DART: Directed Automated Random Testing

Patrice Godefroid   Nils Klarlund
Bell Laboratories, Lucent Technologies
{god,klarlund}@bell-labs.com

Koushik Sen
Computer Science Department
University of Illinois at Urbana-Champaign
ksen@cs.uiuc.edu

Abstract

We present a new tool, named DART, for automatically testing software that combines three main techniques: (1) *automated* extraction of the interface of a program with its external environment using static source-code parsing; (2) automatic generation of a test driver for this interface that performs *random* testing to simulate the most general environment the program can operate in; and (3) 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. Together, these three techniques constitute *Directed Automated Random Testing*, or *DART* for short. The main strength of DART is thus that testing can be performed *completely automatically* on any program that compiles – there is no need to write any test driver or harness code. During testing, DART detects standard errors such as program crashes, assertion violations, and non-termination. Preliminary experiments to unit test several examples of C programs are very encouraging.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Algorithms, Reliability

Keywords Software Testing, Random Testing, Automated Test Generation, Interfaces, Program Verification

1. Introduction

Today, *testing* is the primary way to check the correctness of software. Billions of dollars are spent on testing in the software industry, as testing usually accounts for about 50% of the cost of software development [27]. It was recently estimated that software failures currently cost the US economy alone about $60 billion every year, and that improvements in software testing infrastructure might save one-third of this cost [31].

Among the various kinds of testing usually performed during the software development cycle, *unit testing* applies to the individual components of a software system. In principle, unit testing plays an important role in ensuring overall software quality since its role is precisely to detect errors in the component's logic, check all corner cases, and provide 100% code coverage. Yet, in practice, unit testing is so hard and expensive to perform that it is rarely done properly. Indeed, in order to be able to execute and test a component in isolation, one needs to write test driver/harness code to simulate the environment of the component. More code is needed to test functional correctness, for instance using assertions checking the component's outputs. Since writing all this testing code manually is expensive, unit testing is often either performed very poorly or skipped altogether. Moreover, subsequent phases of testing, such as feature, integration and system testing, are meant to test the overall correctness of the entire system viewed as a black-box, not to check the corner cases where bugs causing reliability issues are typically hidden. As a consequence, many software bugs that should have been caught during unit testing remain undetected until field deployment.

In this paper, we propose a new approach that addresses the main limitation hampering unit testing, namely the need to write test driver and harness code to simulate the external environment of a software application. We describe our tool DART, which combines three main techniques in order to *automate unit testing of software*:

1. *automated* extraction of the interface of a program with its external environment using static source-code parsing;
2. automatic generation of a test driver for this interface that performs *random* testing to simulate the most general environment the program can operate in; and
3. 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.

Together, these three techniques constitute *Directed Automated Random Testing*, or *DART* for short. Thus, the main strength of DART is that testing can be performed *completely automatically* on any program that compiles – there is no need to write any test driver or harness code. During testing, DART detects standard errors such as program crashes, assertion violations, and non-termination.

We have implemented DART for programs written in the C programming language. Preliminary experiments to unit test several examples of C programs are very encouraging. For instance, DART was able to find automatically attacks in various C implementations of a well-known flawed security protocol (Needham-Schroeder's). Also, DART found hundreds of ways to crash 65% of the about 600 externally visible functions provided in the oSIP library, an open-source implementation of the SIP protocol. These experimental results are discussed in detail in Section 4.

The idea of extracting automatically interfaces of software components via static analysis has been discussed before, for model-checking purposes (e.g., [8]), reverse engineering (e.g., [37]), and compositional verification (e.g., [1]). However, we are not aware of any tool like DART which combines automatic interface extraction with random testing and dynamic test generation. DART is complementary to test-management tools that take advantage of interface

PLDI'05 June 12–15, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006…$5.00.

definitions as part of programming languages, such as JUnit [20] for Java, but do not perform automatic test generation.

Random testing is a simple and well-known technique (e.g., [4]), which can be remarkably effective at finding software bugs [11]. Yet, it is also well-known that random testing usually provides low code coverage (e.g., [32]). For instance, the `then` branch of the conditional statement "`if (x==10) then ...`" has only one chance to be exercised out of 2^{32} if `x` is a 32-bit integer program input that is randomly initialized. The contributions of DART compared to random testing are twofold: DART makes random testing automatic by combining it with automatic interface extraction (in contrast with prior work which is API-specific, e.g., [11]), and also makes it much more effective in finding errors thanks to the use of dynamic test generation to drive the program along alternative conditional branches. For instance, the probability of taking the `then` branch of the statement "`if (x==10) then ...`" can be viewed as 0.5 with DART. The novel dynamic test-generation techniques used in DART are presented in Section 2.

Besides testing, the other main way to check correctness during the software development cycle is *code inspection*. Over the last few years, there has been a renewed interest in static source-code analysis for building automatic code-inspection tools that are more practical and usable by the average software developer. Examples of such tools are Prefix/Prefast [6], MC [16], Klocwork [22], and Polyspace [33]. Earlier program static checkers like `lint` [19] usually generate an overly large number of warnings and false alarms, and are therefore rarely used by programmers on a regular basis. The main challenge faced by the new generation of static analyzers is thus to do a better job in dealing with false alarms (warnings that do not actually correspond to programming errors), which arise from the inherent imprecision of static analysis. There are essentially two main approaches to this problem: either report only high-confidence warnings (at the risk of missing some actual bugs), or report all of them (at the risk of overwhelming the user). Despite significant recent progress on techniques to separate false alarms from real errors (for instance, by using more precise analysis techniques to eliminate false alarms, or by using statistical classification techniques to rank warnings by their severity more accurately), analyzing the results of static analysis to determine whether a warning actually corresponds to an error still involves significant human intervention.

We believe DART provides an attractive alternative approach to static analyzers, because it is based on high-precision dynamic analysis instead, while being fully automated as static analysis. The main advantage of DART over static analysis is that every execution leading to an error that is found by DART is guaranteed to be sound. Two areas where we expect DART to compete especially well against static analyzers are the detection of interprocedural bugs and of bugs that arise through the use of library functions (which are usually hard to reason about statically), as will be discussed later in the paper. Of course, DART is overall *complementary* to static analysis since it has its own limitations, namely the computational expense of running tests and the sometimes limited effectiveness of dynamic test generation to improve over random testing. In any case, DART offers a new trade-off among existing static and dynamic analysis techniques.

The paper is organized as follows. Section 2 presents an overview of DART. Section 3 discusses implementation issues when dealing with programs written in the C programming language. In Section 4, experimental results are discussed. We compare DART with other related work in Section 5 and conclude with Section 6.

2. DART Overview

DART's integration of random testing and dynamic test generation using symbolic reasoning is best intuitively explained with an example.

2.1 An Introduction to DART

Consider the function `h` in the file below:

```
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;
}
```

The function `h` is defective because it may lead to an abort statement for some value of its input vector, which consists of the input parameters `x` and `y`. Running the program with random values of `x` and `y` is unlikely to discover the bug. The problem is typical of random testing: it is difficult to generate input values that will drive the program through *all* its different execution paths.

In contrast, DART is able to dynamically gather knowledge about the execution of the program in what we call a *directed search*. Starting with a random input, a DART-instrumented program calculates 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. By repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths.

For the example above, the DART-instrumented `h` initially guesses the value 269167349 for `x` and 889801541 for `y`. As a result, `h` executes the then-branch of the first if-statement, but fails to execute the then-branch of the second if-statement; thus, no error is encountered. Intertwined with the normal execution, the predicates $x_0 \neq y_0$ and $2 \cdot x_0 \neq x_0 + 10$ are formed on-the-fly according to how the conditionals evaluate; x_0 and y_0 are *symbolic variables* that represent the values of the memory locations of variables x and y. Note the expression $2 \cdot x_0$, representing `f(x)`: it is defined through an interprocedural, dynamic tracing of symbolic expressions.

The predicate sequence $\langle x_0 \neq y_0, 2 \cdot x_0 \neq x_0 + 10 \rangle$, called a *path constraint*, represents an equivalence class of input vectors, namely all the input vectors that drive the program through the path that was just executed. To force the program through a different equivalence class, the DART-instrumented `h` calculates a solution to the path constraint $\langle x_0 \neq y_0, 2 \cdot x_0 = x_0 + 10 \rangle$ obtained by negating the last predicate of the current path constraint.

A solution to this path constraint is $(x_0 = 10, y_0 = 889801541)$ and it is recorded to a file. When the instrumented `h` runs again, it reads the values of the symbolic variables that have been solved from the file. In this case, the second execution then reveals the error by driving the program into the `abort()` statement as expected.

2.2 Execution Model

DART runs the program P under test both concretely, executing the actual program with random inputs, and symbolically, calculating constraints on values at memory locations expressed in terms of input parameters. These side-by-side executions require the program P to be instrumented at the level of a RAM (Random Access Memory) machine.

The *memory* $\mathcal{M}$ is a mapping from memory addresses m to, say, 32-bit words. The notation $+$ for mappings denotes updating;

for example, $\mathcal{M}' := \mathcal{M} + [m \mapsto v]$ is the same map as $\mathcal{M}$, except that $\mathcal{M}'(m) = v$. We identify *symbolic variables* by their addresses. Thus in an expression, m denotes either a memory address or the symbolic variable identified by address m, depending on the context. A *symbolic expression*, or just expression, e can be of the form m, c (a constant), $*(e', e'')$ (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), etc. Thus, the symbolic variables of an expression e are the set of addresses m that occur in it. Expressions have no side-effects.

The program P manipulates the memory through *statements* that are specially tailored abstractions of the machine instructions actually executed. There is a set of numbers that denote instruction addresses, that is, statement labels. If ℓ is the address of a statement (other than **abort** or **halt**), then $\ell + 1$ is guaranteed to also be an address of a statement. The initial address is ℓ_0. A statement can be a *conditional statement* **c** of the form `if` (e) `then goto` ℓ' (where e is an expression over symbolic variables and ℓ' is a statement label), an *assignment statement* **a** of the form $m \leftarrow e$ (where m is a memory address), **abort**, corresponding to a program error, or **halt**, corresponding to normal termination.

The concrete semantics of the RAM machine instructions of P is reflected in $evaluate\_concrete(e, \mathcal{M})$, which evaluates expression e in context $\mathcal{M}$ and returns a 32-bit value for e. Additionally, the function *statement_at*(ℓ, $\mathcal{M}$) specifies the next statement to be executed. For an assignment statement, this function calculates, possibly involving address arithmetic, the address m of the left-hand side, where the result is to be stored; in particular, indirect addressing, e.g., stemming from pointers, is resolved at runtime to a corresponding absolute address.[1]

A program P defines a sequence of *input addresses* $\vec{M}_0$, the addresses of the input parameters of P. An *input vector* $\vec{I}$, which associates a value to each input parameter, defines the initial value of $\vec{M}_0$ and hence $\mathcal{M}$.[2]

Let **C** be the set of conditional statements and **A** the set of assignment statements in P. A *program execution* w is a finite[3] sequence in **Execs** $:= (\mathbf{A} \cup \mathbf{C})^*(\mathbf{abort} \mid \mathbf{halt})$. We prefer to view w as being of the form $\alpha_1 \mathbf{c}_1 \alpha_2 \mathbf{c}_2 \ldots \mathbf{c}_k \alpha_{k+1} \mathbf{s}$, where $\alpha_i \in \mathbf{A}^*$ (for $1 \leq i \leq k+1$), $\mathbf{c}_i \in \mathbf{C}$ (for $1 \leq i \leq k$), and $\mathbf{s} \in \{\mathbf{abort}, \mathbf{halt}\}$.

The concrete semantics of P at the RAM machine level allows us to define for each input vector $\vec{I}$ an execution sequence: the result of executing P on $\vec{I}$ (the details of this semantics is not relevant for our purposes). Let **Execs**(P) be the set of such executions generated by all possible $\vec{I}$. By viewing each statement as a node, **Execs**(P) forms a tree, called the *execution tree*. Its assignment nodes have one successor; its conditional nodes have one or two successors; and its leaves are labeled **abort** or **halt**.

2.3 Test Driver and Instrumented Program

The goal of DART is to explore all paths in the execution tree **Execs**(P). To simplify the following discussion, we assume that we are given a theorem prover that decides, say, the theory of integer linear constraints. This will allow us to explain how we handle the transition from constraints within the theory to those that are outside.

DART maintains a *symbolic memory* $\mathcal{S}$ that maps memory addresses to expressions. Initially, $\mathcal{S}$ is a mapping that maps each

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

Figure 1. Symbolic evaluation

$m \in \vec{M}_0$ to itself. Expressions are evaluated symbolically as described in Figure 1. When an expression falls outside the theory, as in the multiplication of two non-constant sub-expressions, *DART simply falls back on the concrete value* of the expression, which is used as the result. In such a case, we also set a flag *all_linear* to 0, which we use to track completeness. Another case where DART's directed search is typically incomplete is when the program dereferences a pointer whose value depends on some input parameter; in this case, the flag *all_locs_definite* is set to 0 and the evaluation falls back again to the concrete value of the expression. With this evaluation strategy, symbolic variables of expressions in $\mathcal{S}$ are always contained in $\vec{M}_0$.

To carry out a search through the execution tree, our instrumented program is run repeatedly. Each run (except the first) is executed with the help of a record of the conditional statements executed in the previous run. For each conditional, we record a *branch* value, which is either 1 (the *then* branch is taken) or 0 (the *else* branch is taken), as well as a *done* value, which is 0 when only one branch of the conditional has executed in prior runs (with the same history up to the branch point) and is 1 otherwise. This information associated with each conditional statement of the last execution path is stored in a list variable called *stack*, kept in a file between executions. For i, $0 \leq i < |stack|$, $stack[i] = (stack[i].branch, stack[i].done)$ is thus the record corresponding to the $i+1$th conditional executed.

More precisely, our test driver *run_DART* is shown in Figure 2. This 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. The two *completeness flags*, namely *all_linear* and *all_locs_definite*, each holds unless a "bad" situation possibly leading to incompleteness has occurred. Thus, if the directed search terminates—that is, if *directed* of the inner loop no longer holds—then the outer loop also terminates provided all of the completeness flags still hold. In this case, DART terminates and safely reports that all feasible program paths have been explored. But if just one of the completeness flags have been turned off at some point, then the outer loop continues forever (modulo resource constraints not shown here).

[1] We do this to simplify the exposition; left-hand sides could be made symbolic as well.

[2] To simplify the presentation, we assume that $\vec{M}_0$ is the same for all executions of P.

[3] We thus assume that all program executions terminate; in practice, this can be enforced by limiting the number of execution steps.

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

Figure 2. Test driver

```
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($\ell$,$\mathcal{M}$)
  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 $\ell'$):
        $b$ = evaluate_concrete($e, \mathcal{M}$)
        $c$ = evaluate_symbolic($e, \mathcal{M}, \mathcal{S}$)
        if $b$ then
          path_constraint = path_constraint ˆ ⟨$c$⟩
          stack = compare_and_update_stack(1, $k$,stack)
          $\ell$ = $\ell'$
        else
          path_constraint = path_constraint ˆ ⟨neg($c$)⟩
          stack = compare_and_update_stack(0, $k$,stack)
          $\ell$ = $\ell + 1$
        $k$= $k + 1$
    s =statement_at($\ell$,$\mathcal{M}$)     // 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

The instrumented program itself is described in Figure 3 (where ˆ denotes list concatenation). It executes as the original program, but with interleaved gathering of symbolic constraints. At each conditional statement, it also checks by calling *compare_and_update_stack*, shown in Figure 4, whether the current execution path matches the one predicted at the end of the previous execution and represented in *stack* passed between runs. Specifically, our algorithm maintains the invariant that when *instrumented_program* is called, *stack*$[|$*stack*$| - 1]$.*done* $= 0$ holds.

```
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 ˆ ⟨(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 $-1 \leq j < h < k_{try}$, 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= ¬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)
```

Figure 5. Solve_path_constraint

This value is changed to 1 if the execution proceeds according to all the branches in *stack* as checked by *compare_and_update_stack*. If it ever happens that a prediction of the outcome of a conditional is not fulfilled, then the flag *forcing_ok* is set to 0 and an exception is raised to restart *run_DART* with a fresh random input vector. Note that setting *forcing_ok* to 0 can only be due to a previous incompleteness in DART's directed search, which was then (conservatively) detected and resulted in setting (at least) one of the completeness flags to 0. In other words, the following invariant always holds: *all_linear* $\wedge$ *all_locs_definite* $\Rightarrow$ *forcing_ok*.

When the original program halts, new input values are generated in *solve_path_constraint*, shown in Figure 5, to attempt to force the next run to execute the last[4] unexplored branch of a conditional along the stack. 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; values corresponding to input parameters not involved in the path constraint are preserved (this update is denoted $\vec{I} + \vec{I}'$).

The main property of DART is stated in the following theorem, which formulates (a) soundness (of error founds) and (b) a form of completeness.

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* **Execs**(P) *have been exercised. (c) Otherwise, run_DART will run forever.*

Proofs of (a) and (c) are immediate. The proof of (b) rests on the assumption that any potential incompleteness in DART's directed search is (conservatively) detected and recorded by setting at least one of the two flags *all_linear* and *all_locs_definite* to 0.

[4] A depth-first search is used for exposition, but the next branch to be forced could be selected using a different strategy, e.g., randomly or in a breadth-first manner.

Since DART performs (typically partial) symbolic executions only as generalizations of concrete executions, a key difference between DART and static-analysis-based approaches to software verification is that any error found by DART is guaranteed to be sound (case (a) above) even when using an incomplete or wrong theory. In order to maximize the chances of termination in case (b) above, setting off completeness flags as described in *evaluate_symbolic* could be done less conservatively (i.e., more accurately) using various optimization techniques, for instance by distinguishing incompleteness in expressions used in assignments from those used in conditional statements, by refining after each conditional statement the constraints stored in $\mathcal{S}$ that are associated with symbolic variables involved in the conditional, by dealing with pointer dereferences in a more sophisticated way, etc.

2.4 Example

Consider the C program:

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

The input address vector is $\vec{M}_0 = \langle m_x, m_y \rangle$ (where $m_x \neq m_y$ are some memory addresses) for f's input parameters $\langle \mathtt{x}, \mathtt{y} \rangle$. Let us assume that the first value for x is 123456 and that of y is 654321, that is, $\vec{I} = \langle 123456, 654321 \rangle$. Then, the initial concrete memory becomes $\mathcal{M} = [m_x \mapsto 123456, m_y \mapsto 654321]$, and the initial symbolic memory becomes $\mathcal{S} = [m_x \mapsto m_x, m_y \mapsto m_y]$. During execution from this configuration, the `else` branch of the outer `if` statement is taken and, at the time **halt** is encountered, the path constraint is $\langle \neg(m_x = m_y) \rangle$. We have $k = 1$, *stack* $= \langle (0,0) \rangle$, $\mathcal{S} = [m_x \mapsto m_x, m_y \mapsto m_y, m_z \mapsto m_y]$, $\mathcal{M} = [m_x \mapsto 123456, m_y \mapsto 654321, m_z \mapsto 654321]$. The subsequent call to *solve_path_constraint* results in an attempt to solve $\langle m_x = m_y \rangle$, which leads to a solution $\langle m_x \mapsto 0, m_y \mapsto 0 \rangle$. The updated input vector $\vec{I} + \vec{I}'$ is then $\langle 0, 0 \rangle$, the branch bit in *stack* has been flipped, and the assignment (*directed, stack,* $\vec{I}$)=(1, $\langle (1,0) \rangle$, $\langle 0,0 \rangle$) is executed in *run_DART*. During the second call of *instrumented_program*, the *compare_and_update_stack* will check that the actually executed branch of the outer `if` statement is now the `then` branch (which it is!). Next, the `else` branch of the inner `if` statement is executed. Consequently, the path constraint that is now to be solved is $\langle m_x = m_y, m_y = m_x + 10 \rangle$. The *run_DART* driver then calls *solve_path_constraint* with (k_{try},*path_constraint, stack*)=(2, $\langle m_x = m_y, m_y = m_x{+}10 \rangle$, $\langle (1,1), (0,0) \rangle$). Since this path constraint has no solution, and since the first conditional has already been covered (*stack*$[0]$.*done* $= 1$), *solve_path_constraint* returns $(0, \_, \_)$. In turn, *run_DART* terminates since all completeness flags are still set.

2.5 Advantages of the DART approach

Despite the limited completeness of DART when based on linear integer constraints, dynamic analysis often has an advantage over static analysis when reasoning about dynamic data. For example, to determine if two pointers point to the same memory location, DART simply checks whether their values are equal and does not require alias analysis. Consider the C program:

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

DART here treats the pointer input parameter by randomly initializing it to NULL or to a single heap-allocated cell of the appropriate type (see Section 3.2). For this example, a static analysis will typically not be able to report with high certainty that `abort()` is reachable. Sound static analysis tools will report "the abort might be reachable", and unsound ones (like BLAST [18] or SLAM [2]) will simply report "no bug found", because standard alias analysis is not able to guarantee that `a->c` has been overwritten. In contrast, DART finds a precise execution leading to the abort very easily by simply generating an input satisfying the linear constraint `a->c == 0`. This kind of code is often found in implementations of network protocols, where a buffer of type `char *` (e.g., representing a message) is occasionally cast into a `struct` (e.g., representing the different fields of the protocol encoded in the message) and *vice versa*.

The DART approach of intertwined concrete and symbolic execution has two important advantages. 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. In particular, whenever we generate a symbolic condition at a branching statement while executing the program under test, and the theorem prover cannot decide whether that symbolic condition is true or false, we simply replace this symbolic condition by its concrete value, i.e., either true or false. This allows us to continue both the concrete and symbolic execution in spite of the limitation of the theorem prover. Note that static analysis tools using predicate abstraction [2, 18] will simply consider both branches from that branching point, which may result in *unsound* behaviors. A test-generation tool using symbolic execution [36], on the other hand, will stop its symbolic execution at that point and may miss bugs appearing down the branch. To illustrate this point, consider the following C program:

```
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  }
```

Given a theorem prover that cannot reason about non-linear arithmetic constraints, a static analysis tool using predicate abstraction [2, 18] will report that both aborts in the above code may be reachable, hence one false alarm since the abort in line 7 is unreachable. This would be true as well if the test `(x*x*x > 0)` is replaced by a library call or if it was dependent on a configuration parameter read from a file. On the other hand, a test-generation tool based on symbolic execution [36] will not be able to generate an input vector to detect any abort because its symbolic execution will be stuck at the branching point in line 2. In contrast, DART can generate randomly an input vector where `x>0` and `y!=10` with almost 0.5 probability; after the first execution with such an input, the directed search of DART will generate another input with the same positive value of `x` but with `y==10`, which will lead the program in its second run to the abort at line 4. Note that, if DART randomly generates a negative value for x in the first run, then DART will generate in the next run inputs where x>0 and y==20 to satisfy the other branch at line 7 (it will do so because no constraint is generated for the branching statement in line 2 since it is non-linear); however, due to the concrete execution, DART will then not take the else branch at line 6 in such a second run. In summary, our mixed strategy of random and directed search along with simultaneous concrete and symbolic execution of the program will allow us to find the only reachable abort statement in the above example with high probability.

3. DART for C

We now discuss how to implement the algorithms presented in the previous section for testing programs written in the C programming language.

3.1 Interface Extraction

Given a program to test, DART first identifies the external interfaces through which the program can obtain inputs via uninitialized memory locations $\vec{M}_0$. In the context of C, we define the external interfaces of a C program as

- its external variables and external functions (reported as "undefined reference" at the time of compilation of the program), and
- the arguments of a user-specified *toplevel function*, which is a function of the program called to start its execution.

The main advantage of this definition is that the external interfaces of a C program can be easily determined and instrumented by a light-weight static parsing of the program's source code. Inputs to a C program are defined as memory locations which are dynamically initialized at runtime through the static external interface. This allows us to handle inputs which are dynamic in nature, such as lists and trees, in a uniform way. Considering inputs as uninitialized runtime memory locations, instead of syntactic objects exclusively such as program variables, also allows us to avoid expensive or imprecise alias analyses, which form the basis of many static analysis tools.

Note that the (simplified) formalization of Section 2.2 assumed that the input addresses $\vec{M}_0$ are the same for all executions of program P. However, our implementation of DART supports a more general model where multiple inputs can be mapped to a same address m when these are obtained by successively reading m during different successive calls to the toplevel function, as will be discussed later, as well as the possibility of a same input being mapped to different addresses in different executions, for instance when the input is provided through an address dynamically allocated with malloc().

For each external interface, we determine the *type* of the input that can be passed to the program via that interface. In C, a type is defined recursively as either a *basic type* (int, float, char, enum, etc.), a *struct type* composed of one or more fields of other types, an *array* of another type, or a *pointer* to another type.

Figure 6 shows a simple example of C program simulating a controller for an air-conditioning (AC) system. The toplevel function is ac_controller, and the external interface is simply its argument message, of basic type int.

It is worth emphasizing that we distinguish three kinds of C functions in this work.

- *Program functions* are functions defined in the program.
- *External functions* are functions controlled by the environment and hence part of the external interface of the program; they can nondeterministically return any value of their specified return type.

```
                         /* 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 */
}
```

Figure 6. AC-controller example (C code)

- *Library functions* are functions not defined in the program but controlled by the program, and hence considered as part of it. Examples of such functions are operating-system functions and functions defined in the standard C library. These functions are treated as unknown but deterministic "black-boxes" which we cannot instrument or analyze.

The ability of DART to handle deterministic but unknown (and arbitrarily complex) library functions by simply executing these makes it unique compared to standard symbolic-execution based frameworks, as discussed in Section 2.4. In practice, the user can adjust the boundary between library and external functions to simulate desired effects. For instance, errors in system calls can easily be simulated by considering the corresponding system functions as external functions instead of library functions.

3.2 Generation of Random Test Driver

Once the external interfaces of the C program are identified, we generate a nondeterministic/random test driver simulating the most general environment visible to the program at its interfaces. This test driver is itself a C program, which performs the random initialization abstractly described at the beginning of the function *instrumented_program()* in Section 2, and which is defined as follows:

- The test driver consists of a function main which initializes all external variables and all arguments of the toplevel function with random values by calling the function random_init defined below, and then calls the application's toplevel function. The user of DART specifies (using the parameter depth) the number of times the toplevel function is to be called iteratively in a single run.
- The test driver also contains code simulating each external function in such a way that, whenever an external function is called during the program execution, a random value of the function's return type is returned by the simulated function.

For example, Figure 7 shows the test driver generated for the AC-controller example of Figure 6.

The initialization of memory locations controlled by the external interface is performed using the procedure random_init shown in Figure 8. This procedure takes as arguments a memory location m and the type of the value to be stored at m, and initializes randomly the location m depending on its type. If m stores a value of

```
void main() {
    for (i=0; i < depth ; i++) {
        int tmp;
        random_init(&tmp,int);
        ac_controller(tmp);
    }
}
```

Figure 7. Test driver generated for the AC-controller example (C code)

```
random_init(m,type) {
  if (type == pointer to type2) {
     if (fair coin toss == head) {
        *m = NULL;
     } else {
        *m = malloc(sizeof(type));
        random_init(*m,type2);
     }
  } else if (type == struct) {
            for all fields f in struct
                random_init(&(m->f),typeof(f));
  } else if (type == array[n] of type3){
            for (int i=0;i<n;i++)
                random_init((m+i),type3);
  } else if (type == basic type) {
            *m = random_bits(sizeof(type));
  }
}
```

Figure 8. Procedure for randomly initializing C variables of any type (in pseudo-C)

basic type, its value `*m`[5] is initialized with the auxiliary procedure `random_bits` which returns n random bits where n is its argument. If its `type` is a pointer, the value of location `m` is randomly initialized with either the value NULL (with a 0.5 probability) or with the address of newly allocated memory location, whose value is in turn initialized according to its type following the same recursive rules. If `type` is a struct or an array, every sub-element is initialized recursively in the same way. Note that, when inputs are data structures defined with a recursive type (such as lists), this general procedure can thus generate data structures of unbounded sizes.

For each external variable or argument to the toplevel function, say `v`, DART generates a call to `random_init(&v,typeof(v))` in the function `main` of the test driver before calling the toplevel function. For instance, in the case of the AC-controller program, the variable `message` forming the external interface is of type `int`, and therefore the corresponding initialization code `random_init(&tmp,int)`[6] is generated (see Figure 7).

Similarly, if the C program being tested can call an external function, say `return_type some_fun()`, then the test driver generated by DART will include a definition for this function, which is as follows:

```
return_type some_fun(){
    return_type tmp;
    random_init(&tmp,return_type);
    return tmp;
}
```

[5] In C, `*m` denotes the value stored at `m`.

[6] In C, `&v` gives the memory location of the variable `v`.

Once the test driver has been generated, it can be combined with the C program being tested to form a *self-executable* program, which can be compiled and executed automatically.

3.3 Implementation of Directed Search

A directed search can be implemented using a dynamic instrumentation as explained in Section 2. The main challenge when dealing with C is to handle all the possible types that C allows, as well as generate and manipulate symbolic constraints, especially across function boundaries (i.e., tracking inputs through function calls when a variable whose value depends on an input is passed as argument to another program function). This is tedious (because of the complexity of C) but conceptually not very hard.

In our implementation of DART for C, the code instrumentation needed to intertwine the concrete execution of the program P with the symbolic calculations performed by DART as described in function *instrumented_program()* (see Section 2) is performed using CIL [28], an OCAML application for parsing and analyzing C code. The constraint solver used by default in our implementation is lp_solve [26], which can solve efficiently any linear constraint using real and integer programming techniques.

3.4 Additional Remarks

For the sake of modeling “realistic” external environments, we have assumed in this work that the execution of external functions do not have any side effects on (i.e., do not change the value of) any previously-defined stack or heap allocated program variable, including those passed as arguments to the function. For instance, an external function returning a pointer to an `int` can only return NULL or a pointer to a newly allocated `int`, not a pointer to a previously allocated `int`. Note that this assumption does not restrict generality: external functions with side effects or returning previously defined heap-allocated objects can be simulated by adding interface code between the program and its environment.

Another assumption we made is that all program variables (i.e., all those not controlled by the environment) are properly initialized. Detecting uninitialized program variables can be done using other analyzes and tools, either statically (e.g., with `lint` [19]) or dynamically (e.g., with Purify [17]) or both (e.g., with CCured [29]).

Instead of using a static definition of interface for C programs as done above in this section, we could have used a dynamic definition, such as considering any uninitialized variable (memory location) read by the program as an input. In general, detecting inputs with such a loose definition can only be done dynamically, using a dynamic program instrumentation similar to one for detecting uninitialized variables. Such instrumentations require a precise, hence expensive, tracking of memory accesses. Discovering and simulating external functions on-the-fly is also challenging. It would be worth exploring further how to deal with dynamic interface definitions.

4. Experimental Evaluation

In this section, we present the results of several experiments performed with DART. We first compare the efficiency of a purely random search with a directed search using two program examples. We then discuss the application of DART on a larger application. All experiments were performed on a Pentium III 800Mhz processor running Linux. Runtime is user+system time as reported by the Unix `time` command and is always roughly equal to elapsed time.

4.1 AC-controller Example

Our first benchmark is the AC-controller program of Figure 6. If we set the depth to 1, the program does not have any execution leading to an assertion violation. For this example, a directed search explores all execution paths upto that depth in 6 iterations and less than a second. In contrast, a random search would thus runs forever without detecting any errors. If we set the depth to 2, there is an assertion violation if the first input value is 3 and the second input value is 0. This scenario is found by the directed search in DART in 7 iterations and less than a second. In contrast, a random search does not find the assertion violation after hours of search. Indeed, if `message` is a 32-bit integer, the probability for a random search to find the specific combination of inputs leading to this assertion violation is one out of $2^{32} \times 2^{32} = 2^{64}$, i.e., virtually zero in practice!

This explains why a directed search usually provides much better code coverage than a simple random search. Indeed, most applications contain *input-filtering* code that performs basic sanity checks on the inputs and discards the bad or irrelevant ones. Only inputs that satisfy these filtering tests are then passed to the core application and can influence its behavior. For instance, in the AC-controller program, only values 0 to 3 are meaningful inputs while all others are ignored; the directed mode is crucial to identify (iteratively) those meaningful input values.

It is worth observing how a directed search can learn through trial and error how to generate inputs that satisfy such filtering tests. Each way to pass these tests corresponds to an execution path through the input-filtering code that leads to the core application code. Every such path will eventually be discovered by the directed search provided it can reason about all the constraints along the path. When this happens, the directed search will reach and start exercizing (in the same smart way) the core application code. In contrast, a purely random search will typically be stuck forever in the input-filtering code and will never exercize the code of the core application.

4.2 Needham-Schroeder Protocol

Our second benchmark example is a C implementation of the Needham-Schroeder public key authentication protocol [30]. This protocol aims at providing mutual authentication, so that two parties can verify each other's identity before engaging in a transaction. The protocol involves a sequence of message exchanges between an *initiator*, a *responder*, and a mutually-trusted key server. The exact details of the protocol are not necessary for the discussion that follows and are omitted here. An attack against the original protocol involving six message exchanges was reported by Lowe in [24]: an intruder I is able to impersonate an initiator A to set up a false session with responder B, while B thinks he is talking to A. The steps of Lowe's attack are as follows:

1. $A \rightarrow I \ : \ \{N_a, A\}_{K_i}$ (A starts a normal session with I by sending it a nonce N_a and its name A, both encrypted with I's public key K_i)
2. $I(A) \rightarrow B \ : \ \{N_a, A\}_{K_b}$ (the intruder I impersonates A to try to establish a false session with B)
3. $B \rightarrow I(A) \ : \ \{N_a, N_b\}_{K_a}$ (B responds by selecting a new nonce N_b and trying to return it with N_a to A)
4. $I \rightarrow A \ : \ \{N_a, N_b\}_{K_a}$ (I simply forwards B's last message to A; note that I does not know how to decrypt B's message to A since it is encrypted with A's key K_a)
5. $A \rightarrow I \ : \ \{N_b\}_{K_i}$ (A decrypts the last message to obtain N_b and returns it to I)
6. $I(A) \rightarrow B \ : \ \{N_b\}_{K_b}$ (I can then decrypt this message to obtain N_b and returns it to B; after receiving this message, B believes that A has correctly established a session with it)

The C implementation of the Needham-Schroeder protocol we considered[7] is described by about 400 lines of C code and is more detailed than the protocol description analyzed in [24]. The C program simulates the behavior of both the initiator A and responder B according to the protocol rules. It can be executed as a single Unix process simulating the interleaved behavior of both protocol entities. It also contains an assertion that is violated whenever an attack to the protocol occurs.[8]. In the C program, agent identifiers, keys, addresses and nounces are all represented by integers. The program takes as inputs tuples of integer values representing incoming messages.

[7] We thank John Havlicek for providing us this implementation.

depth	error?	Random search	Directed search
1	no	-	69 runs (<1 second)
2	yes	-	664 runs (2 seconds)

Figure 9. Results for Needham-Schroeder protocol with a possibilistic intruder model

depth	error?	Iterations (runtime)
1	no	5 runs (<1 second)
2	no	85 runs (<1 seconds)
3	no	6,260 runs (22 seconds)
4	yes	328,459 runs (18 minutes)

Figure 10. Results for Needham-Schroeder protocol with a Dolev-Yao intruder model

Results of experiments are presented in Figure 9. When at most one (depth is 1) message is sent to the initiator or responder, there is no program execution leading to an assertion violation. The table indicates how many iterations (runs) of the program are needed by DART's directed search to reach this conclusion. This number thus represents all possible execution paths of this protocol implementation when executed once. When two input messages are allowed, DART finds an assertion violation in 664 iterations or about two seconds of search. In contrast, a random search is not able to find any assertion violations after many hours of search.

An examination of the program execution leading to this assertion violation reveals that DART only finds *part* of Lowe's attack: it finds the projection of the attack from B's point of view, i.e., steps 2 and 6 above. In other words, DART finds that, when placed in its *most general environment*, there exists a sequence of two input messages that drives this code to an assertion violation. However, the most general environment, which can generate any valid input at any time, is too powerful to model a realistic intruder I: for instance, given a conditional statement of the form `if (input == my_secret) then ...`, DART can set the value of this input to `my_secret` to direct its testing, which is as powerful as being able to guess encryption keys hard-coded in the program. (Such a most powerful intruder model is sometimes called a *possibilistic attacker model* in the literature.)

To find the complete Lowe's attack, it is necessary to use a more constrained model of the environment that models more precisely the capabilities of the intruder I, namely I's ability to only decrypt messages encrypted with its own key K_i, to compose messages with only nonces it already knows, and to forward only messages it has previously seen. (Such a model is called a *Dolev-Yao attacker model* in the security literature.) We then augmented the original code with such a model of I. We quickly discovered that there are many ways to model I and that each variant can have a significant impact on the size of the resulting search space.

Figure 10 presents the results obtained with one of these models, which is at least as unconstrained as the original intruder model of [24] yet results in the smallest state space we could get. In this

[8] A C assertion violation (as defined in `<assert.h>`) triggers an `abort()`.

version, the intruder model acts as an input filter for entities A and B. As shown in the Figure, the shortest sequence of inputs leading to an assertion violation is of length 4 and DART takes about 18 minutes of search to find it. This time, the corresponding execution trace corresponds to the full Lowe's attack:

- (After no specific input) A sends its first message as in Step 1 (depth 1).
- B receives an input and sends an output as in Step 3 (depth 2).
- A receives an input and sends an output as in Step 5 (depth 3).
- B receives an input as in Step 6, which then triggers an assertion violation (depth 4).

Note that, since the initiator I is modeled as an input filter, Steps 2 and 4 are not represented explicitly by additional messages.

The original code we started with contains a flag which, if turned on, implements Lowe's fix to the Needham-Schroeder protocol [25]. By curiosity, we also tested this version with DART and, to our surprise, DART found again an assertion violation after about 22 minutes of search! After examining the error trace produced by DART, we discovered that the implementation of Lowe's fix was incomplete. We contacted the author of the original code and he confirmed this was a bug he was not aware of. After fixing the code, DART was no longer able to find any assertion violation.

It is interesting to compare these results with the ones reported in [13] where the same C implementation of the Needham-Schroeder protocol was analyzed using state-space exploration techniques. Specifically, [13] studied the exploration of the (very large) state space formed by the product of this C implementation in conjunction with a nondeterministic C model of the intruder. The tool VeriSoft [12] was used to explore the product of these two interacting Unix processes. Several search techniques were experimented with. To summarize the results of [13], *neither a systematic search nor a random search* through that state space were able to detect the attack (within 8 hours of search). But a random search guided using application-independent heuristics (essentially maximizing the number of messages exchanged between the two processes) was able to find the attack after 50 minutes of search on average, on a comparable machine. So far, we have not explored the use of heuristics in the context of DART.

Because the intruder model in the implementations of the Needham-Schroeder protocol considered here and in [13] are different, a direct comparison between our results and the results of [13] is not possible. Yet, DART was able to find Lowe's attack using a systematic search (and to discover a previously-unknown bug in the implementation of Lowe's fix), while the experimental setup of [13] was not. This performance difference can perhaps be explained intuitively as follows. A standard model checking approach as taken in [13] (at a protocol implementation level) and also in [24] (at a more abstract protocol specification level) represents the program's environment (here the intruder) by a nondeterministic process that *blindly* guesses possible sequences of inputs (attacks), and then checks the effect of these on the program by performing state-space exploration. In contrast, a directed search as implemented in DART does not treat the program under test as a black-box. Instead, the directed search attempts to partition iteratively the program's input space into equivalence classes, and generate new inputs in order to exhibit new program responses – inputs that trigger a previously considered program behavior are not generated and re-tested over and over again. In this sense, a directed search can be viewed as a more *white-box* approach than traditional model checking since the observation of how the program reacts to specific inputs is used to generate the next test inputs. Since a directed search exploits more information about the program being tested, it is not surprising that it can (and should) be more effective.

4.3 A Larger Application: oSIP

In order to evaluate further the effectiveness and scalability of DART, we applied it to test a large application of industrial relevance: oSIP, an open-source implementation of the *Session Initiation Protocol*. SIP is a telephony protocol for call-establishment of multi-media sessions over IP networks (including *Voice-over-IP*). oSIP is a C library available at

`http://www.gnu.org/software/osip/osip.html`.

The oSIP library (version 2.0.9) consists of about 30,000 lines of C code describing about 600 externally visible functions which can be used by higher-level applications. Two typical such applications are SIP clients (such as softphones to make calls over the internet from a PC) and servers (to route internet calls).

Our experimental setup was as follows. Since there is very little documentation on the API provided by the oSIP library other than the code itself, we considered one-by-one each of the about 600 externally visible functions as the toplevel function that DART calls. These function names were automatically extracted from the library using scripts. For each toplevel function, the inputs controlled by DART were the arguments of the function, and the search was limited to a maximum of 1,000 iterations (runs). In other words, if DART did not find any errors after 1,000 runs, the script would then move on to the next toplevel function, and so on. Since the oSIP code does not contain assertions, the search was limited to finding segmentation faults (crashes) and non-termination.[9]

The results obtained with DART were surprising to us: DART found hundreds of ways to crash externally visible oSIP functions. In fact, DART found a way to crash 65% of the oSIP functions within 1,000 attempts for each function. A closer analysis of the results revealed that 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. It is worth noticing that some oSIP functions do contain code to test for NULL pointers, but most do not perform such tests consistently (i.e., for all execution paths), and the the documentation does not distinguish the former category of functions from the latter. Also note that a simple visual code inspection would have revealed most of these problems.

Because DART reported so many errors and because of the lack of a specification for the oSIP API, it is hard to evaluate how severe these problems really are. Perhaps the implicit assumption for higher-level applications is that they must always pass non-NULL pointers to the oSIP library, but then it is troubling to see that some of the oSIP functions do check their arguments for NULL pointers. All we can conclude with good certainty is that, from the point of view of a higher-level application developer, there are many ways to misuse the API, and that programming errors in higher-level code (such as mistakenly passing a NULL pointer to some unguarded oSIP function) could result in dramatic failures (crashes).

Overwhelmed by the large number of potential problems reported by DART, we decided to focus on the oSIP functions called in a test driver provided with the oSIP library, and to analyze in detail the results obtained for these functions. In the process, we discovered what appears to be a *significant security vulnerability* in oSIP: we found an externally controllable way to crash the oSIP parser. Specifically, the attack is as follows:

- Build an (ASCII) SIP message containing no NULL (zero) or "|" characters, and of more than 2.5 Megabytes (for a cygwin environment – the size may vary on other platforms).

[9] Non-termination is reported by DART after a timer expiration triggered when the program under test does not call any DART instrumentation within a specific time delay.

- Pass it to oSIP parser using the oSIP function "osip_message_parse".
- One of the first thing this function does is to copy this packet in stack space using the system call `alloca(size)`. This system call returns a pointer to size bytes of uninitialized local stack space, or NULL if the allocation failed. Since 2.5 Megabytes is larger than the standard stack space available for cygwin processes, an error is reported and NULL is returned.
- The oSIP code does not check success/failure of the call to `alloca`, and pass the pointer blindly to another oSIP function, which does not check this input argument and then crashes because of the NULL pointer value.

By modifying the test driver that comes with oSIP and generating an input SIP message that satisfies these constraints, we were able to confirm this attack. This is a potentially very serious flaw in oSIP: it could be possible to kill remotely any SIP client or server relying on the oSIP library for parsing SIP messages by simply sending it a message satisfying the simple properties described above! However, we do not know whether existing SIP clients or servers (i.e., higher-level applications) built using oSIP are vulnerable to this attack. Note that, as of version 2.2.0 of the oSIP library (December 2004), this code has been fixed (see comments in the ChangeLog file).

5. Other Related Work

Automatically closing an open reactive program with its most general environment to make it self-executable and to systematically explore all its possible behaviors was already proposed in [8]. However, the approach taken there is to use static analysis and code transformation in order to eliminate the external interface of the open program and to replace with nondeterministic statements all conditional statements whose outcome may depend on an input value. The resulting closed program is a simplified version (abstraction) of the original open program that is guaranteed to simulate all its possible behaviors. In comparison, DART is more precise both because it does not abstract the program under test and because it does not suffer from the inherent imprecision of static analysis. The article [35] explores how to achieve the same goal as [8] by partitioning the program's input domain using static analysis. Because [35] does not rely on abstraction (program simplifications), it can be more precise than [8], but it still suffers from the cost and imprecision of static analysis compared to DART.

There is a rich literature on test-vector generation using symbolic execution (e.g., see [21, 27, 10, 3, 36, 38, 9]). Symbolic execution is limited in practice by the imprecision of static analysis and of theorem provers. As illustrated by the examples in Section 2, DART is able to alleviate some of the limitations of symbolic execution by exploiting dynamic information obtained from a concrete execution matching the symbolic constraints, by using dynamic test generation, and by instrumenting the program to check whether the input values generated next have the expected effect on the program. The ability of DART to handle complex or unknown code segments (including library functions) as black-boxes by simply executing these makes it unique compared to standard symbolic-execution based frameworks, which require some knowledge (minimally regarding termination) about *all* program parts or are inconclusive otherwise. Since most C code usually contains a system or library call every 10 lines or so on average, this distinguishing feature of DART is a significant practical advantage.

The directed search algorithm introduced in Section 2 is closely related to prior work on dynamic test generation (e.g., [23, 15]). The algorithms discussed in these papers generate test inputs to exercise a *specific* program path or branch (to determine if its execution is feasible), starting with some (possibly random) execution path. In contrast, DART attempts to cover *all* executable program paths, in a style similar to systematic testing and model checking (e.g., [12]). It therefore does not use branch/predicate classification techniques as in [23, 15]. Also, prior work on dynamic test generation does not deal with functions calls, unknown code segments (such as library functions), how to check at run-time whether predictions about new test inputs are matched in the next run, and does not discuss completeness. Finally, to the best of our knowledge, dynamic test generation has never been implemented previously for a full-fledged programming language like C nor applied to large examples like the Needham-Schroeder protocol and the oSIP library.

DART is also more loosely related to the following work. QuickCheck [7] is a tool for random testing of Haskell programs which supports a test specification language where the user can assign probabilities to inputs. Korat [5] is a tool that can analyze a Java method's precondition on its input and automatically generate all possible non-isomorphic inputs up to a given (small) size. Continuous testing [34] uses free cycles on a developer's machine to continuously run regression tests in the background, providing feedback about test failures as source code is edited. Random interpretation [14] is an approximate form of abstract interpretation where code fragments are interpreted over a probabilistic abstract domain and their abstract execution sampled via random testing.

6. Conclusions

With DART, we have turned the conventional stance on the role of symbolic evaluation upside-down: symbolic reasoning is an adjunct to real execution. Randomization helps us where automated reasoning is impossible or difficult. For example, when we encounter 'malloc's we use randomization to guess the result of the allocation. Thus symbolic execution degrades gracefully in the sense that randomization takes over, by suggesting concrete values, when automated reasoning fails to suggest how to proceed.

DART's ability to execute and test any program that compiles *without writing any test driver/harness code* is a new powerful paradigm, we believe. Running a program for the first time usually brings interesting feedback (detects bugs), and DART makes this step almost effortless. We wrote "almost" because in practice, the user is still responsible for defining what a suitable self-contained unit is: it makes little sense to test in isolation functions that are tightly coupled; instead, DART should be applied to program/application interfaces where pretty much any input can be expected and should be dealt with. The user can also restrict the most general environment or test for functional correctness by adding interface code to the program in order to filter inputs (i.e., enforce pre-conditions) and analyze outputs (i.e., test post-conditions). We plan to explore how to effectively present to the user the interface identified by DART and let him/her specify constraints on inputs or outputs in a modular way.

Acknowledgments

We thank Dennis Dams, Cormac Flanagan, Alan Jeffrey, Rupak Majumdar, Darko Marinov, Kedar Namjoshi and Vic Zandy for helpful comments on this work. We are grateful to the anonymous reviewers for their comments on a preliminary version of this paper. This work was funded in part by NSF CCR-0341658. The work of Koushik Sen was done mostly while visiting Bell Laboratories, and we thank his advisor, Gul Agha, for the additional time and resources needed to complete this work.

References

[1] R. Alur, P. Cerny, G. Gupta, P. Madhusudan, W. Nam, and A. Srivastava. Synthesis of Interface Specifications for Java Classes. In *Proceedings of POPL'05 (32nd ACM Symposium on Principles of Programming Languages)*, Long Beach, January 2005.

[2] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, July 2001. Springer-Verlag.

[3] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In *Proceedings of ICSE'2004 (26th International Conference on Software Engineering)*. ACM, May 2004.

[4] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of ISSTA'2002 (International Symposium on Software Testing and Analysis)*, pages 123–133, 2002.

[6] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[7] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of ICFP'2000*, 2000.

[8] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically Closing Open Reactive Programs. In *Proceedings of PLDI'98 (1998 ACM SIGPLAN Conference on Programming Language Design and Implementation)*, pages 345–357, Montreal, June 1998. ACM Press.

[9] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *Proceedings of ICSE'2005 (27th International Conference on Software Engineering)*. ACM, May 2005.

[10] J. Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, Linkoping, October 1999.

[11] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, Seattle, August 2000.

[12] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of POPL'97 (24th ACM Symposium on Principles of Programming Languages)*, pages 174–186, Paris, January 1997.

[13] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Proceedings of TACAS'2002 (8th Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, Grenoble, April 2002.

[14] S. Gulwani and G. C. Necula. Precise Interprocedural Analysis using Random Interpretation. In *To appear in Proceedings of POPL'05 (32nd ACM Symposium on Principles of Programming Languages)*, Long Beach, January 2005.

[15] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 219–227, September 2000.

[16] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *Proceedings of PLDI'02 (2002 ACM SIGPLAN Conference on Programming Language Design and Implementation)*, pages 69–82, 2002.

[17] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Usenix Winter 1992 technical Conference*, pages 125–138, Berkeley, January 1992.

[18] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Portland, January 2002.

[19] S. Johnson. Lint, a C program checker, 1978. Unix Programmer's Manual, AT&T Bell Laboratories.

[20] Junit. web page: `http://www.junit.org/`.

[21] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[22] Klocwork. web page: `http://klocwork.com/index.asp`.

[23] B. Korel. A dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

[24] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 1995.

[25] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Proceedings of TACAS'1996 ((Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[26] lp_solve. web page: `http://groups.yahoo.com/group/lp_solve/`.

[27] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[28] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and transformation of C Programs. In *Proceedings of Conference on compiler Construction*, pages 213–228, 2002.

[29] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of POPL'02 (29th ACM Symposium on Principles of Programming Languages)*, pages 128–139, Portland, January 2002.

[30] R. Needham and M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[31] The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and technology, Planning Report 02-3, May 2002.

[32] J. Offut and J. Hayes. A Semantic Model of Program Faults. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 195–200, San Diego, January 1996.

[33] Polyspace. web page: `http://www.polyspace.com`.

[34] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *Proceedings of 2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, March 2004.

[35] S. D. Stoller. Domain Partitioning for Open Reactive Programs. In *Proceedings of ACM SIGSOFT ISSTA'02 (International Symposium on Software Testing and Analysis)*, 200.

[36] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04 (International Symposium on Software Testing and Analysis)*, Boston, July 2004.

[37] J. Whaley, M. C. Martin, and M. S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of ACM SIGSOFT ISSTA'02 (International Symposium on Software Testing and Analysis)*, 2002.

[38] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of TACAS'05 (11th Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 3440 of *LNCS*, pages 365–381. Springer, 2005.