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

PROBLEM: BUGGY SOFTWARE

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

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?
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.
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.
GenProg: automatic program repair using genetic programming.
Four overarching hypotheses.
Empirical evaluations of:
  • Expressive power.
  • Scalability.
Contributions/concluding thoughts.
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**
  
  \[
  \text{ABCDEF} \rightarrow \text{ABADEF}
  \]

- **Crossover**
  
  \[
  \begin{align*}
  \text{ABCDEF} & \quad \text{ABCWVU} \\
  \text{ZYXWVU} & \quad \text{ZYXDEF}
  \end{align*}
  \]
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.
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
EVALUATE FITNESS

INPUT

MUTATE

ACCEPT

DISCARD

OUTPUT

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

(b = 55 - 0)
false
Input:

```
if(a==0)
{
    printf(b);
}

while(b>0)
{
    if(a>b)
    {
        a = a - b;
    }
    else
    {
        b = b - a;
    }

    printf(a);
    return;
}
```
Input: ✓ ✓ ✓ ✗

if(a==0)

printf(b)

while (b>0)

if(a>b)

a = a – b

b = b – a

printf(a)

return

Legend:
- High change probability.
- Low change probability.
- Not changed.
An edit is:

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

```
if(a==0)
{block}
printf(b)
```

```
while (b>0)
{block}
```

```
if(a>b)
{block}
```

```
printf(a)
```

```
return
```

An edit is:

- Insert statement X after statement Y
- Replace statement X with statement Y
- Delete statement X
An edit is:

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

Input:  

```c
if(a == 0)
    return;

while (b > 0)
    if (a > b)
        a = a - b;
    b = b - a;

printf(a);
return;
```

Claire Le Goues
GenProg: automatic program repair using genetic programming.

Four overarching hypotheses.

Empirical evaluations of:

• Expressive power.
• Scalability

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

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Tests</th>
<th>Bugs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fbc</td>
<td>97,000</td>
<td>773</td>
<td>3</td>
<td>Language (legacy)</td>
</tr>
<tr>
<td>gmp</td>
<td>145,000</td>
<td>146</td>
<td>2</td>
<td>Multiple precision math</td>
</tr>
<tr>
<td>gzip</td>
<td>491,000</td>
<td>12</td>
<td>5</td>
<td>Data compression</td>
</tr>
<tr>
<td>libtiff</td>
<td>77,000</td>
<td>78</td>
<td>24</td>
<td>Image manipulation</td>
</tr>
<tr>
<td>lighttpd</td>
<td>62,000</td>
<td>295</td>
<td>9</td>
<td>Web server</td>
</tr>
<tr>
<td>php</td>
<td>1,046,000</td>
<td>8,471</td>
<td>44</td>
<td>Language (web)</td>
</tr>
<tr>
<td>python</td>
<td>407,000</td>
<td>355</td>
<td>11</td>
<td>Language (general)</td>
</tr>
<tr>
<td>wireshark</td>
<td>2,814,000</td>
<td>63</td>
<td>7</td>
<td>Network packet analyzer</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5,139,000</strong></td>
<td><strong>10,193</strong></td>
<td><strong>105</strong></td>
<td></td>
</tr>
</tbody>
</table>
CHALLENGE: GROUNDED COST MEASUREMENTS
READY:
GO!
13 HOURS LATER
## SUCCESS/COST

<table>
<thead>
<tr>
<th>Program</th>
<th>Defects Repaired</th>
<th>Cost per non-repair</th>
<th>Cost per repair</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Hours</td>
<td>US$</td>
</tr>
<tr>
<td>fbc</td>
<td>1/3</td>
<td>8.52</td>
<td>5.56</td>
</tr>
<tr>
<td>gmp</td>
<td>1/2</td>
<td>9.93</td>
<td>6.61</td>
</tr>
<tr>
<td>gzip</td>
<td>1/5</td>
<td>5.11</td>
<td>3.04</td>
</tr>
<tr>
<td>libtiff</td>
<td>17/24</td>
<td>7.81</td>
<td>5.04</td>
</tr>
<tr>
<td>lighttpd</td>
<td>5/9</td>
<td>10.79</td>
<td>7.25</td>
</tr>
<tr>
<td>php</td>
<td>28/44</td>
<td>13.00</td>
<td>8.80</td>
</tr>
<tr>
<td>python</td>
<td>1/11</td>
<td>13.00</td>
<td>8.80</td>
</tr>
<tr>
<td>wireshark</td>
<td>1/7</td>
<td>13.00</td>
<td>8.80</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>55/105</strong></td>
<td><strong>11.22h</strong></td>
<td><strong>1.60h</strong></td>
</tr>
</tbody>
</table>

$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).
Without defect- or program- specific information, GenProg can:

1. repair at least 5 different defect types, and can repair defects in at 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.
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.
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.

http://www.clairelegoues.com
OUTLINE

GenProg: automatic program repair using genetic programming.
Four overarching hypotheses.
Empirical evaluations of:
  • Expressive power.
  • Scalability.

Contributions/concluding thoughts.
**PUBLICATIONS: GENPROG**


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*)


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


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: 

```c
if(a==0) {block} printf(b) {block} {block} {block} printf(a) return

while (b>0) {block} if(a>b) {block} a = a – b {block} b = b – a
```

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

# fitness evaluations to repair

Easy
Medium
Hard
All

Search difficulty

10 : 1
1 : 1.85
1 : 1

http://www.clairelegoues.com
SEARCH SPACE: SUCCESS RATE

GP Success Rate

Search difficulty

Easy | Medium | Hard | All

- **10 : 1**
- **1 : 1.85**
- **1 : 1**

GP Success Rate:

- **Easy**: 100%
- **Medium**: ~90%
- **Hard**: ~70%
- **All**: ~60%

Note: The search difficulty levels are categorized as Easy, Medium, Hard, and All.
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.
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.
$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);
> }

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

Long-running servers with an intrusion detection system that generates/deploy 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

<table>
<thead>
<tr>
<th>Program</th>
<th>Post-patch requests lost</th>
<th>Fuzz Tests Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>General</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>0.00 % ± 0.25%</td>
<td>0 → 0</td>
</tr>
<tr>
<td>lighttpd</td>
<td>0.03% ± 1.53%</td>
<td>1410 → 1410</td>
</tr>
<tr>
<td>php</td>
<td>0.02% ± 0.02%</td>
<td>3 → 3</td>
</tr>
</tbody>
</table>
SEARCH SPACE

![Graph showing the relationship between Fitness Evals to Repair (log) and Weighted Path Length (log). The graph displays a linear trend with the equation $Y = 0.8x + 0.02$ and $R^2 = 0.63$.](http://www.clairelegoues.com)
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.