DART: Directed Automated Random Testing

김효림

Contents

Abstract

- DART: Directed Automated Random Testing
- Main Techniques
  - Automated Extraction of the interface of a program with its external environment using static source-code parsing
  - Automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in
  - Dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths

Abstract

- Main Strength
  - Performed completely automatically on any program that compiles
    - No need: test driver, harness code
  - Detects standard errors
    - Program crashes
    - Assertion violations
    - Non-termination

1. Introduction

- Testing
  - The primary way to check the correctness of software
  - Cost: Software failure > Software Test
- Unit Testing
  - Detect errors in the component's logic
  - Check all corner cases
  - Provide 100% code coverage
  - But,
    - Hard and expensive to perform
      - Need to write test driver/harness code to simulate the environment of the component
      - Often either performed very poorly or skipped
- Subsequent phases of testing(feature, integration and system testing)
  - Not to check the corner cases where bugs causing reliability issues are typically hidden

1. Introduction - cont.

- DART: Directed Automated Random Testing
  - Automate unit testing of software
- Main Techniques
  - Automated Extraction of the interface of a program with its external environment using static source-code parsing
  - Automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in
  - Dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths

1. Introduction - cont.

- Main Strength
  - Performed completely automatically on any program that compiles
    - No need: test driver, harness code
  - Detects standard errors
    - Program crashes
    - Assertion violations
    - Non-termination
- Preliminary experiments to unit test examples
  - Needham-Schroeder's security protocol
  - oSIP library

1. Introduction - cont.

- The idea of extracting automatically interfaces of software components via static analysis
  - Model-checking purposes
  - Reverse engineering
  - Compositional verification
- Combine automatic interface extraction with random testing and dynamic test generation
  - DART
    - Test management tools

1. Introduction - cont.

- Random testing
  - Simple and well-known technique
  - Can be remarkably effective at finding software bugs
  - Usually provide low code coverage
    - If (x ~)-then branch of the conditional statement
      - Has only one chance to be exercised out of 2^32 if x is a 32bit integer(randomly initialized input)
- DART
  - Makes random testing automatic by combining it with automatic interface extraction
  - Makes it much More effective in finding errors thanks to the use of dynamic test generation to drive the program along alternative conditional branches

1. Introduction - cont.

- Code Inspection
  - The way to check correctness during the software development cycle
  - Static source-code analysis for building automatic code-inspection tools
    - Prefix/Prefast
    - MC
    - Klocwork
    - Polyspace
    - lint (static checker)
      - Generate an overly large number of warnings and false alarms
      - Rarely used by programmers on a regular basis

1. Introduction - cont.

- Main challenge faced by the new generation of static analyzers
  - To do a better job in dealing with false alarms, which arise from the inherent imprecision of static analysis
    - Report only high-confidence warnings(at the risk of missing some actual bugs)
    - Report all of them(at the risk of overwhelming the user)
  - Static analysis still need significant human intervention
- DART
  - Based on dynamic analysis and fully automated

2. DART Overview

- Integration of random testing and dynamic test generation
  - Using Symbolic reasoning

2.1 An Introduction to DART

- Function h: defective
  - Abort statement for some value of its input vector(parameter x or y or both)
- Typical of random testing
  - Difficult to generate input values that will drive the program through all its different execution path
- DART
  - Dynamically gather knowledge about the execution of the program
    - Directed search
  - Start with random input
  - Calculate during each execution an input vector for the next execution
    - This vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution
  - The new input vector attempts to force the execution of the program through a new path
  - Repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths

```
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;
}
```

<code> Function h

2.1 An Introduction to DART - cont.

Repeat & Find all its feasible execution paths

Random input	→	Calculate an input vector for the next execution (go to new Path Constraint)	→	Attempts to force the execution through a new path	(Termination)

(Abort)

- Path Constraint: an equivalence class of input vectors, all input vectors that drive the program through the path that was just executed
  - To force the program through a different equivalence class (with Equivalence Partitioning)
    - Negating the last predicate of the current path constraint

2.2 Execution Model

- Under test(concretely, executing the actual program *P*)
- Random input / Path constraints
- Symbolically(on values at memory locations expressed in terms of input parameters)
  - $\mathcal{M}$: memory (domain)(mapping from memory addresses m)
  - m: memory address (codomain(range))
  - +: update
    - $\mathcal{M}' = \mathcal{M} + [m \rightarrow v]$ == $\mathcal{M}'(m) = v$
  - e : set of addresses *m*
  - P: program
    - Defines
      - $\vec{M}_0$: a sequence of input addresses
      - the addresses of the input parameters of P
  - $\vec{I}$: input vector
    - the initial value of $\vec{M}_0$ and hence $\mathcal{M}$
    - associates a value to each input parameter

2.2 Execution Model - cont.

- l: statement of address(other than abort or halt)
    - Initial address l_0
- S : mapped memory address
- c: constance
- $*(e', e'')$: a dyadic term denoting multiplication
- $\leq(e', e'')$: a term denoting comparison
- $\neg(e')$: a monadic term denoting negation
- $*e'$: a monadic term denoting pointer dereference
- $evaluate\_concrete(e, \mathscr{M})$: evaluates expression e in context $\mathscr{M}$ and return a 32-bit value for e
- $statement\_at(l, \mathscr{M})$: the next statement to be executed

2.2 Execution Model - cont.

- *Program execution w*
  - **C**: the set of conditional statements
  - **A**: the set of assignment statements in P
  - finite sequence in **Execs** := (**A**∪**C**)*(abort | halt)
    - $\alpha_1 \mathbf{c}_1 \alpha_2 \mathbf{c}_2 \ldots \mathbf{c}_k \alpha_{k+1} \mathbf{s}$, where $\alpha_i \in \mathbf{A}^*$ (for $1 \le i \le k+1$), $\mathbf{c}_i \in \mathbf{C}$ (for $1 \le i \le k$) , and $\mathbf{s} \in \{$ **abort**, **halt**$\}$.
- **Execs**(P): the set of such executions generated by all possible $\vec{I}$
  - Node: each statement
  - Form: Execution tree
    - Assignment nodes have one successor
    - Conditional nodes have one or two successors
    - Leaves are labeled abort or halt

2.3 Test Driver and Instrumented Program

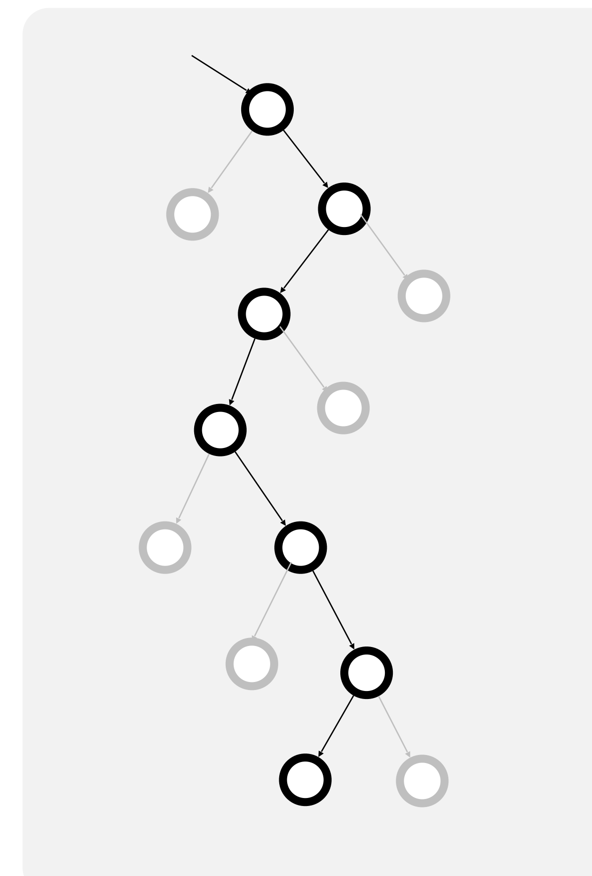
- The goal of DART
  - To explore all paths in the execution tree **Execs**(*P*)
  - Assume
    - The theory of integer linear constraints
      - How we handle the transition from constraints within the theory to those that are outside
- Symbolic evaluation
  - a *symbolic memory S* that maps memory addresses to expressions
    - *S*: mapping that maps each m ∈ $\vec{M}_0$ to itself
  - When an expression falls outside the theory, DART simply falls back on the concrete value of the expression
  - Flag for track completeness
    - *all_linear*
    - *all_locs_definite*
      - Initial value: 1
      - Falls back on the concrete value: 0

```
evaluate_symbolic (e, M, S) =
  match e:
    case m:    //the symbolic variable named m
      if m ∈ domainS then return S(m)
      else return M(m)
    case *(e', e''):    //multiplication
      let f'= evaluate_symbolic(e', M, S);
      let f''= evaluate_symbolic(e'', M, S);
      if not one of f' or f'' is a constant c then
        all_linear = 0
        return evaluate_concrete(e, M)
      if both f' and f'' are constants then
        return evaluate_concrete(e, M)
      if f' is a constant c then
        return *(f', c)
      else return *(c, f'')
    case *e':    //pointer dereference
      let f'= evaluate_symbolic(e', M, S);
      if f' is a constant c then
        if *c ∈ domainS then return S(*c)
        else return M(*c)
      else all_locs_definite = 0
        return evaluate_concrete(e, M)
    etc.
```

Figure 1. Symbolic evaluation

2.3 Test Driver and Instrumented Program - cont.

- Execution Tree
  - Run repeatedly
  - Each run(except the first) is executed with the help of a record of the conditional statements executed in the previous run
  - Each conditional
    - Record a branch value
      - 1: the *then* branch is taken
      - 0: the *else* branch is taken
    - Record *done* value
      - 0: only one branch of the conditional has executed in prior runs
        - With the same history up to the branch point
      - 1: otherwise
  - *Stack*: Store information associated with each conditional statement of the last execution path
    - List variable
    - kept in a file between executions
    - i+1th conditional execution
      - For i, $0 \le i < |stack|$, $stack[i] = (stack[i].branch, stack[i].done)$



2.3 Test Driver and Instrumented Program - cont.

- Test_Driver
  - Combines random testing(the repeat loop) with directed search(the while loop)
  - If the *instrumented_program* throws an exception, then a bug has been found
  - completeness flags
    - *all_linear*
    - *all_locs_definite*
    - Each holds unless a “bad” situation
    - If completeness flags are still hold, then program terminated
      - Found all paths
    - If just one of the completeness flags have been turned off, then the outer loop continues forever

```
run_DART () =
  all_linear, all_locs_definite, forcing_ok = 1, 1, 1
  repeat
    stack = ⟨⟩; I⃗ = [] ; directed = 1
    while (directed) do
      try (directed, stack, I⃗) =
        instrumented_program(stack, I⃗)
      catch any exception →
        if (forcing_ok)
          print “Bug found”
          exit()
        else forcing_ok = 1
  until all_linear ∧ all_locs_definite
```

Figure 2. Test driver

2.3 Test Driver and Instrumented Program - cont.

- Instrument_program
  - Executes as the original program
    - With interleaved gathering of symbolic constraints
    - **s** = *statement_at*(l, $\mathscr{M}$)
  - Each conditional statement
    - Checks by calling *compare_and_update_stack*
  - ^: list concatenation
    - *then*: *path_constraint*^$\langle c \rangle$
    - *else: path_constraint*^$\langle neg(c) \rangle$
  - loop invariant
    - *stack*[|*stack*|-1].done = 0(initial)
      - 1: If the execution proceeds according to all the branches in stack as checked by *compare_and_update_stack*

instrumented_program(*stack*, $\vec{I}$) =
  // Random initialization of uninitialized input parameters in $\vec{M}_0$
  for each input x with $\vec{I}$[x] undefined **do**
    $\vec{I}$[x] = *random*()
  Initialize memory $\mathcal{M}$ from $\vec{M}_0$ and $\vec{I}$
  // Set up symbolic memory and prepare execution
  $\mathcal{S} = [m \mapsto m \mid m \in \vec{M}_0]$.
  $\ell = \ell_0$ // Initial program counter in P
  $k = 0$ // Number of conditionals executed
  // Now invoke P intertwined with symbolic calculations
  s = *statement_at*(ℓ,$\mathcal{M}$)
  while (**s** $\notin$ {**abort**, **halt**}) **do**
    match (**s**)
      case ($m \leftarrow e$):
        $\mathcal{S} = \mathcal{S} + [m \mapsto$ *evaluate_symbolic*$(e, \mathcal{M}, \mathcal{S})]$
        v = *evaluate_concrete*$(e, \mathcal{M})$
        $\mathcal{M} = \mathcal{M} + [m \mapsto v]$; $\ell = \ell + 1$
      case (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* ˆ $\langle c \rangle$
          stack = *compare_and_update_stack*(1, k,*stack*)
          $\ell = \ell'$
        else
          path_constraint = *path_constraint* ˆ $\langle neg(c) \rangle$
          stack = *compare_and_update_stack*(0, k,*stack*)
          $\ell = \ell + 1$
        $k = k + 1$
    s =*statement_at*(ℓ,$\mathcal{M}$) // End of while loop
  if (**s==abort**) **then**
    raise an exception
  else // **s==halt**
    return *solve_path_constraint*(k,*path_constraint*,*stack*)

Figure 3. Instrumented_program

2.3 Test Driver and Instrumented Program - cont.

- *Compare_and_update_stack*
  - Checks whether the current execution path matches the one predicted at the end of the previous execution and represented in stack passed between runs
    - flag *forcing_ok* == 1(initial)
      - 0
        - If prediction of the outcome of a conditional is not fulfilled
        - Restart *run_DART* with a new random input vector
      - $all\_linear \wedge all\_locs\_definite \Rightarrow forcing\_ok$
- *Solve_path_constraint*
  - When the original program halts, new input values are generated
  - To attempt to force the next run to execute the last unexplored branch of a conditional along the stack
    - $\vec{I} + \vec{I}'$: If such a branch exists and if the path constraint that may lead to its execution has a solution $\vec{I}'$, this solution is used to update the mapping $\vec{I}$ to be used for the next run

```
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
```

Figure 4. Compare_and_update_stack

```
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 I⃗') then
            return (1, stack[0..j], I⃗ + I⃗')
        else
            solve_path_constraint(j,path_constraint,stack)
```

Figure 5. Solve_path_constraint

2.3 Test Driver and Instrumented Program - cont.

- THEOREM 1

  a) If *run_DART* prints out “Bug found” for *P*, then there is some input to *P* that leads to an abort

  b) If *run_DART* terminates without printing “Bug found,” then there is no input that leads to an abort statement in P, and all paths in **Execs**(*P*) have been exercised

  c) Otherwise, *run_DART* will run forever

- Proofs
  - (a) and (c) are immediate
  - (b)
    - Any potential incompleteness in DART’s directed search detected and recorded by setting at least one of the two flags to 0.
      - *all_linear*
      - *all_locs_definite*

- Difference between DART and static_analysis_based approaches
  - DART is guaranteed to be sound even when using an incomplete or wrong theory

2.4 Example

- First Run
  - Initial concrete memory: $\mathscr{M} = [m_x \rightarrow 123456, m_y \rightarrow 654321]$
  - Initial symbolic memory: $S = [m_x \rightarrow m_x, m_y \rightarrow m_y]$
  - Path constraint: $\langle m_x = m_y \rangle$
    - halt: $\langle \neg(m_x = m_y) \rangle$
  - $k = 1$, stack $= \langle (0, 0) \rangle$, $S = [m_x \rightarrow m_x, m_y \rightarrow m_y, m_z \rightarrow m_y]$, $\mathscr{M} = [m_x \rightarrow 123456, m_y \rightarrow 654321, m_z \rightarrow 654321]$
    - Solve $\langle m_x = m_y \rangle$
      - $\langle m_x \rightarrow 0, m_y \rightarrow 0 \rangle$
  - Update input vector $\vec{I} + \vec{I}'$ is then $\langle 0, 0 \rangle$
  - The branch bit in stack has been flipped

```
int f(int x, int y) {
  int z;
  z = y;
  if (x == z)
     if (y == x + 10)
        abort();
  return 0;
}
```

<code> Function f

2.4 Example - cont.

- Second Run
  - run_DART
    - (*directed*, *stack*, $\vec{I}$)=(1, ⟨(1, 0)⟩, ⟨0, 0⟩)
  - compare_and_update_stack
    - Check that the actually executed branch of the outer if statement is now *then* branch
    - The else branch of the inner if statement is executed
      - Path constraint
        - ⟨$m_x = m_y$, $m_y = m_x + 10$⟩
  - *run_DART* driver calls solve_path_constraint
    - (k_{try}, *path_constraint*, *stack*)=(2,⟨$m_x = m_y$, $m_y = m_x + 10$⟩,⟨(1,1),(0,0)⟩)
    - This path constraint has no solution and the first conditional has already been covered (*stack*[0].*done* = 1)
    - solve_path_constraint returns (0, _, _)
  - All completeness flags are still set
    - run_DART terminates

```
int f(int x, int y) {
  int z;
  z = y;
  if (x == z)
    if (y == x + 10)
      abort();
  return 0;
}
```

<code> Function f

2.5 Advantages of the DART approach

- Dynamic analysis often has an advantage over static analysis when reasoning about dynamic data
  - Ex. If two pointers point to the same memory location (<code> struct foo)
    - DART simply checks whether their values are equal and does not require alias analysis
    - Static analysis will typically not be able to report with high certainty that abort() is reachable
    - Standard alias analysis is not able to guarantee that a—>c has been overwritten
    - DART finds a precise execution leading to the abort very easily by simply generating an input satisfying the linear constraint a —> c == 0
- DART
  - Concrete + symbolic execution
    - Concolic
      - First, any execution leading to an error detected by DART is trivially sound
      - Second, it allows us to alleviate the limitations of the constraint solver/ theorem prover
  - Find the only reachable abort statement in the above example(<code> foobar) with high probability.

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

<code> struct foo

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

<code> foobar

Question

Static	Dynamic
WhiteBox BlackBox	BlackBox

- What is DART's testing technique?
  - White box(symbolic execution, branch coverage) — input a branch to the stack
  - Black box(Random Testing(concrete execution), Equivalence Partitioning)
    - If so, does it include Boundary Value Analysis?
  - Test drive: dynamic test
  - Directed search
- Figure 3, Why there is no forcing_ok flag after ‘if(s==abort)then’?
- Bug log