EXecution generated Executions: Automatically generating inputs of death.

Dawson Engler
Cristian Cadar, Junfeng Yang, Can Sar, Paul Twohey
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

```
int bad_abs(int x) {
   if(x < 0)
      return -x;
   if(x == 12345678)
      return -x;
   return x;
}
```

- 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

```
int bad_abs(int x) {
   if(x < 0)
      return -x;
   if(x == 12345678)
      return -x;
   return x;
}
```

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

```
int bad_abs_exe(int x) {
   if(fork() == child)
        constrain(x < 0);
        return -x;
   else
        constrain(x >= 0);

   if(fork() == child)
     constrain(x == 12345678);
     return -x;
   else
     constrain(x != 12345678);
   return x;
}
```

The mechanics

- User marks input to treat symbolically using either:

```
void make_symbolic(T *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.

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.

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/ symbolics.
  - Robust: do not need all source code (e.g., OS). Just run

DE2

Untyped memory

- C code observes memory in mutiple 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 array of "a" of size "n" and in-bounds index "i".
  - "(a[i] == 0)" becomes

```
   (i == 0 && a[0] == 0)
|| (i == 1 && a[1] == 0)
|| ..
|| (i == n-1 && a[n-1] == 0)
```

  - "a[i] = 4" could update any entry.
- Sol'n: 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

```
#include <assert.h>
int main() {
    unsigned char i, j, k, a[4] = {11, 13, 17, 19};
    make_symbolic(&i); // these macros make
    make_symbolic(&j); // i, j, and k
    make_symbolic(&k); // symbolic
    if(i >= 4 || j >= 4 || k >= 4) // force in-bounds
        exit(0);
    a[i] = 1;
    if ( (a[j] + a[k] == 14) )
        assert((i != 1));
}
```

Example: symbolic memory reads and writes

```
#include <assert.h>
int main() {
    unsigned char i, j, k, a[4] = {11, 13, 17, 19};
    make_symbolic(&i); // these macros make
    make_symbolic(&j); // i, j, and k
    make_symbolic(&k); // symbolic
    if(i >= 4 || j >= 4 || k >= 4) // force in-bounds
        exit(0);
    a[i] = 1;
    if ( (a[j] + a[k] == 14) )
        assert((i != 1));
}
```

taken branch:
i != 1 && k == 1

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

Slide 9

DE2 "accurately tracking constraints that only involve strongly typed scalars is realtively simple: just name the variables uniquely (as in the original source) and use these names consistenly 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.

```
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.
  - Builtin: div/mod by 0, NULL *p, memory overflow.

```
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!\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 <netinet/in.h>
void main(void) {
    int x;
    make_symbolic(x);
    assert(htonl(ntohl(x)) == x);
}
```

Putting it all together

```
int main(void) {
  unsigned i, t, a[4] = { 1, 0, 5, 2 };
  make_symbolic(&i);

  // ERROR: EXE catches potential overflow i >= 4.
  t = a[i];
  // At this point i < 4.

  // ERROR: EXE catches div by 0 when i = 1.
  t = t / a[i];
  // At this point i != 1 && i < 4.

  // Demonstrate simple gross casting (EXE handles arbitrary
  // casting but for exposition we only show simple casting)
  t = (unsigned)&a[0];
  t += i*4;
  // the value of t is equal to &a[i] at this point.

  // ERROR: EXE catches buffer overflow when i = 2.
  return a[*(unsigned *)t];       // Same as: return a[a[i]];
}
```

Limits

- Missed constraints:
  - If call asm, or CIL cannot eat 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, symbolics passed to varargs not tracked.
  - No floating point.
  - long long support is erratic.

Talk overview

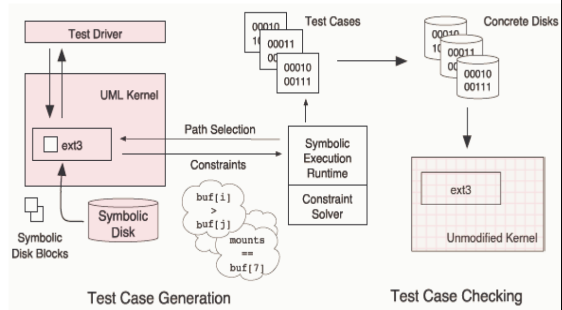
- Goal: complete, accurate constraints on input.
- *IF* can do so, THEN:
  - Automatic all path coverage.
  - All value checking. (Sometimes verification)

  - 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!

A galactic view [Oakland'06]



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 @ fixed (too small) location. Stripped down.
  - CIL could not handle 8 files. Compiled with gcc.
- Hacks to EXE:
  - v = e, with e symbolic: do not make v symbolic if e == val
  - No free of symbolic heap-allocated objects.

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.

Offset	*Hex Values*
00000	0000 0000 0000 0000 0000 0000 0000 0000
. . .	. . .
08000	464a 3153 0000 0000 0000 0000 0000 0000
08010	1000 0000 0000 0000 0000 0000 0000 0000
08020	0000 0000 0100 0000 0000 0000 0000 0000
08030	e004 000f 0000 0000 0002 0000 0000 0000
08040	0000 0000 0000 0000 0000 0000 0000 0000
. . .	. . .
10000	

Create 64K file, set 64^{th} 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 uses valid addresses.
// => Check forgets LDX,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;
```

```
case BPF_LDX|BPF_MEM:
     X = mem[pc->k]; continue;
. . .
case BPF_STX:
     mem[pc->k] = X; continue;
```

Linux Filter

- Generated filter:

```
// other filters that cause this error. . .
//   => BPF_LD|BPF_B|BPF_IND
//   => BPF_LD|BPF_H|BPF_IND
s[0].code = BPF_LD|BPF_B|BPF_ABS;
s[0].k      = 0x7fffffffUL;
s[1].code = BPF_RET;
s[1].k      = 0xfffffff0UL;
```

- offset=s[0].k passed in; len=2,4

```
nline void * skb_header_pointer(struct sk_buff *skb, int offset, int len,
  int hlen = skb_headlen(skb);
  if (offset + len <= hlen)
          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), feed back 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.