Software Vulnerability Detection and Repair

Prof. Abhik Roychoudhury
National University of Singapore
Enhancing local capabilities

Education – NUS (Bachelors in Infosec)

Vulnerability Discovery

Binary Hardening

Verification

Data Protection

Agency Collaboration

Industry Collaboration

Research Outputs – Publications, Tools, Academic Collaboration, Exchanges, Seminars, Workshops
Space of Problems

- **Fuzz Testing**
  - Feed semi-random inputs to find hangs and crashes

- **Continuous fuzzing**
  - Incrementally find new “problems” in software

- **Crash reproduction**
  - Re-construct a reported crash, crashing input not included due to privacy

- **Reaching nooks and corners**

- **Localizing reported observable errors**

- **Patching reported errors from input-output examples**
Space of Techniques

Search
- Random
- Biased-random
- Genetic (AFL Fuzzer)
- ...

- Low set-up overhead
- Fast, less accurate
- Use objective function to steer

Symbolic Execution
- Dynamic Symbolic execution
- Concolic Execution
- Cluster paths based on symbolic expressions of variables
- ....

- High set-up overhead
- Slow, more accurate
- Use logical formula to steer
In this talk ...

Search
- Enhance the effectiveness of search techniques, with symbolic execution as inspiration

- [CCS16, CCS17, ICSE15]

Symbolic Execution
- Explore capabilities of symbolic execution beyond search, in program repair

- [ICSE13, 15, 16, 18]
History of fuzzing

Developed by Barton Miller, see http://pages.cs.wisc.edu/~bart/fuzz/

Fuzz testing is a simple technique for feeding random input to applications. The approach has three characteristics.

- The input is **random**. We do not use any model of program behavior, application type, or system description. This is sometimes called **black box testing**.

- The reliability criteria is **simple**: if the application **crashes or hangs**, it is considered to fail the test, otherwise it passes. Note that the application does not have to respond in a sensible manner to the input, and it can even quietly exit.

- As a result of the first two characteristics, fuzz testing can be **automated** to a high degree and results can be compared across applications, operating systems, and vendors.
Grey-box Fuzzing, as in AFL
Grey-box Fuzzing Algorithm

- **Input**: Seed Inputs $S$
- **1**: $T_x = \emptyset$
- **2**: $T = S$
- **3**: if $T = \emptyset$ then
- **4**: add empty file to $T$
- **5**: end if
- **6**: repeat
- **7**: $t = \text{chooseNext}(T)$
- **8**: $p = \text{assignEnergy}(t)$
- **9**: for $i$ from 1 to $p$ do
- **10**: $t_0 = \text{mutate_input}(t)$
- **11**: if $t_0$ crashes then
- **12**: add $t_0$ to $T_x$
- **13**: else if $\text{isInteresting}(t_0)$ then
- **14**: add $t_0$ to $T$
- **15**: end if
- **16**: end for
- **17**: until timeout reached or abort-signal
- **Output**: Crashing Inputs $T_x$
Programming by experienced people

Schematic

- if (condition1)
  - return  // short path, frequented by many many inputs
- else if (condition2)
  - exit  // short paths, frequented by many inputs
- else ....
Prioritize low probability paths

- Use grey-box fuzzer which keeps track of path id for a test.
- Find probabilities that fuzzing a test $t$ which exercises $\pi$ leads to an input which exercises $\pi'$

$$p$$

- Higher weightage to low probability paths discovered, to gravitate to those -> discover new states in Markov Chain with minimal effort.

```c
void crashme (char* s) {
    if (s[0] == 'b')
        if (s[1] == 'a')
            if (s[2] == 'd')
                if (s[3] == '!')
                    abort();
    }
```
Power-Schedules

- Constant: \( p(i) = \alpha(i) \)
  - AFL uses this schedule (fuzzing ~1 minute)
  - \( \alpha(i) \) .. how AFL judges fuzzing time for the test exercising path \( i \)

- Cut-off Exponential:
  \[
  p(i) = 0, \text{ if } f(i) > \mu \\
  \min(\alpha(i)/\beta \cdot 2^{s(i)}, M), \text{ otherwise}
  \]

  \( \beta \) is a constant
  \( s(i) \) #times the input exercising path \( i \) has been chosen for fuzzing
  \( f(i) \) #fuzz exercising path \( i \) (path-frequency)
  \( \mu \) mean #fuzz exercising a discovered path (avg. path-frequency)
  \( M \) maximum energy expendable on a state
Independent evaluation found crashes 19x faster on DARPA Cyber Grand Challenge (CGC) binaries

Integrated into main-line of AFL fuzzer within a year of publication (CCS16), which is used on a daily basis by corporations for finding vulnerabilities
Comments on the technologies

The paper [1] on AFLFast is, IMO, a great example of where academia shines: carefully looking at how and why something works, developing some theory and a working model, and then using that to get a substantial improvement on the state of the art (and doing a nice evaluation to show that it really works).

The abstract sounds like it. They said with no program analysis, though. I thought program analysis was good enough that it could probably auto-generate tests for every branch in a program, possibly in less time or with more assurance. W as I wrong or is this a parallel subfield?

The question of whether randomized testing or program analysis gives you more coverage of a program is a really interesting one. Böhme actually has an earlier paper that addresses this question: https://www.comp.nus.edu.sg/~mboehme/paper/FSE14.pdf
Use of Grey-box Fuzzing

- **Greybox Fuzzing** is frequently used, daily in corporations
  - State-of-the-art in automated vulnerability detection
  - Extremely efficient coverage-based input generation
    - All program analysis before/at instrumentation time.
    - Start with a seed corpus, choose a seed file, fuzz it.
    - Add to corpus only if new input increases coverage.
  - Cannot be directed, unlike symbolic execution!
In this talk ...

Search
- Enhance the effectiveness of search techniques, with symbolic execution as inspiration
  - Enhance coverage, how to make it directed?

Symbolic Execution
- Explore capabilities of symbolic execution beyond directed search
Directed Fuzzing instead of Coverage

- In-house debugging and fixing
- Vulnerability checking
Using symbolic execution

Reproduced vulnerabilities in Acrobat Reader, Media Player with 24 hour time bound

1. Directed Search Algorithm
2. Guided Selective Symbolic Execution
Symbolic Analyzer

Reproduced vulnerabilities in Acrobat Reader, Media Player with 24 hour time bound
Directed Fuzzing: classical constraint satisfaction prob.

- Program analysis to identify program paths that reach given program locations.

- Symbolic Execution to derive path conditions for any of the identified paths.

- Constraint Solving to find an input that satisfies the path condition and thus reaches a program location that was given.

\[ \varphi_1 = (x > y) \land (x + y > 10) \]
\[ \varphi_2 = \neg (x > y) \land (x + y > 10) \]
(Later) View-point

- Directed Fuzzing as optimization problem!

1. Instrumentation Time:
   - Instrument program to aggregate distance values.

2. Runtime, for each input
   - decide how long to be fuzzed based on distance.
     - If input is closer to the targets, it is fuzzed for longer.
     - If input is further away from the targets, it is fuzzed for shorter.
Power Schedules - Recap

- Input: Seed Inputs $S$
  1: $T_x = \emptyset$
  2: $T = S$
  3: if $T = \emptyset$ then
  4: add empty file to $T$
  5: end if
  6: repeat
  7: $t = \text{chooseNext}(T)$
  8: $p = \text{assignEnergy}(t)$
  9: for $i$ from 1 to $p$ do
  10: $t_0 = \text{mutate_input}(t)$
  11: if $t_0$ crashes then
  12: add $t_0$ to $T_x$
  13: else if $\text{isInteresting}(t_0)$ then
  14: add $t_0$ to $T$
  15: end if
  16: end for
  17: until timeout reached or abort-signal
- Output: Crashing Inputs $T_x$

Exercises common path that rejects invalid PDF
Short paths rejecting syntactically invalid inputs are exercised by fuzzer too frequently
Instrumentation

- **Function-level target distance** using call graph (CG)
- **BB-level target distance** using control-flow graph (CFG)

1. Identify target BBs and assign distance 0
2. Identify BBs that call functions and assign 10*FLTD
3. For each BB, compute harmonic mean of (length of shortest path to any function-calling BB + 10*FLTD).

**CFG for function b**

8.7

30

11

13

10

12

N/A

KLEE Workshop London 2018
Directed fuzzing as optimization

- Integrating Simulated Annealing as power schedule
  - In the beginning (t = 0min), assign the **same energy** to all seeds.
  - Later (t=10min), assign a **bit more energy** to seeds that are **closer**.
  - At exploitation (t=80min), assign **maximal energy** to seeds that are **closest**.
In this talk ...

**Search**
- Enhance the effectiveness of search techniques, with symbolic execution as inspiration
  - Enhance coverage
  - Achieve directed search

**Symbolic Execution**
- Explore capabilities of symbolic execution beyond directed search

KLEE Workshop London 2018
Program Synthesis and Repair

Input-output Examples
Quality Criterion
Tests
Vulnerable program
Synthesizer
Program
Repair System
Repaired Program

KLEE Workshop London 2018
Search-based approach

- 2009: GenProg [Weimer-et-al-ICSE]

EVALUATE
FITNESS

DISCARD

ACCEPT

MUTATE

KLEE Workshop London 2018
Over-fitting in Tests -> Program

Avoid

if (input1) return output1
else if (input2) return output2
else if (input3) return output3
....

Generalize beyond the provided tests using symbolic reasoning.

Artifacts (symbolic formulae)

Tests

Vulnerable program

Repair System

Repaired Program

KLEE Workshop London 2018
View-point on Repair

1. Where to fix – in which line?
2. Generate the candidate patches in this line.
3. Validate the candidate patches.

2. What values should be returned by these lines? \(<\text{inp}=1, \text{ret}=0>\)
3. What are the expressions which will return these values?

Syntactic approach

Semantic approach
Example

```c
int is_upward(int inhibit, int up_sep, int down_sep) {
    int bias;
    if (inhibit)
        bias = down_sep; // bias = up_sep + 100
    else  bias = up_sep ;
    if (bias > down_sep)
        return 1;
    else return 0;
}
```

<table>
<thead>
<tr>
<th>inhibit</th>
<th>up_sep</th>
<th>down_sep</th>
<th>Observed output</th>
<th>Expected Output</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>110</td>
<td>0</td>
<td>1</td>
<td>fail</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>pass</td>
</tr>
<tr>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>0</td>
<td>1</td>
<td>fail</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
</tbody>
</table>

KLEE Workshop London 2018
Patch synthesis

Inhibit  |  up_sep  |  down_sep
== 1    |  == 11  |  == 110

1 \textbf{int} \texttt{is\_upward}( \textbf{int} \texttt{inhibit}, \textbf{int} \texttt{up\_sep}, \textbf{int} \texttt{down\_sep})\{ \\
2 \hspace{1em} \textbf{int} \texttt{bias}; \\
3 \hspace{1em} \textbf{if} (\texttt{inhibit}) \\
4 \hspace{3em} \textbf{bias} = \texttt{f(inhibit, up\_sep, down\_sep)} \\
5 \hspace{1em} \textbf{else} \hspace{1em} \texttt{bias} = \texttt{up\_sep} ; \\
6 \hspace{1em} \textbf{if} (\texttt{bias} > \texttt{down\_sep}) \\
7 \hspace{3em} \textbf{return} 1; \\
8 \hspace{1em} \textbf{else} \hspace{1em} \textbf{return} 0; \\
9 \} \\

\texttt{f(1,11,110) > 110}

Symbolic Execution
Patch synthesis

- Accumulated constraints
  - $f(1, 11, 110) > 110$
  - $f(1, 0, 100) \leq 100$
  - ... 

- Find a $f$ satisfying this constraint
  - By fixing the set of operators appearing in $f$

- Candidate methods
  - Search over the space of expressions
  - Program synthesis with fixed set of operators

- Generated fix
  - $f(\text{inhibit, up}_\text{sep, down}_\text{sep}) = \text{up}_\text{sep} + 100$
Semantic Repair

Concrete Execution

Program

Test input

Concrete values

Output: Value-set or Constraint

Symbolic execution

Expected output of program

KLEE Workshop London 2018
High-level view

Buggy Program

Concrete Execution

Test input

var = \(a + b - c\);

Concrete Execution

Symbolic Execution with \(x\) as the only unknown

Path conditions, Output Expressions

\(x = f(\text{Live Vars})\)

Get properties of function \(f\) via symbolic execution.

Construct a function \(f\) which satisfies these properties!
Program Repair given tests

- Generate and-test patches \((\text{GenProg, CMU/Michigan})\)

- Specification inference and patch synthesis
  - Infer specification or properties about the patch to be synthesized.
  - Meet the specification by enumeration, or by solving constraints.
  - Various works – \text{SemFix, Angelix (NUS), Nopol (KTH), SPR (MIT)}, ...

- Ordering of search space of patches
  - Use minimality to prioritize the search space. \((\text{NUS})\)
  - Use learning approaches to prioritize the search space. \((\text{MIT})\)
    - Patch templates can be learnt from human fixes. \((\text{HKUST})\)
http://angelix.io  [ICSE13,16]
### State-of-the Technology

#### Defect

<table>
<thead>
<tr>
<th>Subject</th>
<th>LoC</th>
<th>Repair time (min)</th>
<th>Fixed Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>wireshark</td>
<td>2814K</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>php</td>
<td>1046K</td>
<td>62</td>
<td>2</td>
</tr>
<tr>
<td>gzip</td>
<td>491K</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>gmp</td>
<td>145K</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>libtiff</td>
<td>77K</td>
<td>14</td>
<td>3</td>
</tr>
</tbody>
</table>

#### Scalability

**Quality:** Less functionality-deleting repair than any other tool.
Heartbleed

(a) The buggy part of the Heartbleed-vulnerable OpenSSL

```c
if (hbtype == TLS1 HB REQUEST) {
    ... 
    memcpy(bp, pl, payload);
    ...
}
```

(b) A fix generated automatically

```c
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0;
... 
if (hbtype == TLS1_HB_REQUEST) {
    ...
}
else if (hbtype == TLS1_HB_RESPONSE) {
    ...
}
return 0;
```

(c) The developer-provided repair
Use a reference implementation


(a) Correct linear search

1 int search(int x, int a[], int length) {
2    int i;
3    for (i=0; i<length; i++) {
4      if (x == a[i])
5         return i;
6      }
7    return -1;
8 }

(b) Buggy binary search

1 int search(int x, int a[], int length) {
2    int L = 0;
3    int R = length-1;
4    do {
5      int m = (L+R)/2;
6      if (x == a[m]) {
7         return m;
8      } else if (x < a[m]) { // bug fix: x > a[m]
9          L = m+1;
10      } else {
11          R = m-1;
12      }
13    } while (L <= R);
14    return -1;
15 }

Experiments on embedded Linux Busybox

Verification condition
SemGraft (ICSE18)

- Buggy program
- Reference program
- Symbolic analysis
- Verification condition
- Candidate patch
- Component library
- Angelic forest
- Counterexample
- Is SAT?
- Negate
- No
- Patch found
- Yes
- Is SAT?
- Yes
- Is SAT?
- Yes
- Buggy program
## SemGraft results

### GNU Coreutils as reference

<table>
<thead>
<tr>
<th>Program</th>
<th>Commit</th>
<th>Bug</th>
<th>Angelix</th>
<th>SemGraft</th>
</tr>
</thead>
<tbody>
<tr>
<td>sed</td>
<td>c35545a</td>
<td>Handle empty match</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>seq</td>
<td>f7d1c59</td>
<td>Wrong output</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>sed</td>
<td>7666fa1</td>
<td>Wrong output</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
<tr>
<td>sort</td>
<td>d1ed3e6</td>
<td>Wrong output</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
<tr>
<td>seq</td>
<td>d86d20b</td>
<td>Don’t accepts 0</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
<tr>
<td>sed</td>
<td>3a9365e</td>
<td>Handle s///</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
</tbody>
</table>

### Linux Busybox as reference

<table>
<thead>
<tr>
<th>Program</th>
<th>Commit</th>
<th>Bug</th>
<th>Angelix</th>
<th>SemGraft</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>f7d1c59</td>
<td>Segmentation fault</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
<tr>
<td>mkfifo</td>
<td>cdb1682</td>
<td>Segmentation fault</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
<tr>
<td>mknod</td>
<td>cdb1682</td>
<td>Segmentation fault</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
<tr>
<td>copy</td>
<td>f3653f0</td>
<td>Failed to copy a file</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>md5sum</td>
<td>739cf4e</td>
<td>Segmentation fault</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>cut</td>
<td>6f374d7</td>
<td>Wrong output</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
</tbody>
</table>
Novel applications

Use program repair in intelligent tutoring systems to give the students' individual attention.

Study in IIT-Kanpur (FSE17)
Acknowledgments

Co-authors:

Umair Ahmed & Amey Karkare (IIT-K)
Marcel Boehme (Monash),
Satish Chandra (Facebook),
Lars Grunske & Yannic Noller (Humboldt)
Sergey Mechtaev (NUS)
HDT Nguyen and Dawei Qi
Manh-Dung Nguyen & Van-Thuan Pham (NUS)
Mukul Prasad & Hiroaki Yoshida (Fujitsu),
Shin Hwei Tan (SUSTech)
Jooyong Yi (Innopolis)

Relevant papers:
http://www.comp.nus.edu.sg/~abhik/projects/Fuzz/

Grants:

NRF NCR program TSUNAMi project (2015-2020)
DSO grant (2013-15).
Airbus grant (2017-18).
Vulnerability detection and patching

Search
- Enhance the effectiveness of search techniques, with symbolic execution as inspiration
  - Enhance coverage
  - Achieve directed search

Symbolic Execution
- Explore capabilities of symbolic execution beyond directed search
  - Specification inference
  - Program synthesis/repair
For more details

Own website

- http://www.comp.nus.edu.sg/~abhik

Project website

- http://www.comp.nus.edu.sg/~tsunami/

Links on Repair

Links on Fuzzing
http://www.comp.nus.edu.sg/~abhik/projects/Fuzz/

Let us talk in the reception now, or tomorrow – if you are interested.
Reflections on Symbolic Execution

Reachability of a location. e.g. KATCH, AFLGo

Bug finding. e.g. KLEE, AFLFast

Specification Inference from buggy program e.g. SemFix, Angelix