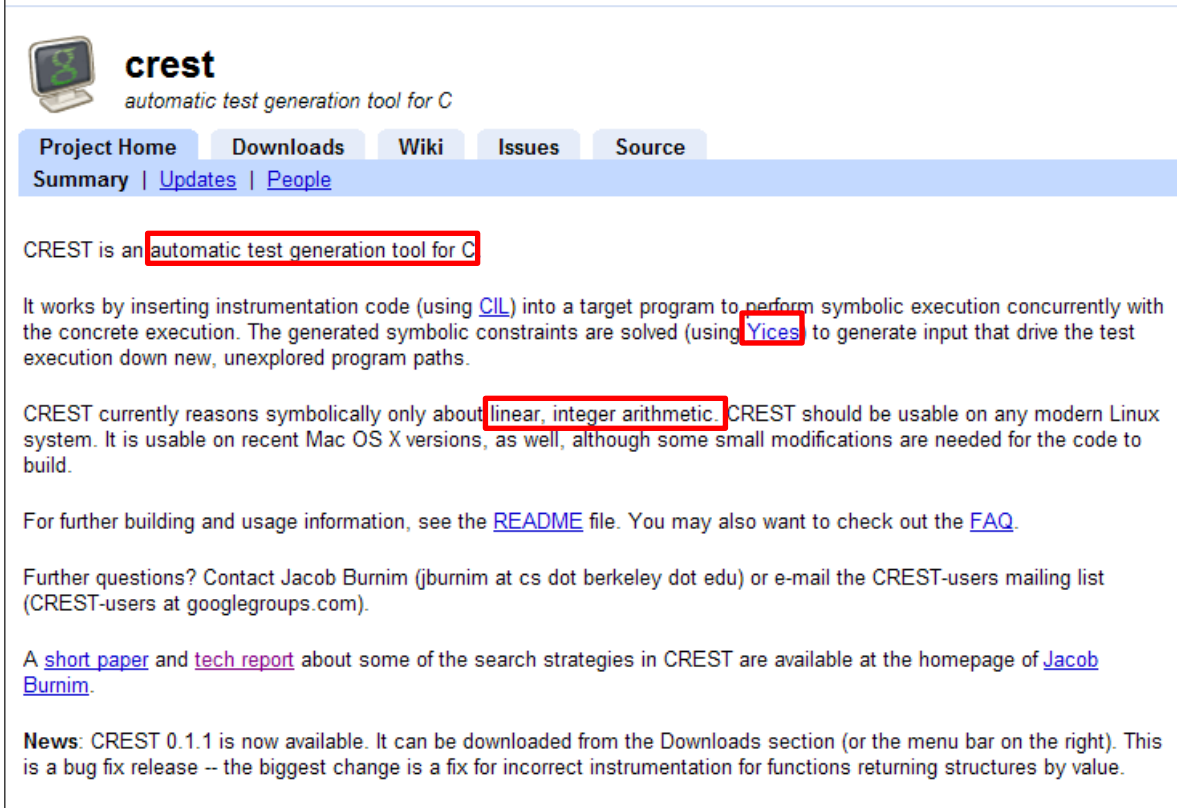
CREST Tutorial

Moonzoo Kim

CS Dept. KAIST



crest

automatic test generation tool for C

Project Home | Downloads | Wiki | Issues | Source

Summary | Updates | People

CREST is an automatic test generation tool for C.

It works by inserting instrumentation code (using CIL) into a target program to perform symbolic execution concurrently with the concrete execution. The generated symbolic constraints are solved (using Yices) to generate input that drive the test execution down new, unexplored program paths.

CREST currently reasons symbolically only about linear, integer arithmetic. CREST should be usable on any modern Linux system. It is usable on recent Mac OS X versions, as well, although some small modifications are needed for the code to build.

For further building and usage information, see the README file. You may also want to check out the FAQ.

Further questions? Contact Jacob Burnim (jburnim at cs dot berkeley dot edu) or e-mail the CREST-users mailing list (CREST-users at googlegroups.com).

A short paper and tech report about some of the search strategies in CREST are available at the homepage of Jacob Burnim.

News: CREST 0.1.1 is now available. It can be downloaded from the Downloads section (or the menu bar on the right). This is a bug fix release -- the biggest change is a fix for incorrect instrumentation for functions returning structures by value.

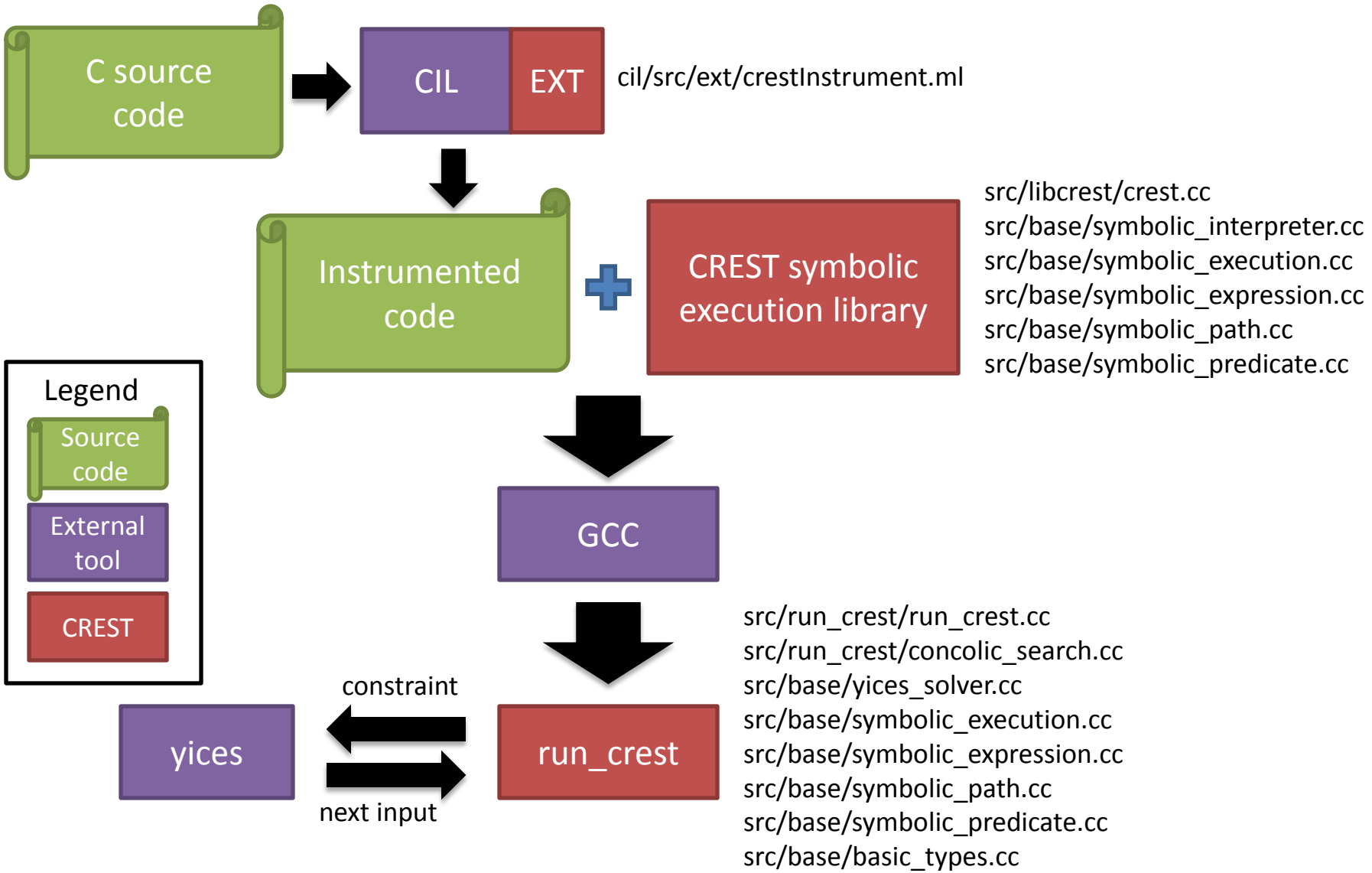
CREST

- CREST is a concolic testing tool for C programs
  - Generate test inputs automatically
  - Execute target under test on generated test inputs
  - Explore all possible execution paths of a target systematically

- CREST is a open-source re-implementation of CUTE
  - mainly written in C++
    - CREST's instrumentation is implemented as a module of CIL(C Intermetiate Language) written in Ocaml

Moonzoo Kim
SWTV Group
KAIST

Overview of CREST code



Moonzoo Kim
SWTV Group
KAIST

4 Main Steps of Concolic Testing

1. Instrumentation of a target program
   - To insert probes to build symbolic path formula
2. Transform a constructed symbolic path formula to SMT-compatible format
   - SMT solvers can solve simple formula only
3. Select one branch condition to negate
   - Core technique impacting both effectiveness and efficiency
4. Invoking SMT solvers on the SPF SMT formula
   - Selection of a SMT solver and proper configuration parameters

Preprocessor of Concolic Testing: CIL and Ocaml

Back-end Engine of Concolic Testing: SMT solver and API Application in CREST

Real-world Concolic Testing 1: Memory model and CFG algorithm

Real-world Concolic Testing 2: Hybrid Concolic + Genetic and distributed DFS algorithm

4 Main Tasks of Human Engineers

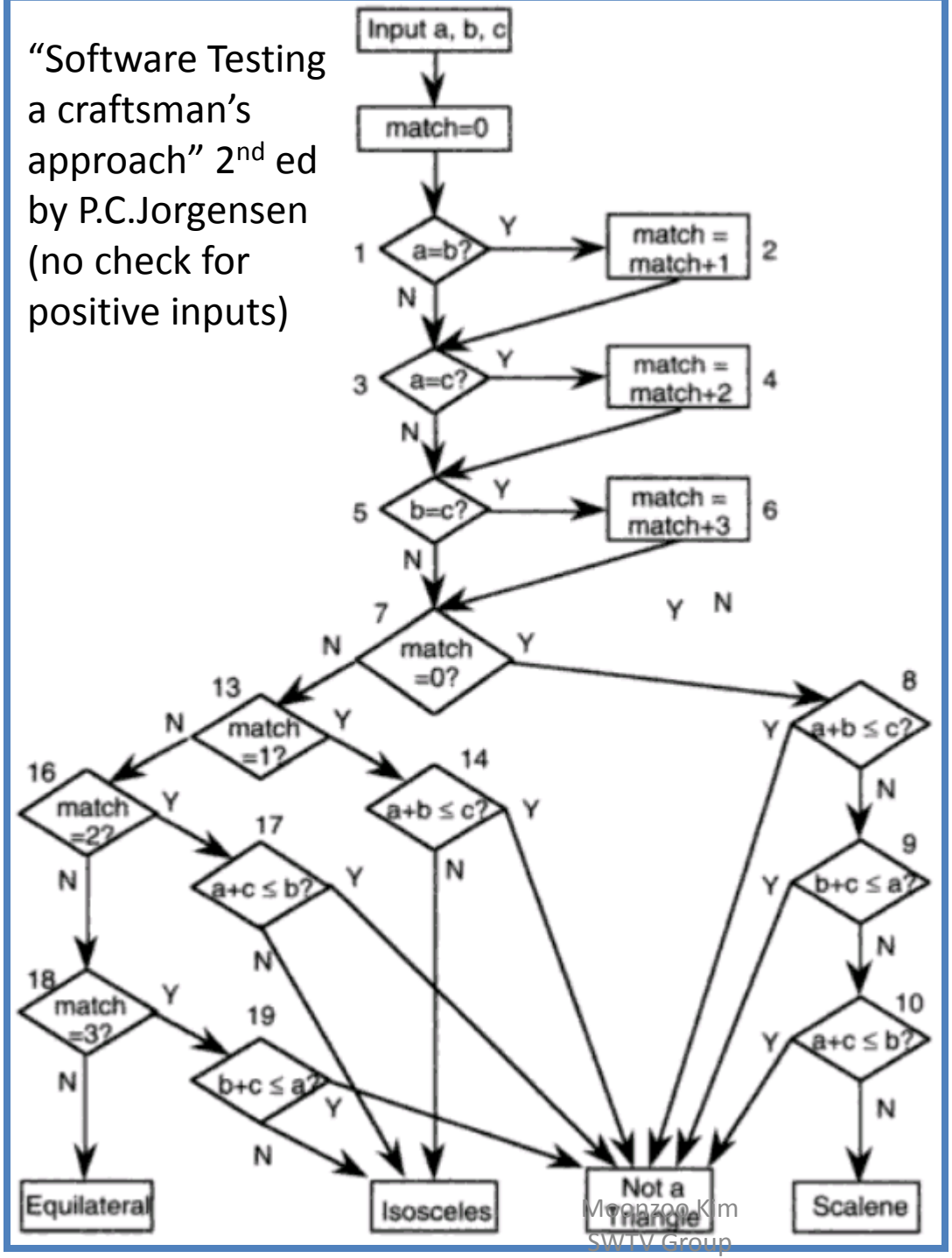
1. Adding proper assert() statements
   - W/o assert(), no test results obtained
2. Selection of symbolic variables in a target program
   - Identify which parts of a target program are most important
3. Construction of symbolic external environment
   - To detect real bugs
4. Performance tuning and debugging
   - To obtain better concolic testing results

SAT based Automated Program Analysis Technique: a Case Study on Flash Memory File System

Real-world case study: Libexif (system level testing) and security lib (unit level testing)

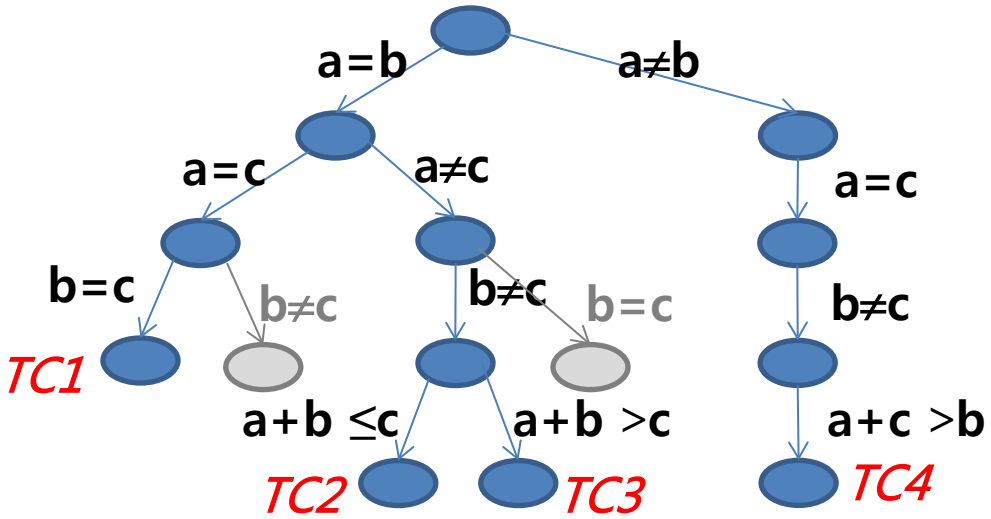
```
1 #include <crest.h>
2 main() {
3    int a,b,c, match=0;
4    CREST_int(a); CREST_int(b); CREST_int(c);
     // filtering out invalid inputs
5    if(a <= 0 || b<= 0 || c<= 0) exit();
6    printf("a,b,c = %d,%d,%d:",a,b,c);
7
8    //0: Equilateral, 1:Isosceles,
     // 2: Not a traiangle, 3:Scalene
9    int result=-1;
10    if(a==b) match=match+1;
11    if(a==c) match=match+2;
12    if(b==c) match=match+3;
13    if(match==0) {
14       if( a+b <= c) result=2;
15       else if( b+c <= a) result=2;
16       else if(a+c <= b) result =2;
17       else result=3;
18    } else {
19       if(match == 1) {
20          if(a+b <= c) result =2;
21          else result=1;
22       } else {
23          if(match ==2) {
24             if(a+c <=b) result = 2;
25             else result=1;
26          } else {
27             if(match==3) {
28                if(b+c <= a) result=2;
29                else result=1;
30             } else result = 0;
31          } }}
32    printf("result=%d\n",result);
33 }
```

"Software Testing a craftsman's approach" 2nd ed by P.C.Jorgensen (no check for positive inputs)



Concolic Testing the Triangle Program

Test case	Input (a,b,c)	Executed path conditions (PC)	Next PC	Solution for the next PC
1	1,1,1	a=b ∧ a=c ∧ b=c	a=b ∧ a=c ∧ b≠c	Unsat
			a=b ∧ a≠c	1,1,2
2	1,1,2	a=b ∧ a≠c ∧ b≠c ∧ a+b ≤c	a=b ∧ a≠c ∧ b≠c ∧ a+b > c	2,2,3
3	2,2,3	a=b ∧ a≠c ∧ b≠c ∧ a+b >c	a=b ∧ a≠c ∧ b=c	Unsat
			a≠b	2,1,2
4	2,1,2	a≠b ∧ a=c ∧ b≠c ∧ a+c>b	a≠b∧ a=c ∧ b≠c ∧ a+c ≤b	2,5,2



CREST Commands

- `crestc <filename>.c`
  - Output
    - `<filename>.cil.c` // instrumented C file
    - `branches` // lists of paired branches
    - `<filename>` // executable file
- `run_crest ./filename <n> -[dfs|cfg|random|random_input|hybrid]`
  - <n>: # of iterations/testings
  - `dfs`: depth first search (but in reverse order)
  - `cfg`: uncovered branch first
  - `random`: negated branch is randomly selected
  - `random_input`: pure random input
  - `hybrid`: combination of dfs and random

Instrumented C Code

```
#line 10
 { /* Creates symbolic expression a==b */
  __CrestLoad(36, (unsigned long )(& a), (long long )a);
  __CrestLoad(35, (unsigned long )(& b), (long long )b);
  __CrestApply2(34, 12, (long long )(a == b));
  if (a == b) {

    __CrestBranch(37, 11, 1); //extern void __CrestBranch(int id , int bid , unsigned char b )
    __CrestLoad(41, (unsigned long )(& match), (long long )match);
    __CrestLoad(40, (unsigned long )0, (long long )1);
    __CrestApply2(39, 0, (long long )(match + 1));
    __CrestStore(42, (unsigned long )(& match));
     match ++;

  } else {
    __CrestBranch(38, 12, 0);
  } }
```

Execution Snapshot

```
[moonzoo@verifier crest]$ run_crest ./triangle 10000 -dfs
Iteration 0 (0s): covered 0 branches [0 reach funs, 0 reach branches].
Iteration 1 (0s): covered 1 branches [1 reach funs, 32 reach branches].
Iteration 2 (0s): covered 3 branches [1 reach funs, 32 reach branches].
Iteration 3 (0s): covered 5 branches [1 reach funs, 32 reach branches].
a,b,c = 1,1,1:result=0
Iteration 4 (0s): covered 13 branches [1 reach funs, 32 reach branches].
a,b,c = 2,1,1:result=2
Iteration 5 (0s): covered 17 branches [1 reach funs, 32 reach branches].
a,b,c = 2,1,2:result=1
Iteration 6 (0s): covered 20 branches [1 reach funs, 32 reach branches].
a,b,c = 1,2,1:result=2
Iteration 7 (0s): covered 21 branches [1 reach funs, 32 reach branches].
a,b,c = 3,2,1:result=2
Iteration 8 (0s): covered 24 branches [1 reach funs, 32 reach branches].
a,b,c = 2,1,3:result=2
Iteration 9 (0s): covered 25 branches [1 reach funs, 32 reach branches].
a,b,c = 4,3,2:result=3
Iteration 10 (0s): covered 27 branches [1 reach funs, 32 reach branches].
a,b,c = 2,3,1:result=2
Iteration 11 (0s): covered 28 branches [1 reach funs, 32 reach branches].
a,b,c = 3,2,2:result=1
Iteration 12 (0s): covered 29 branches [1 reach funs, 32 reach branches].
a,b,c = 2,2,1:result=1
Iteration 13 (0s): covered 31 branches [1 reach funs, 32 reach branches].
a,b,c = 1,1,2:result=2
Iteration 14 (0s): covered 32 branches [1 reach funs, 32 reach branches].
elapsed time = 0.0015093
```

```
[moonzoo@verifier crest]$ cat coverage
3 /* covered branch ids*/
4
5
6
7
8
11
12
14
15
17
18
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
```

Supported Symbolic Datatypes

- #define CREST_unsigned_char(x) __CrestUChar(&x)
- #define CREST_unsigned_short(x) __CrestUShort(&x)
- #define CREST_unsigned_int(x) __CrestUInt(&x)
- #define CREST_char(x) __CrestChar(&x)
- #define CREST_short(x) __CrestShort(&x)
- #define CREST_int(x) __CrestInt(&x)

Decision/Condition Coverage Analysis by CREST

```
1 int main(){
2     int A, B, C, D;
3     if (A && B || C && D){
4         printf("Yes\n");
5     }else{
6         printf("No\n");
7     }
8 }
```

- CREST consider all possible cases with short-circuit
- Thus, coverage reported by CREST might be lower than actual branch coverage

```
1   if (A != 0) {
2     __CrestBranch(5, 2, 1);
3     if (B != 0) {
4       __CrestBranch(10, 3, 1);
5       printf("Yes\n");
6     } else {
7       __CrestBranch(11, 4, 0);
8       goto _L;
9     }
10  } else {
11    __CrestBranch(6, 5, 0);
12    _L: /* CIL Label */
13    if (C != 0) {
14      __CrestBranch(16, 6, 1);
15      if (D != 0) {
16        __CrestBranch(21, 7, 1);
17        printf("Yes\n");
18      } else {
19        __CrestBranch(22, 8, 0);
20        printf("No\n");
21      }
22    } else {
23      __CrestBranch(17, 9, 0);
24      printf("No\n");
25    }
26  }
```

A == T

A == T && B == T

A == T && B != T

A != TRUE

(A != T || A == T && B != T) && C == T

(A != T || A == T && B != T) && C == T && D == T

(A != T || A == T && B != T) && C == T && D != T

(A != T || A == T && B != T) && C != T