Directed Automated Randomized Testing (DART)

Motivation

Verification is *really* hard

Unit testing is also hard and rarely done properly

- Have to check all corner cases
- Have to simulate external environment
- Have to set up a driver

Static analysis is imprecise

- Tools like lint generate a lot of false positives

What does DART do?

- Automatically extracts a programs interface
- Automatically generates a test driver for all externally visible functions
- Automatically performs randomized testing

Randomized testing produces poor coverage

```
int f(int x) { return 2 * x; }
int h(int x, int y) {
  if (x != y)
    if (f(x) == x + 10)
      abort(); /* error */
  return 0;
}
```

Will happen w.h.p.

Hard to do

Want a solution to
x + 10 == 2*x
x=10

Overview

1. Start with randomized input
2. Determine predicates that must be satisfied to enter conditionals
3. Generate new input satisfying these constraints
4. Repeat until all paths have been traversed

Program Model

Random Access Memory (RAM) Machine:

- A Memory *M* is a mapping between address and 32 bit words
- + denotes updating; $M' = M + [m \rightarrow v]$ means replace value at m with v

DART models

- Symbolic memory *S*, which maps addresses to expressions
- Concrete memory *M*, which maps addresses to concrete values

A program consists of statements which can either be:

- Assignment
- Conditional

The instrumented program

case $(m \leftarrow e)$:
$\mathcal{S} = \mathcal{S} + [m \mapsto \mathit{evaluate\_symbolic}(e, \mathcal{M}, \mathcal{S})]$
$v = \mathit{evaluate\_concrete}(e, \mathcal{M})$
$\mathcal{M} = \mathcal{M} + [m \mapsto v]; \ell = \ell + 1$

Update symbolic memory

Update concrete memory / PC

case (if (e) then goto ℓ'):
$b = \mathit{evaluate\_concrete}(e, \mathcal{M})$
$c = \mathit{evaluate\_symbolic}(e, \mathcal{M}, \mathcal{S})$
if b **then**
  $\mathit{path\_constraint} = \mathit{path\_constraint} \;\hat{}\; \langle c \rangle$
  $\mathit{stack} = \mathit{compare\_and\_update\_stack}(1, k, \mathit{stack})$
  $\ell = \ell'$
else
  $\mathit{path\_constraint} = \mathit{path\_constraint} \;\hat{}\; \langle \mathit{neg}(c) \rangle$
  $\mathit{stack} = \mathit{compare\_and\_update\_stack}(0, k, \mathit{stack})$
  $\ell = \ell + 1$
$k = k + 1$

The instrumented program

ase $(m \leftarrow e)$:
  $\mathcal{S} = \mathcal{S} + [m \mapsto evaluate\_symbolic(e, \mathcal{M}, \mathcal{S})]$
  $v = evaluate\_concrete(e, \mathcal{M})$
  $\mathcal{M} = \mathcal{M} + [m \mapsto v]$; $\ell = \ell + 1$
ase (if (e) then goto ℓ'):
  $b = evaluate\_concrete(e, \mathcal{M})$
  $c = evaluate\_symbolic(e, \mathcal{M}, \mathcal{S})$
  if b **then**
    $path\_constraint = path\_constraint \mathbin{\hat{}} \langle c \rangle$
    $stack = compare\_and\_update\_stack(1, k, stack)$
    $\ell = \ell'$
  else
    $path\_constraint = path\_constraint \mathbin{\hat{}} \langle neg(c) \rangle$
    $stack = compare\_and\_update\_stack(0, k, stack)$
    $\ell = \ell + 1$
  $k = k + 1$

Record a list of all constraints taken to get to this conditional

Check to ensure that we're on the expected path and record if given conditionals are "done"

The stack

Kept as a record of execution so far

Stores two pieces of information for each conditional

- The branch taken (if = 1, else = 0)
- Whether the if and else branch have been explored (done)

Enables depth-first exploration of conditionals

Updating the stack

compare_and_update_stack(*branch*, *k*, *stack*) =

if $k < |stack|$ **then**

  if $stack[k].branch \neq branch$ **then**

    $forcing\_ok = 0$

    raise an exception

  else if $k = |stack| - 1$ **then**

    $stack[k].branch = branch$

    $stack[k].done = 1$

else $stack = stack \mathbin{\hat{}} \langle (branch, 0) \rangle$

return *stack*

All other conditionals exception the one of interest should take the same branch as the previous execution

If we successfully reached the branch we were shooting for, that conditional is done

New conditionals are simply push on the top of the stack

Solving for new path

olve_path_constraint(k_{try},*path_constraint*,*stack*) =
  let j be the smallest number such that
    for all h with $-1 \leq j < h < k_{try}$, $stack[h].done = 1$
  if $j = -1$ **then**
    return (0, _, _) // This directed search is over
  else
    $path\_constraint[j] = neg(path\_constraint[j])$
    $stack[j].branch = \neg stack[j].branch$
    if ($path\_constraint[0, \ldots, j]$ has a solution $\vec{I}'$) **then**
      return $(1, stack[0..j], \vec{I} + \vec{I}')$
    else
      solve_path_constraint(j,*path_constraint*,*stack*)

Find the first conditional
has not been fully explor

Flip the conditional to take
the opposite branch

Overall Algorithm

run_DART () =
  all_linear, *all_locs_definite*, *forcing_ok* = 1, 1, 1
  repeat
    stack = $\langle\rangle$; $\vec{I}$ = [] ; *directed* = 1
    while (*directed*) **do**
      try (*directed*, *stack*, $\vec{I}$) =
        instrumented_program(*stack*, $\vec{I}$)
      catch any exception $\rightarrow$
        if (*forcing_ok*)
          print "Bug found"
          exit()
        else *forcing_ok* = 1
  until *all_linear* $\wedge$ *all_locs_definite*

Advantages over static analysis

```
foobar(int x, int y){
  if (x*x*x > 0){
    if (x>0 && y==10)
      abort();
  } else {
    if (x>0 && y==20)
      abort();
  }
}
```

Can function even when theorem solvers fail

```
struct foo { int i; char c; }
bar (struct foo *a) {
    if (a->c == 0) {
      *((char *)a + sizeof(int)) = 1;
      if (a->c != 0)
        abort();
    }
}
```

Can handle aliasing

Limitations

- Incomplete in the presence of non-linear path constraints
  - e.g., x*x
  - all_linear = 0 -> DART will run forever
- Library functions
  - Can be explored via execution
  - Can't be used to form path constraints; e.g., x = libFun(); if(x){} else {}

Results

Needham-Schoeder Protocol

- Protocol for handshake
- Has a known security vulnerability (man in the middle)

oSIP

- Was able to crash 65% of the external functions
- Most of these turned out to be due to non-uniform handling of NULL
- Found a security vulnerability that caused the parser to crash

Discussion

- Their results on real oSIP aren't very motivating
  - Most of the errors are null pointers
  - How successful would DART be on coreutils?
- Can DART be applied to incremental codes changes?