Week 11A: Directed Fuzzing I

Stefan Nagy

University of Utah
How are semester projects going?

Smoothly?

Obstacles?
Questions?
Fuzzing Specific Locations
Recap: guided fuzzing

- **Idea:** track some measure of exploration “progress”
  - Coverage of program code
  - Stack traces
  - Memory accesses

- Pinpoint inputs that further progress over the others

- **Mutate only those inputs**
What if I only want to fuzz one location?
What if I only want to fuzz *one* location?
What if I only want to fuzz one location?

- **Regression testing**
  - Did my PR break the software?

- **Patch testing**
  - Have I actually fixed this vulnerability?

- **Crash reproduction**
  - Is this random person’s bug report valid?
“Directed” Fuzzing

- Guided fuzzing steered to **specific locations**
  - E.g., Patch-changed code lines
  - E.g., An ASAN-reported crash line

- **Key differences versus guided fuzzing:**
  - **Instrumentation:**
    - Track **distance** relative to targeted site(s)
    - Compute this for every generated test case
  - **Seed selection:**
    - Pick inputs that get you **closer** to target(s)
    - Progress stalls? Pick a new input and restart

```c
1   if (input < 100) 2
2     f(0);          1
3
4   if (input > 100) 3
5     if (input > 200) 2
6       f(input) 1
7
8   void f(int x) {
9     if (x == 999) 1
10       // target 0
11   }
```

Source: KATCH: High-Coverage Testing of Software Patches
Directed Fuzzing
Recap: Symbolic Execution

- Solve paths as **symbolic expressions**

```
0. def f (x, y):
1.    if (x > y):
2.       x = x + y
3.       y = x - y
4.       x = x - y
5.    if (x - y > 0):
6.        assert false
7.    return (x, y)
```

Possible path constraints:
- *(A > B) and (B-A > 0)* = unsatisfiable
- *(A > B) and (B-A <= 0)* = satisfiable
- *(A <= B)* = satisfiable

```
L2: x : A+B
    y : B
L3: x : A+B
    y : (A+B) - B = A
L4: x : (A+B) - A = B
    y : A
L5: x : B
    y : A
L6: B - A > 0
L7: B - A <= 0
```

- **L2** satisfied
- **L3** satisfied
- **L4** satisfied
- **L5** unsatisfiable
- **L6** unsatisfiable
- **L7** satisfiable

Stefan Nagy
Directed Symbolic Execution

- **Early directed testing relied on SE**
  - E.g., KATCH (built atop of KLEE)
  - Primarily used for patch testing

- **Idea:** perform SE on specific paths
  - **Recap:** SE models paths symbolically
    - Find all satisfiable assignments
    - Generates branch-solving inputs

- **Trade-offs:**
  - Far too **heavyweight** to be practical
    - Not great on complex programs
Directed Fuzzing

- Direct successor to DSE
  - Originator: AFL-Go

- Idea: minimize seed–target distance
  - Obtain each basic block’s distance to target(s)
    - Computed during instrumentation time
  - Aggregate seed distance over block distances
    - Ideally minimize this over time
Distance Measurements

(a) Arithmetic Mean

Source: Directed Greybox Fuzzing
Distance Measurements

(a) Arithmetic Mean

(b) Harmonic Mean

Source: Directed Greybox Fuzzing
Distance Measurements

(a) Arithmetic Mean

(b) Harmonic Mean

Source: Directed Greybox Fuzzing
Function-level Distances

- Obtain the program’s call graph
  - Relationships among all subroutines
  - Here, our target function is E
Function-level Distances

- Obtain the program’s **call graph**
  - Relationships among all subroutines
  - Here, our target function is E

- Assign each $f$ a harmonic distance
  - Relative to the **target function(s)**
  - No path to target? No score (e.g., D)
Block-level distances

- Obtain control-flow graph for each \( f \)
  - Transitions between basic blocks in \( f \)
  - Here, we have a CFG for function \( B \)
Block-level distances

- Obtain **control-flow graph** for each $f$
  - Transitions between basic blocks in $f$
  - Here, we have a CFG for function $B$

- Identify basic blocks that call **functions**
  - Here, calls to functions $A$ and $C$
Block-level distances

- Obtain **control-flow graph** for each \( f \)
  - Transitions between basic blocks in \( f \)
  - Here, we have a CFG for function \( B \)

- Identify basic blocks that call **functions**
  - Here, calls to functions \( A \) and \( C \)

- Assign distances to each \( b \) in \( f \)
Block-level distances

- Obtain **control-flow graph** for each $f$
  - Transitions between basic blocks in $f$
  - Here, we have a CFG for function $B$

- Identify basic blocks that call **functions**
  - Here, calls to functions $A$ and $C$

- Assign distances to each $b$ in $f$
  - **Callers:** $10 \times$ (callee’s function-level distance)
**Block-level distances**

- Obtain **control-flow graph** for each $f$
  - Transitions between basic blocks in $f$
  - Here, we have a CFG for function $B$

- Identify basic blocks that call **functions**
  - Here, calls to functions $A$ and $C$

- Assign distances to each $b$ in $f$
  - **Callers**: $10 \times$ (callee’s function-level distance)
    - Choice of 10 seems arbitrary
Block-level distances

- Obtain control-flow graph for each $f$
  - Transitions between basic blocks in $f$
  - Here, we have a CFG for function $B$

- Identify basic blocks that call functions
  - Here, calls to functions $A$ and $C$

- Assign distances to each $b$ in $f$
  - Callers: $10 \times$ (callee’s function-level distance)
    - Choice of 10 seems arbitrary
  - Rest: harmonic distances to caller blocks
    - No path to a caller? No score

\[
8.7 = \frac{1}{\frac{1}{1+30} + \frac{1}{2+10}}
\]

Diagram:

- Nodes represent basic blocks.
- Edges indicate transitions between blocks.
- Distances are assigned based on the call graph structure.

Stefan Nagy
Aggregate seed distance

- Normalize cumulative block distances over edges taken
Aggregate seed distance

- Normalize cumulative block distances over edges taken
  - E.g., seed one = \((8.7 + 30) / 2\)
    - Seed Distance = \(19.35\)
Aggregate seed distance

- Normalize cumulative block distances over edges taken
  - E.g., seed one = (8.7 + 30) / 2
    - Seed Distance = 19.35
  - E.g., seed two = (8.7 + 11 + 10 + 12) / 4
    - Seed Distance = 10.42
Closing the distance

- By minimizing distance, we are treating programs as gradients
  - Want to converge on this gradient’s global minima
Closing the distance

- By minimizing distance, we are treating programs as **gradients**
  - Want to converge on this gradient’s **global minima**

- **Problem:** programs are spaghetti code
  - More likely to reach a **local minima** at first
  - Can get stuck really easily on bad paths
Closing the distance

- By minimizing distance, we are treating programs as **gradients**
  - Want to converge on this gradient’s **global minima**

- **Problem:** programs are spaghetti code
  - More likely to reach a **local minima** at first
  - Can get stuck really easily on bad paths

- **Solution:** **simulated annealing**
  - Mutate candidate inputs at random
  - Eventually converge on global minima

![Simulated annealing for a global maxima](image-url)
Results

- Unsurprisingly, **significantly faster** than Directed Symbolic Execution
  - **Cool finding**: able to reproduce the HeartBleed bug in 20 minutes!

<table>
<thead>
<tr>
<th>CVE</th>
<th>Fuzzer</th>
<th>Runs</th>
<th>Mean TTE</th>
<th>Median TTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heart</td>
<td>AFLGo</td>
<td>30</td>
<td>19m19s</td>
<td>17m04s</td>
</tr>
<tr>
<td></td>
<td>KATCH</td>
<td>1</td>
<td>&gt; 1 day</td>
<td>&gt; 1 day</td>
</tr>
</tbody>
</table>

Figure 3: Time-to-Exposure (TTE), AFLGo versus KATCH.
Problem: indirect control flow

- **Indirect control-flow edges:**
  - E.g., CALL $R1, JMP $R1

- **Cannot be recovered statically**
  - Destinations resolved only at runtime
  - General case is undecidable
  - Potentially miss shorter paths
Problem: indirect control flow

- **Solution 1:** dynamic control-flow graph
  - Initialize CFG with whatever edges are obtainable statically
  - As fuzzing continues, incorporate indirect edges as they are covered

- **Trade-offs:**
  - Higher runtime overhead
    - Tracking, bookkeeping
  - Only considers seen paths
    - CFG still incomplete
Problem: indirect control flow

- **Solution 2:** value set analysis
  - Statically determine possible values that flow into all indirect calls, jumps

- **Trade-offs:**
  - Very high analysis cost
    - Enumerate all instructions
    - Track all memory accesses
  - Most severely over-approximate
    - E.g., D's set may be all functions
Questions?