Automatically Finding Patches Using Genetic Programming

Westley Weimer, Stephanie Forrest, Claire Le Goues, ThanVu Nguyen, Ethan Fast, Briana Satchell, Eric Schulte
Motivation

- Software Quality remains a key problem
  - Over one half of 1 percent of US GDP each year [NIST02]
  - The cost of fixing a defect increases ($25 - $16k) [IBM08]
  - Even security-critical bugs take 28 days (avg) [Symantec06]
  - Despite bug detection and test suites
  - Programs ship with known bugs

- How can we reduce debugging costs?
  - Bug reports accompanied by patches are addressed more rapidly

- Thus: Automated Patch Generation
Main Claim

• We can automatically and efficiently repair certain classes of bugs in off-the-shelf, unannotated legacy programs.

• Basic idea: Biased search through the space of certain nearby programs until you find a variant that repairs the problem. Key insights:
  • Use existing **test cases** to evaluate variants.
  • Search by perturbing parts of the program **likely** to contain the error.

ICSE'09 Best Paper, GECCO'09 Best Paper, SBST'09 Best Short Paper, 2009 IFIP TC2 Manfred Paul Award, 2009 Gold Human-Competitive Award
Repair Process Preview

• Input:
  • The program source code
  • System/regression tests passed by the program
  • A test case failed by the program (= the bug)

• Genetic Programming Work:
  • Create variants of the program
  • Run them on the test cases
  • Repeat, retaining and combining variants

• Output:
  • New program source code that passes all tests
  • or “no solution found in time”
This Talk

• Fixing Real Bugs In Real Programs
  • Representation and Operations
• The Quality of Automated Repairs
  • Self-Healing Systems and Metrics
• Test Suite Selection
  • Success and Explanations
• Open Questions in Automated Repair
Genetic Programming

• Genetic programming is the application of evolutionary or genetic algorithms to program source code.
  • Representing a population of program variants
  • Mutation and crossover operations
  • Fitness function
• GP serves as a search heuristic
  • Others (random search, brute force, etc.) also work
• Similar in ways to search-based software engineering:
  • Regression tests to guide the search
Useful Insight #1 - Where To Fix

• In a large program, not every line is equally likely to contribute to the bug.

• **Fault localization**: given a bug, find its location in the program source.

• Insight: since we have the test cases, run them and collect coverage information.

• The bug is *more likely to be found on lines visited when running the failed test case.*

• The bug is less likely to be found on lines visited when running the passed test cases.
Useful Insight #2 - How To Fix

- Developers often use statements or lines of code as atomic units representing actions
- Insight: operate on statements or lines
  - Not on assembly ops or expressions
  - Factor of 10 reduction in search space each time
- Insight: do not invent new code
  - Instead, copy and modify existing statements
- We assume the program “contains the seeds of its own repair”
  - e.g., has another null check somewhere
Fault Localization Formalism

- We define a weighted path to be a list of \langle\text{statement}, \text{weight}\rangle pairs.

- We use this weighted path:
  - The statements are those visited during the failed test case.
  - The weight for a statement $S$ is
    - High (1.0) if $S$ is not visited on a passed test
    - Low (0.0-0.1) if $S$ is also visited on a passed test
  - (Other weight sources are possible: e.g., Cooperative Bug Isolation or Daikon predicates)
Genetic Programming for Program Repair: Mutation

• Population of Variants:
  • Each variant is an <AST, weighted path> pair

• Mutation:
  • To mutate a variant \( V = <\text{AST}_V, \text{wp}_V> \), choose a statement \( S \) from \( \text{wp}_V \) biased by the weights
  • Replacement. Replace \( S \) with \( S_1 \)
  • Insertion. Replace \( S \) with \{ S_2 ; S \}
  • Deletion. Replace \( S \) with \{ \}
  • Choose \( S_1 \) and \( S_2 \) from the entire AST
  • All variants retain weighted path length
Genetic Programming for Program Repair: Fitness

- Compile a variant
  - If it fails to compile, Fitness = 0
  - Otherwise, run it on the test cases
  - Fitness = number of test cases passed
  - Weighted: passing the bug test case is worth more

- Selection and Crossover
  - Higher fitness variants are retained and combined into the next generation
  - Tournament selection and one-point crossover

- Repeat until a solution is found
Example: GCD

```c
/* requires: a >= 0, b >= 0 */
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: Abstract Syntax Tree

if (a==0)
{
    { block }
    printf(... a)
}

while (b != 0)
{
    { block }
    printf(... b)
    a = a - b

    { block }
    b = b - a

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

return

Example: Weighted Path (1/3)

Nodes visited on Negative test case (a=0, b=55): (printf ... b)
Example: Weighted Path (2/3)

Nodes visited on Positive test case (a=1071, b=1029):
- if (a==0)
- while (b != 0)
- printf(... a)
- return

b = b - a
a = a - b

Nodes visited on Negative test case (a=0, b=55):
- (printf ...b)
if (a==0)
while (b != 0)
printf(... a)
return

if (isLeapYear)
if (a > b)
printf(... b)
a = a - b
b = b - a

Example: Weighted Path (3/3)

Weighted Path:
Example: Mutation (1/2)

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

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

Mutation Source: Anywhere in AST
Mutation Destination: Weighted Path
if (a==0)
while (b != 0)
printf(... a)

if (isLeapYear)
if (a > b)

return

{ block }

printf(... b)

return

{ block }

a = a - b

{ block }

b = b - a

Mutation Source: Anywhere in AST
Mutation Destination: Weighted Path
if (a==0)
{ block }

if (isLeapYear)
{ block }

while (b != 0)
{ block }

printf(... a)

return

if (a > b)
{ block }

if (a == 0)
{ block }

printf(... b)

return

a = a - b

b = b - a

Example: Final Repair
Minimize The Repair

- Repair Patch is a diff between orig and variant
- Mutations may add unneeded statements
  - (e.g., dead code, redundant computation)
- In essence: try removing each line in the diff and check if the result still passes all tests
- Delta Debugging finds a 1-minimal subset of the diff in $O(n^2)$ time
  - Removing any single line causes a test to fail
- We use a tree-structured diff algorithm (diffX)
  - Avoids problems with balanced curly braces, etc.
## Experimental Results: 20 Repairs

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of Code</th>
<th>Program Description</th>
<th>Defect Repaired</th>
<th>Time</th>
<th># of Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Repaired</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcd</td>
<td>22</td>
<td>-</td>
<td>Euclid’s algorithm</td>
<td>infinite loop</td>
<td>276 s</td>
</tr>
<tr>
<td>zune</td>
<td>28</td>
<td>-</td>
<td>MS Zune example</td>
<td>infinite loop</td>
<td>78 s</td>
</tr>
<tr>
<td>tiff</td>
<td>1084</td>
<td>84067</td>
<td>image processing</td>
<td>segfault</td>
<td>45 s</td>
</tr>
<tr>
<td>uniq</td>
<td>1146</td>
<td>-</td>
<td>text processing</td>
<td>segfault</td>
<td>32 s</td>
</tr>
<tr>
<td>look-ultrix</td>
<td>1169</td>
<td>-</td>
<td>dictionary lookup</td>
<td>segfault</td>
<td>42 s</td>
</tr>
<tr>
<td>look-svr4</td>
<td>1363</td>
<td>-</td>
<td>dictionary lookup</td>
<td>infinite loop</td>
<td>51 s</td>
</tr>
<tr>
<td>units</td>
<td>1504</td>
<td>-</td>
<td>metric conversion</td>
<td>segfault</td>
<td>1528 s</td>
</tr>
<tr>
<td>deroff</td>
<td>2236</td>
<td>-</td>
<td>document processing</td>
<td>segfault</td>
<td>132 s</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>5575</td>
<td>-</td>
<td>threaded webserver</td>
<td>heap buff. overflow †</td>
<td>1394 s</td>
</tr>
<tr>
<td>openldap</td>
<td>6159</td>
<td>308777</td>
<td>authentication server</td>
<td>non-overflow D.O.S. †</td>
<td>665 s</td>
</tr>
<tr>
<td>ccrypt</td>
<td>6413</td>
<td>7515</td>
<td>Rijndael cryptography</td>
<td>segfault</td>
<td>330 s</td>
</tr>
<tr>
<td>leukocyte</td>
<td>6718</td>
<td>-</td>
<td>computational biology</td>
<td>segfault</td>
<td>544 s</td>
</tr>
<tr>
<td>indent</td>
<td>9906</td>
<td>-</td>
<td>pretty printing</td>
<td>infinite loop</td>
<td>7614 s</td>
</tr>
<tr>
<td>python</td>
<td>11227</td>
<td>496527</td>
<td>web app. interpreter</td>
<td>overflow error</td>
<td>1393 s</td>
</tr>
<tr>
<td>lighttpd</td>
<td>13984</td>
<td>51895</td>
<td>webserver CGI</td>
<td>heap buff. overflow †</td>
<td>395 s</td>
</tr>
<tr>
<td>imagemagick</td>
<td>16851</td>
<td>450416</td>
<td>image processing</td>
<td>incorrect image output</td>
<td>1240 s</td>
</tr>
<tr>
<td>flex</td>
<td>18775</td>
<td>-</td>
<td>scanner generator</td>
<td>segfault</td>
<td>4660 s</td>
</tr>
<tr>
<td>atris</td>
<td>21553</td>
<td>-</td>
<td>graphical game</td>
<td>stack buff. overflow †</td>
<td>84 s</td>
</tr>
<tr>
<td>php</td>
<td>26044</td>
<td>797749</td>
<td>web app. interpreter</td>
<td>integer overflow †</td>
<td>2678 s</td>
</tr>
<tr>
<td>wu-ftpd</td>
<td>35109</td>
<td>-</td>
<td>FTP server</td>
<td>format string vuln. †</td>
<td>5397 s</td>
</tr>
<tr>
<td>total or avg</td>
<td>186,866</td>
<td>2,302,050</td>
<td>(20 distinct programs)</td>
<td>(20 defects in 8 classes)</td>
<td>1428 s</td>
</tr>
</tbody>
</table>

Many defects from “black hat” lists; avg minimization time: 12 seconds.
The Story Thus Far

• How does the approach work?
  • Create programs in a **restricted** search space

• Can it produce repairs?
  • Yes, for many types of programs and defects

• Can I afford to use it?
  • Are the repairs trustworthy?
  • Does the approach scale?
Repair Quality

- Repairs are typically not what a human would have done
  - Example: our technique adds bounds checks to one particular network read, rather than refactoring to use a safe abstract string class in multiple places

- Recall: any proposed repair must pass all regression test cases
  - When POST test is omitted from nullhttpd, the generated repair eliminates POST functionality
  - Tests ensure we do not sacrifice functionality
  - Minimization prevents gratuitous deletions
  - Adding more tests helps rather than hurting
Repair Quality Experiment

- A high-quality repair …
  - Retains required functionality
  - Does not introduce new bugs
  - Is not a “fragile memorization” of the buggy input
  - Works as part of an entire system
- If humans are present, they can inspect it
- Let's consider a human-free situation, such as:
  - A long-running server with an anomaly intrusion detection system that will generate and deploy repairs for all detected anomalies.
Repair Quality Benchmarks

• Two webservers with buffer overflows
  • nullhttpd (simple, multithreaded)
  • lighttpd (used by Wikimedia, etc.)
  • 138,226 requests from 12,743 distinct client IP addresses (held out; one day of data)

• One web application language interpreter
  • php (integer overflow vulnerability)
  • 15kloc secure reservation system web app
  • 12,375 requests (held out; one day of data)
Repair Quality Experimental Setup

• Apply indicative workloads to vanilla servers
  • Record result contents and times
• Send attack input
  • Caught by anomaly intrusion detection system
• Generate and deploy repair
  • Using attack input and six test cases
• Apply indicative workload to patched server
  • Each request must yield exactly the same output (bit-per-bit) in the same time or less!
## Closed-Loop Outcomes

<table>
<thead>
<tr>
<th>Case</th>
<th>Anomaly Detected?</th>
<th>Successful Repair?</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True Neg.</td>
<td>N/A</td>
<td>Legitimate request handled correctly; no repair</td>
</tr>
<tr>
<td>2</td>
<td>False Neg.</td>
<td>N/A</td>
<td>Attack succeeds; Repair not attempted</td>
</tr>
<tr>
<td>3</td>
<td>True Pos.</td>
<td>Yes</td>
<td>Attack stopped and bug fixed. Later requests could be lost if repair breaks functionality</td>
</tr>
<tr>
<td>4</td>
<td>True Pos.</td>
<td>No</td>
<td>Attack detected; bug not repaired</td>
</tr>
<tr>
<td>5</td>
<td>False Pos.</td>
<td>No</td>
<td>Legitimate request dropped; repair not found</td>
</tr>
<tr>
<td>6</td>
<td>False Pos.</td>
<td>Yes</td>
<td>Legitimate request dropped; later requests may be harmed if “repair” is incorrect</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>
</tr>
</thead>
<tbody>
<tr>
<td>nullhttpd</td>
<td>2.38% ± 0.83%</td>
<td>0.00% ± 0.25%</td>
</tr>
<tr>
<td>lighttpd</td>
<td>0.98% ± 0.11%</td>
<td>0.03% ± 1.53%</td>
</tr>
<tr>
<td>php</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.83% ± 0.49%</td>
<td>0.00% ± 2.22%</td>
<td>0 → 0</td>
<td>n/a</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>False Pos #2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.04% ± 0.29%</td>
<td>0.57% ± 3.91%</td>
<td>0 → 0</td>
<td>n/a</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>False Pos #3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.92% ± 0.09% (no repair!)</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Repair Quality Conclusions

• It is possible to create repairs that
  • Retain required functionality
  • Do not introduce new bugs
  • Are not a fragile memorizations
  • Work as part of an entire system

• This reduces to the problem of supplying a good test suite

• For webservers and php, a few indicative end-to-end system tests suffice
  • But in general we may need more test cases …
Algorithm Scalability

- We want to quickly produce high-quality repairs for complicated defects in large programs with arbitrary test suites
- GP is a heuristic search strategy
- Worst-case run time is effectively
  - Size of search space multiplied by
  - Time to evaluate a point in the search space
- Examine fitness cost first, then search space
Fitness Scalability

- 1000 fitness evaluations means 1000 complete runs of your test suite
  - This task can be done in parallel (two ways)
- We view it as an advantage that we can repair programs with only a few test cases
- But we want to scale to more larger test suites
- For both performance and correctness
  - Test cases encode required behavior!
Test Suite Purposes

• Thus far, the full test suite determines:
  • Do we keep this variant in the next generation?
  • Is this a candidate repair that passes all tests?
• Insight: split these tasks
  • Use a small subset of tests to decide “keep/drop”
  • GP structure allows noise
  • Use the full suite to evaluate candidate repairs
Test Suite Selection

- Can choose subset at random or by some other metric (e.g., max coverage, min time)

- In intermediate steps, poor test selection:
  - Retains variants that should be dropped
  - Drops variants that should be kept (rare)
  - Distorts the view of the fitness landscape

- Thus requiring more generations

- Does the time saved in fitness evaluations exceed the cost of being “led astray”? 
Tests Suite Selection Algorithms

• Random Subset
  • Pick next test at random without replacement

• Time-Aware Test Suite Prioritization
  • Includes test time and test coverage
    (Walcott, Soffa, Kapfhammer, Roos. ISSTA'06)

• Greedy Coverage
  • Pick next test to maximize coverage gains
Test Suite Selection Results

- 10 programs, each with 100+ test cases

<table>
<thead>
<tr>
<th>Program</th>
<th>Total LOC</th>
<th>Module LOC</th>
<th># Tests</th>
<th>Test Suite Description</th>
<th>Defect Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deroff utx 4.3</td>
<td>2236</td>
<td>2236</td>
<td>100</td>
<td>fuzz input (as typesetting directives)</td>
<td>segfault</td>
</tr>
<tr>
<td>look utx 4.3</td>
<td>1169</td>
<td>1169</td>
<td>100</td>
<td>fuzz input (for both needle and haystack strings)</td>
<td>segfault</td>
</tr>
<tr>
<td>uniq utx 4.3</td>
<td>1146</td>
<td>1146</td>
<td>100</td>
<td>fuzz input (as duplicate-containing text file)</td>
<td>segfault</td>
</tr>
<tr>
<td>zune</td>
<td>28</td>
<td>28</td>
<td>100</td>
<td>fuzz input (as days since 1 January 1980)</td>
<td>infinite loop</td>
</tr>
<tr>
<td>gcd</td>
<td>22</td>
<td>22</td>
<td>100</td>
<td>fuzz input (as pairs of integers)</td>
<td>infinite loop</td>
</tr>
<tr>
<td>lighttpd 1.4.15</td>
<td>51895</td>
<td>3829</td>
<td>200</td>
<td>HTTP requests from cs.virginia.edu</td>
<td>buffer overrun</td>
</tr>
<tr>
<td>nullhttpd 0.5.0</td>
<td>5575</td>
<td>5575</td>
<td>200</td>
<td>HTTP requests from cs.virginia.edu</td>
<td>buffer overrun</td>
</tr>
<tr>
<td>leukocyte</td>
<td>6718</td>
<td>6718</td>
<td>100</td>
<td>video microscopy images</td>
<td>segfault</td>
</tr>
<tr>
<td>tiff 3.8.2</td>
<td>84067</td>
<td>1084</td>
<td>106</td>
<td>image suite bundled with tifflib</td>
<td>segfault</td>
</tr>
<tr>
<td>imagemagick 6.5.2</td>
<td>450416</td>
<td>5858</td>
<td>100</td>
<td>images from pngsuite test suite</td>
<td>wrong output</td>
</tr>
<tr>
<td>total</td>
<td>603272</td>
<td>27665</td>
<td>1206</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Selection reduces time-to-repair by 81%
- Yields equivalent-quality repairs
- leukocyte with 100 tests: 90 mins to 6 mins
- imagemagick with 100 tests: 36 mins to 3 mins
Test Suite Selection Explained

- Small changes between variants mean most variants have similar test case behavior
  - True. However, an optimal safe impact analysis could only reduce time-to-repair by 29% (cf. 81%)
- The test cases are all dependent, and thus running one is as good as running another
  - False. There was only a 3% performance increase between high-overlap and low-overlap suites.
- The fitness function can tolerate noise
  - True. Test suite selection on gcd distorts the fitness function by 27%. (cf. Fitness Distance Correlation)
Search Space Scalability

• So each variant can be evaluated rapidly
• The other factor in cost is the number of variants examined
  • i.e., the size of the search space
• This is related to fault localization precision, not overall program size
  • Since we only mutate and crossover statements along the weighted path
Search Space vs. Fault Localization

![Graph showing the relationship between the log of average number of fitness evaluations and the log of weighted path length. The graph includes labels for various projects such as 'indent', 'flex', 'nullhttpd', 'units', 'deroff', 'look', 'look svr4', 'lighttpd', 'opendlap'.]
Outline

- Fixing Real Bugs In Real Programs
  - Representation and Operations
- The Quality of Automated Repairs
  - Self-Healing Systems and Metrics
- Test Suite Selection
  - Success and Explanations
- Open Questions in Automated Repair
Can Formal Specifications Be Used?

- Use Local Annotations in Mutation
- Typestate Repair
  - Algorithms for repairing programs with respect to a temporal safety policy
  - Provably safe with respect to that one policy
- Synthesis
  - Use GP to identify regions, not to copy statements
- Refactoring for Formal Verification
  - If repair is correct but cannot easily be verified
What Fault Localization is Possible?

- Standard approaches (e.g., Tarantula)
- Cooperative Bug Isolation
  - Instrument program with Daikon-style predicates
  - Measure which are false on normal runs but true on failing runs (etc.)
  - Have repaired a program using a weighted path induced from CBI information
- Impact on mutation operator:
  - Guide changes to flip predicates
What Mutations are Possible?

- **Goal:** increase “expressive power”

- **Expression-level Mutation**
  - Increases size of search space
  - In practice, reduces time-to-repair by 30%
    - “x=3;” vs. “x=0; x++; x++; x++;”

- **“Typed” Repair Templates**
  - “if (local_ptr != NULL) { local_stmt(local_ptr); }”
  - Manually crafted or mined automatically
    - Distance metric on changes
Can We Handle Threads?

• Currently we assume a deterministic fitness function and the ability to localize faults.

• VM Integration
  • Add scheduler constraints to the representation
  • Repair = code changes plus scheduler directives

• Context-Bounded
  • Existing tools can prove the presence or absence of race conditions assuming at most \( k \) thread interleavings
Can We Do More Than Repair?

- Evolutionary approaches are traditionally strong at small optimization and synthesis
- Vertex and Pixel Shaders
  - Small C programs used by modern graphics cards
  - Optimize for space or speed
  - Can be “10% blurrier” than original
  - What is “fault localization” here?
Is Our Fitness Function Reasonable?

- A good fitness function increases for more desirable variants
- Ours is accurate at the extremes
  - But weak in the middle
- Correlation with an “optimal” tree structured distance metric for known repairs is ~0.3
- Can we combine test case counts with …
  - Invariants retained, anomaly detection signals, …
Is Competitive Coevolution Possible?

• In the security domain, white hats and black hats both want to identify the next attack as quickly as possible

• Can we simulate this “arms race”?  
  • Many exploits are themselves C programs
  • For a small exploitable program we have evolved a repair, then evolved the exploit to work again, then evolved a second repair

• Automated Hardening and Synthetic Diversity
  • Repair old programs against various signals
  • Does it defeat attacks that came out later?
Conclusions

- We can automatically and efficiently repair certain classes of bugs in off-the-shelf legacy programs.
  - 20 programs totaling 186kloc in about 5 minutes each, on average
- We use regression tests to encode desired behavior.
  - Existing tests encode required behavior
- The genetic programming search focuses attention on parts of the program visited during the bug but not visited during passed test cases.
Questions

• I encourage difficult questions.
Bonus Slide: Test Cases

```bash
#!/bin/sh
# Positive Test Case for nullhttpd (POST data)
ulimit -t 5
/usr/bin/wget --tries=1 --post-data 'name=my_name&submit=submit'
    "http://localhost:\$PORT/cgi-bin/hello.pl"
if diff hello.pl ../known-good-hello.pl-result ; then
    # if the current output matches the known-good output
    echo "passed hello.pl test case" >> ../list-of-tests-passed
fi
```

Figure 2: Positive test case for `nullhttpd`. `wget` is a command-line HTTP client; `ulimit` cuts the test off after five seconds. The test assumes that the sandboxed webserver is accepting connections on `PORT` and has its own copy of `htdocs`, including `cgi-bin/hello.pl`. Note the oracle comparison using `diff` against `known-good-hello.pl-result` on line 6.

```bash
#!/bin/sh
# Negative Test Case for nullhttpd
ulimit -t 5
./nullhttpd-exploit -h localhost -p $PORT -t2
/usr/bin/wget --tries=1 "http://localhost:$PORT/index.html"
if diff index.html ../known-good-index.html-result ; then
    # if the current output matches the known-good output
    echo "passed exploit test case" >> ../list-of-tests-passed
fi
```

Figure 3: Negative test case for `nullhttpd`. If the exploit (line 4) disables the webserver then the request for `index.html` (line 5) will fail.
Evolution of Zune Repair

(5 normal test cases weighing 1 each, 2 buggy test cases weighing 10 each)