Execution generated Executions:
Automatically generating inputs of death.

Dawson Engler
Cristian Cadar, Junfeng Yang, Can Sar, Paul Trowey
Stanford University

Goal: find many bugs in systems code

- Generic features:
  - Baroque interfaces, tricky input, rats nest of conditionals.
  - Enormous undertaking to hit with manual testing.

- Random "fuzz" testing
  - Charm: no manual work
  - Blind generation makes hard to hit errors for narrow input range.
  - Also hard to hit errors that require structure

This talk: a simple trick to finesse.

EXE: Execution generated Executions

- Basic idea: use the code itself to construct its input!
- Basic algorithm:
  Symbolic execution + constraint solving.

  Run code on symbolic input, initial value = "anything"
  As code observes input, it tells us values input can be.
  At conditionals that use symbolic input, fork
    - On true branch, add constraint that input satisfies check
    - On false that it does not.

- exit() or error: solve constraints for input.
- Rerun on uninstrumented code = No false positives.
- IF complete, accurate, solvable constraints = all paths!

The toy example

```c
int bad_abs(int x) {
  if(x < 0)
    return -x;
  else
    return x;
}
```

Initial state: x unconstrained
Code will return 3 times.
Solve constraints at each return = 3 test cases.

```c
int bad_abs_exe(int x) {
  if(x < 0)
    return -x;
  else
    return x;
}
```

Isn't exponential expensive?

- Only fork on symbolic branches.
  - Most concrete (linear).
- Loops? Heuristics.
  - Default: DFS. Linear processes with chain depth.
  - Can get stuck.
  - "Best first" search: chose branch, backtrack to point that will run code hit fewest times.
  - Can do better...
- However:
  - Happy to let run for weeks as long as generating interesting test cases. Competition is manual and random.

The mechanics

- User marks input to treat symbolically using either:
  ```c
  void make_symbolic(char **obj);
  void make_symbolic_bytes(void *bytes, unsigned nbytes);
  ```
- Compile with EXE compiler, exe-cc. Uses CIL to
  - Insert checks around every expression: if operands all concrete, run as normal. Otherwise, add as constraint
  - Insert fork calls when symbolic could cause multiple acts
  - ./a.out: forks at each decision point.

  When path terminates use STP to solve constraints.
  Terminates when: (1) exit, (2) crash, (3) EXE detects err
  - Rerun concrete through uninstrumented code.
Where we're going and why.

- One main goal:
  - At any point on program path have accurate, complete set of constraints on symbolic input.
- "IF* EXE has and can solve THEN"
  - Can drive execution down all paths.
  - Can use path constraints to check if any input value exists that causes error such as div 0, deref NULL, etc.
- Entire motivation: all path => all value for much code.
- Next:
  - Mechanics of supporting symbolic execution
  - Universal checks.
  - Results.

Mixed execution

- Basic idea: given expression (e.g., deref, ALU op)
  - If all of its operands are concrete, just do it.
  - If any are symbolic, add as constraint.
  - If current constraints are impossible, stop.
  - If current path hits error or exit(), solve emit.
  - If calls uninstrumented code: do call, or solve and do call
- Example: "x = y + z"
  - If y, z both concrete, execute. Record x = concrete.
  - Otherwise set "x = y + z", record x = symbolic.
- Result:
  - Most code runs concretely: small slice deals w/ symbols.
  - Robust: do not need all source code (e.g., OS). Just run

Untyped memory

- C code observes memory in multiple ways
  - Signed to unsigned casts
  - Cast array of bytes to inode, superblock, pkt header
- Soln:
  - Cannot bind types to memory, must do to expressions
  - Represent symbolic memory using STP primitives: array of 8-bit bitvectors.
  - Bitvector=untyped, array=pointers (next)
  - Each read of memory generates constraints based on static type of read. Does not persist. Just encoded in constraint.

Symbolic memory expressions.

- Given an array of "a" of size "n" and in-bounds index "i",
  "a[i] = 0" becomes

<table>
<thead>
<tr>
<th>Condition</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i == 0 &amp;&amp; a[0] == 0)</td>
<td>(i == 1 &amp;&amp; a[1] == 0)</td>
</tr>
</tbody>
</table>

"a[i] = 4" could update any entry.
- Soln: map to STP array (translates to SAT).
  - Given "a[i]" where "i" is symbolic (other cases similar)
  - If "a" has no symbolic counterpart create one, "a_sym"
  - Record "a" corresponds to "a_sym"
  - Build constraints using a_sym[i_sym]

Example: symbolic memory reads and writes

```c
1: #include <assert.h>
2: int main() {
3:   unsigned char i, j, k, a[8] = {11, 13, 17, 19};
4:   make_symbolic(&i); // these macros make
5:   make_symbolic(&j); // i, j, and k
6:   make_symbolic(&k); // symbolic
7:   if (i >= 4 && j >= 4 && k >= 4) // force in-bounds
8:     exit(0);
9:   a[i] = 1;
10:  if ( (a[i] + a[8] == 14) )
11:    assert(i <= 5));
12: }
```

Example: symbolic memory reads and writes

```c
1: #include <assert.h>
2: int main() {
3:   unsigned char i, j, k, a[8] = {11, 13, 17, 19};
4:   make_symbolic(&i); // these macros make
5:   make_symbolic(&j); // i, j, and k
6:   make_symbolic(&k); // symbolic
7:   if (i >= 4 && j >= 4 && k >= 4) // force in-bounds
8:     exit(0);
9:   a[i] = 1;
10:  if ( (a[i] + a[8] == 14) )
11:    assert(i <= 5));
12: }

A non-taken soln:
  i = 0 && k == 2
```

mixed content
"accurately tracking constraints that only involve strongly typed scalars is relatively simple: just name the variables uniquely (as in the original source) and use these names consistently in the original source."

Dawson Engler, 3/20/2006
Automatic, systematic corner cases hitting

- Conditional: fork, both branches.
- Overflow: can "\(x + y\), "\(x - y\), "\(x * y\) ... overflow? Build two symbolic expressions
  
  E1: expression at precision of ANSI C’s expression types.
  E2: expression at essentially infinite precision.

  If E1 could be different than E2, force it.

  ```c
  if(query(E1 != E2) == satisfiable) {
    if(fork() == child)
      add_constraint(E1 == E2);
    else
      add_constraint(E1 != E2);
  }
  ```

- Others: truncation casts, signed->unsigned.

Universal checks.

- Key: Symbolic reasons about many possible values simultaneously. Concrete about just current ones.
- Universal checks:
  
  When reach dangerous op, EXE checks if any input exists that could cause it to blow up.

  Built-in: div/mod by 0, NULL * p, memory overflow.

  ```c
  sym_expr div_transformation(sym_expr x, sym_expr y)
  if(query(y != 0) == satisfiable)
    if(fork())
      add_constraint(y != 0);
    return symbolic_expression(x / y);
  else
    add_constraint(y == 0);
    terminate_with_error("Found div by 0 in\n  ```

Generalized checking.

- "assert(sym_expr)"
  
  EXE will systematically try to violate sym_expr.

  Complete, accurate, solved path constraints = verification

- Scales with sophistication of correctness checks.

  E.g., given f and inv can verify correct: inv(f(x)) = x.

```
#include <assert.h>
#include <verif/lnk.h>
void main(void) {
  int x;
  make_symbolic(x);
  assert(hnotl(hnotl(x)) == x);
}
```

Putting it all together

```
int main(void) {
  unsigned i, x[4] = {1, 0, 5, 2};
  make_symbolic(i);
  // ERROR: EXE catches potential overflow i >> 4.
  // ERROR: EXE catches implicit division by i when i = 1.
  // Demonstrate simple gross casting (EXE handles arithmetic)
  // casting but not expression we only show simple casting)
  t = {unsigned}(t); // here // the value of t is equal to (unsigned) a[i] at this point
  // ERROR: EXE catches buffer overflow when i = 2
  return a[2](unsigned *)[0]; // Same as: return a[i][0];
}
```

Limits

- Missed constraints:
  
  If call asm, or CIL cannot catch file.

  STP cannot do div/mod: constraint to be power of 2, shift, mask respectively.

  Cannot handle **p where "p" is symbolic: must concretize *p. (Note: **p still symbolic.)

  Stops path if cannot solve; can get lost in exponentials.

- Missing:
  
  No symbolic function pointers, symbols passed to variables not tracked.

  No floating point.

  long long support is erratic.

Talk overview

- Goal: complete, accurate constraints on input.
- "IF" can do so, THEN:
  
  Automatic all path coverage.

  All value checking. (Sometimes verification)

  Limits: missed constraints, NP-hard problem, loops.

- Does it work? Next.

  Automatic generation of malicious disks.

  Automatic generation of inputs of death.
Automatically generating malicious disks.

- File systems:
  - Mount untrusted data as file systems (CD-rom, USB)
  - Let untrusted users mount files as file systems.
- Problem: bad people.
  - Must check disk as aggressively as networking code.
  - More complex.
  - FS guys are not paranoid.
  - Hard to random test: 40 if-statements of checking.
  - Result: easy exploits.
- Basic idea:
  - Make disk symbolic, jam up through kernel.
  - Cool: automatically make disk image to blow up kernel!

Checking Linux FSes with EXE

- Why UML?
  - Hard to cut Linux FS out of kernel. UML=check in situ.
  - Need to clone/wait for process.
  - Hard to debug OS on raw machine.
- Hacks to get Linux working
  - Disable threading
  - Replace asm functions (strlen, memcpy) with EXE versions
  - UML linked at fixed (too small) location. Stripped down.
  - CIL could not handle 8 files. Compiled with gcc.
- Hacks to EXE:
  - \( v = e \), with \( v \) symbolic: do not make \( v \) symbolic if \( e = e \)
  - No free of symbolic heap-allocated objects.

A galactic view [Oakland’06]

Results

- Ext2:
  - Four bugs.
  - One buffer overflow = \( r/w \) arbitrary kernel memory
  - Three = kernel crash.
- Ext3:
  - Four bugs (copied from ext2)
- JFS:
  - One null pointer dereference.

Generated disk for JFS, Linux 2.4.27.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hex Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>08000</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08010</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08020</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08030</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08040</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10000</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

Create 64K file, set 64th sector to above. Mount.

BPF, Linux packet filters

- "We'll never find bugs in that"
  - Some of most heavily audited, best written open source
  - Easy to pull out of kernel.
- Mark filter: packet as symbolic.
  - Symbolic = turn check into generator of concretes.
  - Safe filter check: generates all valid filters of length \( N \).
  - Interpreter: will produce all valid filter programs that pass check of length \( N \).
  - Filter on message: generates all packets that accept, reject.
- Results!
Results: BPF, trivial exploit.

Check that memory operations only use valid addresses.

<table>
<thead>
<tr>
<th>Check for xent LDX,STX!</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
</tr>
</tbody>
</table>

| ![Image](image2.png) |

| ![Image](image3.png) |

| ![Image](image4.png) |

Linux Filter

- **Generated filter:**
  ```c
  if (BPFCLASS(p->code) == BPF_ST) || (BPFCLASS(p->code) == BPF_LD & &
  (p->code & (15<<4)) == BPF_MEM)) & & p->k >= BPF_MEMWORDS )
  return 0;
  ```

- offset=s[0] k passed in; len=2,4
  ```c
  void * skb_header_pointer(struct sk_buff * skb, int offset, int len,
  int blen = skb_headerlen(skb);
  if (offset + len < blen)
  return skb->data + offset;
  ```

Conclusion [Spin'05, Oakland'06]

- **Automatic all-path execution, all-value checking**
  - Make input symbolic. Run code. If operation concrete, do it. If symbolic, track constraints. Generate concrete solution at end (or on way), feedback to code.
  - Finds bugs in real code.
  - Zero false positives.
  - But, still very early in research cycle.

- **Three ways to look at what's going on**
  - Grammar extraction.
  - Turn code inside out from input consumer to generator.
  - Sort of Heisenberg effect: observations perturb symbolic inputs into increasingly concrete ones. More definitive observation = more definitive perturbation.

Future work

- **Automatic “hardening”**
  - Assume: EXE finds error and has accurate, complete path constraints.
  - Then: can translate constraints to if-statements and reject concrete input that satisfies.
  - Example: wrap up disk reads. “Cannot mount.” Or reject network packets that crash system.

- **Automatic exploit generation.**
  - Compile Linux with EXE. Mark data from copy_from_user as symbolic. (System call params if fancy)
  - Find paths to bugs.
  - Generate concrete input + C code to call kernel.
  - Mechanized way to produce exploits.