DART - Directed Automated Random Testing

Patrice Godefroid, Nils Klarlund, Koushik Sen
Bell Laboratories, Lucent Technologies
University of Illinois at Urbana-Champaign

Presenters: Angelina Martineau, Bingsheng Yao
Background

- Testing – 50% of software development
- Software Failures – $60 Billion / year
- Unit Testing
  - Hard to complete
  - Expensive
  - Rarely done correctly, sometimes skipped
    - Corner bugs are missed!
Overview

- Automate unit testing
  1) **Automatically** extract interfaces
  2) Generate test driver - random tests
  3) **Direct** execution along certain paths
- Written for C programming language
- *Code Inspection* - static checkers produce too many warnings
  - Need to handle false alarms
  - DART - static analysis is automated, uses high precision dynamic analysis
  - All exceptions found are sound
Why use DART? - Example One

```plaintext
if(x == 10)  // x is 32 bit integer

- X could be any of $2^{32}$ integers
- $1/2^{32}$ chance it is 10
- DART can change the probability to 0.5
  - Route the tests along the two branches
```
Why use DART? - Example Two

- x = 269167349, y = 889801541
- Predicates: \( x_0 \neq y_0 \) and \( 2 \times x_0 \neq x_0 + 10 \)
- Path Constraint: \( <x_0 \neq y_0, 2 \times x_0 \neq x_0 + 10> \)
  - Vectors that follow that path of execution
- Second predicate is negated:
  - \( <x_0 \neq y_0, 2 \times x_0 = x_0 + 10> \)
  - Solution: \( x = 10, y = 889801541 \)
  - Forces execution down error branch
Execution Model Syntax

- $P$ - Program, instrumented at level of RAM machine
- $M$ - Memory mapping of $m$ to 32-bit words
- $m$ - memory address or symbolic variable identified by address $m$
- $e$ - expression of the form $m \mid c \mid *(e', e'') \mid \leq (e', e'') \mid \neg (e', e'') \mid *e'$
- $\ell$ - address of a statement
  - If $\ell$ is address of statement (excluding abort and halt), then $\ell+1$ is as well
- $\ell_0$ is initial statement

4 types of statements:

1) $a$ - Assignment - $m \leftarrow e$
2) $c$ - Conditional - if $(e)$ then goto $\ell'$
3) abort - program error
4) halt - normal termination

$\bar{I}$ - input vector - initializes $m_0$ vector, which initializes $M$
Execution Model - Program $P$

$C$ - set of conditional statements, $A$ - set of assignment statements

$\bowtie$ - program execution, said to be a finite sequence if $\text{Execs} := (A \cup C) \ast (\text{abort} \mid \text{halt})$

$\bowtie = a_1 c_1 a_2 c_2 \ldots c_k a_k s$, where $a_i \in A^*$, $c_i \in C$, $s \in \{\text{abort, halt}\}$

$\text{Execs}(P)$ = the set of executions with all possible $\tilde{I}$

- Each statement is a node
- $\text{Execs}(P)$ forms an execution tree
  - Assignment nodes have one child
  - Conditional nodes have one or two child nodes
  - Leaves are $\text{abort}$ or $\text{halt}$
Test Driver

- **GOAL:** Explore all paths in execution tree
- **Symbolic memory** - map expressions to addresses
  - Expression are evaluated symbolically
  - Completeness flags are used: `all_linear` and `all_locs_definite`
- Run through execution tree repeatedly
  - Keep track of path used in conditional statements in previous run
  - 1 for then branch, 0 for else branch
- Test driver uses **random testing** and **directed search**
  - DART terminates → all paths have been explored, no issues
  - DART continues infinitely → one of the completeness flags is off
Instrumented Program

- Executes as the original program while gathering symbolic constraints
- At conditional statements
  - Compare_and_update_stack()
  - Checks that execution path matches the predicted path from previous run
  - If not, restart DART with fresh random input vector
    - Flag forcing_ok is set to 0
- Invariant that always holds:
  \[ all_{linear} \land all_{locs}_{definite} \implies forcing_ok \]
- On halt, solve_path_constraint() is used to find new input values
  - Explore last unexplored branch
Theorem 1

Consider a program $P$ as defined in Section 2.2.

(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 $\text{Execs}(P)$ have been exercised.

(c) Otherwise, run DART will run forever.
Advantages

- DART does not require alias analysis for pointers
  - Static analysis tools will not express certainty
  - DART will have a specific execution that leads to error
- Static analysis tools will generate false alarms
  - Static analysis tools will say that statement 7 is reachable
  - DART will recognize through execution that it is not
  - Input vector where \( x > 0 \) and \( y \neq 10 \) will be generated with about 0.5 probability
  - Then same input vector is used with \( y = 10 \)
  - If \( x \) is negative, DART will use \( x > 0 \) and \( y = 20 \) on the next run
    - Statement 7 will never be reached
DART for C

Interface Extraction

- Identify the external interfaces
  - Program’s external variables and functions
  - Arguments of a user-specified top-level function
- DART for C supports more general model like malloc()
- Determine the type of input via the interface
  - In C, basic type, struct type, array of another type or a pointer
- Distinguish 3 kinds of C functions
  - Program functions, external functions, library functions
DART for C

- **Generation of Random Test Driver**
  - C program performs random initialization
    - Initialize all external variables and arguments of the top-level function with random values, then call the top-level function.
    - Simulating each external function that generate a random value of the external function’s return type whenever it is called

- **Implementation of Directed Search**
  - Using a dynamic instrumentation
  - Main challenge is to handle all possible types that C allows, generate and manipulate symbolic constraints across function boundaries.
Experimental Evaluation

Compare the efficiency of a purely random search with a directed search using an AC-controller Example

DART found an assertion violation much faster (within 7 iters vs forever)

```c
/* initially, */
int is_room_hot=0; /* room is not hot */
int is_door_closed=0; /* and door is open */
int ac=0; /* so, ac is off */

void ac_controller(int message) {
    if (message == 0) is_room_hot=1;
    if (message == 1) is_room_hot=0;
    if (message == 2) { is_door_closed=0; ac=0; }
    if (message == 3) {
        is_door_closed=1;
        if (is_room_hot) ac=1; }
    if (is_room_hot && is_door_closed && !ac)
        abort(); /* check correctness */
...
Experimental Evaluation

Needham–Schroeder public key authentication Protocol

- The protocol involves a sequence of message exchanges between an initiator, a responder and a mutually-trusted key server.
- An attack involves an intruder trying to impersonate an initiator to set up a false session with responder.
- DART finds an assertion violation much faster and more efficient than a random search
- DART has limitation - can only find the projection of the attack from B’s point of view
Evaluation on A Large Application oSIP

- An open-source implementation of the Session Initiation Protocol
  - Since there’s little documentation on the API of oSIP, DART is tested on ~600 externally visible functions
    - DART found a way to crash 65% of oSIP functions within 1,000 attempts for each function
    - Most of these crashes share the same basic pattern: an oSIP function takes as argument a pointer to a data structure and then de-references later that pointer without checking first whether the pointer is non-NULL
    - Provide specific method to crash oSIP (oSIP being fixed now)
Related Work and Conclusion

- Similar works have more limitations
  - Use static analysis and code transformation to eliminate external interface of the open program and replace conditional statements with nondeterministic statements
  - Symbolic execution is limited in practice by the imprecision of static analysis and of theorem provers
  - Does not deal with function calls and unknown code segments

- Directed Randomization is very helpful, and DART’s ability to execute and test without writing any test driver/harness code is efficient