Static and Dynamic Analysis for Vulnerability Detection
Vulnerability Analysis

- Programmer checks the program, and corrects the errors
- Cycle repeated until all relevant bugs are fixed
Terminology

- **False Positives**: A warning or error is generated, but there is no real vulnerability.
- **False Negatives**: A vulnerability exists, but it is not being identified by the analysis.
- **Complete**: A technique that is guaranteed to be free of false positives.
- **Sound**: A technique that is guaranteed to detect all vulnerabilities (i.e., no FNs).
- **Note**: A technique cannot be sound and complete, since most program properties are undecidable in general.
- **Useful bug-finding tools suffer from both FNs and FPs**.
Benefits and Drawbacks

❖ Benefits
  ▪ Does not rely on bugs being exercised: fix the bug before it strikes you
  ▪ No runtime overhead
  ▪ Leverage programmer knowledge

❖ Drawbacks
  ▪ Not applicable for operator use
    ▼ May not have source code access
    ▼ May not be able to understand the logic of the program
  ▪ Suffers from false positives
    ▼ A programmer can cope with these, but not an operator
Vulnerability Analysis Techniques

- **Static analysis**
  - Analysis performed before a program starts execution
  - Works mainly on source code
    - Binary static analysis techniques are rather limited
  - Not very effective in practice, so we won’t discuss in depth

- **Dynamic analysis**
  - Analysis performed by executing the program
  - Key challenge: How to generate input for execution?
  - Two main approaches to overcome challenge
    - Fuzzing: random, black-box testing (primarily)
    - Symbolic execution: systematic technique for generating inputs that exercise “interesting program paths.”
      - More of a white-box approach.
Black-box fuzzing

BlackBoxFuzzing
Input: initial test suite TestSuite
Output: bug triggering inputs Crashers

Mutations (helper function)
Input: test input t
Output: new test inputs with some bits flipped in t

while TestSuite not empty:
  t = PickFrom(TestSuite)
  for each m in Mutations(t):
    RunAndCheck(m)
    if Crashes(m):
      add m to Crashers

Drawbacks
- Blind search: a successful mutation does not help subsequent search in any way
Coverage guided fuzzing

CoverageGuidedFuzzing
Input: initial test suite TestSuite
Output: bug triggering inputs Crashers

while TestSuite not empty:
    t = PickFrom(TestSuite)
    for each m in Mutations(t):
        RunAndCheck(m)
        if Crashes(m):
            add m to Crashers
        if NewCoverage(m)
            add m to TestSuite

Note: A successful mutation feeds into other mutations.

AFL Fuzzer by Zalewski ‘14
Coverage metrics

- **Statement coverage**
  - A statement is *covered* by a test input if it is executed at least once by the program when processing that input.

- **Edge (or branch) coverage**
  - An edge is covered by a test input if it is taken at least once when processing that input.

- **Counted coverage**
  - Take into account the number of times a statement (or edge) is executed. Variant: use log(count) instead of exact count.

- **Path coverage**
  - Similar, but applies to a full execution path.
  - Note: number of possible execution paths can be extremely large, or even be infinite, so it is not used.
Coverage metric

```c
void path_explosion(char *input) {
    int count = 0;
    for (int i = 0; i < 100; i++)
        if (input[i] == 'A')
            count++;
}
```
Coverage metric

```c
int walk_maze(char *steps) {
    ...
    int x, y; // Player position.
    for (int i = 0; i < ITERS; i++) {
        switch (steps[i]) {
            case 'U': y--; break;
            case 'D': y++; break;
            case 'L': x--; break;
            case 'R': x++; break;
            default: // Wrong command, lose.
        }
        if (maze[y][x] != ' ') // Lose.
            if (maze[y][x] == '#') // Win!
    ...
}
```

Winning input: **DDDDRRRUULUUU**
AFL – state-of-the-art fuzzing

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time</td>
<td>cycles done: 0</td>
</tr>
<tr>
<td>last new path</td>
<td>total paths: 195</td>
</tr>
<tr>
<td>last uniq crash</td>
<td>uniq crashes: 0</td>
</tr>
<tr>
<td>last uniq hang</td>
<td>uniq hangs: 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
<th>findings in depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing</td>
<td>map density: 1217 (7.43%)</td>
<td></td>
</tr>
<tr>
<td>paths timed out</td>
<td>count coverage: 2.55 bits/tuple</td>
<td></td>
</tr>
<tr>
<td></td>
<td>favored paths: 128 (65.64%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new edges on: 85 (43.59%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>total crashes: 0 (0 unique)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>total hangs: 1 (1 unique)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>now trying</td>
<td>levels: 3</td>
</tr>
<tr>
<td>stage execs</td>
<td>pending: 178</td>
</tr>
<tr>
<td>total execs</td>
<td>pend fav: 114</td>
</tr>
<tr>
<td>exec speed</td>
<td>imported: 0</td>
</tr>
<tr>
<td>fuzzing strategy yields</td>
<td>variable: 0</td>
</tr>
<tr>
<td>bit flips</td>
<td>latent: 0</td>
</tr>
<tr>
<td>byte flips</td>
<td></td>
</tr>
<tr>
<td>arithmetics</td>
<td></td>
</tr>
<tr>
<td>known ints</td>
<td></td>
</tr>
<tr>
<td>havoc</td>
<td></td>
</tr>
<tr>
<td>trim</td>
<td></td>
</tr>
</tbody>
</table>

- American Fuzzy Lop (0.47b) reading (readpng)
- Process timing:
  - Run time: 0 days, 0 hrs, 4 min, 43 sec
  - Last new path: 0 days, 0 hrs, 0 min, 26 sec
  - Last uniq crash: none seen yet
  - Last uniq hang: 0 days, 0 hrs, 1 min, 51 sec
- Cycle progress:
  - Now processing: 38 (19.49%)
  - Paths timed out: 0 (0.00%)
- Stage progress:
  - Now trying: interest 32/8
  - Stage execs: 0/9990 (0.00%)
  - Total execs: 654k
  - Exec speed: 2306/sec
- Fuzzing strategy yields:
  - Bit flips: 88/14.4k, 6/14.4k, 6/14.4k
  - Byte flips: 0/1804, 0/1786, 1/1750
  - Arithmetics: 31/126k, 3/45.6k, 1/17.8k
  - Known ints: 1/15.8k, 4/65.8k, 6/78.2k
  - Havoc: 34/254k, 0/0
  - Trim: 2876 B/931 (61.45% gain)
Bugs found by AFL

IJG jpeg 1, libjpeg-turbo 1 2, libpng 1, libtiff 1 2 3 4 5, mozjpeg 1, PHP 1 2 3 4 5, Mozilla Firefox 1 2 3 4, Internet Explorer 1 2 3 4, Apple Safari 1, Adobe Flash / PCRE 1 2 3 4, sqlite 1 2 3 4, OpenSSL 1 2 3 4 5 6 7, LibreOffice 1 2 3 4, poppler 1, freetype 1 2, GnuTLS 1, GnuPG 1 2 3 4, OpenSSH 1 2 3, PuTTY 1 2, ntpd 1, nginx 1 2 3, bash (post-Shellshock) 1 2, tcpdump 1 2 3 4 5 6 7 8 9, JavaScriptCore 1 2 3 4, pdfium 1 2, ffmpeg 1 2 3 4 5, libmatroska 1, libarchive 1 2 3 4 5 6 7 8, wireshark 1 2 3, ImageMagick 1 2 3 4 5 6 7 8 9, BIND 1 2 3, QEMU 1 2, Icms 1, Oracle BerkeleyDB 1 2, Android / libstagefright 1 2, iOS / ImageIO 1, FLAC audio library 1 2, libsndfile 1 2 3, less / lesspipe 1 2 3, strings (+ related tools) 1 2 3 4 5 6 7, file 1 2 3 4, dpkg 1 2, rcs 1, systemd-resolved 1 2, libyaml 1, Info-Zip unzip 1 2, libtasn1 1 2, OpenBSD pfctl 1, NetBSD bpf 1, man & mandoc 1 2 3 4 5 6 7, IDA Pro [reported by authors], clamav 1 2 3 4 5, libxml2 1 2 3 4 5 6 7 8 9, glibc 1, clang / llvm 1 2 3 4 5 6 7 8 9, nasm 1 2, ctags 1, mutt 1, procmail 1, fontconfig 1, pdksh 1 2, Qt 1, wavpack 1, redis / lua-cmsgpack 1, taglib 1 2 3, privoxy 1 2 3, perl 1 2 3 4 5 6 7 8 9, libxmp, radare2 1 2, SleuthKit 1, fwknop [reported by author], X.Org 1 2, exifprobe 1, jhead [?], capnproto 1, Xerces-C 1 2 3, metacam 1, djvu-libre 1, exiv 1, Linux btrfs 1 2 3 4 5 6 7 8, Knot DNS 1, curl 1 2, wpa_supplicant 1, libde265 [reported by author], dnsmasq 1, libbpg (1), lame 1, libwmf 1, uudecode 1, MuPDF 1, imlib2 1, libraw 1, libbson 1, libsass 1, yara 1 2 3 4, W3C tidy-html5 1, VLC 1, FreeBSD syscons 1 2 3, John the Ripper 1 2, screen 1 2 3, tmux 1 2, mosh 1, UPX 1, indent 1, openjpeg 1, MMIX 1, OpenMPT 1 2, rxvt 1 2, dhcpacd 1, Mozilla NSS 1, Nettle 1, mbed TLS 1, Linux netlink 1, Linux ext4 1, Linux xfs 1, botan 1, expat 1 2, Adobe Reader 1, libav 1, libical 1, OpenBSD kernel 1, collectd 1, libidn 1 2
JPEGs out of thin air
Fuzzing: strength and weaknesses

```java
if (input == 0x1badc0de) {
    ...
}

if (adler32(input) == 0x3eb52a45) {
    ...
}
```
Dynamic symbolic execution (DSE)

**DynamicSymbolicExecution**
Input: initial test suite *TestSuite*
Output: bug triggering inputs *Crashers*

while *TestSuite* not empty:  
  
  \[ t = \text{PickFrom}(*TestSuite*) \]
  
  for each *m* in DSENewInputs(*t*):
    
    RunAndCheck(*m*)
    
    if Crashes(*m*):
      
      add *m* to *Crashers*
      add *m* to *TestSuite*
Dynamic symbolic execution

DSENewInputs
Input: test case $t$
Output: new test cases $Children$

$PC = \text{ExecuteSymbolically}(t)$
for each condition $c$ in $PC$

$NEW\_PC = PC[0..i-1] \text{ and not } c$
$new\_input = \text{SMTSolve}(NEW\_PC)$
if $new\_input \neq \text{UNSAT}$:
    add $new\_input$ to $Children$
Constraint solvers (SAT/SMT)

• Complete solvers (most used)
  • Complete = always returns an answer (*given enough time*)
  • **Backtracking** based algorithms
  • Typically based on Conflict-Driven Clause Learning (CDCL) algorithm
  • E.g., **Z3** from Microsoft Research, STP (used by KLEE)

• Incomplete solvers
  • Incomplete = may return “don’t know”
  • Trade-off between complexity and the quality of the search
  • **Stochastic local search (SLS)** based algorithms
  • E.g., **SLS-SMT** by Frohlich et al. [AAAI’15]

• Note that theoretically complete solvers are indeed incomplete in their practical use, since implementations call the solver, and time out after a specific period.
Fuzzing vs. DSE

<table>
<thead>
<tr>
<th>Technique</th>
<th>Replayable</th>
<th>Semantic Insight</th>
<th>Scalability</th>
<th>Crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Symbolic Execution</td>
<td>Yes</td>
<td>High</td>
<td>Low</td>
<td>16</td>
</tr>
<tr>
<td>Veritest</td>
<td>Yes</td>
<td>High</td>
<td>Medium</td>
<td>11</td>
</tr>
<tr>
<td>Dynamic Symbolic Execution + Veritest</td>
<td>Yes</td>
<td>High</td>
<td>Medium</td>
<td>23</td>
</tr>
<tr>
<td>Fuzzing (AFL)</td>
<td>Yes</td>
<td>Low</td>
<td>High</td>
<td>68</td>
</tr>
</tbody>
</table>

“In reality, fuzzing identified almost three times as many vulnerabilities [as DSE]. In a sense, this mirrors the recent trends in the security industry: symbolic analysis engines are criticized as impractical while fuzzers receive an increasing amount of attention. However, this situation seems at odds with the research directions of recent years, which seem to favor symbolic execution.” ANGR Study (The Art of War) [Oakland’16]
DSE: strength and weaknesses

• Symbolic state maintenance is costly
  • Overhead of executing symbolically can be ~1000x [SAGE]

• Constraint solving does not scale well (NP-hard problem)
  • time complexity: complex formulas often time out

• Path condition solved by the solver is not guaranteed to take the targeted path
  • Due to imperfections of the symbolic memory model and environment model
  • Path divergence in 60% of the case [SAGE]

• The probability of a new test case exercising a new path is still much higher than in case of blind fuzzing