Symbolic Execution

Kevin Wallace, CSE504
2010-04-28
Problem

• Attacker-facing code must be written to guard against all possible inputs

• There are many execution paths; not a single one should lead to a vulnerability

• Current techniques are helpful, but have weaknesses
Symbolic Execution

- Insight: code can generate its own test cases
- Run program on ‘symbolic’ input
- When execution path diverges, fork, adding constraints on symbolic values
- When we terminate (or crash), use a constraint solver to generate concrete input
Advantages

• Tests many code paths
• Generates concrete attacks
• Zero false positives
Fuzzing

• Idea: randomly apply mutations to well-formed inputs, test for crashes or other unexpected behavior

• Problem: usually, mutations have very little guidance, providing poor coverage

• if(x == 10) bug(); -- fuzzing has a 1 in $2^{32}$ chance of triggering a bug
Today

- **EXE**
  - Fast - uses a custom constraint-to-SAT converter (STP)
- **Whitebox fuzz testing (SAGE)**
  - Targeted execution - focuses search around a user-provided execution path
EXE: Automatically Generating Inputs of Death
Using EXE

- Mark which regions of memory hold symbolic data
- Instrument code with exe-cc source-to-source translator
- Compile instrumented code with gcc, run
```c
#include <assert.h>

int main(void) {
    unsigned i, t, a[4] = { 1, 3, 5, 2 };
    make_symbolic(&i);
    if(i >= 4)
        exit(0);
    // cast + symbolic offset + symbolic mutation
    char *p = (char *)a + i * 4;
    *p = *p - 1; // Just modifies one byte!

    // ERROR: EXE catches potential overflow i=2
    t = a[*p];
    // At this point i != 2.

    // ERROR: EXE catches div by 0 when i = 1.
    t = t / a[i];
    // At this point: i != 0 && i != 2.

    // EXE determines that neither assert fires.
    if(t == 2)
        assert(i == 1);
    else
        assert(i == 3);
}
```

Mark i as symbolic
```c
#include <assert.h>

int main(void) {
    unsigned i, t, a[4] = { 1, 3, 5, 2 }; 
    make_symbolic(&i);

    if(i >= 4)
        exit(0);

    char *p = (char *)a + i;
    *p = *p - 1; // Just modifies one byte!

    // cast + symbolic offset + symbolic mutation
    int test2 = t/a[i];
    if(i < 4 && i > 0)
        assert(i == 3);
    exit(0);
}
```

Fork, add constraints

- Constraint: \( i \geq 4 \)
  - Result: \( i = 2 \) (line 12)
  - Constraint: \( t = a[i] \) (line 13)
  - Constraint: \( i \neq 2 \)
  - Result: \( i = 3 \) (line 15)
  - Constraint: \( t = t / a[i] \) (line 16)
  - Constraint: \( i \neq 2 \)
  - Result: \( i = 0 \) (line 18)
  - Constraint: \( i = 0 \) (line 19)
  - Result: \( i = 0 \) (line 20)
  - Constraint: \( t == 2 \) (line 21)
  - Result: \( t = 0 \) (line 22)
  - Constraint: \( t == 0 \) (line 23)

- Constraint: \( i < 4 \)
  - Result: \( i = 0 \) (line 18)
  - Constraint: \( i = 0 \) (line 19)
  - Result: \( i = 0 \) (line 20)
  - Constraint: \( t == 2 \) (line 21)
  - Result: \( t = 0 \) (line 22)
  - Constraint: \( t == 0 \) (line 23)
  - Result: \( t = 0 \) (line 24)
Add constraints:
"p equals (char*)a + i * 4"
"p[0]' equals p[0] - 1"
```c
#include <assert.h>
int main(void) {  
  unsigned i, t, a[4] = { 1, 3, 5, 2 };  
  make_symbolic(&i);  
  if(i >= 4)  
    exit(0);  
  // cast + symbolic offset + symbolic mutation  
  char *p = (char *)a + i * 4;  
  *p = *p - 1; // Just modifies one byte!
  // ERROR: EXE catches potential overflow i=2
  t = a[*p];  
  // At this point i != 2.

  // ERROR: EXE catches div by 0 when i = 0.
  t = t / a[i];  
  // At this point: i != 0 && i != 2.

  // EXE determines that neither assert fires.
}
```

**Could cause invalid dereference or division.**

**Fork, add constraints for invalid/valid cases.**
```c
#include <assert.h>

int main(void) {
    unsigned i, t, a[4] = { 1, 3, 5, 2 };
    make_symbolic(&i);
    if(i >= 4) exit(0);
    // cast + symbolic offset + symbolic mutation
    char *p = (char *)a + i * 4;
    *p = *p - 1; // Just modifies one byte!

    // ERROR: EXE catches potential overflow i=2
    t = a[*p];
    // At this point i != 2.

    // ERROR: EXE catches div by 0 when i = 0.
    t = t / a[i];
    // At this point: i != 0 && i != 2.

    // EXE determines that neither assert fires.
    if(t == 2)
        assert(i == 1);
    else
        assert(i == 3);
}
```

Fork, add constraints.
On false branch, emit error
Using exe-cc

% exe–cc simple.c
% ./a.out
% ls exe–last
test1.forks test2.out test3.forks test4.out
test1.out test2.ptr.err test3.out test5.forks
test2.forks test3.div.err test4.forks test5.out
% cat exe–last/test3.div.err
ERROR: simple.c:16 Division/modulo by zero!
% cat exe–last/test3.out
# concrete byte values:
0 # i[0]
0 # i[1]
0 # i[2]
0 # i[3]
% cat exe–last/test3.forks
# take these choices to follow path
0 # false branch (line 5)
0 # false (implicit: pointer overflow check on line 9)
1 # true (implicit: div–by–0 check on line 16)
% cat exe–last/test2.out
# concrete byte values:
2 # i[0]
0 # i[1]
0 # i[2]
0 # i[3]
Constraint solving: STP

• Insight: if memory is a giant array of bits, constraint solving can be reduced to SAT

• Idea: turn set of constraints on memory regions into a set of boolean clauses in CNF

• Feed this into an off-the-shelf SAT solver (MiniSAT)
Caveat - pointers

• STP doesn’t directly support pointers
• EXE takes a similar approach to CCured and tags each pointer with a ‘home’ region
• Double-dereferences resolved with concretization, at the cost of soundness
## STP results

<table>
<thead>
<tr>
<th>Solver</th>
<th>Total Time</th>
<th>Timeouts</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVCL</td>
<td>60,366s</td>
<td>546</td>
</tr>
<tr>
<td>STP (no optimizations)</td>
<td>3,378s</td>
<td>36</td>
</tr>
<tr>
<td>STP (substitution)</td>
<td>1,216s</td>
<td>1</td>
</tr>
<tr>
<td>STP (refinement)</td>
<td>624s</td>
<td>1</td>
</tr>
<tr>
<td>STP (simplifications)</td>
<td>336s</td>
<td>0</td>
</tr>
<tr>
<td>STP (subst+refinement)</td>
<td>513s</td>
<td>1</td>
</tr>
<tr>
<td>STP (simplif+subst)</td>
<td>233s</td>
<td>0</td>
</tr>
<tr>
<td>STP (simplif+refinement)</td>
<td>220s</td>
<td>0</td>
</tr>
<tr>
<td>STP (all optimizations)</td>
<td>110s</td>
<td>0</td>
</tr>
</tbody>
</table>

(Pentium 4 machine at 3.2 GHz, with 2 GB of RAM and 512 KB of cache)
## EXE Results

<table>
<thead>
<tr>
<th></th>
<th>bpf</th>
<th>expat</th>
<th>pcre</th>
<th>tcpdump</th>
<th>udhcpd</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test cases</strong></td>
<td>7333</td>
<td>360</td>
<td>866</td>
<td>2140</td>
<td>328</td>
</tr>
<tr>
<td><strong>None</strong></td>
<td>30.6</td>
<td>28.4</td>
<td>31.3</td>
<td>28.2</td>
<td>30.4</td>
</tr>
<tr>
<td><strong>Caching</strong></td>
<td>32.6</td>
<td>30.8</td>
<td>34.4</td>
<td>27.0</td>
<td>36.4</td>
</tr>
<tr>
<td><strong>Independence</strong></td>
<td>17.8</td>
<td>25.2</td>
<td>10.0</td>
<td>24.9</td>
<td>30.5</td>
</tr>
<tr>
<td><strong>All</strong></td>
<td>10.3</td>
<td>26.3</td>
<td>7.5</td>
<td>23.6</td>
<td>32.1</td>
</tr>
<tr>
<td><strong>STP cost</strong></td>
<td>6.9</td>
<td>24.6</td>
<td>2.8</td>
<td>22.4</td>
<td>23.1</td>
</tr>
</tbody>
</table>

(number of test cases generated, times in minutes on a dual-core 3.2 GHz Intel Pentium D machine with 2 GB of RAM, and 2048 KB of cache)
Results (detail)

<table>
<thead>
<tr>
<th></th>
<th>Symbolic input size (bytes)</th>
<th>bpf</th>
<th>expat</th>
<th>pcre</th>
<th>tcpdump</th>
<th>udhcpd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>96</td>
<td>10</td>
<td>16</td>
<td>84</td>
<td>548</td>
</tr>
<tr>
<td>2</td>
<td>Total statements run (not unique)</td>
<td>298,195</td>
<td>41,345</td>
<td>423,182</td>
<td>40,097</td>
<td>15,258</td>
</tr>
<tr>
<td>3</td>
<td>% of statements symbolic</td>
<td>29.2%</td>
<td>8.5%</td>
<td>34.7%</td>
<td>41.7%</td>
<td>23.6%</td>
</tr>
<tr>
<td>4</td>
<td>Explicit symbolic branch points</td>
<td>77,024</td>
<td>1,969</td>
<td>98,138</td>
<td>11,425</td>
<td>888</td>
</tr>
<tr>
<td>5</td>
<td>% with both branches feasible</td>
<td>11.3%</td>
<td>19.3%</td>
<td>0.9%</td>
<td>19.4%</td>
<td>52.8%</td>
</tr>
<tr>
<td>6</td>
<td>Avg. # symbolic branches per path</td>
<td>38.33</td>
<td>43.44</td>
<td>55.72</td>
<td>103.37</td>
<td>200.14</td>
</tr>
<tr>
<td>7</td>
<td>Symbolic checks</td>
<td>1,490</td>
<td>904</td>
<td>4,451</td>
<td>552</td>
<td>1,535</td>
</tr>
<tr>
<td>8</td>
<td>Pointer concretizations</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Symbolic args. to uninstr. calls</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5: Dynamic counts from EXE execution runs.
Results (detail)

<table>
<thead>
<tr>
<th></th>
<th>bpf</th>
<th>expat</th>
<th>pcre</th>
<th>tcpdump</th>
<th>udhcpd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Symbolic input size (bytes)</td>
<td>96</td>
<td>10</td>
<td>16</td>
<td>84</td>
</tr>
<tr>
<td>2</td>
<td>Total statements run (not unique)</td>
<td>298,195</td>
<td>41,345</td>
<td>423,182</td>
<td>40,097</td>
</tr>
<tr>
<td>3</td>
<td>% of statements symbolic</td>
<td>29.2%</td>
<td>8.5%</td>
<td>34.7%</td>
<td>41.7%</td>
</tr>
<tr>
<td>4</td>
<td>Explicit symbolic branch points</td>
<td>77,024</td>
<td>1,969</td>
<td>98,138</td>
<td>11,425</td>
</tr>
<tr>
<td>5</td>
<td>% with both branches feasible</td>
<td>11.3%</td>
<td>19.3%</td>
<td>0.9%</td>
<td>19.4%</td>
</tr>
<tr>
<td>6</td>
<td>Avg. # symbolic branches per path</td>
<td>38.33</td>
<td>43.44</td>
<td>55.72</td>
<td>103.37</td>
</tr>
<tr>
<td>7</td>
<td>Symbolic checks</td>
<td>1,490</td>
<td>904</td>
<td>4,451</td>
<td>552</td>
</tr>
<tr>
<td>8</td>
<td>Pointer concretizations</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>9</td>
<td>Symbolic args. to uninstr. calls</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5: Dynamic counts from EXE execution runs.
Search heuristics

• Need to limit the number of simultaneously running forked processes
  • (unless you like forkbombs)
• What order do we run forked processes in?
• Currently using a modified best-first search
Search heuristics

![Chart showing basic block coverage vs. number of test cases for BFS and DFS]

Figure 4: Best-first search vs. depth-first search.

![Chart comparing basic block coverage for EXE with best-first search vs. random testing]

Figure 5: EXE with best-first search vs. random testing.
EXE finds real bugs

s[0].code = BPF_STX; // also: (BPF_LDX|BPF_MEM)
s[0].k = 0xffffffffUL;
s[1].code = BPF_RET;

Figure 6: A BPF filter of death

// Code extracted from bpf_validate. Rejects
// filter if opcode’s memory offset is more than
// BPF_MEMWORDS.
// Forgets to check opcodes LDX and STX!
if((BPF_CLASS(p->code) == BPF_ST
 || (BPF_CLASS(p->code) == BPF_LD &&
   (p->code & 0xe0) == BPF_MEM))
   && p->k >= BPF_MEMWORDS )
   return 0;

... // Code extracted from bpf_filter: pc points to current
// instruction. Both cases can overflow mem[pc->k].
case BPF_LDX|BPF_MEM:
   X = mem[pc->k]; continue;
...

case BPF_STX:
   mem[pc->k] = X; continue;

Figure 7: The BPF code Figure 6’s filter exploits.

- FreeBSD BPF accepts filter rules in custom opcode format
- Forgets to check memory read/write offset in some cases, leading to arbitrary kernel memory access
EXE finds real bugs

• 2 buffer overflows in BSD Berkeley Packet Filter
• 4 errors in Linux packet filter
• 5 errors in udhcpcd
• A class of errors in pcre
• Errors in ext2, ext3, JFS drivers in Linux
Automated Whitebox Fuzz Testing
Whitebox fuzz testing

- Insight: valid input gets us close to the interesting code paths
- Idea: execute with valid input, record constraints that were made along the way
- Systematically negate these constraints one-by-one, and observe the results
Example

```c
void top(char input[4]) {
    int cnt=0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 3) abort(); // error
}
```

- With input “good”, we collect the constraints $i_0 \neq b$, $i_1 \neq a$, $i_2 \neq d$, $i_3 \neq !$
- Generate all inputs that don’t match this, choose one to use as next input, repeat
Figure 2. Search space for the example of Figure 1 with the value of the variable \( \text{cnt} \) at the end of each run and the corresponding input string.

Paths of this program can be exercised. If this systematic search is performed in depth-first order, these 16 executions are explored from left to right on the Figure. The error is then reached for the first time with \( \text{cnt} = 3 \) during the 8th run, and full branch/block coverage is achieved after the 9th run.

2.2 Limitations

Systematic dynamic test generation \([16, 7]\) as briefly described above has two main limitations.

Path explosion: systematically executing all feasible program paths does not scale to large, realistic programs. Path explosion can be alleviated by performing dynamic test generation compositionally \([14]\), by testing functions in isolation, encoding test results as function summaries expressed using function input preconditions and output postconditions, and then re-using those summaries when testing higher-level functions. Although the use of summaries in software testing seems promising, achieving full path coverage when testing large applications with hundreds of millions of instructions is still problematic within a limited search period, say, one night, even when using summaries.

Imperfect symbolic execution: symbolic execution of large programs is bound to be imprecise due to complex program statements (pointer manipulations, arithmetic operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision at a reasonable cost. Whenever symbolic execution is not possible, concrete values can be used to simplify constraints and carry on with automated reasoning. Whenever an actual execution path does not match the program path predicted by symbolic execution for a given input vector, we say that a divergence has occurred. A divergence can be detected by recording a predicted execution path as a bit vector (one bit for each conditional branch outcome) and checking that the expected path is actually taken in the subsequent test run.

2.3 Generational Search

We now present a new search algorithm that is designed to address these fundamental practical limitations. Specifically, our algorithm has the following prominent features:

- it is designed to systematically yet partially explore the state spaces of large applications executed with large inputs (thousands of symbolic variables) and with very deep paths (hundreds of millions of instructions);
- it maximizes the number of new tests generated from each symbolic execution (which are long and expensive in our context) while avoiding any redundancy in the search;
- it uses heuristics to maximize code coverage as quickly as possible, with the goal of finding bugs faster;
- it is resilient to divergences: whenever divergences occur, the search is able to recover and continue.

This new search algorithm is presented in two parts in Figures 3 and 4. The main Search procedure of Figure 3 is mostly standard. It places the initial input \( \text{inputSeed} \) in a \( \text{workList} \) (line 3) and runs the program to check whether any bugs are detected during the first execution (line 4). The inputs in the \( \text{workList} \) are then processed (line 5) by selecting an element (line 6) and expanding it (line 7) to generate new inputs with the function...
Limitations

• Path explosion
  • $n$ constraints leads to $2^n$ paths to explore
• Must prioritize
• Imperfect symbolic execution
  • Calls to libraries/OS, pointer tricks, etc. make perfect symbolic execution difficult
### Generational search

#### Code Snippet

```c
1  Search(inputSeed) {
2      inputSeed.bound = 0;
3      workList = {inputSeed};
4      Run&Check(inputSeed);
5      while (workList not empty) { // new children
6         input = PickFirstItem(workList);
7         childInputs = ExpandExecution(input);
8         while (childInputs not empty) {
9            newInput = PickOneItem(childInputs);
10           Run&Check(newInput);
11           Score(newInput);
12           workList = workList + newInput;
13       }
14   }
15 }

1  ExpandExecution(input) {
2      childInputs = {};
3      // symbolically execute (program,input)
4      PC = ComputePathConstraint(input);
5      for (j=input.bound; j < |PC|; j++) {
6         if((PC[0...(j-1)] and not(PC[j])) has a solution I) {
7            newInput = input + I;
8            newInput.bound = j;
9            childInputs = childInputs + newInput;
10       }
11      return childInputs;
12   }
```

- **BFS with a heuristic to maximize block coverage**
- **Score returns the number of new blocks covered**

---

**Systematic dynamic test generation** as briefly described.

**Imperfect symbolic execution:**

Functions that are hard or impossible to reason about symbolically can also help by suggesting concrete values whenever implied, partial symbolic execution. Randomized whenever symbolic execution is not possible, concrete values can be assigned.

Large programs are bound to be imprecise due to complex search periods, say, one night, even when using summaries. Software testing seems promising, achieving full path coverage.

Program paths do not scale to large, realistic programs. Higher-level functions. Although the use of summaries in practice is mostly standard. It places the initial input in isolation, encoding test results as symbolic variables representing values of input preconditions and output post.

**Generational Search**

- BFS with a heuristic to maximize block coverage
- Score returns the number of new blocks covered
ANI bug

Failure to check the length of the second anih record

Was blackbox fuzz tested, but no test case had more than one anih

Zero-day exploit of this bug was used in the wild

Figure 5. On the left, an ASCII rendering of a prefix of the seed ANI file used for our search. On the right, the SAGE-generated crash for MS07-017. Note how the SAGE test case changes the LIST to an additional anih record on the next-to-last line.
Crash triage

• Idea: most found bugs can be uniquely identified by the call stack at time of error

• Crashes are bucketed by stack hash, which includes information about the functions on the call stack, and the address of the faulting instruction
# Results

<table>
<thead>
<tr>
<th>Media 1:</th>
<th>wff-1</th>
<th>wff-1nh</th>
<th>wff-2</th>
<th>wff-2nh</th>
<th>wff-3</th>
<th>wff-3nh</th>
<th>wff-4</th>
<th>wff-4nh</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1 (46)</td>
<td>1 (32)</td>
<td>1(23)</td>
<td>1(12)</td>
<td>1(32)</td>
<td>1(26)</td>
<td>1(13)</td>
<td>1(1)</td>
</tr>
<tr>
<td>ReadAV</td>
<td>1 (40)</td>
<td>1 (16)</td>
<td>2(32)</td>
<td>2(13)</td>
<td>7(94)</td>
<td>4(74)</td>
<td>6(15)</td>
<td>5(45)</td>
</tr>
<tr>
<td>WriteAV</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1(1)</td>
<td>1(3)</td>
<td>1(1)</td>
</tr>
<tr>
<td>SearchTime</td>
<td>10h7s</td>
<td>10h11s</td>
<td>10h4s</td>
<td>10h20s</td>
<td>10h7s</td>
<td>10h12s</td>
<td>10h34s</td>
<td>9h29m2s</td>
</tr>
<tr>
<td>AnalysisTime(s)</td>
<td>5625</td>
<td>4388</td>
<td>16565</td>
<td>11729</td>
<td>5082</td>
<td>6794</td>
<td>5545</td>
<td>7671</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Media 1:</th>
<th>wff-5</th>
<th>wff-5nh</th>
<th>bogus-1</th>
<th>bogus-1nh</th>
<th>bogus-2</th>
<th>bogus-3</th>
<th>bogus-4</th>
<th>bogus-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1(25)</td>
<td>1(15)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ReadAV</td>
<td>3(44)</td>
<td>3(56)</td>
<td>3(3)</td>
<td>1(1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WriteAV</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SearchTime</td>
<td>10h8s</td>
<td>10h4s</td>
<td>10h8s</td>
<td>10h14s</td>
<td>10h29s</td>
<td>9h47m15s</td>
<td>5m23s</td>
<td>5m39s</td>
</tr>
<tr>
<td>AnalysisTime(s)</td>
<td>21614</td>
<td>22005</td>
<td>11640</td>
<td>13156</td>
<td>3885</td>
<td>4480</td>
<td>214</td>
<td>234</td>
</tr>
</tbody>
</table>
Results

Figure 8. Histograms of test cases and of crashes by generation for Media 1 seeded with wff-4.

Most crashes found within a few generations.
Discussion

• Generational search is better than DFS
• Bogus files find few bugs
• Different files find different bugs
• Block coverage heuristic doesn’t help much
  • Generation much better heuristic
Comparison

- Generational search vs. modified BFS
- Bad input is usually only a few mutations away from good
- Incomplete search, but can effectively find bugs in large applications without source
- EXE closer to sound - how much does this matter?