Automated Program Repair
Motivation

• **Software maintenance is expensive**
  - Up to 90% of the cost of software
    • [Seacord]
  - Up to $70 Billion per year in US
    • [Jorgensen, Sutherland]
  - Bug repair is the majority of maintenance
    • [Erlikh, Ramamoothy, Williamson]

• **Bugs are ubiquitous**
  - Outstanding bugs exceed the resources available to address them
    • [Anvik, Liblit]
Fixing Bugs Early Would Help

• The cost of a defect is $25 during the coding phase, $100 during the build phase, $450 during the QA/testing phase, and $16,000 once released as a product.
  - Leigh Williamson, IBM Distinguished Engineer

• On average, a software vendor loses 0.63% of its market value on the day of any vulnerability announcement.
  - [Telang and Wattal]
Fixing Many Bugs Would Help

• “Everyday, almost 300 bugs appear … far too much for only the Mozilla programmers to handle.”

- Mozilla Developer [Anvik]
Central Claim

- We can **automatically repair** certain classes of bugs in off-the-shelf, unannotated legacy software.
  - We can automatically repair many types of high-impact security bugs.
  - We can automatically repair many “general software engineering” bugs.
Insights

- Given a buggy program, search the space of **nearby programs** until a valid repair is found.
- Use **test cases** to encode the program specification and the defect.
- Find nearby programs by mimicking human edits and leveraging existing **human insight**.
- **Reduce the search space** by restricting attention to areas likely to contain the bug.
- **Genetic programming** guides the search, tolerating noise and admitting parallelism.
Correctness

- **Testing** gives confidence that a program implementation adheres to its specification (as refined from its requirements).

- Loosely, in the **Oracle-Comparator** model, a test case includes:
  - A Test Input
  - An Oracle Answer (expected answer)
  - A Comparator (is an answer close enough?)

- The **current bug** is demonstrated by a test case that currently fails
Fault Localization

- Even in large programs, not every component is equally likely to contribute to a given fault
- Given a program, a bug, and a test suite, fault localization produces a mapping from program components to weights
  - “High weight” means “likely related to the bug”
- Many techniques exist
  - [Renieris and Reiss, Jones and Harrold]
- Loosely: print statement debugging, note all statements only visited on bug test case
Fix Localization

• There are a large number of ways to change a given statement
  - e.g., deleting it, adding more code to it, replacing it with something completely different, etc.
• When inserting, we leverage human insight by using code from elsewhere in the program
  - Program is probably correct elsewhere [Engler]
• Simple Operations: Insert, Replace, Delete
• Fix localization is the set of things to insert
Genetic Algorithm Overview

- **Genetic Programming** is a search heuristic
  - Based on a computational analog of biology
  - **Represent** solutions to the problem (genotypes)
  - Uses a **population** of variant solutions
  - Applies simple **mutations**
  - Evaluate the **fitness** of a variant (phenotype)
  - High-fitness variants survive and mate (**crossover**)
  - Repeat until a solution is found
Patch Representation

• A **patch** is represented as a **list of edits**
• Edits are **statement-level** operations
  - Example: \(< \text{Delete}(5), \text{Insert}(33,2) >\)
• **Mutation**: add to edit list
  - Choose a statement \(X\) from Fault Localization
  - Choose: \(\text{Delete } X\) or \(\text{Insert}(X, \text{FixLocalization}(X))\) or \(\text{Replace}(X, \text{FixLocalization}(X))\)
• **Crossover**: random sublist of parent lists
• **Fitness**: print out patched program, run tests
Overall Repair Algorithm

- Input: Program P, Test Suite T
- Loc := Compute Fault and Fix Localizations
- Population := $n$ random mutants of P
- Repeat Until Solution Found:
  - HighFit := Select(Population, $n$, T)
  - Offspring := Crossover(HighFit)
  - Population := Mutate(HighFit + Offspring, Loc)
Example: GCD

```c
void print_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;
}
```

Bug: when a==0 and b>0, it loops forever!
Example: Representation

```c
if (a == 0)
    printf(... a)
else
    while (b != 0)
        printf(... b)
        a = a - b
        b = b - a
    return

Example: Representation
```
Example: Fault Location

Fault Localization:

(printf ...b)

```
if (a==0)
while (b != 0)
printf(... a)
return

if (isLeapYear)
if (a > b)
{

}

printf(... b)

a = a - b

if (a > b)
{

}

b = b - a
```
Example: Mutation (1/2)

```c
if (a == 0)
    while (b != 0)
        if (isLeapYear)
            if (a > b)
                { block }
                return
                { block }
        printf(... a)
        b = b - a
        { block }
printf(... b)
    a = a - b
    { block }
```

Example: Mutation (2/2)

```c
if (a==0)
    while (b != 0)
        printf(... a)
        if (isLeapYear)
            if (a > b)
                { block }
                { block }
        return
        printf(... b)
        a = a - b
    b = b - a
return
```

```c
{ block }
while (b != 0)
    printf(... a)
    if (isLeapYear)
        if (a > b)
            { block }
            { block }
        return
        printf(... b)
        a = a - b
    b = b - a
return
```
Example: Final Repair

```c
if (a==0)
    { block }

while (b != 0)
    { block }

if (isLeapYear)
    { block }

if (a > b)
    { block }

printf(... a)
return

a = a - b
{ block }

b = b - a
{ block }
```

Example: Final Repair
## Initial Program Repair Results

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Program Description</th>
<th>Defect Repaired</th>
<th>Time</th>
<th># Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>22</td>
<td>Euclid’s algorithm</td>
<td>infinite loop</td>
<td>276 s</td>
<td>909</td>
</tr>
<tr>
<td>zune</td>
<td>28</td>
<td>MS Zune example</td>
<td>infinite loop</td>
<td>78 s</td>
<td>460</td>
</tr>
<tr>
<td>uniq</td>
<td>1146</td>
<td>text processing</td>
<td>segfault</td>
<td>32 s</td>
<td>139</td>
</tr>
<tr>
<td>look-ultrix</td>
<td>1169</td>
<td>dictionary lookup</td>
<td>segfault</td>
<td>42 s</td>
<td>120</td>
</tr>
<tr>
<td>look-svr4</td>
<td>1363</td>
<td>dictionary lookup</td>
<td>infinite loop</td>
<td>51 s</td>
<td>42</td>
</tr>
<tr>
<td>units</td>
<td>1504</td>
<td>metric conversion</td>
<td>segfault</td>
<td>1528 s</td>
<td>9014</td>
</tr>
<tr>
<td>deroff</td>
<td>2236</td>
<td>document processing</td>
<td>segfault</td>
<td>132 s</td>
<td>227</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>5575</td>
<td>webserver</td>
<td>heap buff. overflow</td>
<td>1394 s</td>
<td>1800</td>
</tr>
<tr>
<td>openldap io</td>
<td>6159</td>
<td>authentication server</td>
<td>non-overflow D.O.S.</td>
<td>665 s</td>
<td>8</td>
</tr>
<tr>
<td>leukocyte</td>
<td>6718</td>
<td>computational biology</td>
<td>segfault</td>
<td>544 s</td>
<td>12</td>
</tr>
<tr>
<td>indent</td>
<td>9906</td>
<td>source code processing</td>
<td>infinite loop</td>
<td>7614 s</td>
<td>13628</td>
</tr>
<tr>
<td>python complexobject</td>
<td>11227</td>
<td>web app. interpreter</td>
<td>overflow error</td>
<td>1393 s</td>
<td>23222</td>
</tr>
<tr>
<td>lighttpd fastcgi</td>
<td>13984</td>
<td>webserver CGI module</td>
<td>heap buff. overflow</td>
<td>395 s</td>
<td>52</td>
</tr>
<tr>
<td>imagemagick fx</td>
<td>16851</td>
<td>image processing</td>
<td>incorrect image output</td>
<td>1240 s</td>
<td>131</td>
</tr>
<tr>
<td>flex</td>
<td>18775</td>
<td>scanner generator</td>
<td>segfault</td>
<td>4660 s</td>
<td>9560</td>
</tr>
<tr>
<td>atris</td>
<td>21553</td>
<td>graphical game</td>
<td>stack buff. overflow</td>
<td>84 s</td>
<td>285</td>
</tr>
<tr>
<td>php string</td>
<td>26044</td>
<td>web app. interpreter</td>
<td>integer overflow</td>
<td>2678 s</td>
<td>18081</td>
</tr>
<tr>
<td>wu-ftp</td>
<td>35109</td>
<td>FTP server</td>
<td>format string vuln.</td>
<td>5397 s</td>
<td>74</td>
</tr>
<tr>
<td>total</td>
<td>179369</td>
<td>(18 distinct programs)</td>
<td>(18 defects in 8 classes)</td>
<td>1567 s</td>
<td>4320</td>
</tr>
</tbody>
</table>
Research Questions

- Your results may not generalize to my situation because the programs and bugs I deal with ...
  - Have huge, long-running test suites (scalability)
  - Require high quality repairs (expressive power, overfitting)
  - Do not have source code (input assumptions, expressive power)
  - Are large and indicative (overfitting)
  - Require an economic business case
Large Test Suites and Fitness

- Thus far, test cases both
  - Validate a variant as a final repair
  - *and also* Determine which variants are retained
- What if there are 100+ test cases?
- Idea:
  - Use **all** tests to validate final repairs
  - Sample **some** tests to decide which variants are retained into the next iteration
Large Test Suites

- Use test suite selection techniques
  - Or random sampling
- Sampling reduces repair time by 81%
  - First evaluation on 10 bugs, 1200+ test cases
- "leukocyte was repaired in 6 minutes instead of over 90 minutes ... and imagemagick was repaired in 3 minutes instead of 36"
- Helps performance and correctness
Repair Quality

• Produced repairs pass *all* tests + minimization

• Objective measures
  - Retains required functionality
  - Does not introduce new bugs
  - Is not a “fragile memorization” of buggy input

• Subjective measures
  - Code review
  - Assurance argument
Repair Quality Benchmarks

• Two webservers with buffer overflows
  - nullhttpd, lighttpd
  - 138,226 held-out requests from 12,743 clients

• One web app language interpreter
  - php (integer overflow vulnerability)
  - 15 kLOC secure reservation system web app
  - 12,375 requests (held out)
Repair Quality Experiments

- Conduct repairs
  - Using ~10 test cases

- Evaluate repairs
  - Using all held-out test cases
  - Need same result bit-for-bit in same time or less
  - Also evaluate using held-out fuzz testing
## Repair Quality Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Requests Lost Making Repair</th>
<th>Requests Lost to Repair Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nullhttpd</strong></td>
<td>2.38% ± 0.83%</td>
<td>0.00% ± 0.25%</td>
</tr>
<tr>
<td><strong>lighttpd</strong></td>
<td>0.98% ± 0.11%</td>
<td>0.03% ± 1.53%</td>
</tr>
<tr>
<td><strong>php</strong></td>
<td>0.12% ± 0.00%</td>
<td>0.02% ± 0.02%</td>
</tr>
</tbody>
</table>
# Repair Quality Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Requests Lost Making Repair</th>
<th>Requests Lost to Repair Quality</th>
<th>General Fuzz Tests Failed</th>
<th>Exploit Fuzz Tests Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>nullhttpd</td>
<td>2.38% ± 0.83%</td>
<td>0.00% ± 0.25%</td>
<td>0 → 0</td>
<td>10 → 0</td>
</tr>
<tr>
<td>lighttpd</td>
<td>0.98% ± 0.11%</td>
<td>0.03% ± 1.53%</td>
<td>1410 → 1410</td>
<td>9 → 0</td>
</tr>
<tr>
<td>php</td>
<td>0.12% ± 0.00%</td>
<td>0.02% ± 0.02%</td>
<td>3 → 3</td>
<td>5 → 0</td>
</tr>
</tbody>
</table>
# Repair Quality Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Requests Lost Making Repair</th>
<th>Requests Lost to Repair Quality</th>
<th>General Fuzz Tests Failed</th>
<th>Exploit Fuzz Tests Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>nullhttpd</td>
<td>2.38% ± 0.83%</td>
<td>0.00% ± 0.25%</td>
<td>0 → 0</td>
<td>10 → 0</td>
</tr>
<tr>
<td>lighttpd</td>
<td>0.98% ± 0.11%</td>
<td>0.03% ± 1.53%</td>
<td>1410 → 1410</td>
<td>9 → 0</td>
</tr>
<tr>
<td>php</td>
<td>0.12% ± 0.00%</td>
<td>0.02% ± 0.02%</td>
<td>3 → 3</td>
<td>5 → 0</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>False Pos #1</td>
<td>7.83% ± 0.49%</td>
<td>0 → 0</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00% ± 2.22%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nullhttpd</td>
<td>False Pos #2</td>
<td>3.04% ± 0.29%</td>
<td>0 → 0</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.57% ± 3.91%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nullhttpd</td>
<td>False Pos #3</td>
<td>6.92% ± 0.09%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(no repair!)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cherry Picking

• Thus far, all repaired programs were chosen arbitrarily by the authors
  - That is, the benchmark set may not be a representative sample of real-world programs

• Let's address that and program size concerns in one fell swoop with a systematic study
  - Use version control, take entire ranges of versions
  - Find all reproducible bugs within that range
  - Must be important enough for devs to fix and test
  - Try to repair them all
### “Many Bugs” Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Tests</th>
<th>Defects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fbc</td>
<td>97,000</td>
<td>773</td>
<td>3</td>
<td>legacy programming</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>web programming</td>
</tr>
<tr>
<td>python</td>
<td>407,000</td>
<td>355</td>
<td>11</td>
<td>general programming</td>
</tr>
<tr>
<td>wireshark</td>
<td>2,814,000</td>
<td>63</td>
<td>7</td>
<td>network packet analyzer</td>
</tr>
</tbody>
</table>

**total**     | **5,139,000** | **10,193** | **105** |

**Table I**

**Subject C Programs, Test Suites and Historical Defects**: Tests were taken from the most recent version available in May, 2011; Defects are defined as test case failures fixed by developers in previous versions.
**“Many Bugs” Results**

<table>
<thead>
<tr>
<th>Program</th>
<th>Defects Repaired</th>
<th>Avg. Cost per Non-Repair Hours</th>
<th>Avg. Cost Per Repair Hours</th>
<th>US$</th>
</tr>
</thead>
<tbody>
<tr>
<td>fbc</td>
<td>1 / 3</td>
<td>8.52</td>
<td>5.56</td>
<td>4.08</td>
</tr>
<tr>
<td>gmp</td>
<td>1 / 2</td>
<td>9.93</td>
<td>6.61</td>
<td>0.44</td>
</tr>
<tr>
<td>gzip</td>
<td>1 / 5</td>
<td>5.11</td>
<td>3.04</td>
<td>0.30</td>
</tr>
<tr>
<td>libtiff</td>
<td>17 / 24</td>
<td>7.81</td>
<td>5.04</td>
<td>0.04</td>
</tr>
<tr>
<td>lighttpd</td>
<td>5 / 9</td>
<td>10.79</td>
<td>7.25</td>
<td>0.25</td>
</tr>
<tr>
<td>php</td>
<td>28 / 44</td>
<td>13.00</td>
<td>8.80</td>
<td>0.62</td>
</tr>
<tr>
<td>python</td>
<td>1 / 11</td>
<td>13.00</td>
<td>8.80</td>
<td>0.16</td>
</tr>
<tr>
<td>wireshark</td>
<td>1 / 7</td>
<td>13.00</td>
<td>8.80</td>
<td>0.17</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>
<td></td>
</tr>
</tbody>
</table>

Results can be reproduced for $403; successful repairs cost $7.32 on average.
Repair Quality and Bug Bounties

- All patches pass all available tests (e.g., 8471)
- Humans can still inspect patches [Weimer]
  - If this cuts developer time in half, the economic argument still holds ($25 vs. $25/2 + $7.32)
- One commercial developer paid $1,625 for
  - 63 repairs to “harmless” severity bugs
  - 11 repairs to “minor” severity bugs, 1 repair to “major”
  - 125 “false positive” candidate patches
  - Avg: $21 per non-trivial repair, one every 40 hours
- “Worth the money? Every penny.”
- [Link](http://www.daemonology.net/blog/2011-08-26-1265-dollars-of-tarsnap-bugs.html)
Project Scope

- Two workshop paper (FoSER, SBST)
- Two journal paper (TSE, Comm. ACM)
- Four conference papers (ASE, ASE, GECCO, ICSE)
- Three best paper awards (ICSE, GECCO, SBST)
- Some associated papers (SIGGRAPH, etc.)
- IFIP TC2 Manfred Paul Award (1024 Euros)
- Gold Human Competitive Award ($5000)
- Four grant proposal awards
- … since 2009.
Discussion Section

• Possible stories:
• Boring grant meeting
• Fail Early
• Bug finding fatigue
• Early experimental result problems
• Don't be all things to all people
• Dagstuhl Seminar
• Tom Ball predictions
Homework

• Project Writeup Due!
  - Need help? Stop by my office or send email.

• Project Presentation Due