.c`)

```
436. object = $test3->a ($this)
```

…

```
449. zval_ptr_dtor(object)
```

If `object` points to `$this` and `$this` is global, its memory is completely freed, which is a problem.

EXAMPLE: PHP BUG #54372

Human :

```
% 449c449,453
< zval_ptr_dtor(&object);
> if (*retval != object)
> { // expected
>  zval_ptr_dtor(&object);
> } else {
>  Z_DELREF_P(object);
> }
```

GenProg :

```
% 448c448,451
> Z_ADDROF_P(object);
> if (PZVAL_IS_REF(object))
> {
>  SEPARATE_ZVAL(&object);
> }
 zval_ptr_dtor(&object)
```

Human: if the result of the get is not the original object (is not self), call the original destructor. Otherwise, just delete the one reference to the object.

GenProg: if the object is a global reference, create a copy of it (deep increment), and then call the destructor.

EXPERIMENTAL SETUP

Apply indicative workloads to vanilla servers.

- Record results and times.

Send attack input.

- Caught by intrusion detection system.

Generate, deploy repair using attack input and regression test cases.

Apply indicative workload to patched server.

Compare requests processed pre- and post-repair.

- Each request must yield exactly the same output (bit-per-bit) in the same time or less!

SCENARIO

Webservers: 138,226 requests, 12,743 distinct IP addresses
php: 15k loc reservation system, 12,375 requests

Long-running servers with an intrusion detection system that generates/deploys repairs for detected anomalies.

- Worst-case: no humans around to vet the repairs!

Workloads: unfiltered requests to the UVA CS webserver.

Webservers with buffer overflows:

- nullhttpd (simple, multithreaded)
- lighttpd (used by Wikimedia, etc.)

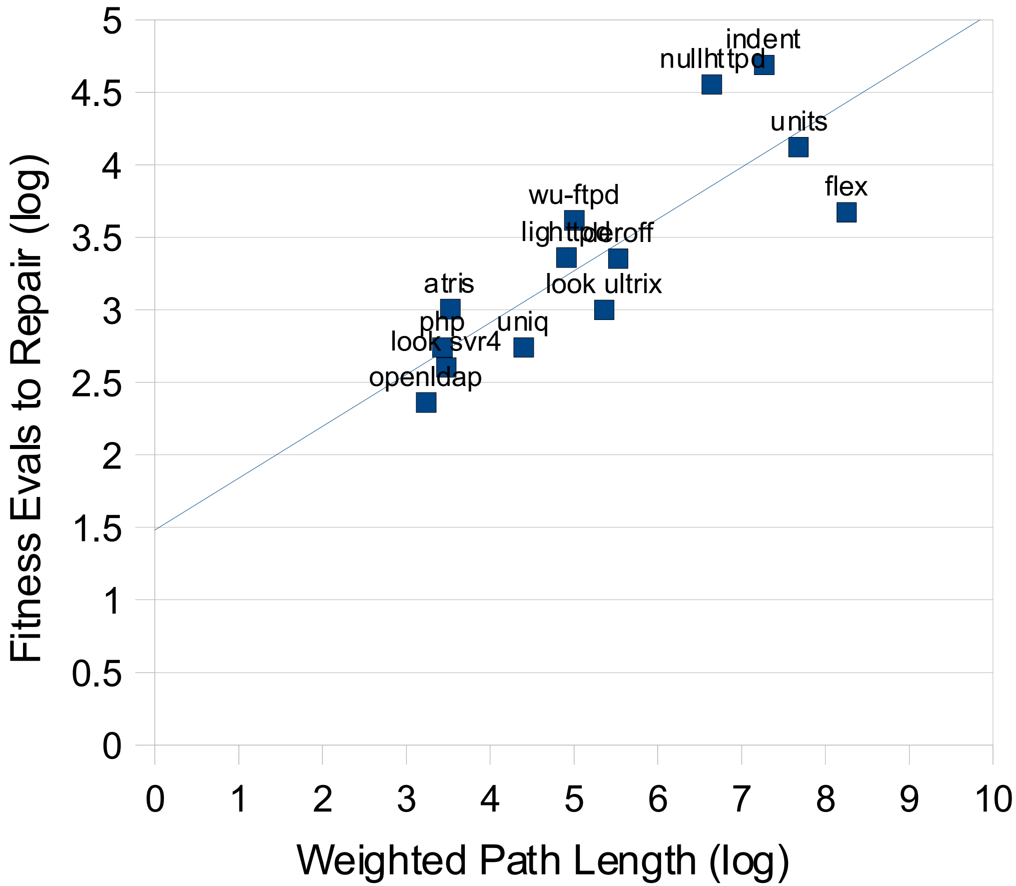
Language interpreter with integer overflow vulnerability:

- php

REPAIR QUALITY RESULTS

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	0.02% ± 0.02%	3 → 3	5 → 0

SEARCH SPACE



Y = 0.8x + 0.02
$R^2 = 0.63$

DEFINITIONS

Bug (colloquialism): a mistake in a program's source code that leads to undesired behavior when the program is executed.

- E.g., a deviation from the functional specification, a security vulnerability, or a service failure of any kind
- Also referred to as a **defect.**

Repair: a set of changes (patch) to program source, intended to fix a bug.