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 program's interface
- Automatically generates a test driver for all externally visible functions
- Automatically performs randomized testing
Randomized testing produces poor coverage

```c
int f(int x) { return 2 * x; }
int h(int x, int y) {
  if (x != y) {
    if (f(x) == x + 10)
      abort();
  }
  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; $M' = M + [m \to v]$ 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

\[
\begin{align*}
\text{case } (m \leftarrow e): & \quad \text{Update symbolic memory} \\
S & \leftarrow S + [m \mapsto \text{evaluate-symbolic}(e, \mathcal{M}, S)] \\
v & \leftarrow \text{evaluate-concrete}(e, \mathcal{M}) \\
\mathcal{M} & \leftarrow \mathcal{M} + [m \mapsto v]; \ell = \ell + 1 \\
\text{case } (\text{if } (e) \text{ then goto } \ell'): & \quad \text{Update concrete memory / PC} \\
b & \leftarrow \text{evaluate-concrete}(e, \mathcal{M}) \\
c & \leftarrow \text{evaluate-symbolic}(e, \mathcal{M}, S) \\
\text{if } b \text{ then} & \\
\text{path-constraint} & \leftarrow \text{path-constraint} \land \langle c \rangle \\
\text{stack} & \leftarrow \text{compare-and-update-stack}(1, k, \text{stack}) \\
\ell & \leftarrow \ell' \\
\text{else} & \\
\text{path-constraint} & \leftarrow \text{path-constraint} \land \langle \neg(c) \rangle \\
\text{stack} & \leftarrow \text{compare-and-update-stack}(0, k, \text{stack}) \\
\ell & \leftarrow \ell + 1 \\
k & \leftarrow k + 1
\end{align*}
\]
The instrumented program

```
\begin{align*}
\text{case } (m \leftarrow e): \\
S &= S + [m \mapsto \text{evaluate\_symbolic}(e, M, S)] \\
v &= \text{evaluate\_concrete}(e, M) \\
M &= M + [m \mapsto v]; \ell = \ell + 1 \\
\text{case } (if\ (e)\ then\ goto\ \ell'):\ \\
b &= \text{evaluate\_concrete}(e, M) \\
c &= \text{evaluate\_symbolic}(e, M, S) \\
\text{if } b \text{ then } \\
\quad \text{path\_constraint} = \text{path\_constraint} \ ^\wedge \langle c \rangle \\
\quad \text{stack} = \text{compare\_and\_update\_stack}(1, k, \text{stack}) \\
\quad \ell = \ell' \\
\text{else } \\
\quad \text{path\_constraint} = \text{path\_constraint} \ ^\wedge \langle \text{neg}(c) \rangle \\
\quad \text{stack} = \text{compare\_and\_update\_stack}(0, k, \text{stack}) \\
\quad \ell = \ell + 1 \\
k &= k + 1
\end{align*}
```

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

```python
compare_and_update_stack(branch, k, stack) =
    if k < |stack| then
        if stack[k].branch ≠ 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 \ (branch, 0)
    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

```python
solve_path_constraint(k_{try}, path_constraint, stack) =
  let j be the smallest number such that
    for all h with -1 ≤ 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 = ~stack[j].branch
    if (path_constraint[0, ..., 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 that has not been fully explored.
Flip the conditional to take the opposite branch.
Overall Algorithm

\[
\text{run}_\text{DART}() = \\
\text{all\_linear, all\_locs\_definite, forcing\_ok} = 1, 1, 1 \\
\text{repeat} \\
\quad \text{stack} = \langle \rangle; \vec{I} = []; \text{directed} = 1 \\
\text{while} (\text{directed}) \text{ do} \\
\quad \text{try} (\text{directed, stack, } \vec{I}) = \\
\quad \quad \text{instrumented\_program}(\text{stack, } \vec{I}) \\
\quad \text{catch any exception} \rightarrow \\
\quad \quad \text{if} (\text{forcing\_ok}) \\
\quad \quad \quad \text{print} \text{“Bug found”} \\
\quad \quad \quad \text{exit()} \\
\quad \quad \text{else forcing\_ok} = 1 \\
\text{until all\_linear} \land \text{all\_locs\_definite}
Advantages over static analysis

```c
1  foobar(int x, int y){
2     if (x*x*x > 0){
3         if (x>0 && y==10)
4             abort();
5         } else {
6             if (x>0 && y==20)
7                 abort();
8         }
9     }
```

```c
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 function even when theorem solvers fail

Can handle aliasing
Limitations

Incomplete in the presence of non-linear path constraints

• e.g., $x^2$
• all_linear = 0 -> DART will run forever

Library functions

• Can be explored via execution
• Can’t be used to form path constraints; e.g., $x = \text{libFun}();$ 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?