

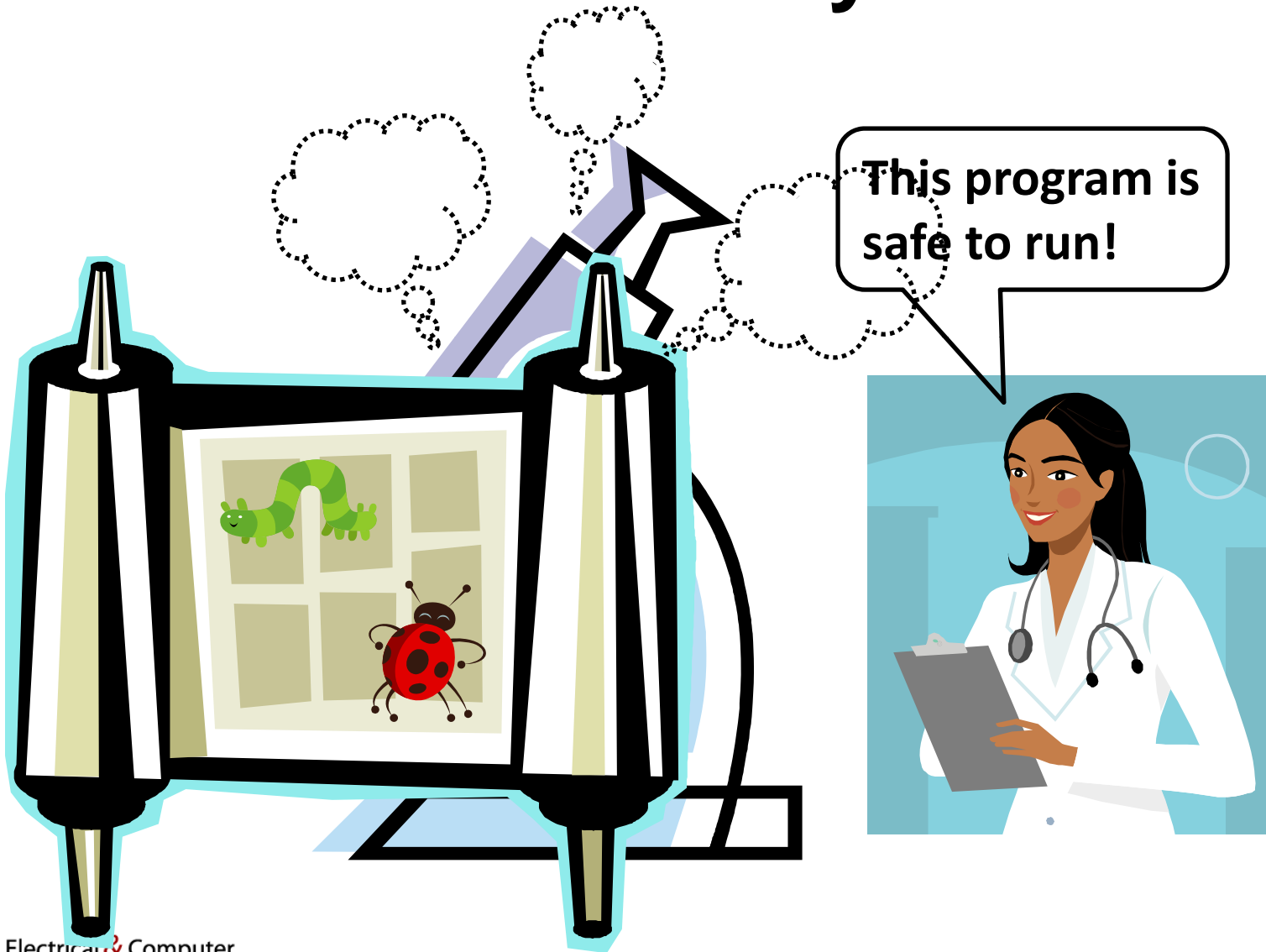
Dynamic Analysis: DART, CUTE, and EXE

Lujo Bauer

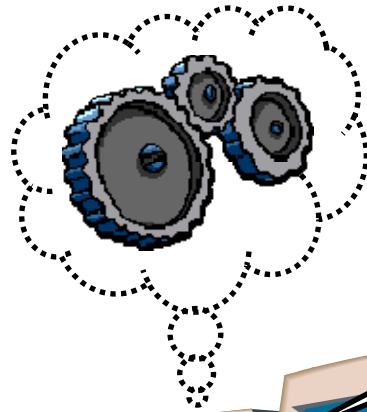
18-732

Fall 2010

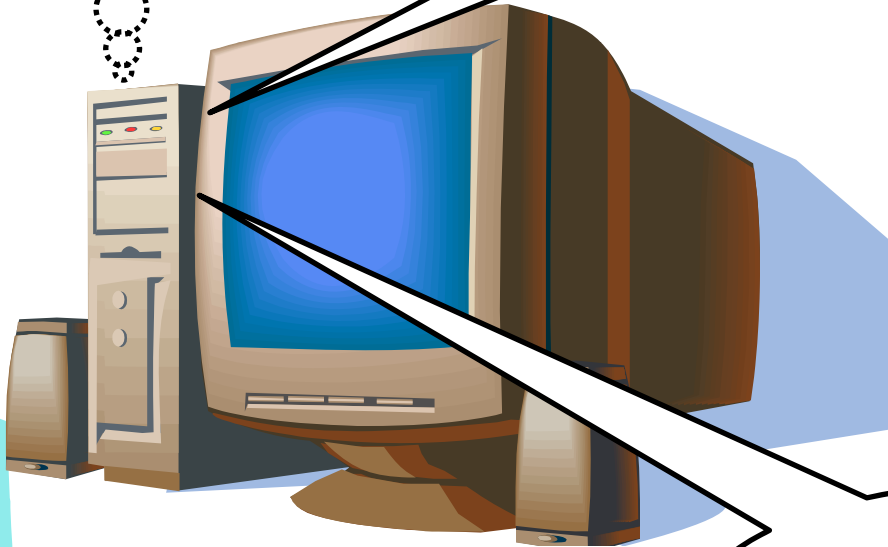
Static Analysis



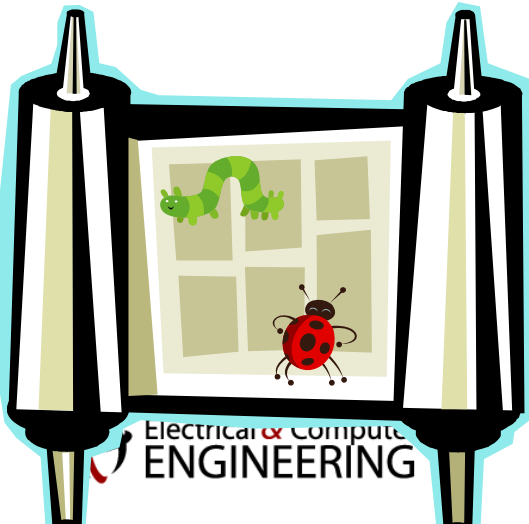
Dynamic Analysis



Program looks safe!



Program has bugs!
[if input=3, crash at line 12]



Today: DART, CUTE and EXE

- Some tools that perform dynamic analysis
 - DART: Directed automated random testing
 - CUTE: Concolic unit testing
 - EXE: Execution generated executions
- Goals
 - *Catch bugs that static analysis will miss*
 - Discover whether some inputs will cause the program to crash
 - If the program might crash, exhibit inputs that will cause this
- Mechanisms
 - Run program (in a safe way)
 - Test with random inputs
 - Test with symbolic inputs

DART

- DART algorithm
 1. Automatically extract interface to program from source
 2. Generate driver for random testing via this interface
 3. Dynamically direct executions along alternate paths
- Detects program crashes and assertion violations
- Can be applied to any program with little effort
 - Interface extraction and driver generation is automated

Step 1: Extract Interface

```
int double(int x) {  
    return 2 * x;  
}
```

```
void test_me (int x, int y) {  
    int z = double(x);  
    if (z==y) {  
        if (y == x+10)  
            error();  
    }  
}
```

- User specifies top-level function
- DART detects:
 - Arguments to top-level function
 - External variables
 - External functions

Step 2: Generate Testing Driver

```
int double(int x) {  
    return 2 * x;  
}
```

```
void test_me (int x, int y) {  
    int z = double(x);  
    if (z==y) {  
        if (y == x+10)  
            error();  
    }  
}
```

- Generate driver function that will allow automated testing with random inputs

```
main() {  
    int tmp1 = randomInt();  
    int tmp2 = randomInt();  
    test_me (tmp1,tmp2);  
}
```

- Also initialize (randomly) external variables, generate (random) drivers for external functions
 - None in this example

Step 2: Generate Testing Driver

```
main() {  
    int tmp1 = randomInt();  
    int tmp2 = randomInt();  
    test_me (tmp1,tmp2);  
}  
  
int double(int x) {  
    return 2 * x;  
}  
  
void test_me (int x, int y) {  
    int z = double(x);  
    if (z==y) {  
        if (y == x+10)  
            error();  
    }  
}
```

- Generate driver function that will allow automated testing with random inputs
- However, unlikely that random testing will pick values that will cause execution to reach “error()”

Step 3: Directed Search

```
main() {  
    int tmp1 = randomInt();  
    int tmp2 = randomInt();  
    test_me (tmp1,tmp2);  
}  
  
int double(int x) {  
    return 2 * x;  
}  
  
void test_me (int x, int y) {  
    int z = double(x);  
    if (z==y) {  
        if (y == x+10)  
            error();  
    }  
}
```

- For every if-then statement, attempt to derive inputs that will cause execution to go along each branch
- Approach: combined concrete and symbolic execution

Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

x=36, y=99,
z=72

x, y
z=2*x



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

x=36, y=99,
z=72

x, y

z=2*x

2*x != y



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```

Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

Solve: $2*x = y$
Solution: $x=1, y=2$

x, y

$2*x \neq y$

$x=36, y=99,$
 $z=72$

$z=2*x$



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

x=1, y=2,
z=2

x, y
z=2*x



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

$x=1, y=2,$
 $z=2$

x, y

$z=2*x$

$2*x == y$



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

$x=1, y=2,$
 $z=2$

x, y

$z=2*x$

$2*x == y$
 $y != x+10$



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```

Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

Solve: $2*x == y$
 $y == x+10$
 Solution: $x=10, y=20$

$x=1, y=2,$
 $z=2$

x, y

$z=2*x$

$2*x == y$
 $y != x+10$



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

x=10, y=20

x, y



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

x=10, y=20,
z=20

x, y
z=2*x



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

$x=10, y=20,$
 $z=20$

x, y

$z=2*x$

$2*x == y$



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

$x=10, y=20,$
 $z=20$

x, y

$z=2*x$

$2*x == y$
 $y == x+10$



Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



Concrete
execution

Symbolic
execution

concrete
state

symbolic
state

path
constraints

$x=10, y=20,$
 $z=20$

x, y

$z=2*x$

$2*x == y$
 $y == x+10$

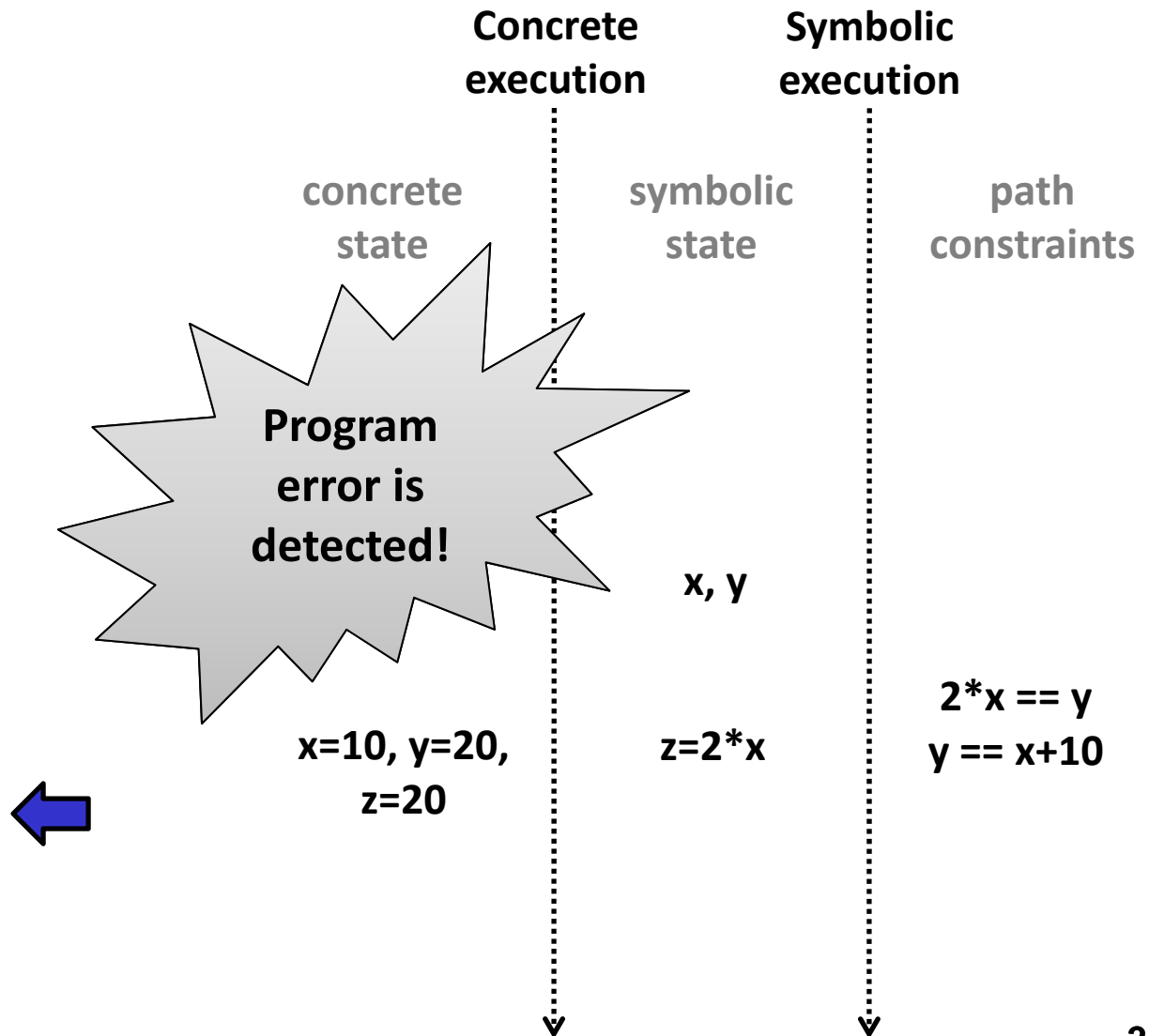


Step 3: Directed Search

```
main() {
  int tmp1 = randomInt();
  int tmp2 = randomInt();
  test_me (tmp1,tmp2);
}
```

```
int double(int x) {
  return 2 * x;
}
```

```
void test_me (int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      error();
  }
}
```



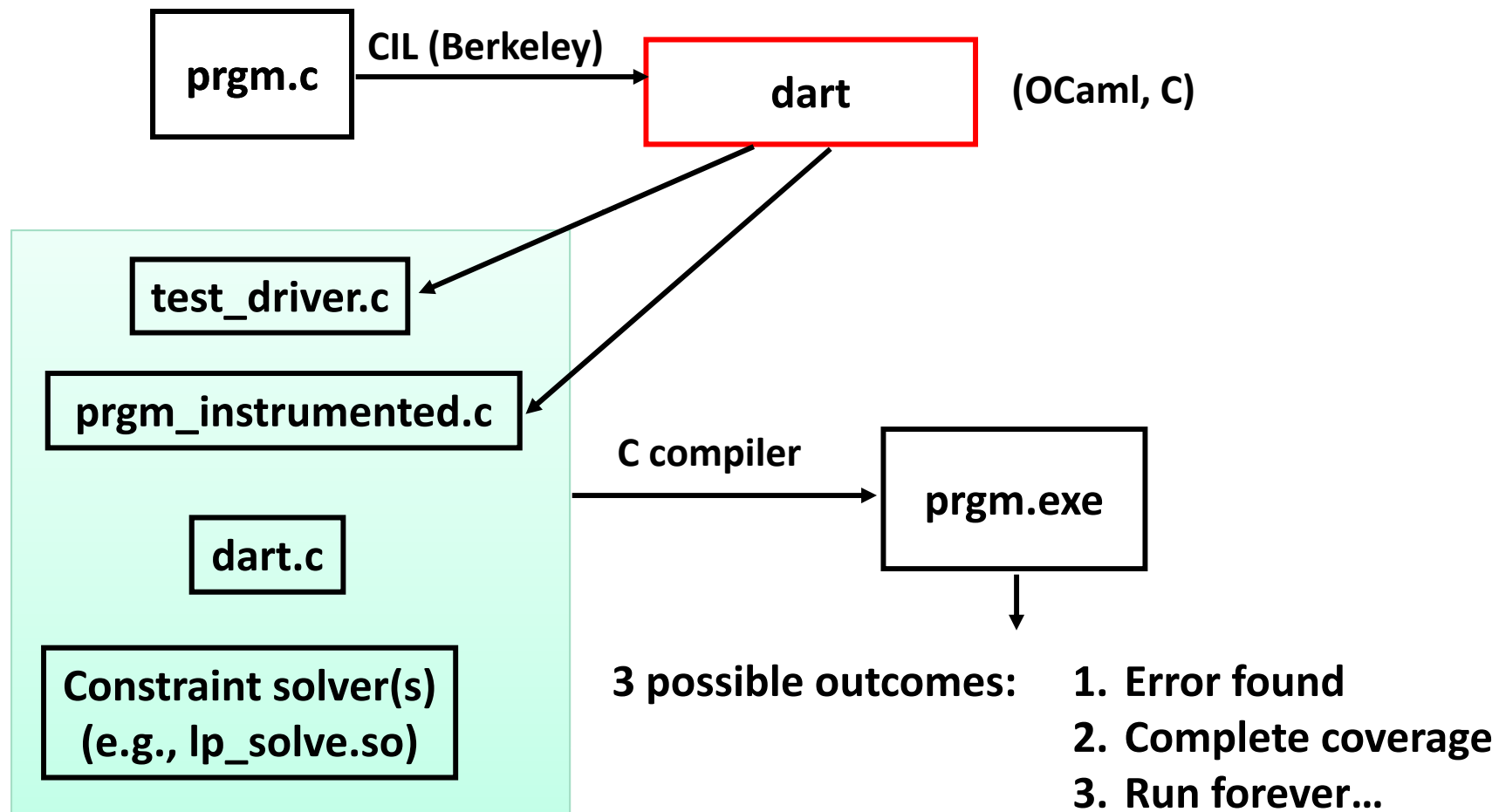
Directed Search: Summary

- Dynamic test generation to direct executions along alternative program paths
 - Collect symbolic constraints at branch points (whenever possible)
 - Negate one constraint at a branch point to take other branch (say b)
 - Call constraint solver with new path constraint to generate new test inputs
 - Next execution driven by these new test inputs to take alternative branch b
 - Check with dynamic instrumentation that branch b is indeed taken
- Repeat this process until all execution paths are covered
 - May never terminate!
- Significantly improves code coverage vs. pure random testing

Directed Search: Unsolvable Constraints

- What if a symbolic constraint can't be solved?
 - $z = x * x * x$
- DART: substitute concrete value for symbolic value, continue execution
 - Suppose $x=3, z=27$
 - Then, path constraint becomes $z \neq 27$
(instead of $z \neq x * x * x$)
- Made possible by concurrent *concrete* and *symbolic* execution

DART for C: Implementation Details



DART in Practice: NS Auth. Protocol

- Tested a C implementation of a security protocol (Needham-Schroeder public key authentication) with a known attack
 - About 400 lines of C code; experiments on 800Mz P-III machine
 - < 2 seconds (664 runs) to discover a (partial) attack, with an unconstrained (possibilistic) intruder model
 - 18 minutes (328,459 runs) to discover a (full) attack, with a realistic (Dolev-Yao) intruder model
 - found a new bug in C implementation of Lowe's fix to the NS protocol (after 22 minutes of search; bug confirmed by the code's author)
- In contrast:
 - Systematic state-space search of program composed with concurrent nondeterministic intruder model using VeriSoft (a software model checker) does not find the attack

DART in Practice

- oSIP library (call establishment over IP)
 - 30k lines of code, 600 externally visible functions
 - Found numerous bugs, mostly null pointer dereferences

Limitations of DART

- Constraints may get too complicated
 - Some constraints can't be solved symbolically
 - Control flow too complicated (loops)
 - Constraints too complicated ($x=y^3$)
 - Random guesses might miss
 - Options for dealing with complicated branches:
 - Skip branch (possible false negative)
 - Force execution down branch (possible false positive + no example)
 - Keep trying new random inputs (possible nontermination)

Limitations of DART

- Can't detect all unwanted behavior
 - Null pointer dereferencing is easy to detect
 - Semantically incorrect behavior is very hard to detect
 - C isn't type-safe

Limitations of DART

- Unaware of external constraints
 - There may be external checks of arguments passed to tested code
- Can't always distinguish between reasons for termination
 - Did program terminate correctly or because of a fault
- Exploring some branches of execution may be less productive (or more expensive) than exploring others
 - How to decide what to test first?

Advantages of DART

- Catches many errors in real programs
 - Including errors that static analysis would miss
- No false positives
 - Under certain assumptions (e.g., no external constraints)
- Allows automated testing

Other Similar Tools: CUTE

Sen and Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, 2006

- Follow-on work to DART
 - Same approach
- Handles more complicated constraints
 - Pointer arithmetic, some data structures (C structs)

```
a->c = 1;
```

```
memset(a,0,sizeof(struct foo));
```

```
if (a->c != 1) { ... }
```

Other Similar Tools: CUTE

- Much more efficient constraint solver
 - 100x to 1000x faster
- Doesn't automate test driver generation
 - User specifies functions, *preconditions*
 - Less automation (more work), but more precision
- Bounded depth-first search
- Roughly speaking, more capable than DART

Other Similar Tools: EXE

Cadar, Ganesh, Pawlowski, Dill, and Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006

- Developed independently from and simultaneously with DART & CUTE
- Focus on symbolic execution and constraint solving
 - Fast, capable solver (STP)
 - E.g., pointer offsets
 - No automated driver generation
 - No random generation of inputs
- Heuristics for directing search

Other Similar Tools: EXE

```
#include <assert.h>
int main(void) {
    unsigned i,t,a[4]={1,3,5,2};
    make_symbolic(&i);
    if (i>=4)
        exit(0);
    char *p = (char*)a + i*4;
    *p = *p - 1;
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
}
```

- Variables must be explicitly (and manually) labeled as symbolic
 - Must know what to label

Other Similar Tools: EXE

```
#include <assert.h>
int main(void) {
    unsigned i,t,a[4]={1,3,5,2};
    make_symbolic(&i);
    if (i>=4)
        exit(0);
    char *p = (char*)a + i*4;
    *p = *p - 1;
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
}
```

- Variables must be explicitly (and manually) labeled as symbolic
 - Must know what to label
- Handles pointer offsets

Other Similar Tools: EXE

```
#include <assert.h>
int main(void) {
    unsigned i,t,a[4]={1,3,5,2};
    make_symbolic(&i);
    if (i>=4)
        exit(0);
    char *p = (char*)a + i*4;
    *p = *p - 1;
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
}
```

- Variables must be explicitly (and manually) labeled as symbolic
 - Must know what to label
- Handles pointer offsets
- Found bugs in packet filters (BPF), DHCP servers (udhcpd), file systems (ext2, ext3, JFS)
- Hard to compare evenly with DART & CUTE
 - More powerful solver (maybe), but no random generation of inputs, more manual effort

Random Testing + Symbolic Execution

Wrap Up

- **Pros:**
 - Find class of errors not detectable by static analysis
 - Including on many real programs
 - No false positives
 - Can be fairly automated
- **Cons:**
 - Can't find all errors (or all classes of errors)
 - Complicated control flow or data structures
 - Constraints that can't be solved
 - Potentially slow
 - Minutes or hours
 - May have false negatives (EXE) or not terminate (DART)

Sources

- Cadar, Ganesh, Pawlowski, Dill, and Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.
- Godefroid, Klarlund, and Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.
- Slides from R. Alur's CIS 670: Program Analysis, Fall 2007, University of Pennsylvania (including slides from DART talk at PLDI '05)