



# Learn to Build Automated Software Analysis Tools with Graph Paradigm and Interactive Visual Framework

GLAN Jaipur, September 12-16, 2016

**Suresh C. Kothari**  
**Richardson Professor**  
**Department of Electrical and Computer Engineering**

**Ben Holland, Iowa State University**

## Module IV: Accuracy and Scalability Barriers for Analyzing Large Software

**Acknowledgement:** Team members at Iowa State University and EnSoft, DARPA contracts FA8750-12-2-0126 & FA8750-15-2-0080

# Module Outline

- *Exponentiality of points to sets*
- *Exponentiality of control flow paths*
- *Exponentiality of path feasibility*
- A tug-of-war between *scalability* and *accuracy*
- Solutions to counter-act exponentiality:
  - Binary Decision Diagram (BDD)
  - Projected Control Graph

# Terminology

A **control block** is the sequence of statements forming the body of a control statement.

A **branch node** corresponds to a conditional statement from which mutually exclusive branches emanate. A conditional statement could be a conditional loop.

**Structured Program**: Every control block has exactly one entry. Could have multiple exits.

**Unstructured Program**: A control block can have multiple entries.

**Control Flow Graph (CFG)**: The graph with one nodes for each statement and each edge depicting the control flow from one statement to another.

A **CFG path (execution path)** follows branches with one branch at each branch node it hits. Each such branch is called a path segment.

A **path is infeasible** if it includes mutually exclusive path segments

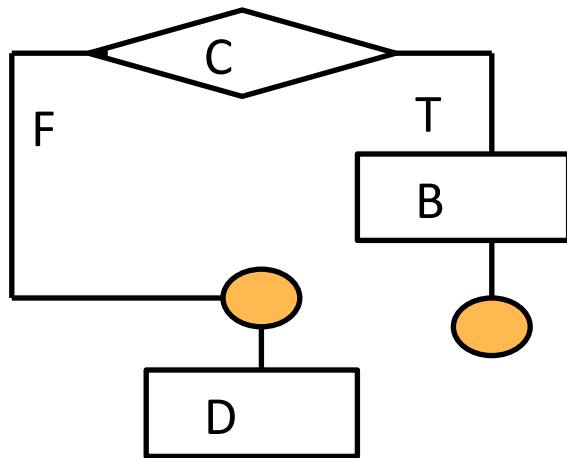
An **event** is execution result of a statement relevant to validating a safety property

**Conservative analysis** is reasoning based on aggregate of events along all branches at a branch node

**Path-sensitive analysis** reasoning based on only the events along individual paths

# Multiple exits in control block

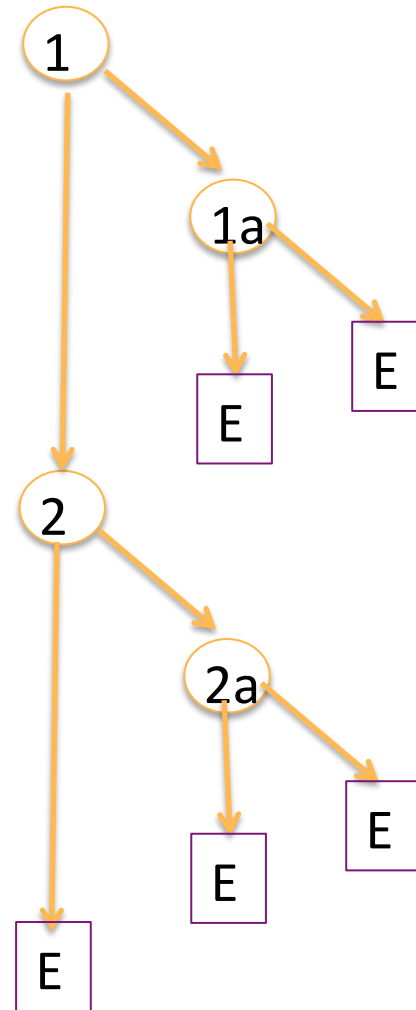
```
if ( (q=dsptr->dreqlst) == DRNULL ) {  
    dsptr->dreqlst = drptr;  
    drptr->drnext = DRNULL;  
    dskstrt(dsptr);  
    return(DONQ);  
}
```



Two execution paths

# Execution Path Analysis Examples - I

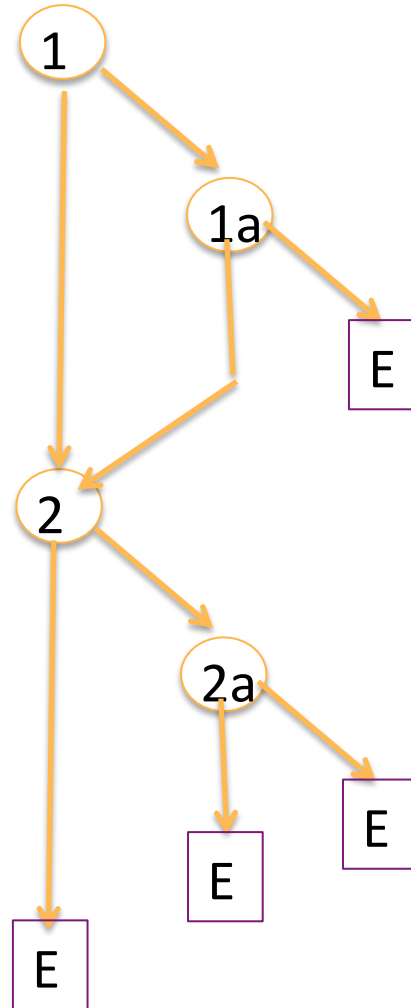
```
if ( C1 ) {  
    if ( C2 ) {  
        do something ;  
        return();  
    }  
    do something;  
    return();  
}  
  
if ( C3 ) {  
    if ( C4 ) {  
        do something ;  
        return();  
    }  
    do something;  
    return();  
}
```



Five execution paths

# Execution Path Analysis Examples - II

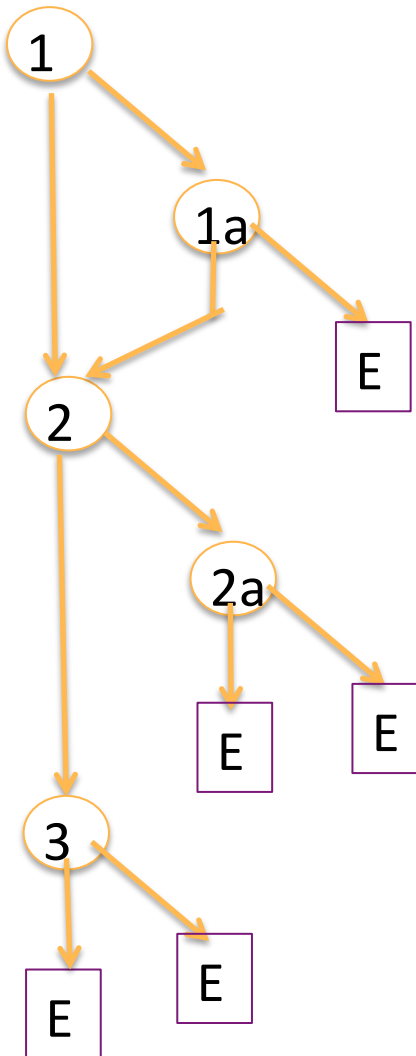
```
if ( C1 ) {  
    if ( C2 ) {  
        do something ;  
        return();  
    }  
    do something;  
}  
  
if ( C3 ) {  
    if ( C4 ) {  
        do something ;  
        return();  
    }  
    do something;  
    return();  
}
```



Seven execution paths

# Execution Path Analysis Examples - II

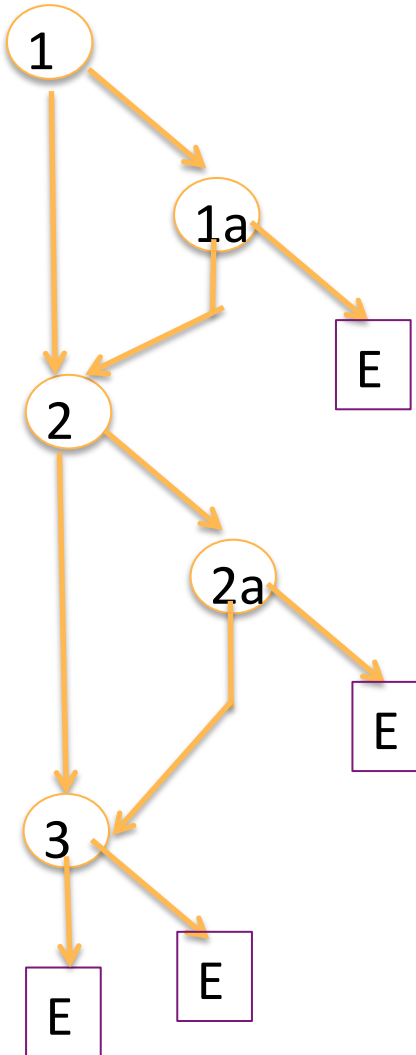
```
if ( C1 ) {  
    if ( C2 ) {  
        do something ;  
        return();  
    }  
    do something;  
}  
if ( C3 ) {  
    if ( C4 ) {  
        do something ;  
        return();  
    }  
    do something;  
    return();  
}  
if ( C5 ){  
    do something ;  
    return();  
}
```



Nine execution paths

# Execution Path Analysis Examples - III

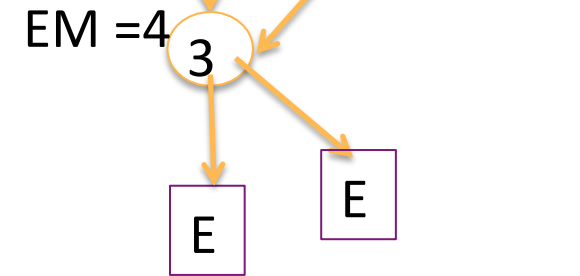
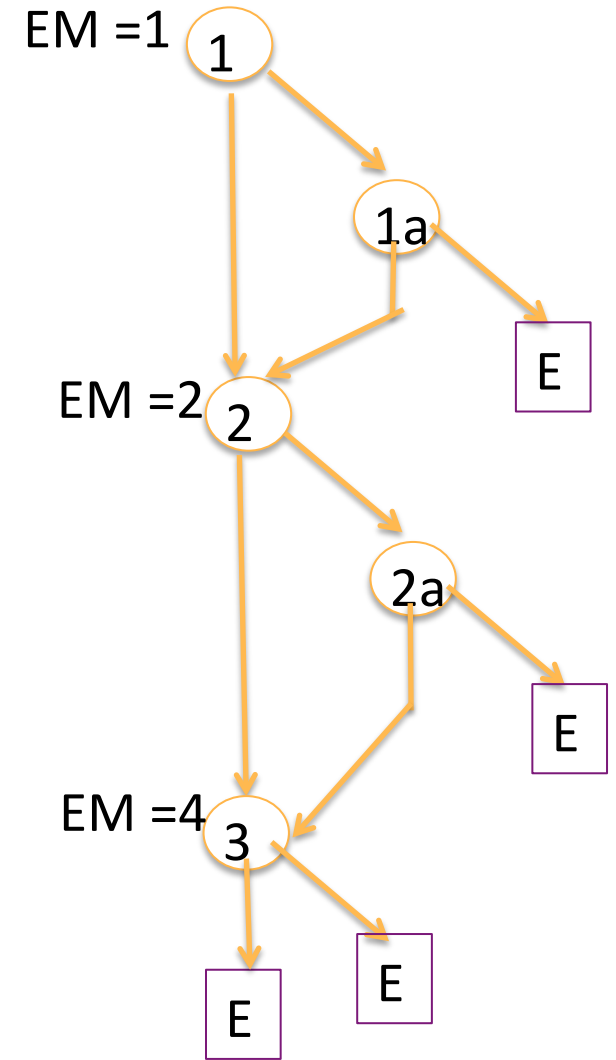
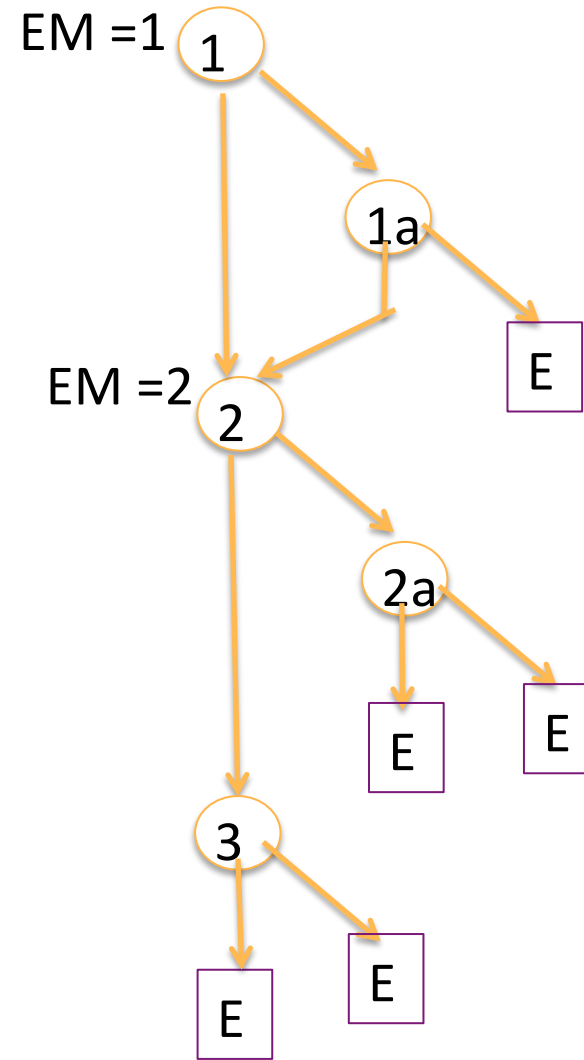
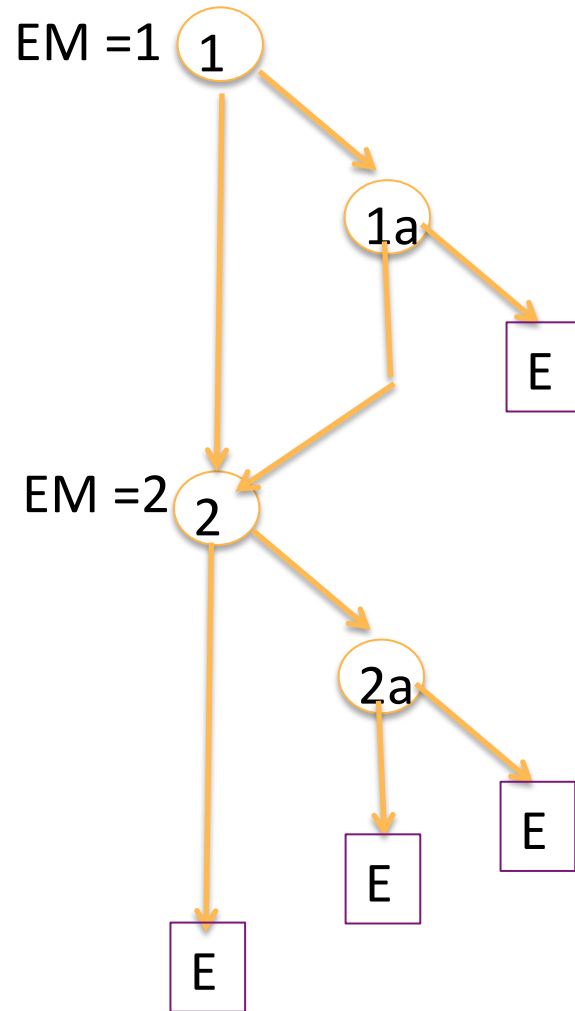
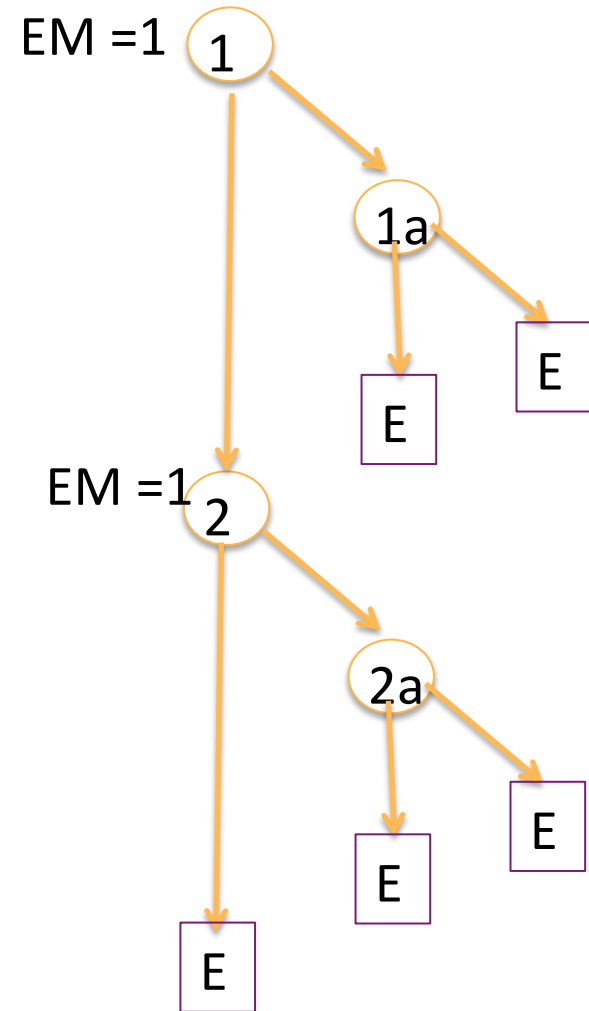
```
if ( C1 ) {  
    if ( C2 ) {  
        do something ;  
        return();  
    }  
    do something;  
}  
if ( C3 ) {  
    if ( C4 ) {  
        do something ;  
        return();  
    }  
    do something;  
}  
if ( C5 ){  
    do something ;  
    return();  
}
```



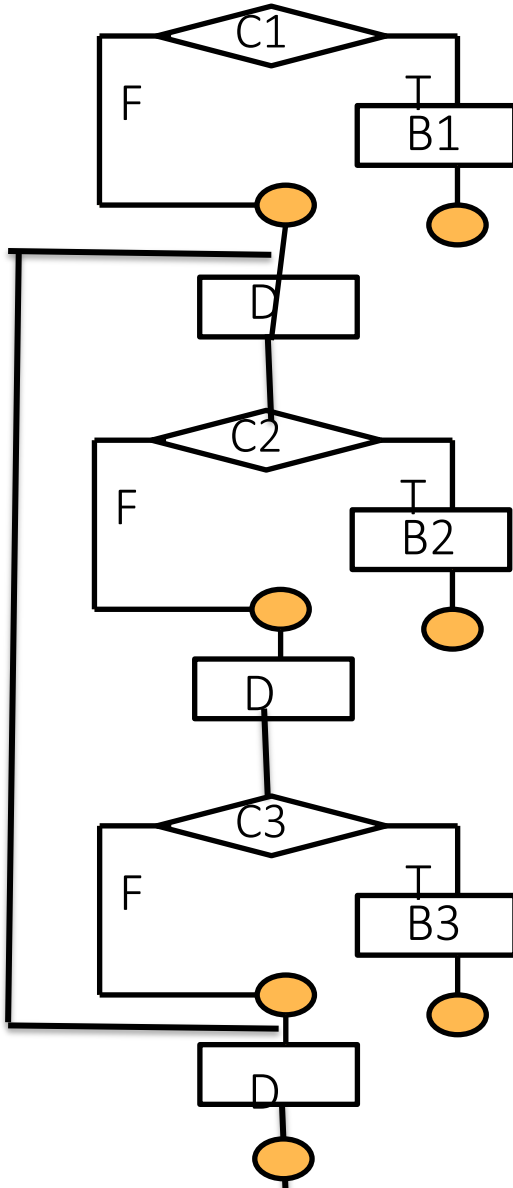
Eleven execution paths



# Entry point multiplicity (EM)



# A Loop Example



Path 1: C1 = F, C2 = F, C3 = F, Loop executed

Path 2: C1 = F, Loop not executed

Path 3: C1 = F, C2 = T, BREAK B2

Path 4: C1 = F, C2 = F, C3 = T, BREAK B3

Path 5: C1 = T, BREAK B1

# Loops without BREAKs

- Scenario 1: Loop contains A(X) (allocation) followed by D(X) (deallocation) on all feasible execution paths within a loop - safe, no matter how many times the loop iterates
- Scenario 2: Loop contains D(X) (deallocation) followed by A(X) (allocation) on all feasible execution paths within a loop – safe if the loop is preceded A(X) and succeeded by D(X).
- OK to treat multiple iterations of the Loop as a single execution path.

# Loops with BREAK

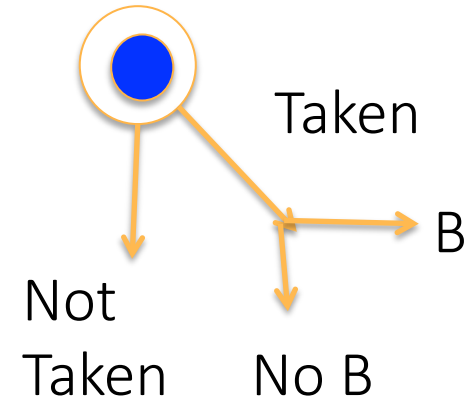
- Scenario 3: Loop contains A(X) (allocation) followed a followed by D(X) (deallocation) on all feasible execution paths within a loop but with a BREAK in between on some paths.
- OK to treat multiple iterations of the Loop as two equivalent execution paths:
  - One path without a BREAK – safe.
  - Another path with a BREAK – not safe without D(X) on the path following BREAK.

# Loops with BREAK

- Scenario 4: Loop contains D(X) (deallocation) followed a followed by A(X) (allocation) on all feasible execution paths within a loop but with a BREAK in between on some paths.
- OK to treat multiple iterations of the Loop as two equivalent execution paths:
  - One path without a BREAK – safe if loop is followed by D(X).
  - Another path with a BREAK – safe without D(X) on the path following BREAK.

# A Flow diagram example for a loop

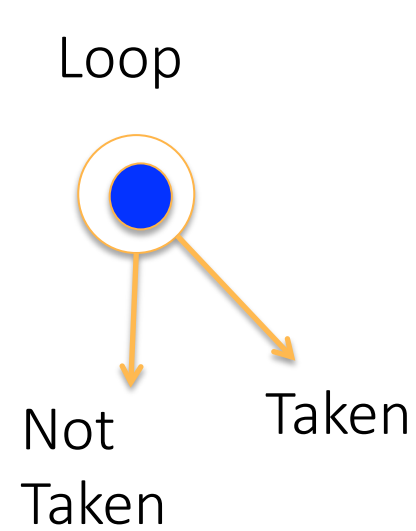
```
for(i=1;i<=n;++i){  
    read num;  
    if(num<0)  
        break;  
    sum= sum+ num;  
}
```



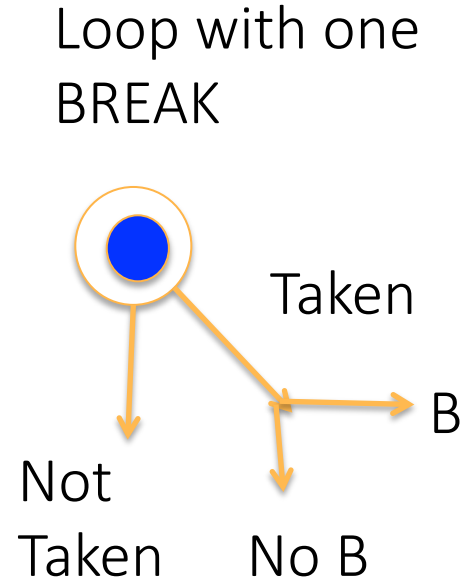
Three execution paths: (i) FOR skipped, (ii) FOR exited with BREAK, (iii) FOR exited without BREAK

Three paths

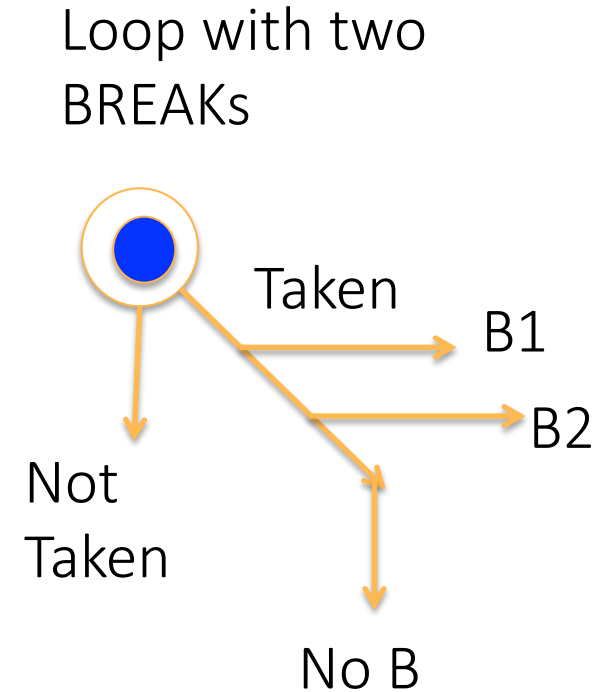
# Flow diagrams for loops



Two paths

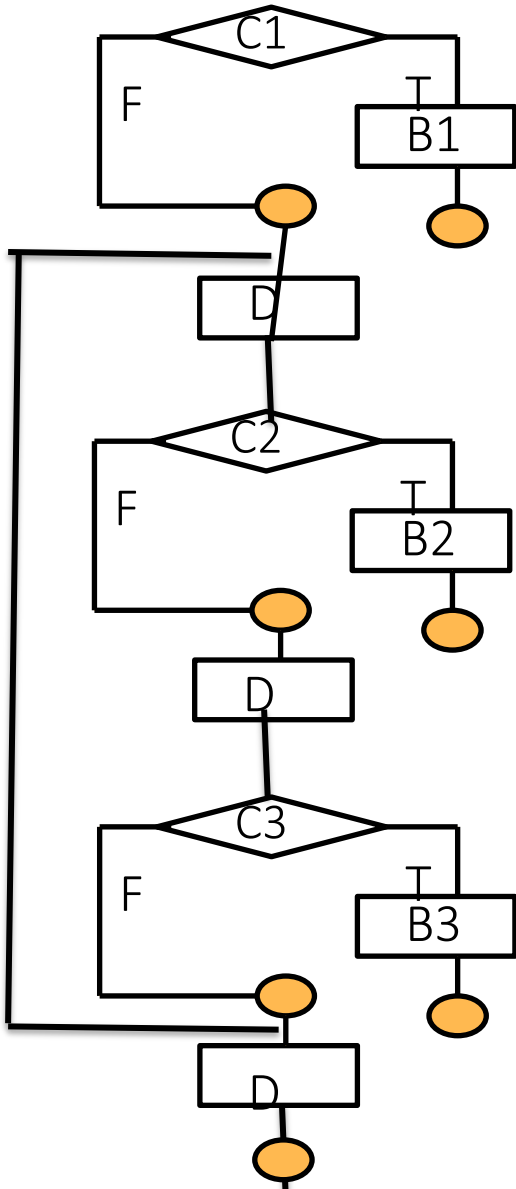


Three paths

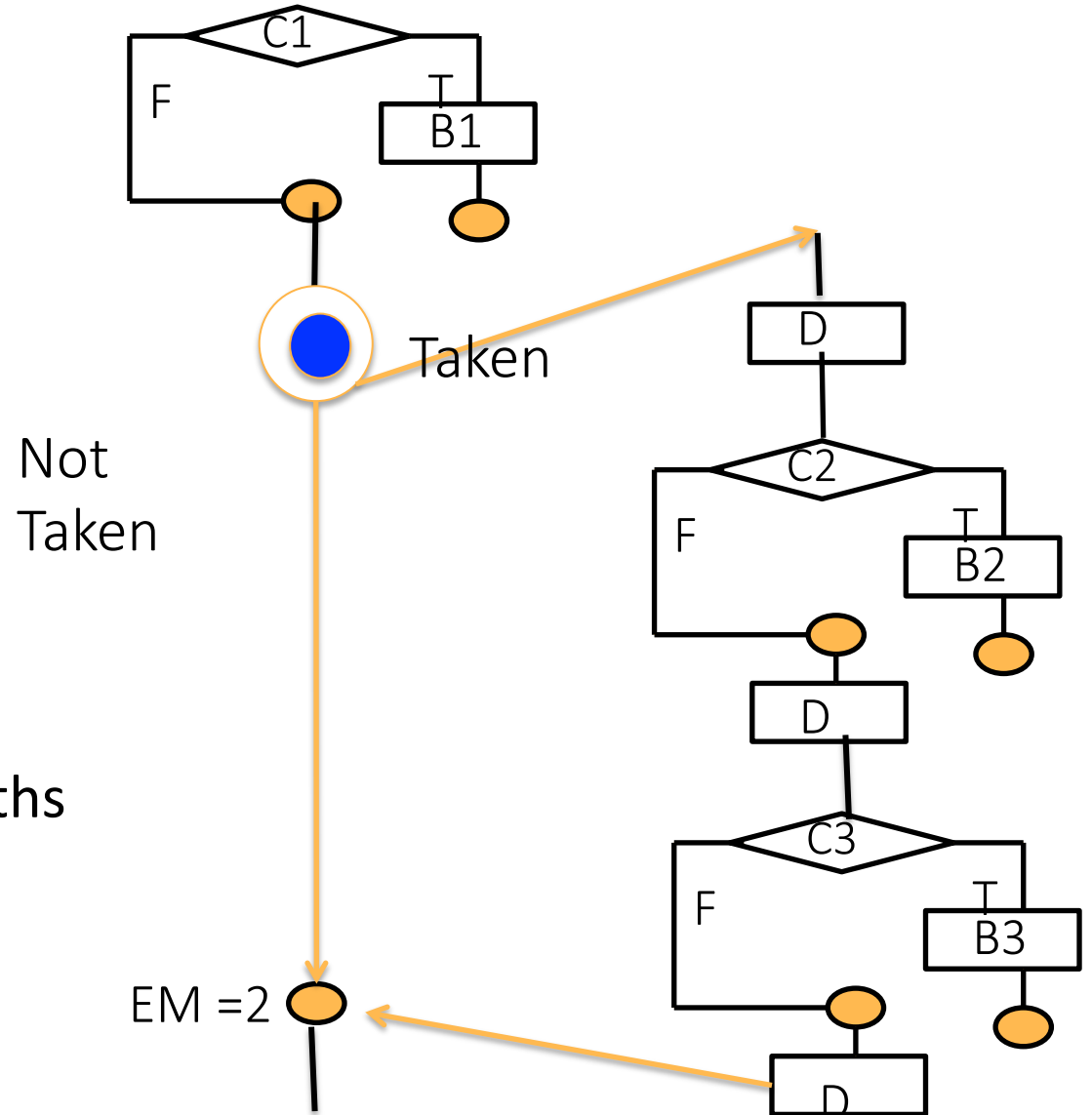


Four paths

# A Loop Example



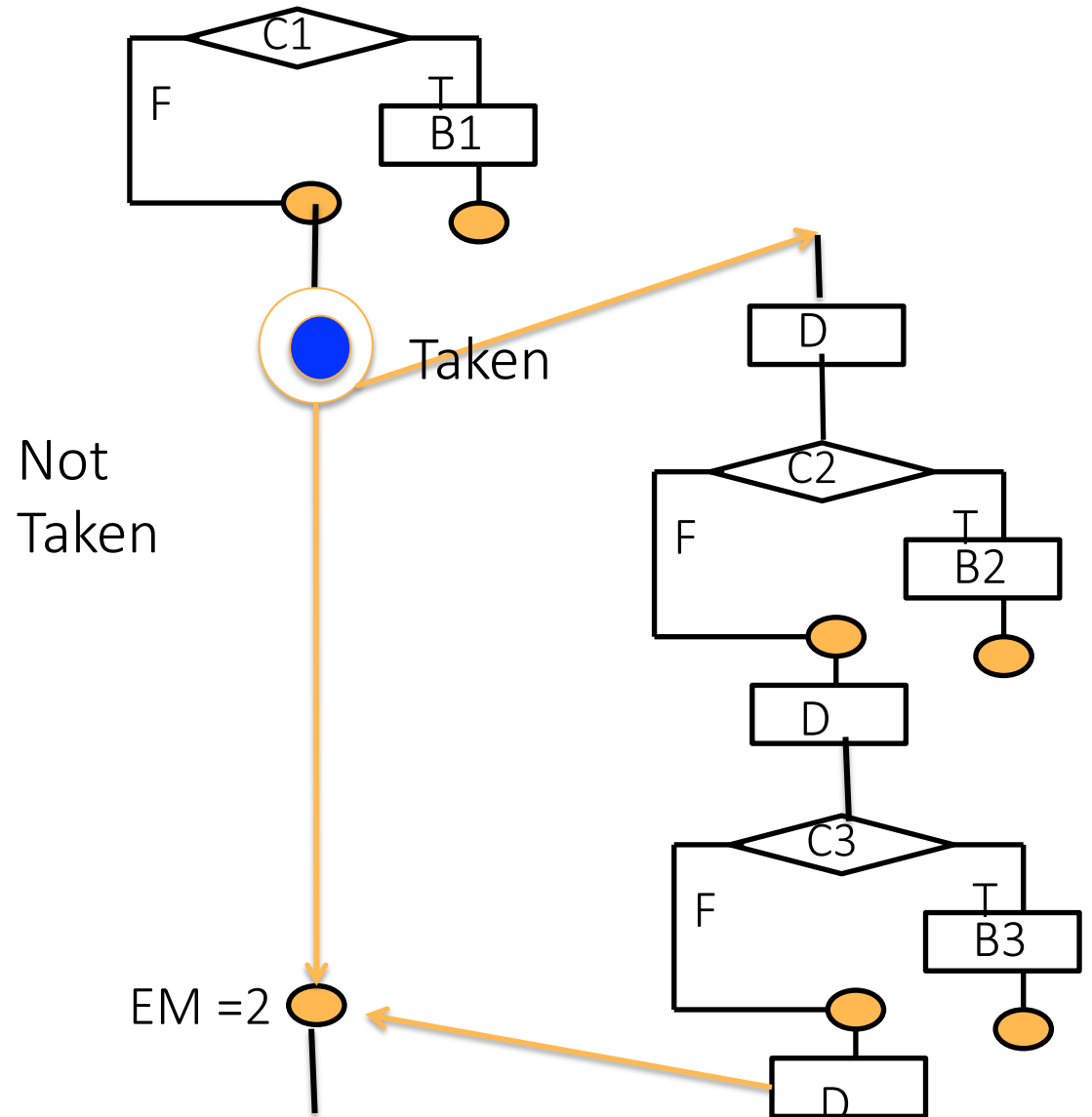
Five Execution Paths



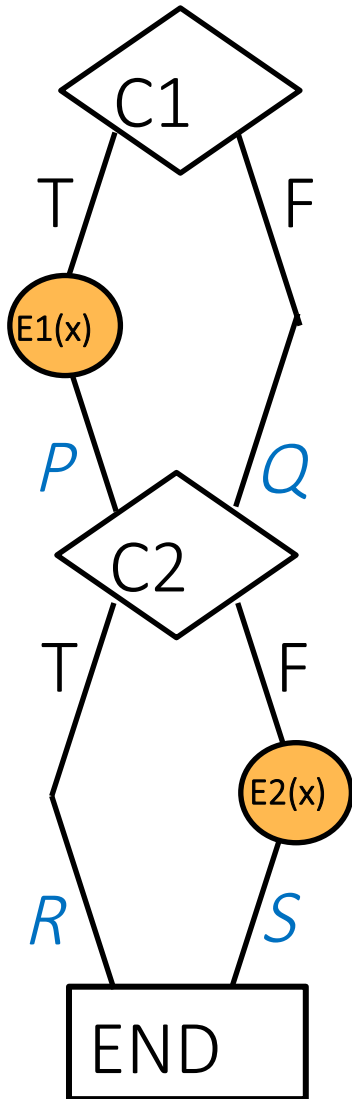


# A Loop Example

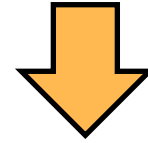
```
if (C2) {  
    dosomething;  
    return();  
}  
for ( ) {  
    if (C2) {  
        dosomething;  
        return();  
    }  
    if ( C3) {  
        dosomething;  
        return();  
    }  
}  
dosomething;
```



# Failed Verification of Infeasible Path



When C1 is TRUE C2 *cannot* be TRUE



*PR* is an infeasible path

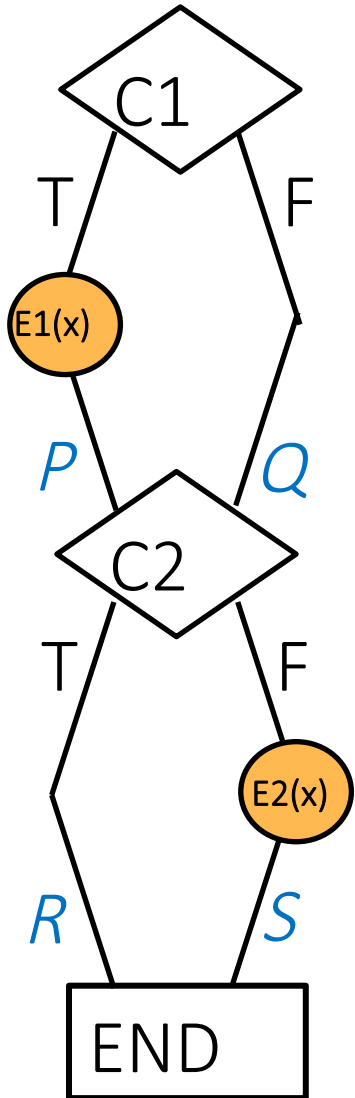
For MP, a FP if the path *PR* is infeasible.

For AMP, a FP if the path *PS* is infeasible.

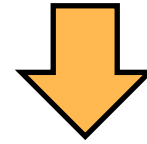
Definitions:

1. Matching Pair (MP) Property:  $E1(x)$  is followed by  $E2(x)$  on all feasible execution paths.
2. Anti-Matching Pair (AMP) Property:  $E1(x)$  is *not* followed by  $E2(x)$  on all feasible execution paths

# Verification Misses a Path



Both C1 and C2 *can* be TRUE



*PR* is a feasible path

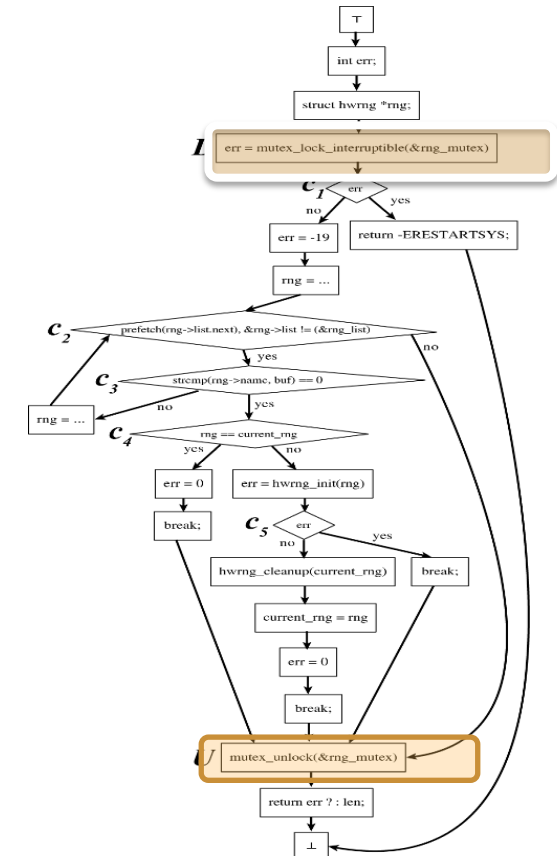
For MP, a FN if the feasible path *PR* is missed.

For AMP, a FN if the feasible path *PS* is missed.

# An example from Linux kernel

```
1 static ssize_t hwrng_attr_current_store(...) {  
2     int err;  
3     struct hwrng *rng;  
4     err = mutex_lock_interruptible(&rng_mutex);  
5     if (err)  
6         return -ERESTARTSYS;  
7     err = -ENODEV;  
8     list_for_each_entry(rng, &rng_list, list) {  
9         if (strcmp(rng->name, buf) == 0) {  
10             if (rng == current_rng) {  
11                 err = 0;  
12                 break;  
13             }  
14             err = hwrng_init(rng);  
15             if (err)  
16                 break;  
17             hwrng_cleanup(current_rng);  
18             current_rng = rng;  
19             err = 0;  
20             break;  
21         }  
22     }  
23     mutex_unlock(&rng_mutex);  
24     return err ? : len;  
25 }
```

Linux kernel (v 2.6.31)

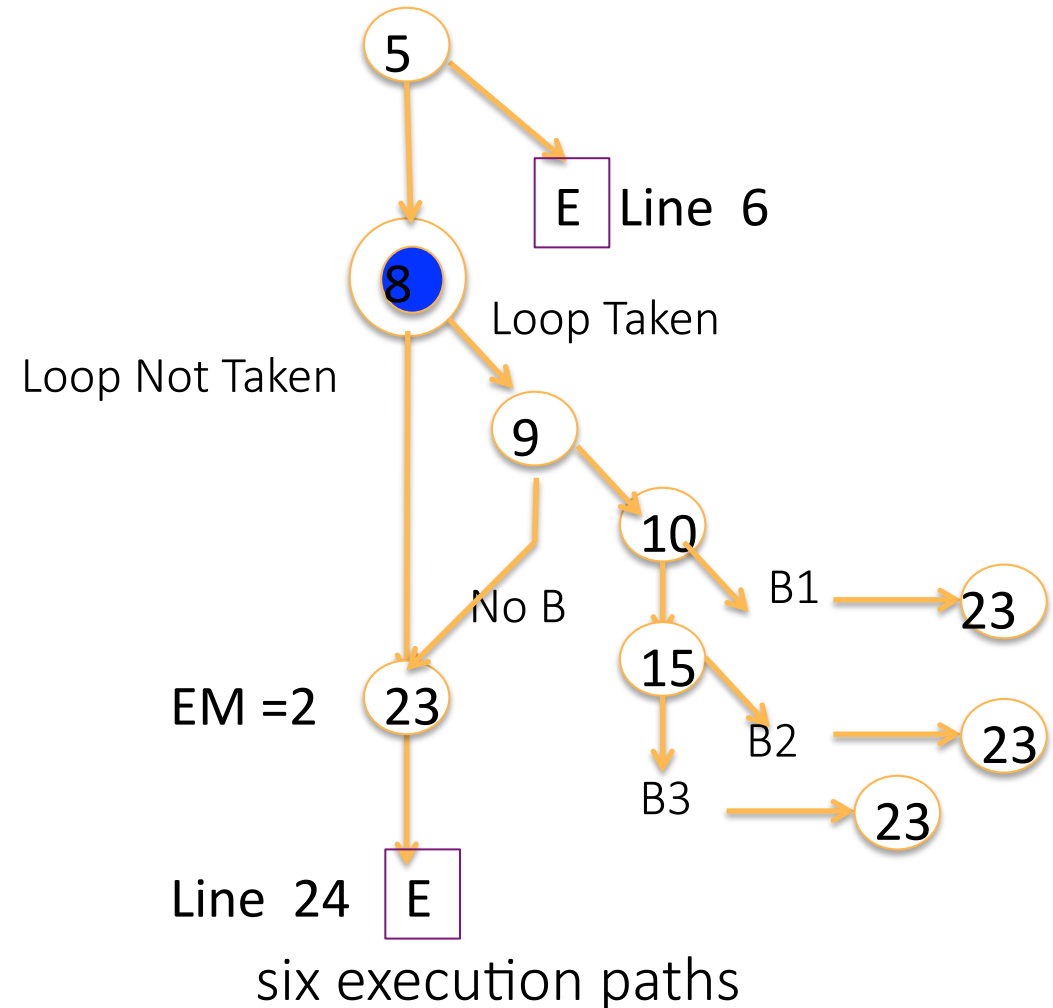


How many execution paths?

# An example from Linux kernel

```
1 static ssize_t hwrng_attr_current_store(...) {  
2     int err;  
3     struct hwrng *rng;  
4     err = mutex_lock_interruptible(&rng_mutex);  
5     if (err)  
6         return -ERESTARTSYS;  
7     err = -ENODEV;  
8     list_for_each_entry(rng, &rng_list, list) {  
9         if (strcmp(rng->name, buf) == 0) {  
10             if (rng == current_rng) {  
11                 err = 0;  
12                 break;  
13             }  
14             err = hwrng_init(rng);  
15             if (err)  
16                 break;  
17             hwrng_cleanup(current_rng);  
18             current_rng = rng;  
19             err = 0;  
20             break;  
21         }  
22     }  
23     mutex_unlock(&rng_mutex);  
24     return err ? : len;  
25 }
```

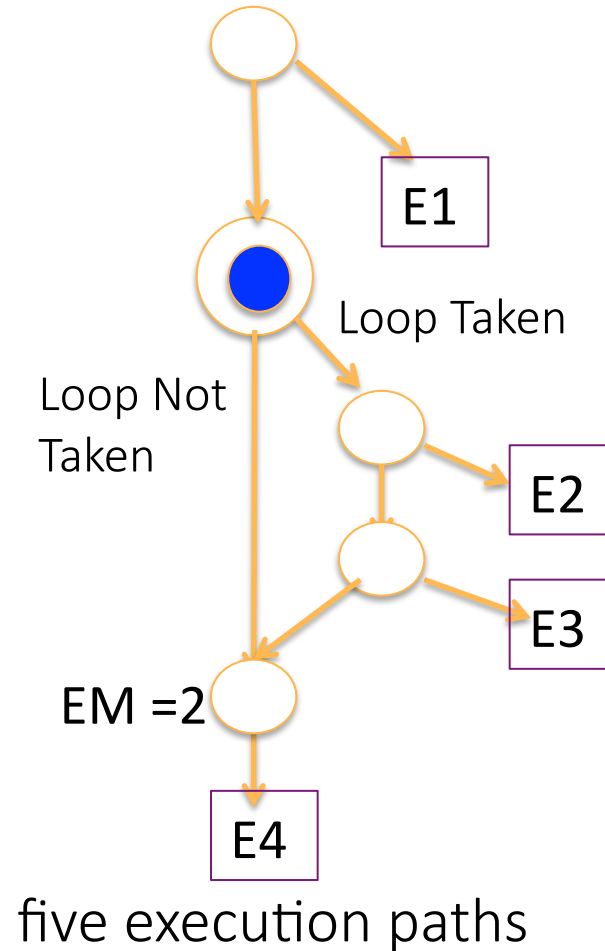
Linux kernel (v 2.6.31)



# Execution path analysis of a call

dskeng(drprr, dsprr)

```
{  
  if () {  
    return(DONQ);  
  }  
  for () {  
    if () {  
      return(st);  
    }  
    if () {  
      return(DONQ);  
    }  
  }  
  return(DONQ);  
}
```



# Analysis of dskenq

```
dskenq(drprr, dsprr)
{
    if () {
        return(DONQ);          EXIT1, EM=1
    }
    for () {
        if () {
            return(st);        EXIT2, EM=1
        }
        if () {
            return(DONQ);      EXIT3, EM=1
        }
    }
    return(DONQ); EXIT4, EM=2
}
```

5 execution paths

# Analysis of dskqopt

```
dskqopt(p, q, drpctr)

    if ( ) {
        return(SYSERR);
    }
    if (drpctr->drop == DSEEK) {
        return(OK);
    }
    if (p->drop == DSEEK) {
        return(OK);
    }
    if (p->drop==DWRITE && drpctr->drop==DWRITE) {
        return(OK);
    }
    if (drpctr->drop==DREAD && p->drop==DWRITE) {
        for ( );
        return(OK);
    }
    if (drpctr->drop==DWRITE && p->drop==DREAD) {
        for ( ) {
            for ( );
        }
        return(OK);
    }
    return(SYSERR);
}
```

EXIT1, EM=1

EXIT2, EM=1

EXIT3, EM=1

EXIT4, EM=1

EXIT5, EM=2

EXIT6, EM=3

EXIT7, EM=1

10 execution paths

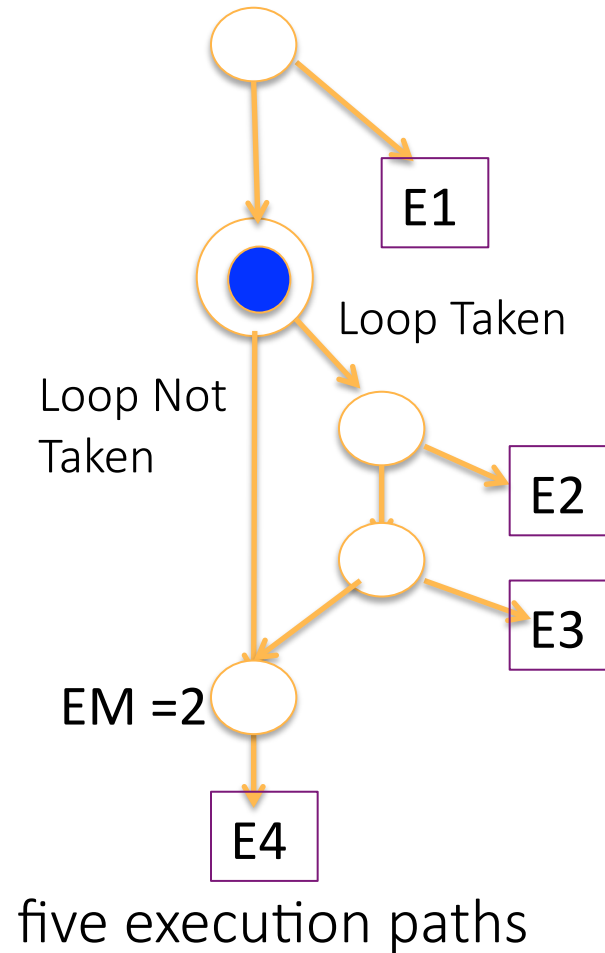
Total # paths in dskqopt(), which is its total exit multiplicity (EM), is the sum of all exit multiplicities = 10.



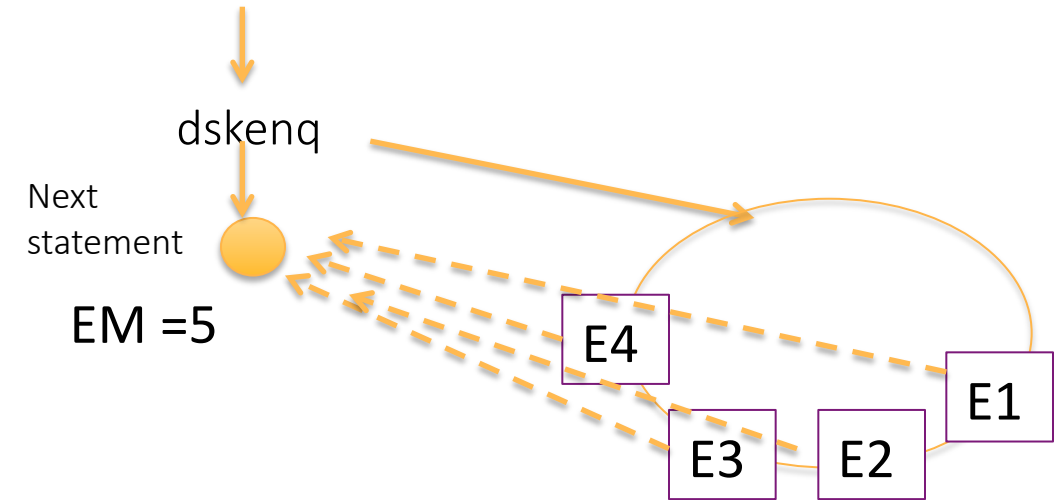
# Execution path analysis of a call

dskeng(drprr, dsprr)

```
{  
  if () {  
    return(DONQ);  
  }  
  for () {  
    if () {  
      return(st);  
    }  
    if () {  
      return(DONQ);  
    }  
  }  
  return(DONQ);  
}
```



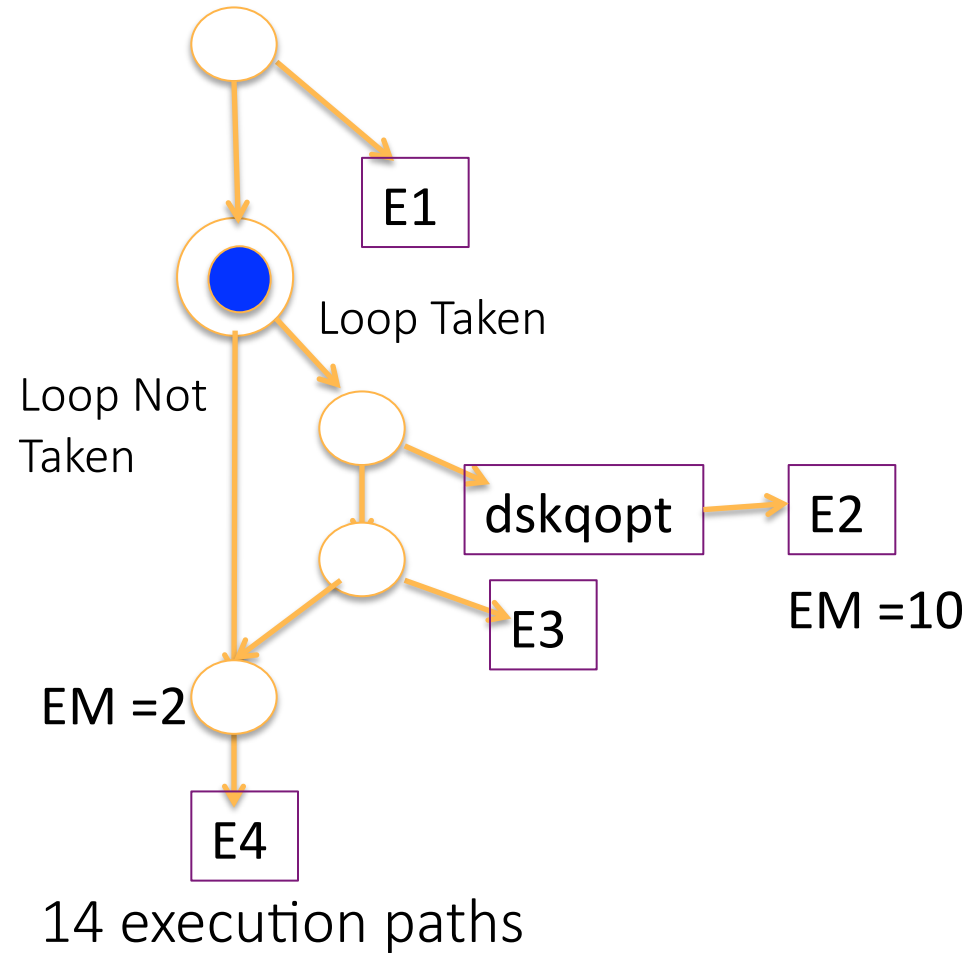
Modeling a call to dskeng



# Execution path analysis of with nested calls

```
dskmq(drpqr, dsptr)
```

```
{  
  if () {  
    return(DONQ);  
  }  
  for () {  
    if () {  
      return(st);  
    }  
    if () {  
      return(DONQ);  
    }  
  }  
  return(DONQ);  
}
```



# Three accuracy and scalability challenges

- Points-to analysis – exponential due to the power set of objects.
- Path-sensitive analysis – exponential due to the multiplicative growth of the number of paths
- Path feasibility analysis – exponential due to the satisfiability problem.
- We will discuss two approaches to address these problems:
  - Binary Decision Diagram (PCG)
  - Projected Control Graph (PCG)

# On Binary Decision Diagram (BDD)

Binary decision diagrams (BDDs) are wonderful, and the more I play with them the more I love them. For fifteen months I've been like a child with a new toy, being able now to solve problems that I never imagined would be tractable.

~Donald Knuth, The Art of Computer Programming, Volume 4, Preface

BDD is a compressed representation of a Boolean Function.

# Boolean function representations

- $X_1, X_2, X_3$  – three Boolean variables
- $f(X_1, X_2, X_3)$  – a Boolean function
- $f(0,0,0) = 0, f(0,0,1) = 0, f(0,1,0) = 0, f(0,1,1) = 1, f(1,0,0) = 0, f(1,0,1) = 1, f(1,1,0) = 1, f(1,1,1) = 1$
- Boolean vector representation of  $f$ : 00010111
- A Truth Table representation of  $f$ :

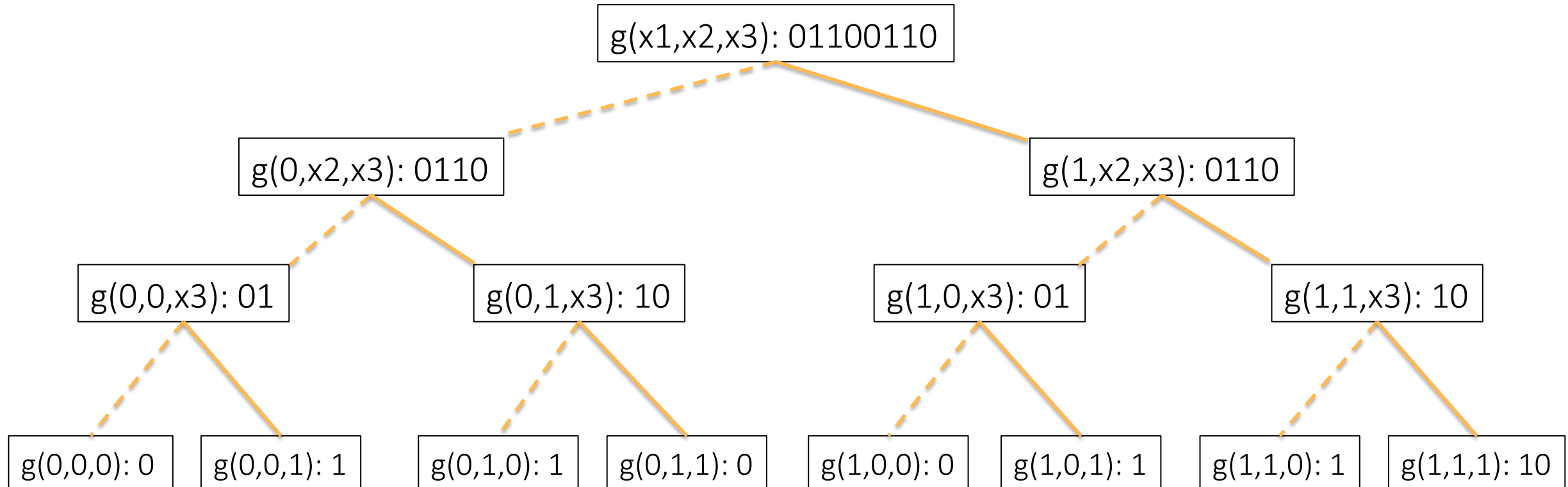
x1	0	0	0	0	1	1	1	1
x2	0	0	1	1	0	0	1	1
x3	0	1	0	1	0	1	0	1
f	0	0	0	1	0	1	1	1

# How can the representation be made compact?

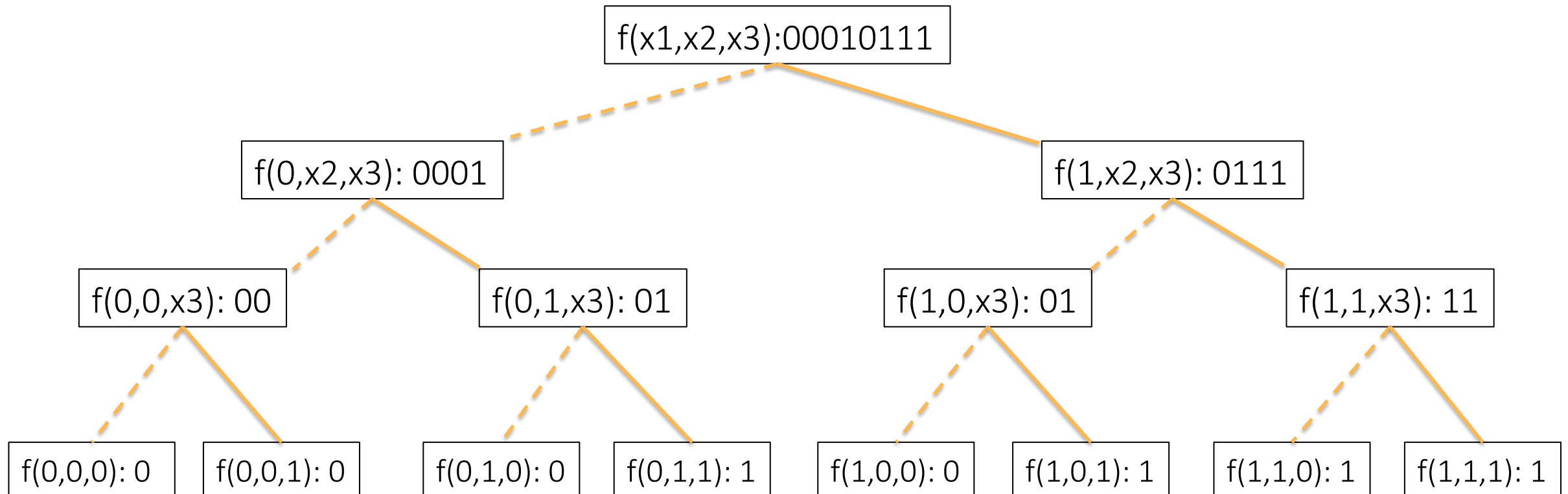
- Consider two Boolean functions:
  - $f(x_1, x_2, x_3)$ : 00010111
  - $g(x_1, x_2, x_3)$ : 01100110

Which function can be compressed? Why?

# Sub-functions of a function

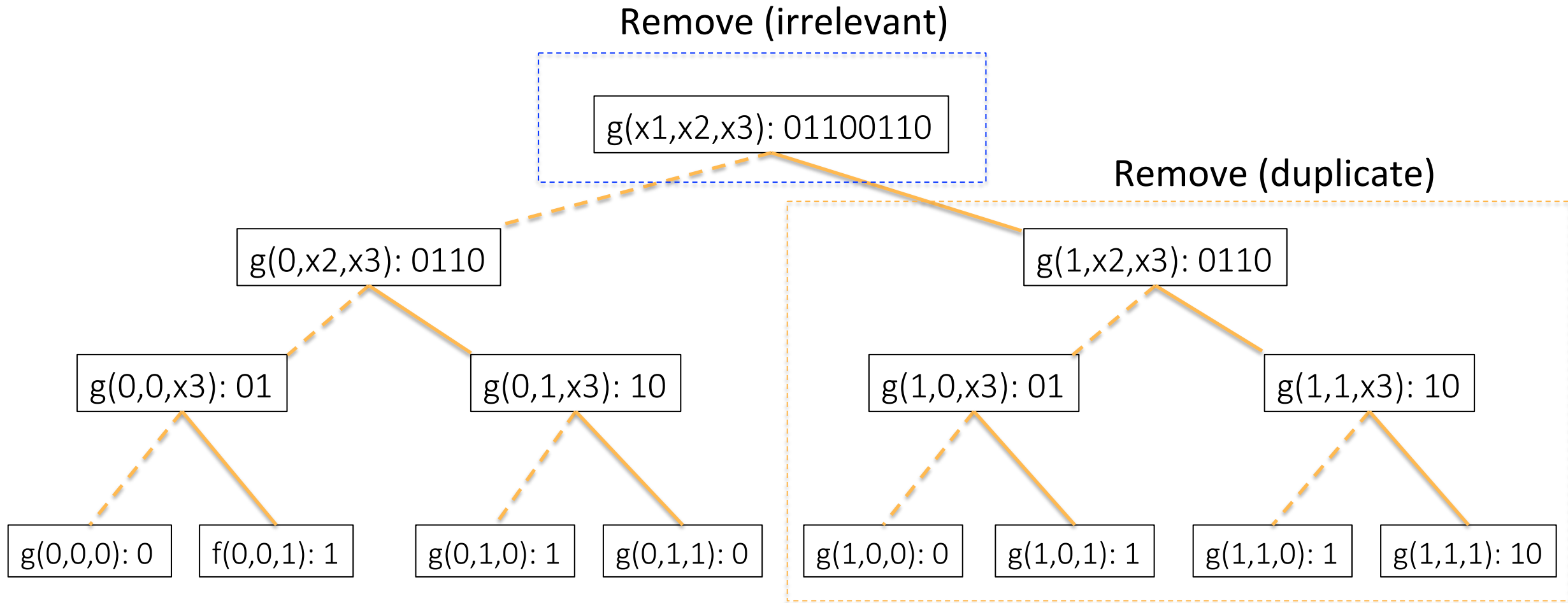


# Sub-functions of a function

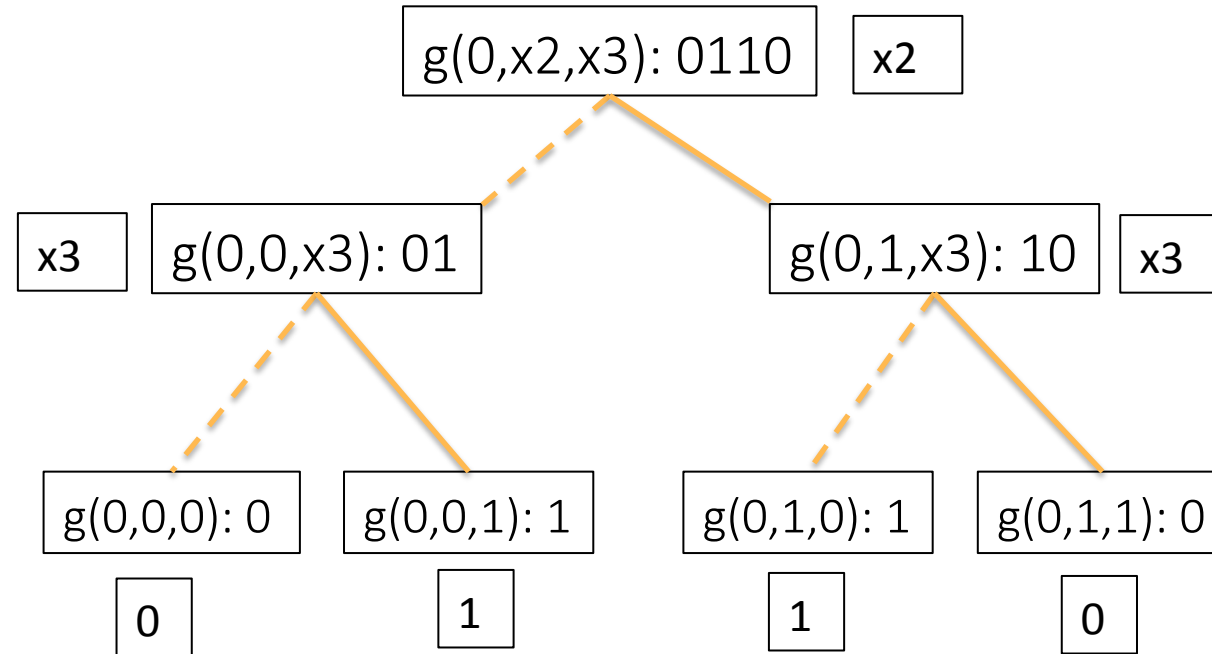




# Retain relevant sub-functions & remove duplicates



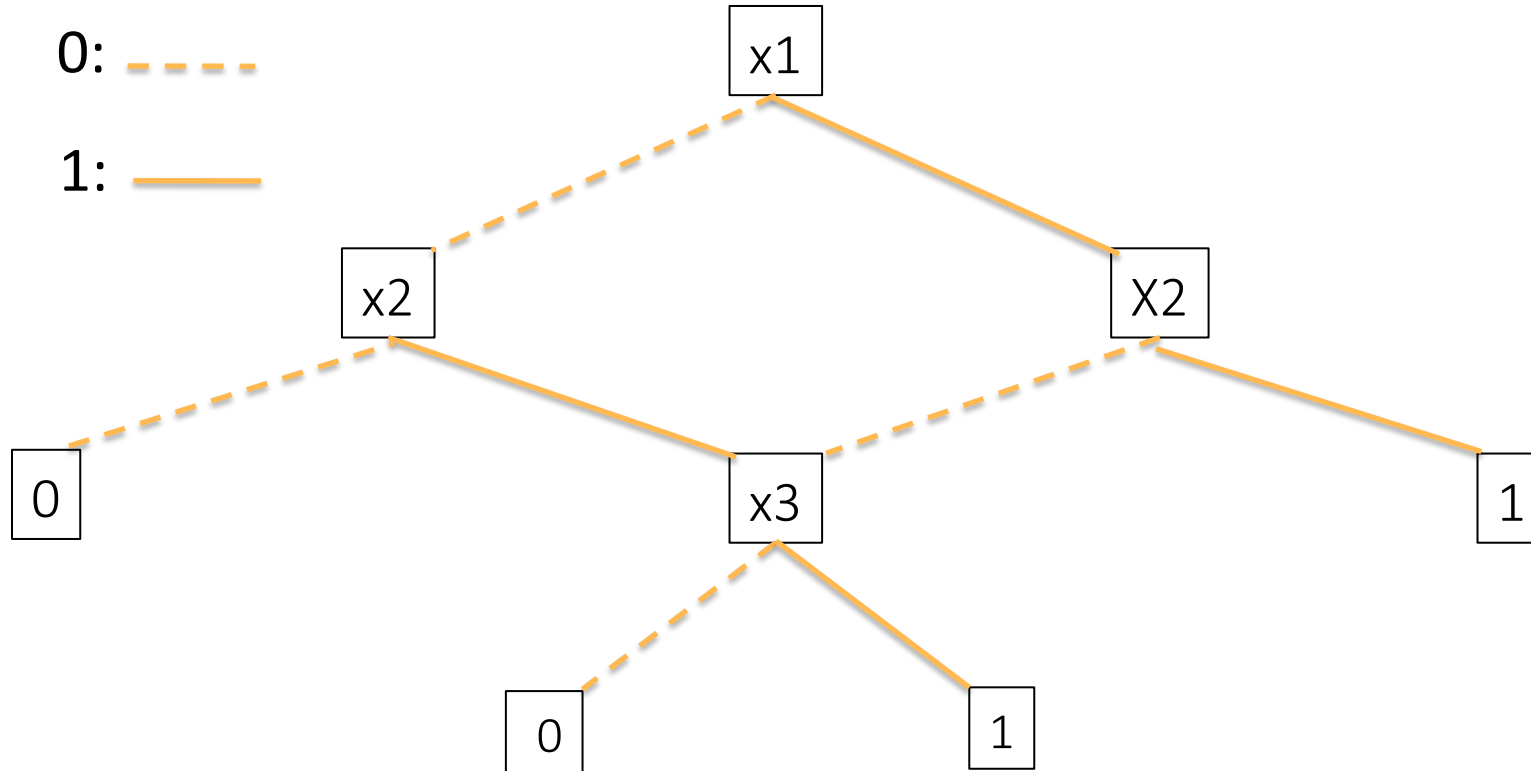
# Resulting BDD



The boxes should be relabeled in BDD as shown

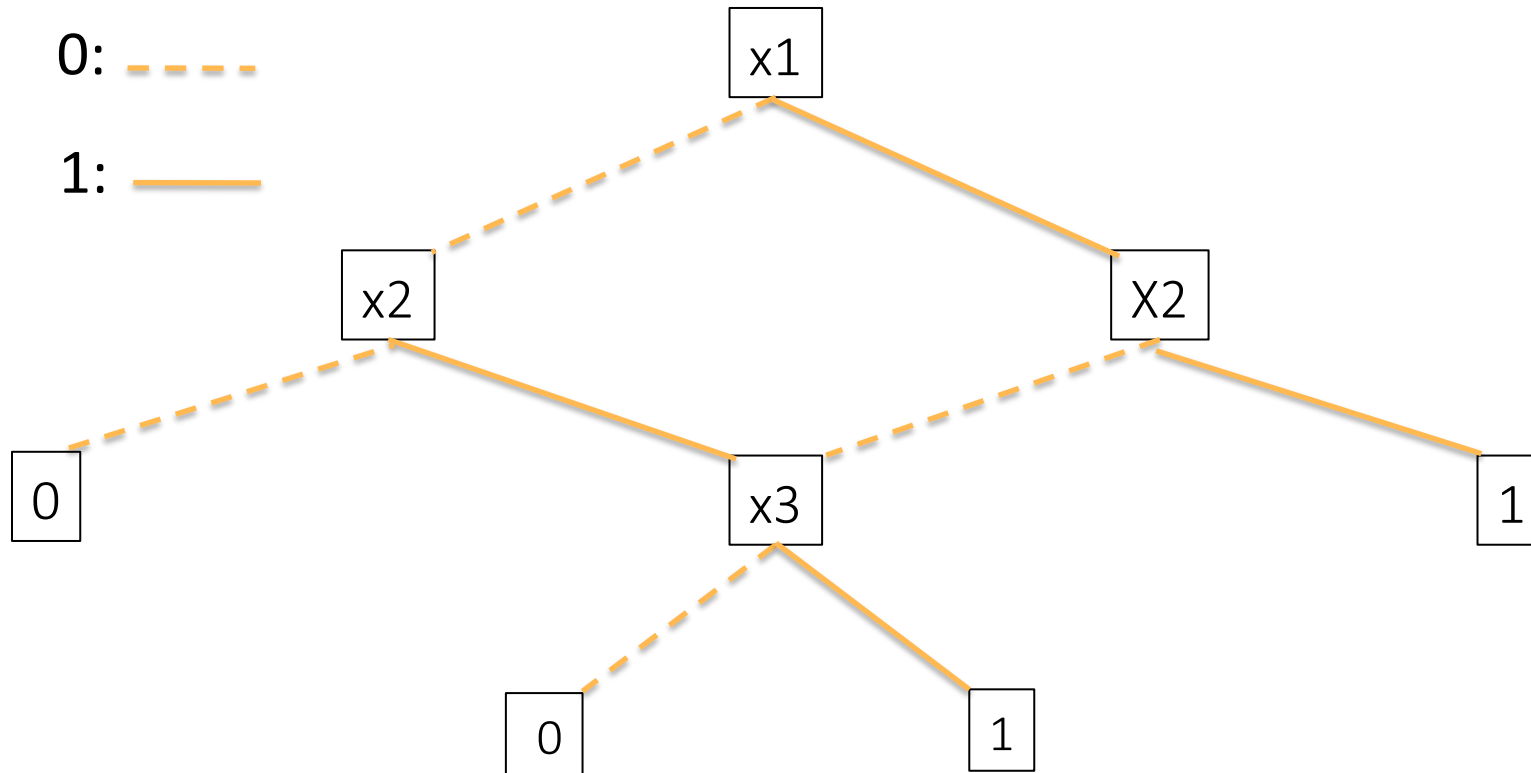
The input values of  $x_1$  are irrelevant for computing the Boolean function.

# What is the function represented by the BDD?



x1	x2	x3	f()
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

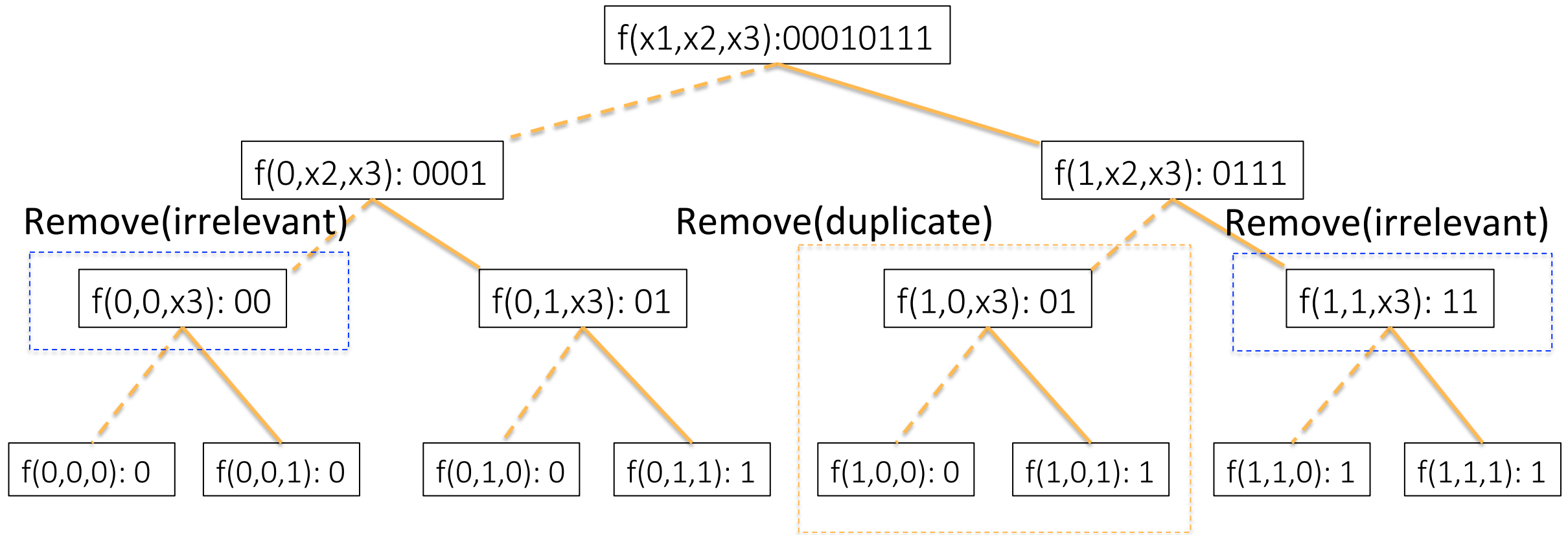
# What is the Boolean function represented by the BDD?



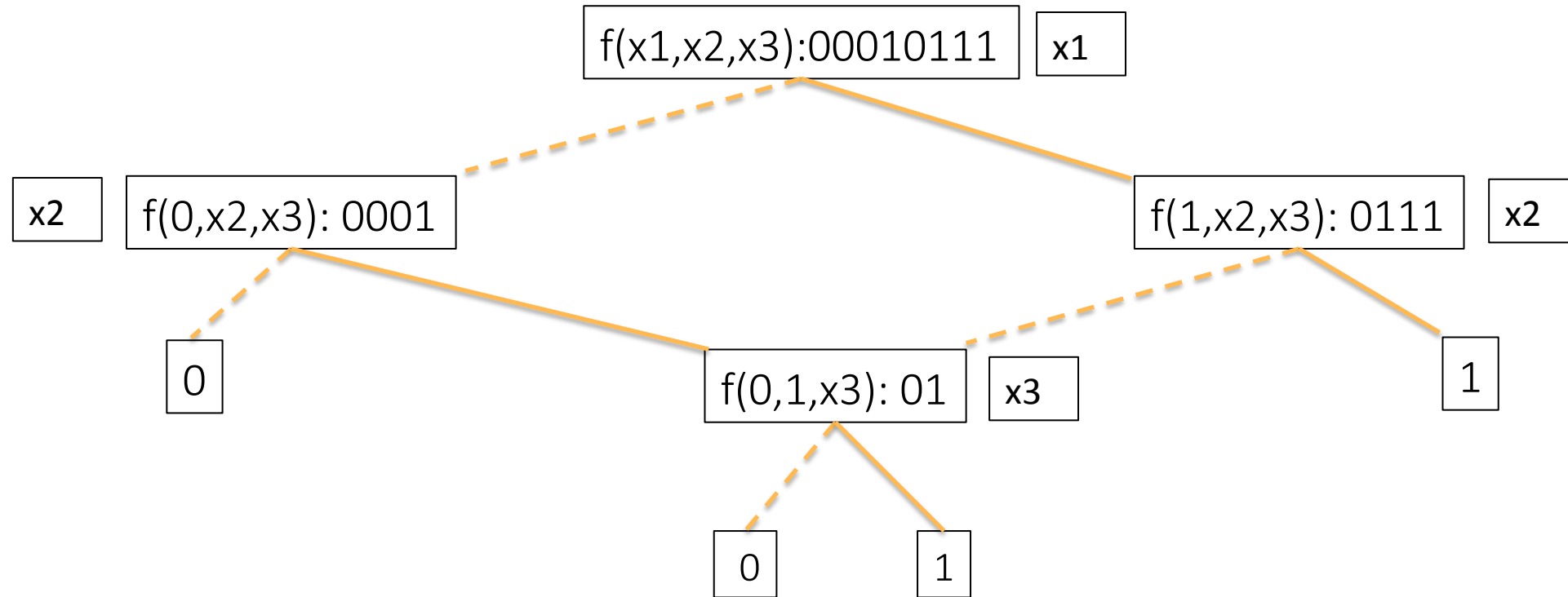
x1	x2	x3	f()
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Binary vector representation of the Boolean function: 00010111

# Retain relevant sub-functions and remove duplicate



# Resulting BDD



The boxes should be relabeled in BDD as shown

# Observations from BDD experiment with two Boolean functions

- Boolean function: 00010111
  - Retained sub-functions in the BDD: 00010111, 0001, 0111, 01
  - Eliminated sub-functions: 00, 11
- Boolean function: 01100110
  - Retained sub-functions in the BDD: 0110, 01, 10
  - Eliminated sub-functions: 01101110
- Definitions to describe the observations succinctly:
  - A sub-function is *non-primitive* if the binary vector is of the form  $B^2$  (BB), otherwise it is *primitive*. The primitive sub-functions are also called the *beads*.
  - Only one copy of each primitive will be retained.

# A succinct definition of BDD

- BDD retains only the primitive sub-functions of a Boolean function.
- BDD nodes correspond to an ordered set (sequence) of Boolean variables  $X_1$ ,  $X_2$ , and  $X_n$ .
- Each node has two edges.
- *Convention*: left edge from node for  $X_i$  is dotted and it corresponds to  $X_i == 0$ .
- BDD is acyclic if there is an edge from  $X_i$  node to  $X_j$  node then  $i < j$ .
- The leaves of the BDD are the 0 or 1 values of the Boolean function.
- The paths in the BDD correspond to  $2^n$  binary inputs.
- For the visualization, we may show multiple leaves for 0 or 1 but for the actual storage we need only two leaves.



# Ordering of Boolean variables can affect the size of the BDD

a	b	c	f()
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

c	b	a	f()
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

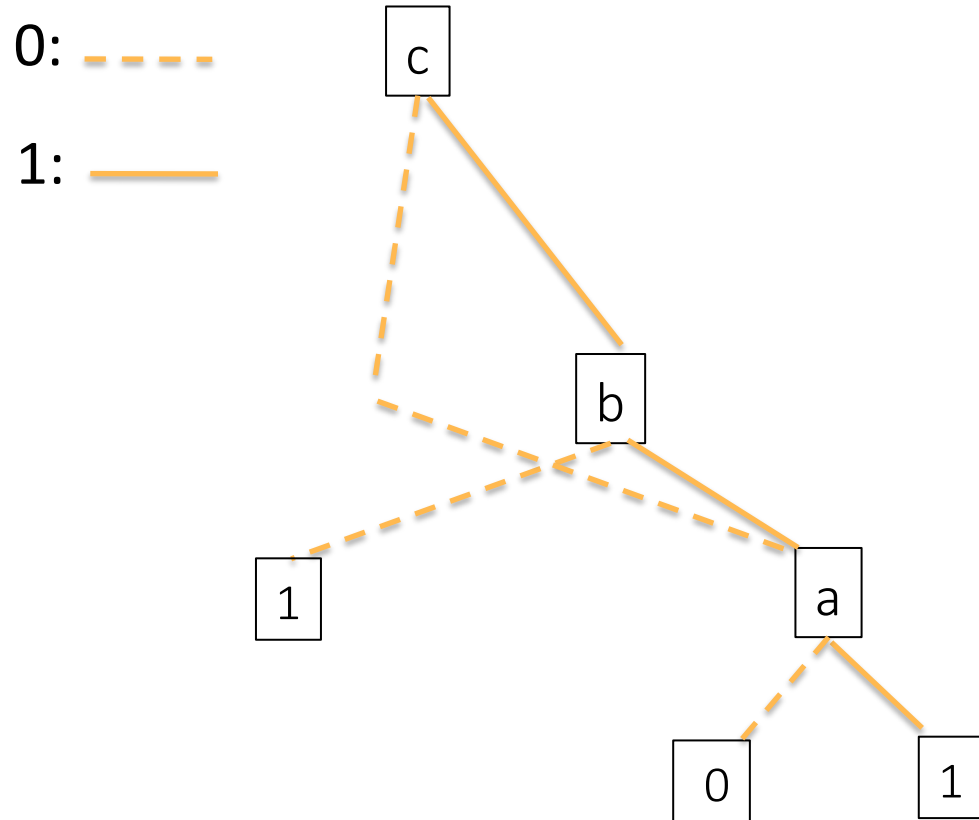
Boolean representation for the ordering a, b, c:  
01001111

Boolean representation for the ordering c, b, a:  
01011101

Primitives: 01001111, 0100, 01

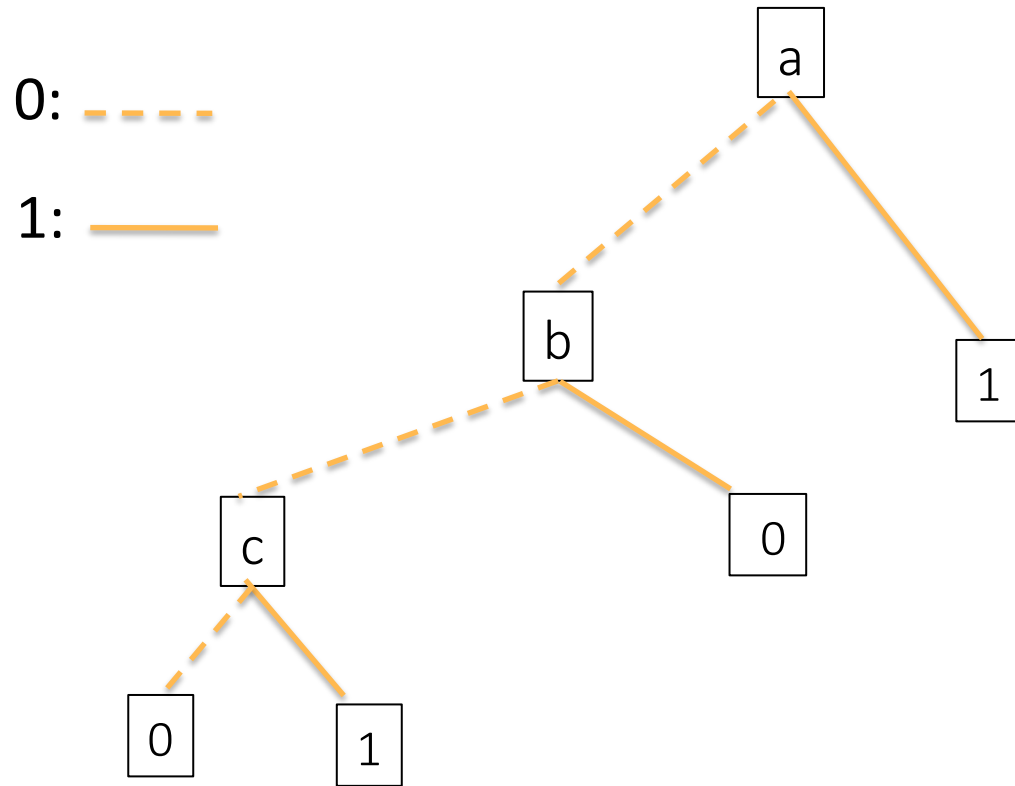
Primitives: 01011101, 1101, 01

# What is the function represented by the BDD?



a	b	c	f()
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

# What is the function represented by the BDD?



a	b	c	f()
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

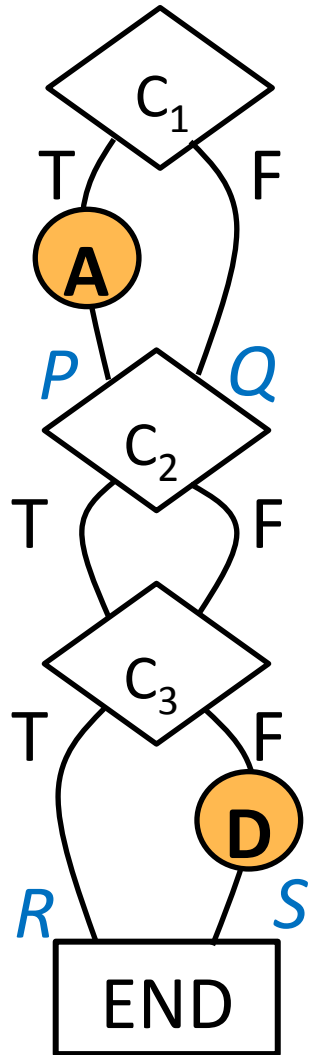
# Equivalence between Boolean functions and subsets of $\mathcal{P}(S)$

- $S = \{O_1, O_2, O_3\}$ , Powerset:  $\mathcal{P}(S) = \{\Phi, \{O_1\}, \{O_2\}, \{O_3\}, \{O_1, O_2\}, \dots\}$
- Each subset corresponds to a Boolean input:
  - $(x_1=0, x_2=0, x_3=0)=\{\Phi\}$ ,  $(x_1=0, x_2=0, x_3=1)=\{O_3\}$ ,  $(x_1=0, x_2=1, x_3=0)=\{O_2\}$ ,  $(x_1=0, x_2=1, x_3=1)=\{O_2, O_3\}$ ,  $(x_1=1, x_2=0, x_3=0)=\{O_1\}$ ,  $(x_1=1, x_2=0, x_3=1)=\{O_1, O_3\}$ ,  $(x_1=1, x_2=1, x_3=0)=\{O_1, O_2\}$ ,  $(x_1=1, x_2=1, x_3=1)=\{O_1, O_2, O_3\}$
- Subset  $T$  of  $\mathcal{P}(S)$  maps to Boolean function. Given a set  $S$  with  $n$  elements, the cardinality of  $\mathcal{P}(S)$  is  $k = 2^n$  and the number of Boolean functions is  $2^k$ .
  - $T = \{\{O_1, O_2\}, \{O_3\}, \{O_2, O_3\}, \{O_1, O_2, O_3\}\}$  corresponds to the Boolean function 01010011
- Points-to sets corresponds to subset of  $T$  of  $\mathcal{P}(S)$ 
  - e.g.  $P(V_1) = \{O_1, O_2\}$ ,  $P(V_2) = \{O_3\}$ ,  $P(V_3) = \{O_2, O_3\}$ ,  $P(V_4) = \{O_1, O_2, O_3\}$ , corresponds to  $T = \{\{O_1, O_2\}, \{O_3\}, \{O_2, O_3\}, \{O_1, O_2, O_3\}\}$ .
  - Every variable  $V_i$  maps to one path in the BDD
- Implication: BDDs can serve as compact representations for Boolean functions as well as for points-to sets.

# Intuition: Efficient Path-Sensitive Analysis

- A large number of paths could be partitioned into a small number of groups.
- All Paths in a group are equivalent – have the same execution behavior w.r.t. the property to be verified.
- Efficient computation by examining only one path from each group.
- Challenge: How can the groups be formed without examining each path at least once?

# Irrelevant Branch Conditions

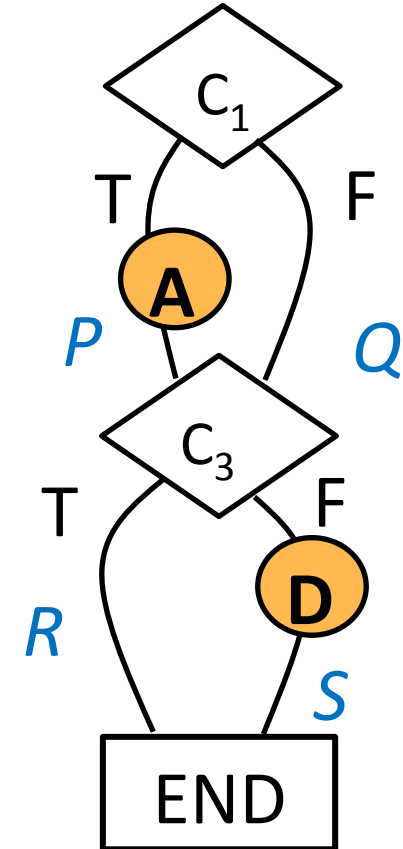


$C_2$  Irrelevant to  
path-sensitive analysis



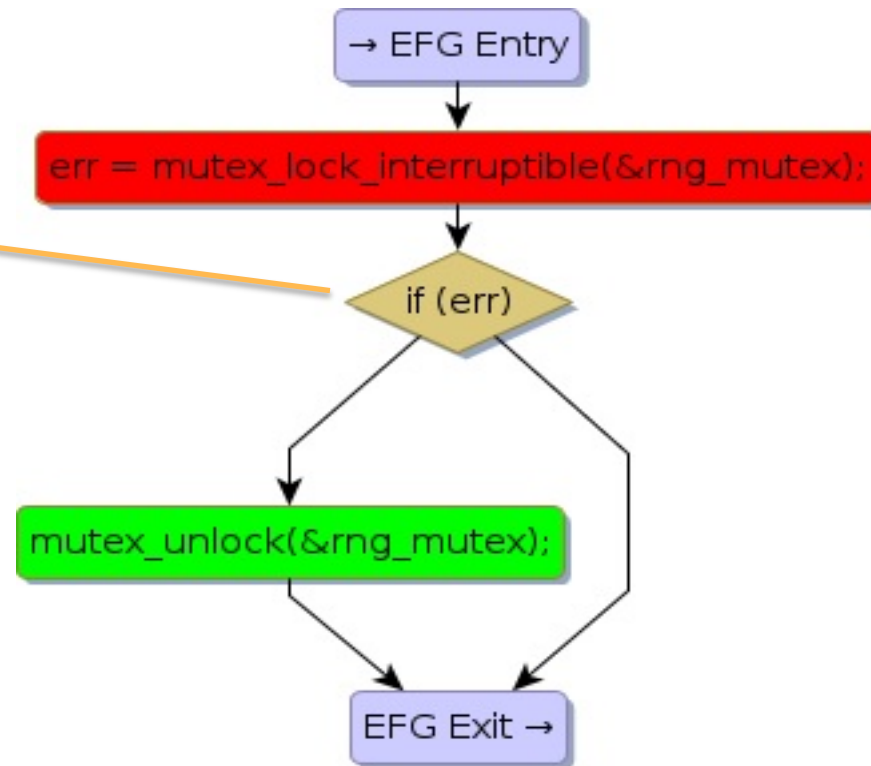
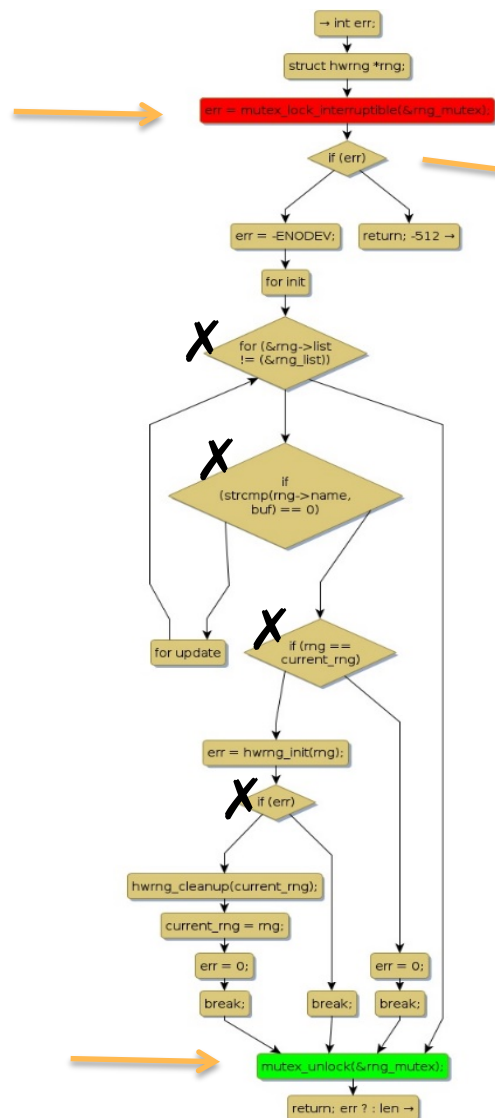
Remove the Irrelevant branch nodes to  
avoid unnecessary path explosion & also  
simplify the path feasibility check.

# paths reduced from 8 to 4



# conditions for feasibility check  
reduced from 3 to 2

# A Linux Example to Illustrate Efficient Analysis



# EFG for Efficient Path-Sensitive Analysis

- Event Flow Graph (EFG) is derived from the Control Flow Graph (CFG).
- Relevant events: CFG nodes relevant to the property (e.g. getbuf, freebuf, nodes of type p = drptr (pointer to the allocated memory) ).
- Behavior: An event trace along a CFG path
- EFG has one path per group of CFG paths with identical behaviors.
- EFG construction:  $O(K)$  where  $K = N + E$ ,  $N$ : # nodes in CFG,  $E$ : # edges in CFG.



# Linux Lock/Unlock Stats by L-SAP

- Versions 3.17-rc1, 3.18-rc1, 3.19-rc1
  - 37 MLOC & 66,609 instances
  - 62,663 Intra-Procedural ( MPG size =1), average MPG size 1.3, the maximum MPG size 40 for one MPG
  - 55,251 functions in the union of all MPGs
  - 4 nodes per EFG, the maximum EFG size 63 for one EFG

# CFG to EFG Reduction – Top Ten

Linux version 3-19-rc-1

Function Name	Nodes		Edges		Branch Nodes	
	CFG	EFG	CFG	EFG	CFG	EFG
<code>client_common_fill_super</code>	1,101	15	1,179	28	249	13
<code>kiblnd_create_conn</code>	731	18	925	34	197	15
<code>CopyBufferToControlPacket</code>	392	20	559	39	180	18
<code>kiblnd_cm_callback</code>	662	38	831	56	170	15
<code>kiblnd_passive_connect</code>	622	22	784	44	164	20
<code>dst_ca_ioctl</code>	349	2	518	1	163	0
<code>qib_make_ud_req</code>	621	10	821	15	156	5
<code>cfs_cpt_table_al</code>	522	7	672	13	153	6
<code>private_ioctl</code>	569	16	732	24	148	8
<code>vCommandTimer</code>	490	47	623	75	143	28