CE 874 - Secure Software Systems

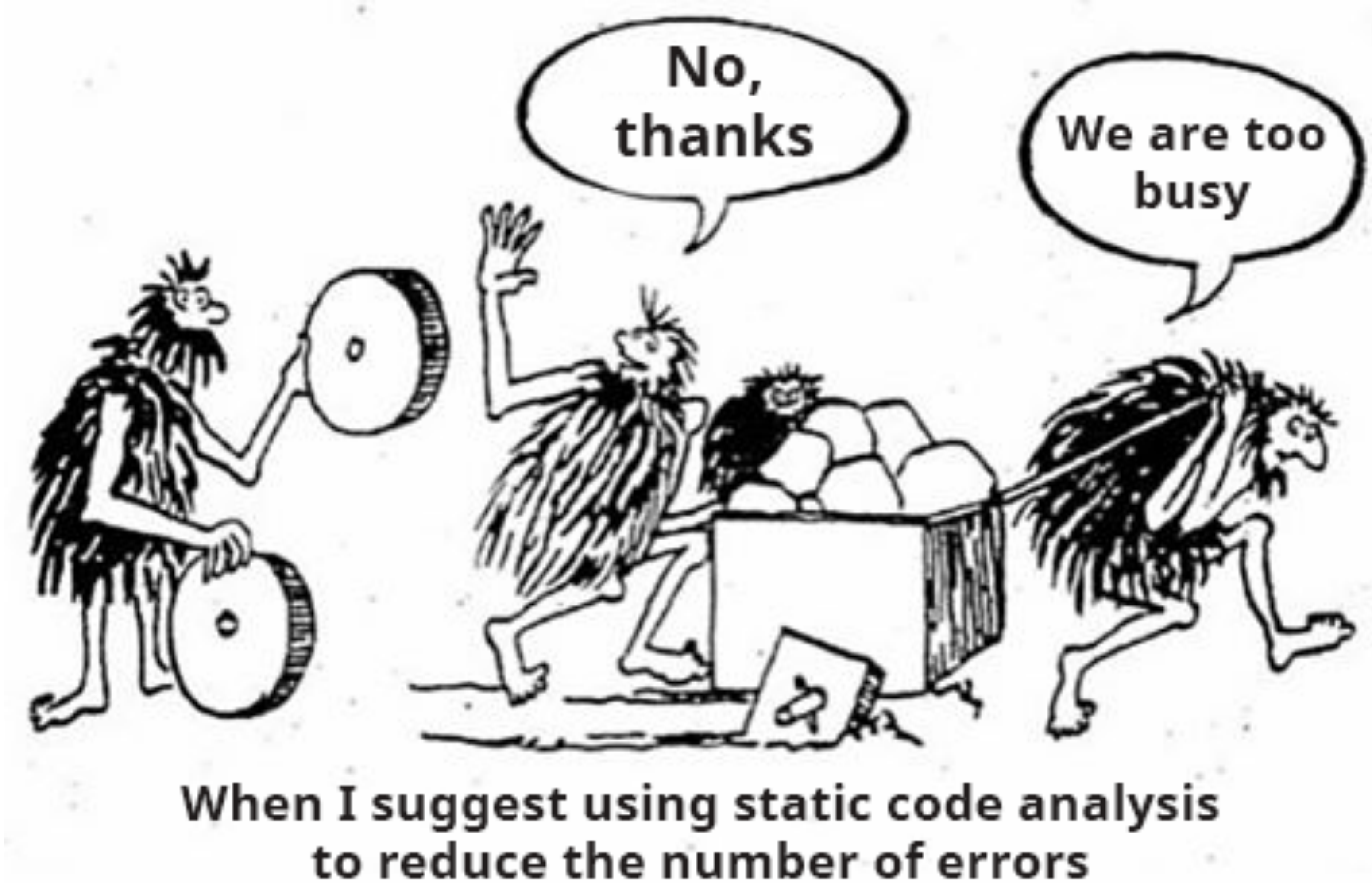
Symbolic Execution

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

Program Analysis

- How could we analyze a program (with source code) and look for problems?
- How accurate would our analysis be without executing the code?
- If we execute the code, what input values should we use to test/analyze the code?
- What if we don't have the source code?



https://www.viva64.com

Symbolic Execution

Symbolic Execution --- History

- 1976: A system to generate test data and symbolically execute programs (Lori Clarke)
- 1976: Symbolic execution and program testing (James King)
- 2005-present: practical symbolic execution
  - Using SMT solvers
  - Heuristics to control exponential explosion
  - Heap modeling and reasoning about pointers
  - Environment modeling
  - Dealing with solver limitations

[Chowdhury’15]

Motivation

- Writing and maintaining tests is tedious and error-prone
- Idea: Automated Test Generation
  - Generate regression test suite
  - Execute all reachable statements
  - Catch any assertion violations

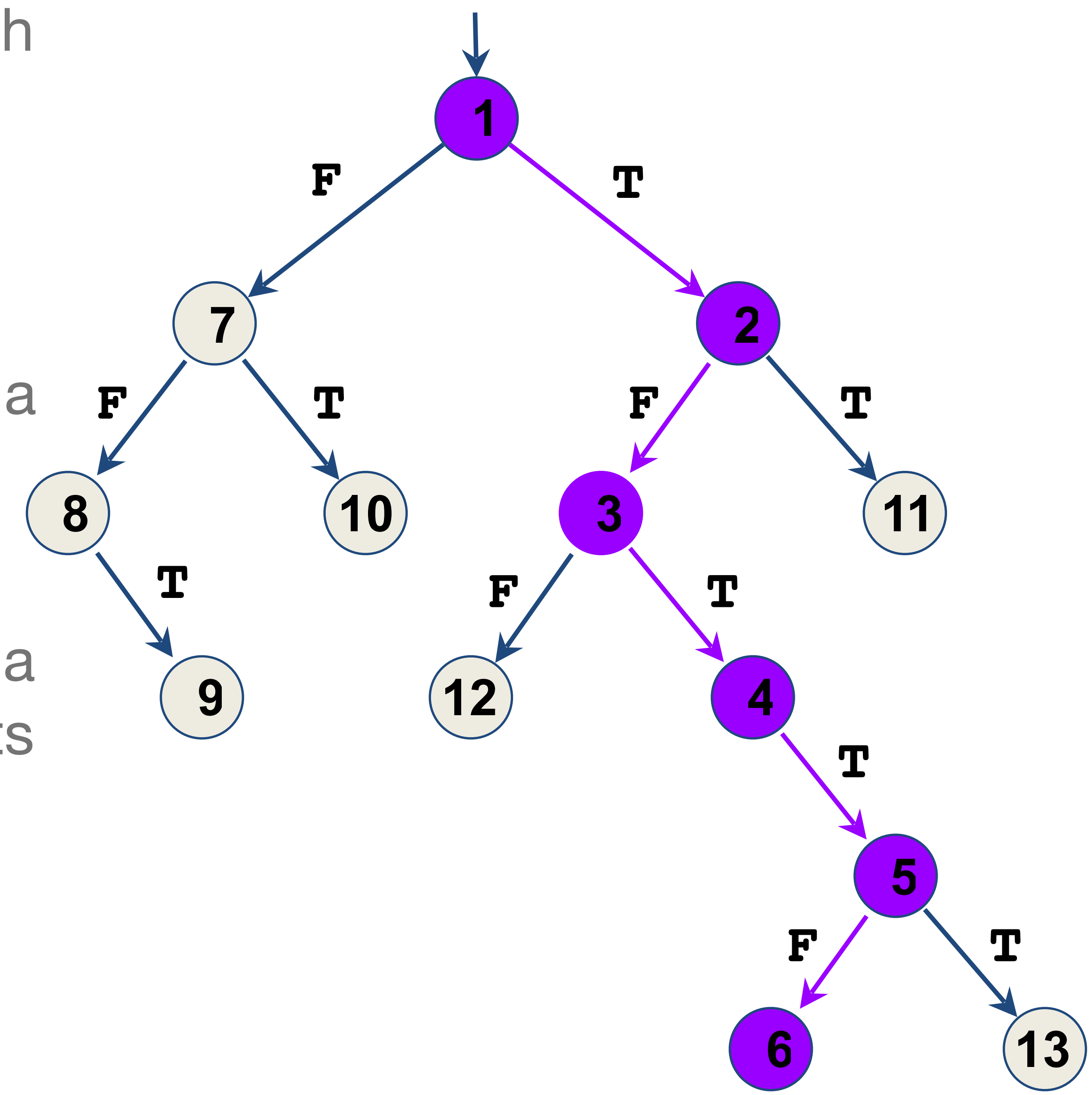
Approach

- Dynamic Symbolic Execution
  - Stores program state concretely and symbolically
  - Solves constraints to guide execution at branch points
  - Explores all execution paths of the unit tested

- Example of Hybrid Analysis
  - Collaboratively combines dynamic and static analysis

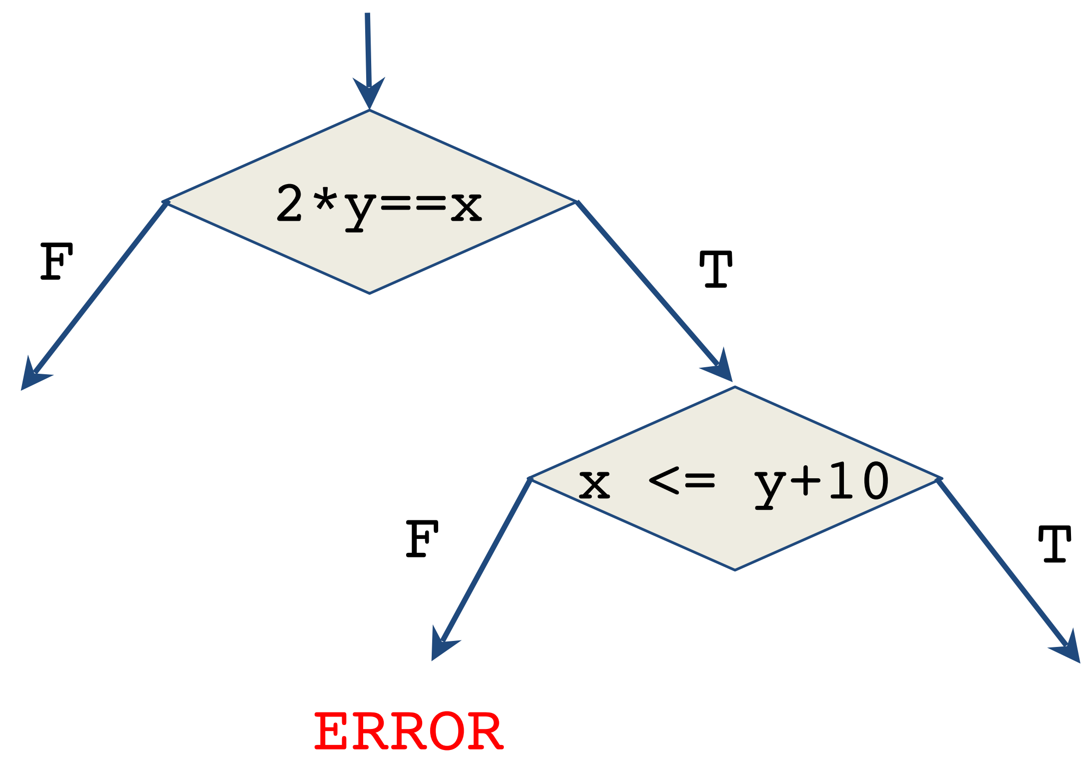
Execution Paths of a Program

- Program can be seen as binary tree with possibly infinite depth
  - Called Computation Tree
- Each node represents the execution of a conditional statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs



Example of Computation Tree

```
void test_me(int x, int y) {
   if (2*y == x) {
      if (x <= y+10)
         print("OK");
      else {
         print("something bad");
         ERROR;
      }
   } else
      print("OK");
}
```



Existing Approach I

- Random Testing:
  - Generate random inputs
  - Execute the program on those (concrete) inputs

```
void test_me(int x) {
    if (x == 94389) {
        ERROR;
    }
}
```

- Problem:
  - Probability of reaching error could be astronomically small

Probability of ERROR:

$1/2^{32} \approx 0.000000023\%$

Existing Approach II

- Symbolic Execution
  - Use symbolic values for inputs
  - Execute program symbolically on symbolic input values
  - Collect symbolic path constraints
  - Use theorem prover to check if a branch can be taken
- Problem:
  - Does not scale for large programs

```
void test_me(int x) {
    if (x*3 == 15) {
        if (x % 5 == 0)
            print(“OK”);
        else {
            print(“something
bad”);
            ERROR;
        }
    } else
        print(“OK”);
}
```

Existing Approach II

- Symbolic Execution
  - Use symbolic values for inputs
  - Execute program symbolically on symbolic input values
  - Collect symbolic path constraints
  - Use theorem prover to check if a branch can be taken
- Problem:
  - Does not scale for large programs

```
void test_me(int x) {
    // c = product of two
    // large primes
    if (pow(2,x) % c == 17) {
        print(“something bad”);
        ERROR;
    } else
        print(“OK”);
}
```

Symbolic execution will say both branches are reachable: False Positive

Combined Approach

- Dynamic Symbolic Execution (DSE)
  - Start with random input values
  - Keep track of both concrete values and symbolic constraints
  - Use concrete values to simplify symbolic constraints
  - Incomplete theorem-prover

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition
	x = 22 y = 7		$x = x_0$ $y = y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22		$x = x_0$	
y = 7		$y = y_0$	
z = 14		$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$2*y_0 \ != x_0$
y = 7	$y = y_0$	
z = 14	$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution: concrete state	Symbolic Execution: symbolic state	path condition
x = 22	$x = x_0$	$2*y_0 \mathrel{!=} x_0$
y = 7	$y = y_0$	
z = 14	$z = 2*y_0$	

Solve: $2*y_0 == x_0$

Solution: $x_0 = 2$, $y_0 = 1$

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 2		$x = x_0$	
y = 1		$y = y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 2		$x = x_0$	
y = 1		$y = y_0$	
z = 2		$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 2		$x = x_0$	$2*y_0 == x_0$
y = 1		$y = y_0$	
z = 2		$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
x = 2	$x = x_0$	$2*y_0 == x_0$
y = 1	$y = y_0$	$x_0 <= y_0+10$
z = 2	$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 2	$x = x_0$	$2*y_0 == x_0$
y = 1	$y = y_0$	
z = 2	$z = 2*y_0$	$x_0 <= y_0+10$

Solve: $(2*y_0 == x_0)$ and $(x_0 > y_0+10)$

Solution: $x_0 = 30$, $y_0 = 15$

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 30 y = 15	$x = x_0$ $y = y_0$		

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition
	x = 30		$x = x_0$	
	y = 15		$y = y_0$	
	z = 30		$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 30	$x = x_0$		$2*y_0 == x_0$
y = 15	$y = y_0$		
z = 30	$z = 2*y_0$		

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Program Error

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 30$	$x = x_0$	$2*y_0 == x_0$
$y = 15$	$y = y_0$	
$z = 30$	$z = 2*y_0$	$x_0 > y_0+10$

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22		$x = x_0$	
y = 7		$y = y_0$	

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22		x = x_0	
y = 7		y = y_0	
z = 601...129		z = secure_hash(y_0)	

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$secure\_hash(y_0) \mathrel{!=} x_0$
y = 7	$y = y_0$	
z = 601...129	$z = secure\_hash(y_0)$	

Solve: $secure\_hash(y_0) == x_0$

Don’t know how to solve! Stuck?

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
```

Not stuck! Use concrete state: replace y_0 by 7

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	x = x_0	secure_hash(y_0) != x_0
y = 7	y = y_0	
z = 601...129	z = secure_hash(y_0)	

Solve: secure_hash(y_0) == x_0

Don't know how to solve! Stuck?

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
x = 22	$x = x_0$	$\text{secure\_hash}(y_0) \mathrel{!=} x_0$
y = 7	$y = y_0$	
z = 601...129	$z = \text{secure\_hash}(y_0)$	

Solve: $601...129 == x_0$

Solution: $x_0 = 601...129$, $y_0 = 7$

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution		Symbolic Execution	
	concrete state	symbolic state		path condition
	x = 601...129 y = 7	$x = x_0$ $y = y_0$		

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 601...129	x = x_0		
y = 7	y = y_0		
z = 601...129	z = secure_hash(y_0)		

[Naik'18]

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution | Symbolic Execution

concrete state

x = 601...129

y = 7

z = 601...129

symbolic state

$x = x_0$

$y = y_0$

$z = \text{secure\_hash}(y_0)$

path condition

$\text{secure\_hash}(y_0) == x_0$

[Naik'18]

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Program Error

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 601...129 y = 7 z = 601...129	$x = x_0$ $y = y_0$ $z = \text{secure\_hash}(y_0)$		$\text{secure\_hash}(y_0) == x_0$ $x_0 > y_0+10$

[Naik'18]

Example Application

- DSE tests the below program starting with input x = 1. What is the input and constraint (C1 ∧ C2 ∧ C3) solved in each run of DSE? Use depth-first search and leave trailing constraints blank if unused.

Run	x	C1	C2	C3
1	1	5 != x0	7 != x0	9 == x0
2				
3				
4				

```
int test_me(int x) {
    int[] A = { 5, 7, 9 };
    int i = 0;
    while (i < 3) {
        if (A[i] == x)
            break;
        i++;
    }
    return i;
}
```

Example Application

- DSE tests the below program starting with input x = 1. What is the input and constraint (C1 ∧ C2 ∧ C3) solved in each run of DSE? Use depth-first search and leave trailing constraints blank if unused.

Run	x	C1	C2	C3
1	1	5 != x0	7 != x0	9 == x0
2	9	5 != x0	7 == x0	
3	7	5 == x0		
4	5			

```
int test_me(int x) {
    int[] A = { 5, 7, 9 };
    int i = 0;
    while (i < 3) {
        if (A[i] == x)
            break;
        i++;
    }
    return i;
}
```

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

	Concrete Execution		Symbolic Execution	
	concrete state	symbolic state		path condition
	x = 22	x = x_0		
	y = 7	y = y_0		

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$x_0 \ != y_0$
y = 7	$y = y_0$	

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

Concrete Execution: concrete state	Symbolic Execution: symbolic state	path condition
x = 22 y = 7	$x = x_0$ $y = y_0$	$x_0 \mathrel{!=} y_0$ $\mathrm{secure\_hash}(x_0) \mathrel{!=} \mathrm{secure\_hash}(y_0)$

Solve: $x_0 \mathrel{!=} y_0$ and $\mathrm{secure\_hash}(x_0) == \mathrm{secure\_hash}(y_0)$

Use concrete state: replace y_0 by 7.

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$x_0 \,!= y_0$
y = 7	$y = y_0$	secure_hash(x_0) != secure_hash(y_0)

Solve: $x_0 \,!= 7$ and secure_hash(x_0) == 601...129

Use concrete state: replace x_0 by 22.

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

False negative!

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22 y = 7	$x = x_0$ $y = y_0$	$x_0 \mathrel{!=} y_0$ $\text{secure\_hash}(x_0) \mathrel{!=} \text{secure\_hash}(y_0)$

Solve: 22 != 7 and
438...861 == 601...129

Unsatisfiable!

Another Example: Testing Data Structures

- Random Test Driver:
  - random value for x
  - random memory graph reachable from p

- Probability of reaching ERROR is extremely low

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 236 p = NULL		$x = x_0$ $p = p_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution
concrete state	symbolic state	path condition
$x = 236$ $p = NULL$	$x = x_0$ $p = p_0$	$x_0 > 0$

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	x = 236	$x = x_0$	$x_0 > 0$
	p = NULL	$p = p_0$	p_0 == NULL

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 236$	$x = x_0$	$x_0 > 0$
$p = NULL$	$p = p_0$	$p_0 == NULL$

Solve: $x_0 > 0$ and $p_0 \;!= NULL$

Solution: $x_0 = 236$, $p_0 \rightarrow$ | 634 | NULL |

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	
p → [634 | NULL]	$p = p_0$	
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
p → [634 | NULL]	$p = p_0$	
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

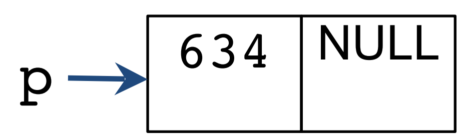
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
p → [634 | NULL]	$p = p_0$	p_0 != NULL
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution | **Symbolic Execution**

concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
	$p = p_0$	p_0 != NULL
	$p\text{->}data = v_0$	$2*x_0+1$!= v_0
	$p\text{->}next = n_0$	



Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	**symbolic state**	**path condition**
x = 236	$x = x_0$	$x_0 > 0$
p → [634 | NULL]	$p = p_0$	p_0 != NULL
	p->data = v_0	$2*x_0+1$!= v_0
	p->next = n_0	

Solve: $x_0 > 0$ and p_0 != NULL and $2*x_0+1 == v_0$

Solution: $x_0 = 1$, p_0 → [3 | NULL]

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$	
p → [3 | NULL]	$p = p_0$	
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 1$	$x = x_0$	$x_0 > 0$
p → [3 | NULL]	$p = p_0$	
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

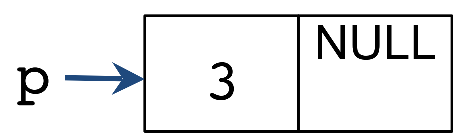
Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution | **Symbolic Execution**

concrete state

x = 1



symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

$x_0 > 0$

$p_0 \; != \; NULL$

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	$x = 1$	$x = x_0$	$x_0 > 0$
	p → [3 | NULL]	$p = p_0$	$p_0 \mathrel{!}= NULL$
		$p\text{->}data = v_0$	$2*x_0+1 == v_0$
		$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
x = 1	x = x_0	x_0 > 0
p → [3 | NULL]	p = p_0	p_0 != NULL
	p->data = v_0	$2*x_0+1$ == v_0
	p->next = n_0	n_0 != p_0

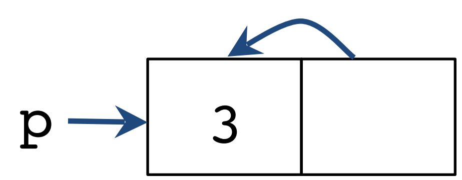
Solve: x_0 > 0 and p_0 != NULL and $2*x_0+1$==v_0 and n_0 == p_0

Solution: x_0 = 1, p_0 → [3 |]

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

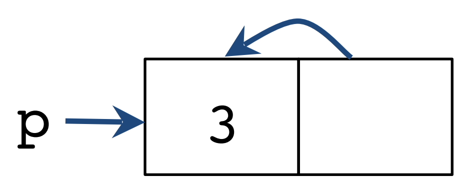
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$ $p = p_0$ $p\text{->}data = v_0$ $p\text{->}next = n_0$	



Data-Structure Example

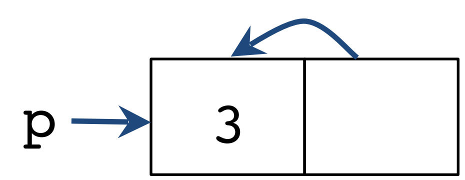
```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$ $p = p_0$ $p\text{->}data = v_0$ $p\text{->}next = n_0$	$x_0 > 0$



Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$	$x_0 > 0$
	$p = p_0$	p_0 != NULL
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

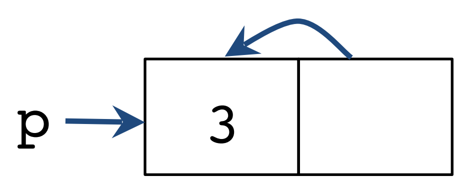
Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution | **Symbolic Execution**

concrete state

$x = 1$



symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

$x_0 > 0$

$p_0 \text{ != NULL}$

$2*x_0+1 == v_0$

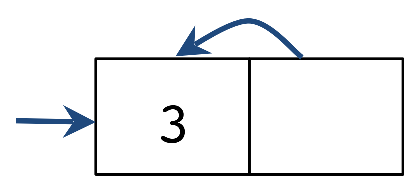
Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Program Error

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$	$x_0 > 0$
	$p = p_0$	p_0 != NULL
	p->data = v_0	$2*x_0+1 == v_0$
	p->next = n_0	n_0 != p_0

p



Approach in a Nutshell

- Generate concrete inputs, each taking different program path
- On each input, execute program both concretely and symbolically
- Both cooperate with each other:
    - Concrete execution guides symbolic execution
        - Enables it to overcome incompleteness of theorem prover
    - Symbolic execution guides generation of concrete inputs
        - Increases program code coverage

Realistic Implementations

- KLEE: LLVM (C family of languages)
- PEX: .NET Framework
- jCUTE: Java
- Jalangi: Javascript
- SAGE and S2E: binaries (x86, ARM, ...)

How does Symbolic Execution Find bugs?

- It is possible to extend symbolic execution to help us catch bugs
- How: Dedicated checkers
  - Divide by zero example --- y = x / z where x and z are symbolic variables and assume current PC (i.e. path constraint) is f
  - Even though we only fork in branches we will now fork in the division operator
  - One branch in which z = 0 and another where z !=0
  - We will get two paths with the following constraints:
  - z = 0 && f, z != 0 && f
  - Solving the constraint z = 0 && f will give us concrete input values that will trigger the divide by zero error.

How does Symbolic Execution Find bugs?

- It is possible to extend symbolic execution to help us catch b
- How: Dedicated checkers
  - Divide by zero example --- y = x / z where x
    and assume current PC (i.e. path constr
  - Even though we only fork in bran division
    operator
  - One branch in which z 0
  - We will get two nstraints:
  - $\boxed{z = 0 \&\&}$
  - So will give us concrete input values that will
    error.

Write a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, integer underflow)

Classic Symbolic Execution --- Practical Issues

- Loops and recursions --- infinite execution tree
- Path explosion --- exponentially many paths
- Heap modeling --- symbolic data structures and pointers
- SMT solver limitations --- dealing with complex path constraints
- Environment modeling --- dealing with native / system/library calls/file operations/network events

EXE vs. KLEE and the UC-KLEE

EXE: Automatically Generating Inputs of Death,
David Dill, Vijay Ganesh, Cristian Cadar, Dawson Engler, Peter Pawlowski, CCS'06

What if you could find all the bugs in your code, automatically ?

EXE: EXecution generated Executions

- The Idea:
    - Code can automatically generate its own (potentially highly complex) test cases

EXE: EXecution generated Executions

- The Algorithm
  - Symbolic execution

    +
  - Constraint solving

EXE: EXecution generated Executions

- As program runs
  - Executes each feasible path, tracking all constraints
- A path terminates upon
  - exit() crash
  - failed 'assert' error detection
- When a path terminates
  - Calls STP to solve the path's constraints for concrete values

EXE: EXecution generated Executions

- Identifies all input values causing these errors
    - Null or Out-of-bounds memory reference
    - Overflow
    - Division or modulo by 0
- Identifies all input values causing assert invalidation

Example

```
int main(void) {
    unsigned int i, t, a[4] = { 1, 3, 5, 2 };



    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);
    return 0;
}
```

Marking Symbolic Data

```
int main(void) {
    unsigned int 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);
    return 0;
}
```

Marks the 4 bytes associated with 32-bit variable ‘i’ as symbolic

 [Leibowitz’13]

Compiling...

example.c

```
int main(void) {
        unsigned int 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);
        return 0;
}
```

EXE compiler ⇒ **example.out**

Executable

1

Inserts checks around every assignment, expression & branch, to determine if its operands are concrete or symbolic

`unsigned int a[4] = {1,3,5,2}`

`if (i >= 4)`

Compiling...

example.c

```
int main(void) {
	unsigned int 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);
	return 0;
}
```

EXE compiler

example.out

Executable

1

Inserts checks around every assignment, expression & branch, to determine if its operands are concrete or symbolic

If any operand is symbolic, the operation is not performed, but is added as a constraint for the current path

Compiling...

example.c

```
int main(void) {
        unsigned int 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);
        return 0;
}
```

EXE compiler

example.out

Executable

2

Inserts code to fork program execution when it reaches a symbolic branch point, so that it can explore each possibility

if (i >= 4)

(i ≥ 4)   (i < 4)

Compiling...

example.c

```c
int main(void) {
        unsigned int 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);
        return 0;
}
```

EXE compiler

2

example.out

Executable

Inserts code to fork program execution when it reaches a symbolic branch point, so that it can explore each possibility

For each branch constraint, queries STP for existence of at least one solution for the current path. If not – stops executing path

Compiling...

example.c

```
int main(void) {
	unsigned int 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);
	return 0;
}
```

EXE compiler

3

example.out

Executable

Inserts code for checking if a symbolic expression could have any possible value that could cause errors

`t = t / a[i]`

Division by Zero ?

Compiling...

example.c

```
int main(void) {
        unsigned int 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);
        return 0;
}
```

EXE compiler

3

example.out

Executable

Inserts code for checking if a symbolic expression could have any possible value that could cause errors

If the check passes – the path has been verified as safe under all possible input values

Running…

```
int main(void) {
    unsigned int 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);
    return 0;
}
```

$4 \leq i$

e.g. $i = 8$

EXE generates a test case

Running…

```c
int main(void) {
    unsigned int 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);
    return 0;
}
```

$0 \leq i \leq 4$

↓

e.g. $i = 2$

↓

$p \rightarrow a[2] = 5$

↓

$a[2] = 5 - 1 = 4$

↓

$t = a[4]$

↓

Out of bounds

↓

EXE generates a test case

Running…

```
int main(void) {
    unsigned int 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);
    return 0;
}
```

$0 \leq i \leq 4$, $i \neq 2$

⇩

e.g. $i = 0$

⇩

$p \rightarrow a[0] = 1$

⇩

$a[0] = 1 - 1 = 0$

⇩

$t = a[0]$

⇩

$t = t / 0$

⇩

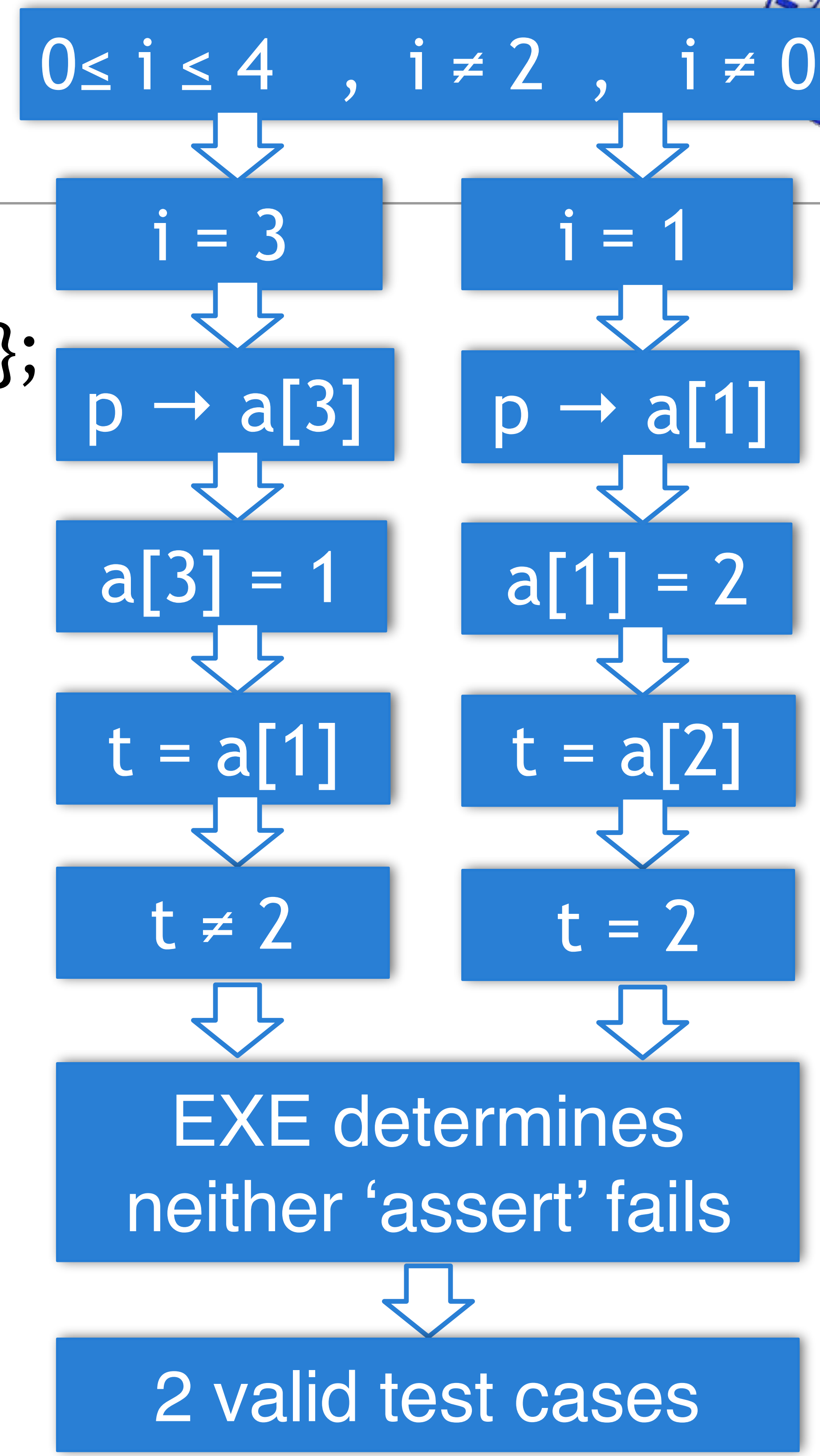
Division by 0

⇩

EXE generates a test case

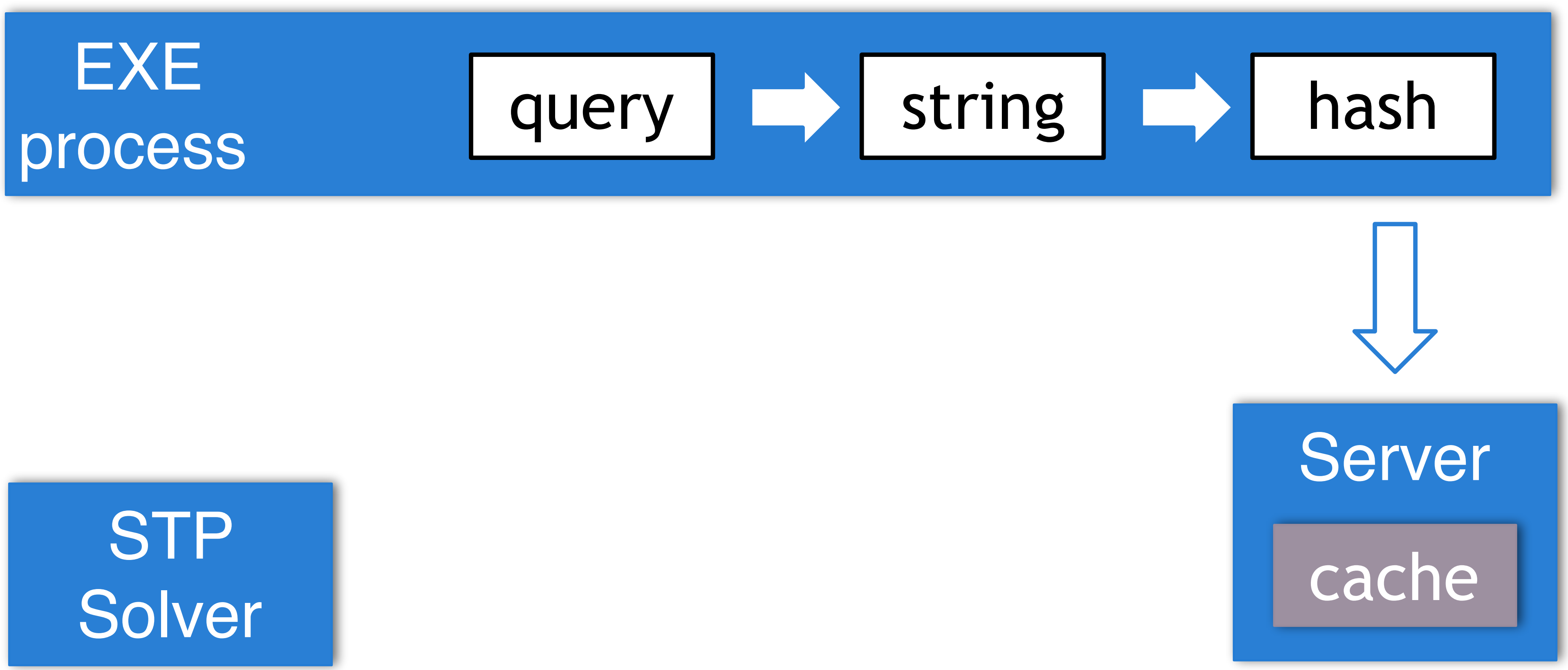
Running…

```
int main(void) {
    unsigned int 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);
    return 0;
}
```



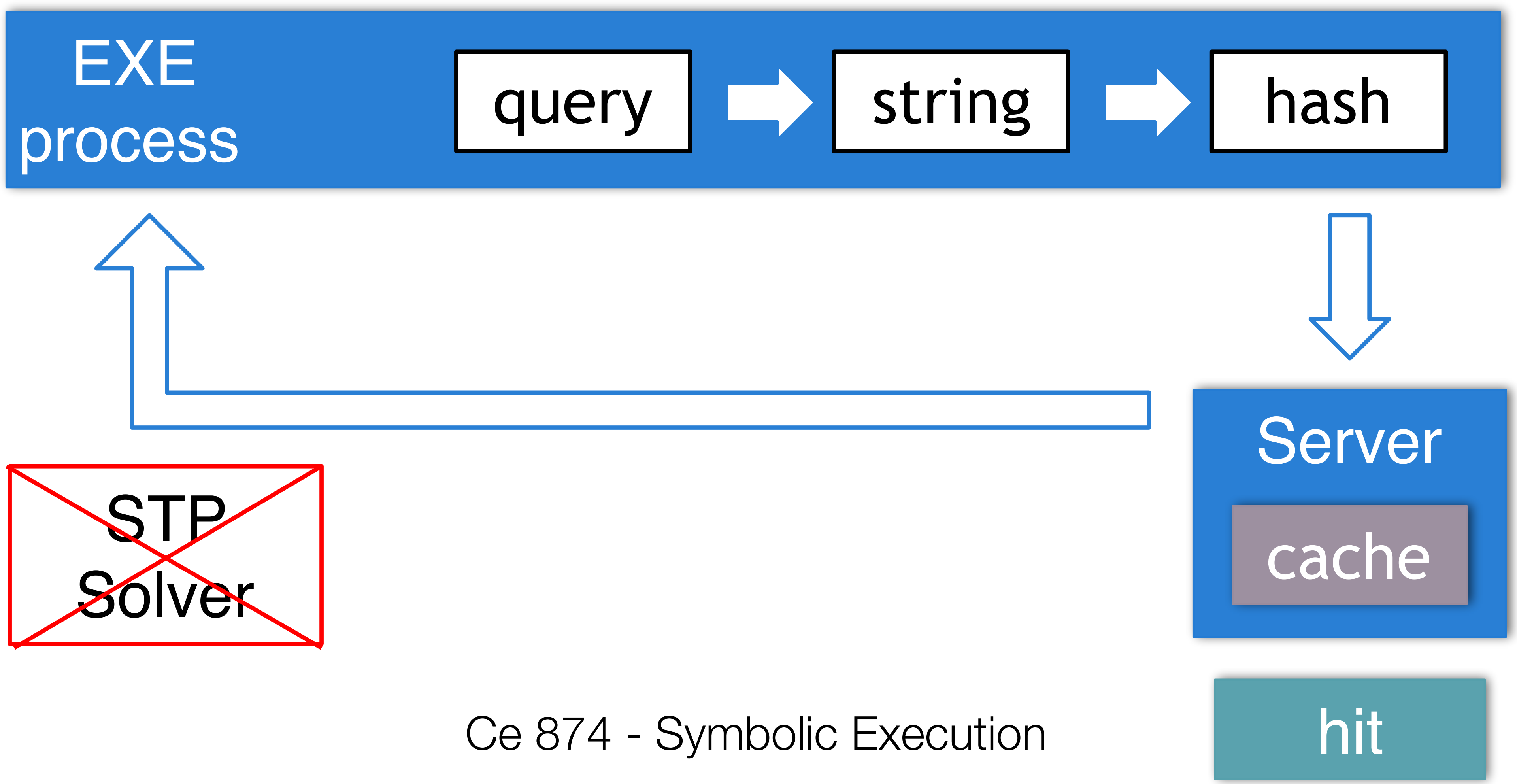
Optimizations

1. Caching constraints to avoid calling STP
   - Goal – avoid calling STP when possible
   - Results of queries and constraint solutions are cached
   - Cache is managed by a server process
   - Naïve implementation – significant overhead



Optimizations

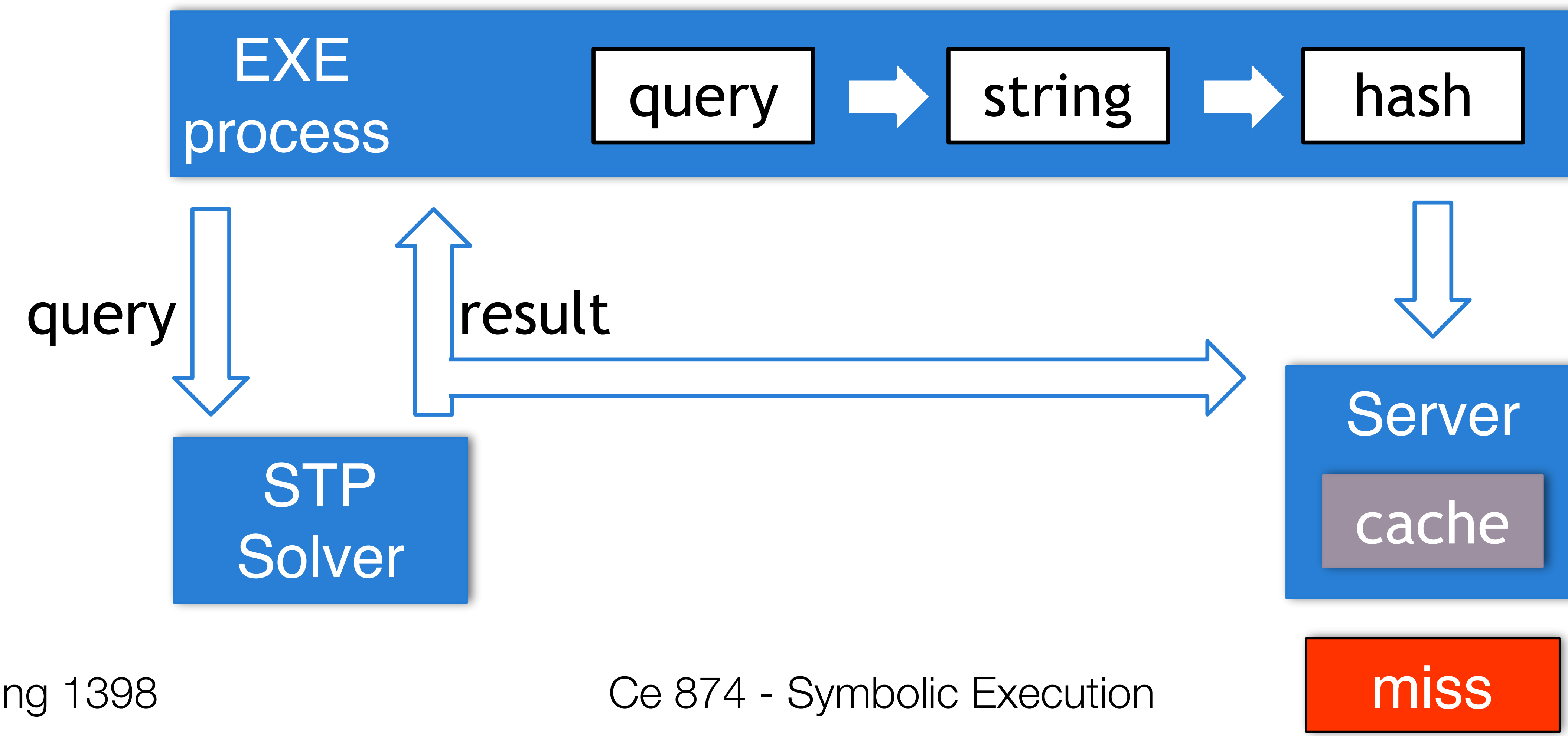
1. Caching constraints to avoid calling STP
    - Goal – avoid calling STP when possible
    - Results of queries and constraint solutions are cached
    - Cache is managed by a server process
    - Naïve implementation – significant overhead



[Leibowitz'13]

Optimizations

1. Caching constraints to avoid calling STP
    - Goal – avoid calling STP when possible
    - Results of queries and constraint solutions are cached
    - Cache is managed by a server process
    - Naïve implementation – significant overhead



 [Leibowitz'13]

Optimizations

2. Constraint Independence
    - Breaking constraints into multiple, independent, subsets
    - Discard irrelevant constraints
    - Small cost for computing independent subsets
    - May yield additional cache hits

Optimizations

2. Constraint Independence

```
if (A[i] > A[i+1]) {
   ...
}

if (B[j] + B[j-1] == B[j+1])
{
   ...
}
```

2 consecutive independent branches

$(A[i] > A[i+1]) \&\& (B[j] + B[j-1] \neq B[j+1])$

$(A[i] \leq A[i+1]) \&\& (B[j] + B[j-1] \neq B[j+1])$

$(A[i] \leq A[i+1]) \&\& (B[j] + B[j-1] = B[j+1])$

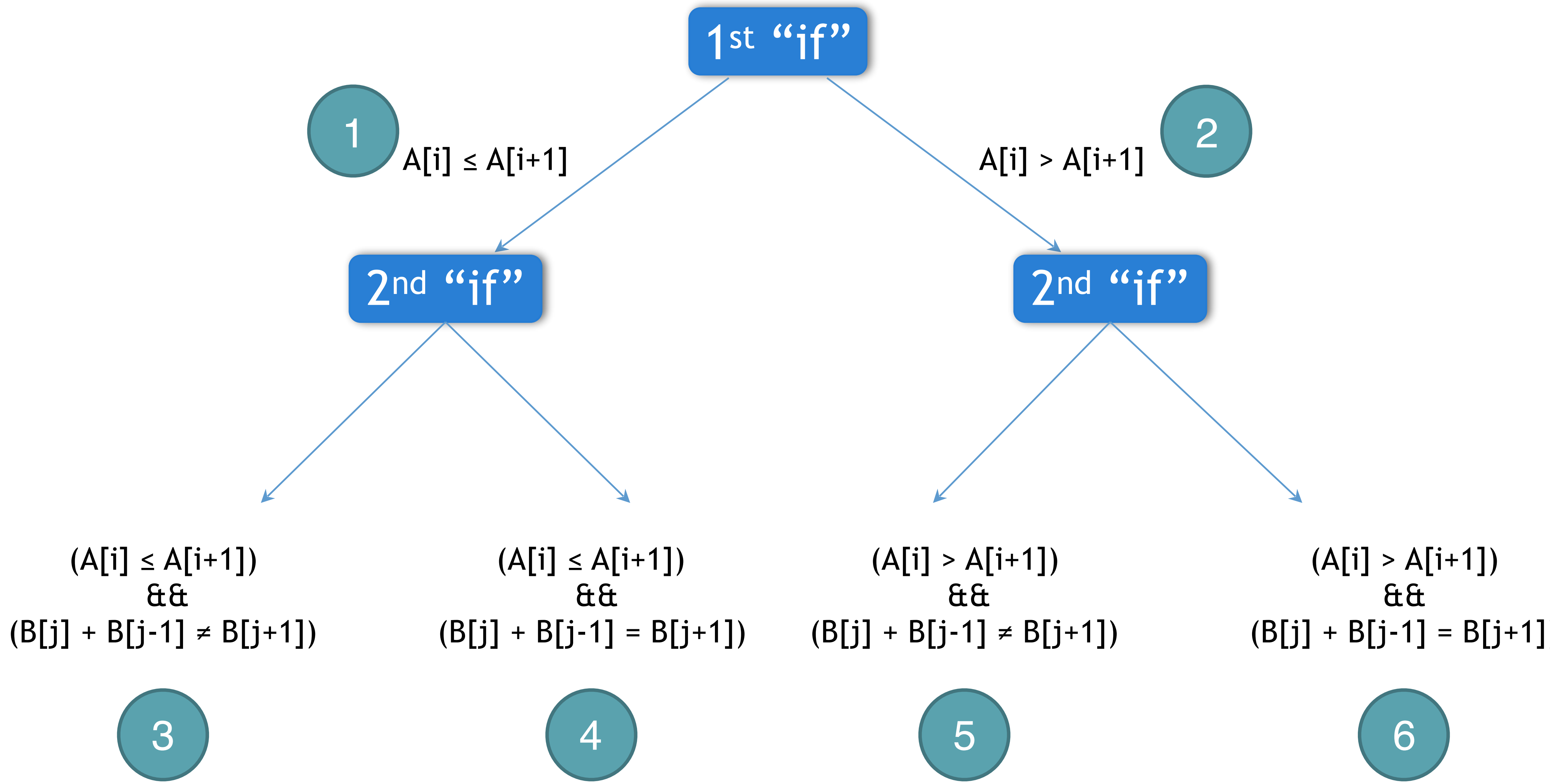
$(A[i] > A[i+1]) \&\& (B[j] + B[j-1] = B[j+1])$

4 possible paths

Optimizations

2. Constraint Independence

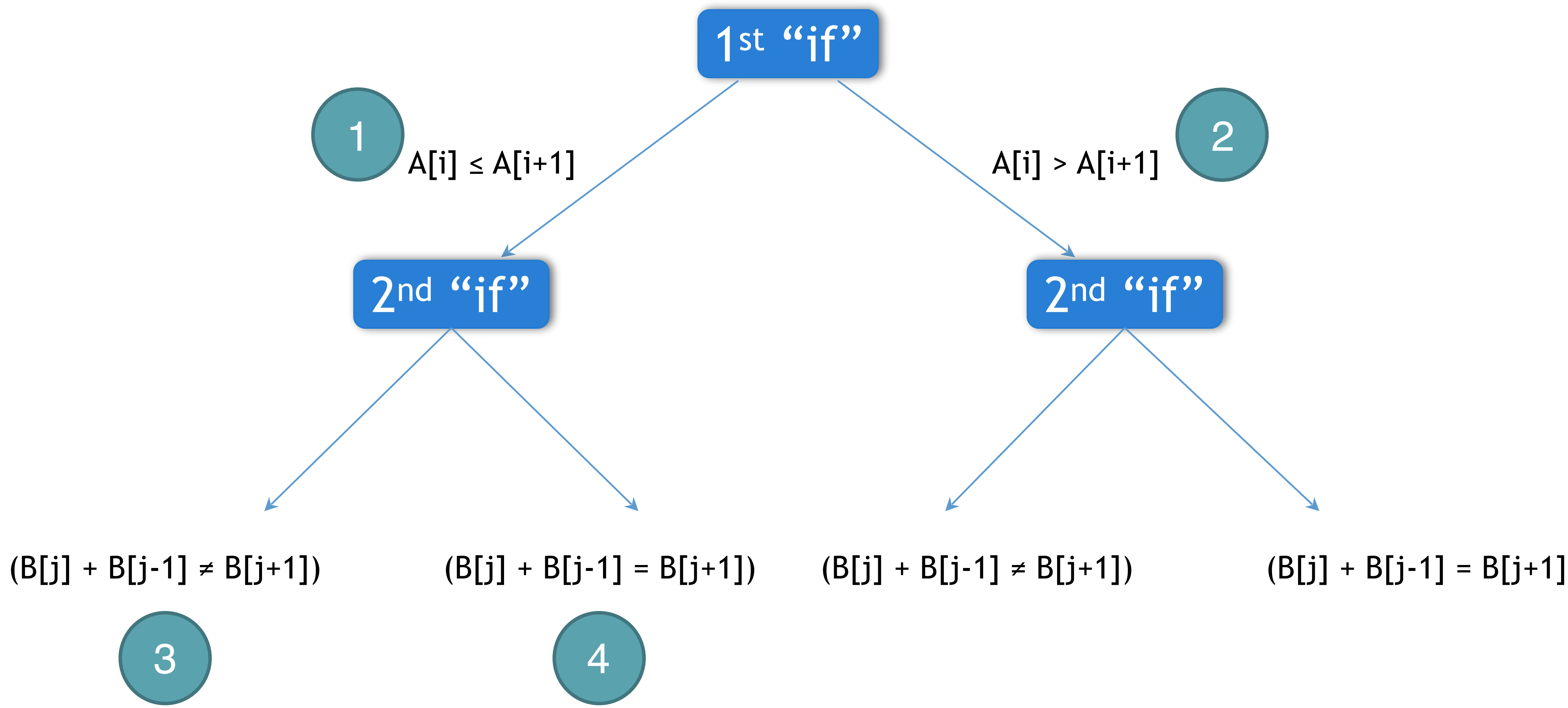
no optimization



Optimizations

2. Constraint Independence

with optimization



Optimizations

2. Constraint Independence

'n' consecutive independent branches

no optimization

$2(2^n-1)$ queries to STP

with optimization

$2n$ queries to STP

Optimizations

3. Search Heuristics – "Best First Search" & DFS
   - By default, EXE uses DFS when forking for picking which branch to follow first
   - Problem – Loops bounded by symbolic variables
   - Solution
     - Each forked process calls search server, and blocks
     - Server picks process blocked on line of code which has run the fewer number of times
     - Picked process and children are run with DFS

Optimizations

- Experimental Performance
  - Used to find bugs in
    - 2 packet filters (FreeBSD & Linux)
    - DHCP server (udhcpd)
    - Perl compatible regular expressions library (pcre)
    - XML parser library (expat)
  - Ran EXE without optimizations, with each optimization separately, and with all optimizations

Optimizations

- Experimental Performance
  - Positive
    - With both caching & independence – Faster by 7%-20%
    - Cache hit rate jumps sharply with independence
    - Cost of independence – near zero
    - Best First Search gets (almost) full coverage more than twice as fast than DFS
    - Coverage with BFS compared to random testing: 92% against 57%

Optimizations

- Experimental Performance
  - Interesting
    - Actual growth of number of paths is much smaller than potentially exponential growth
    - EXE is able to handle relatively complex code
  - Negative
    - Cache lookup has significant overhead, as conversion of queries to string is dominant
    - STP by far remains highest cost (as expected)

Advantages

- Automation – "competition" is manual and random testing
- Coverage - can test any executable code path and (given enough time) exhaust them all
- Generation of actual attacks and exploits
- No false positives

Limitations

- Optimizations – far from perfect implementation
- Benchmarks – hand-picked, small-scaled
- Single threaded – each path is explored independently from others
- Code doesn’t interact with it’s surrounding environment

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, Cristian Cadar, Daniel Dunbar, Dawson Engler, OSDI'08

KLEE

- Shares main idea with EXE, but completely redesigned
- Deals with the external environment
- More optimizations, better implemented
- Targeted at checking system-intensive programs “out of the box”
- Thoroughly evaluated on real, more complicated, environment-intensive programs

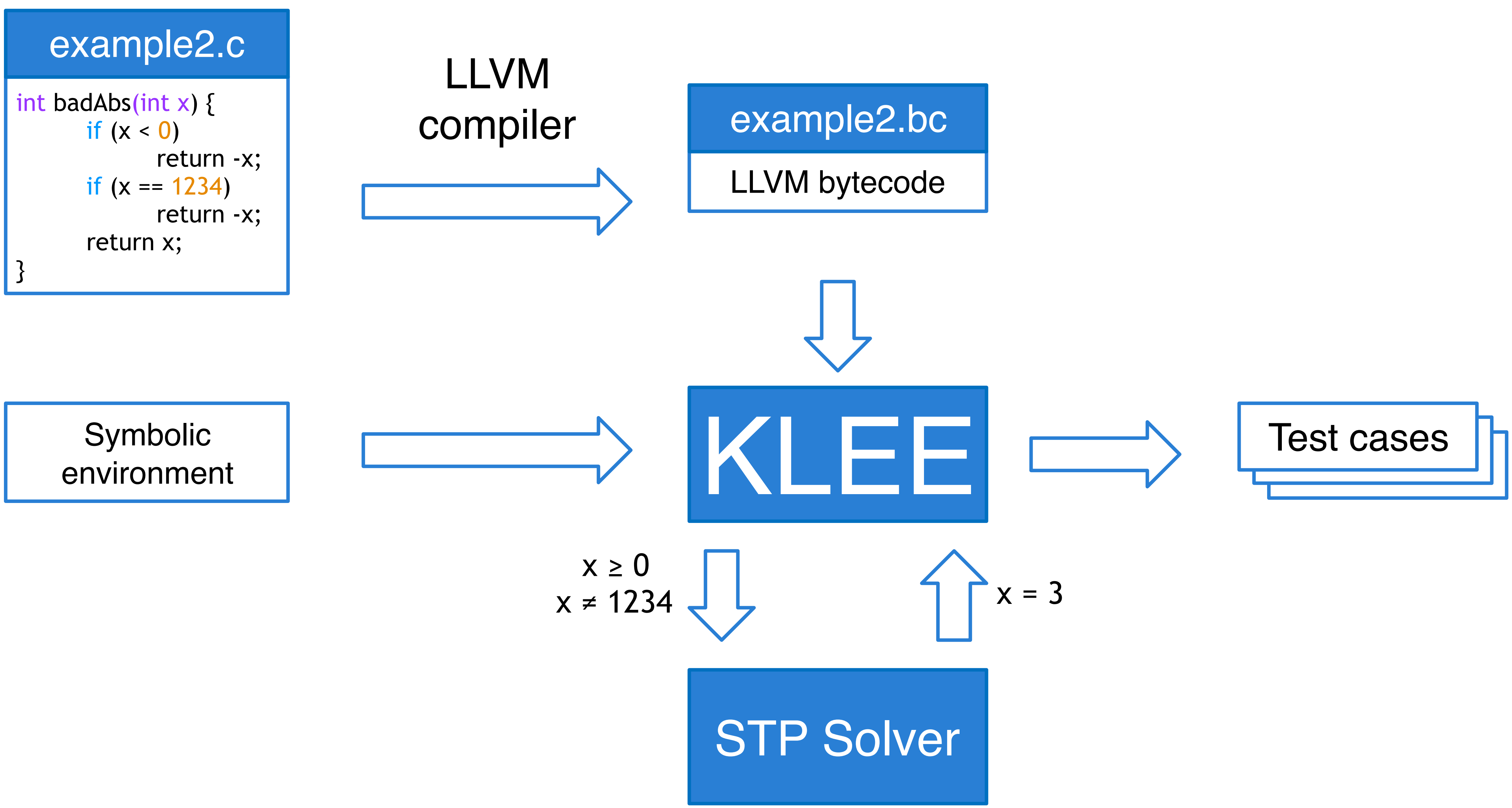
KLEE

- A hybrid between an **operating system for symbolic processes** and an **interpreter**
  - Programs are compiled to virtual instruction sets in LLVM assembly language
  - Each symbolic process (“state”) has a symbolic environment
    - register file   stack   heap
    - program counter   path condition
  - Symbolic environment of a state (unlike a normal process)
    - Refers to symbolic expressions and not concrete data values

KLEE

- Able to execute a large number of states simultaneously
- At its core – an interpreter loop
  - Selects a state to run (search heuristics)
  - Symbolically executes a single instruction in the context of the state
  - Continues until no remaining states
  - (or reaches user-defined timeout)

Architecture



Execution

- Conditional Branches
  - Queries STP to determine if the branch condition is true or false
  - The state’s instruction pointer is altered suitably
  - Both branches are possible?
    - State is cloned, and each clone’s instruction pointer and path condition are updated appropriately

Execution

- Targeted Errors
  - As in EXE
    - Division by 0
    - Overflow
    - Out-of-bounds memory reference

Modeling the Environment

- Code reads/writes values from/to its environment
    - Command line arguments
    - Environment variables
    - File data
    - Network packets
- Want to return all possible values for these reads
- How?
    - Redirecting calls that access the environment to custom models

Modeling the Environment

- Example: Modeling the File System
  - File system operations
    - Performed on an actual concrete file on disk?
      - Invoke the corresponding system call in the OS
    - Performed on a symbolic file?
      - Emulate the operation's effect on a simple symbolic file system (private for each state)
  - Defined simple models for 40 system calls

Modeling the Environment

- Example: Modeling the File System
  - Symbolic file system
    - Crude
      - Contains a single directory with N symbolic files
      - User can specify N and size of files
    - Coexists with real file system
      - Applications can use files in both

Modeling the Environment

- Failing system calls
  - Environment can fail in unexpected ways
    - write() when disk is full
    - Unexpected, hard-to-diagnose bugs
  - Optionally simulates environmental failures
    - Failing system calls in a controlled manner

Optimizations

1. Compact State Representation
    - Number of concurrent states grows quickly (even >100,000)
    - Implements copy-on-write at object level
        - Dramatically reduces memory requirements per state
        - Heap structure can be shared amongst multiple states
        - Can be cloned in constant time (very frequent operation)

Optimizations

2. Simplifying queries
    - Cost of constraint solving dominates everything else
        - Make solving faster
        - Reduce memory consumption
        - Increase cache hit rate (to follow)

Optimizations

2. Simplifying queries

   a. Expression Rewriting

   - Simple arithmetic simplifications

     $\boxed{x + 0} \Rightarrow \boxed{x}$

   - Strength reduction

     $\boxed{x * 2^n} \Rightarrow \boxed{x << n}$

   - Linear simplification

     $\boxed{2*x - x} \Rightarrow \boxed{x}$

 [Leibowitz'13]

Optimizations

2. Simplifying queries

   b. Constraint Set Simplification

   - Constraints on same variables tend to become more specific
   - Rewrites previous constraints when new, equality constraints, are added to the set

$$\boxed{x < 10}$$

Optimizations

2. Simplifying queries

    b. Constraint Set Simplification

- Constraints on same variables tend to become more specific
- Rewrites previous constraints when new, equality constraints, are added to the set

$\boxed{x < 10}$

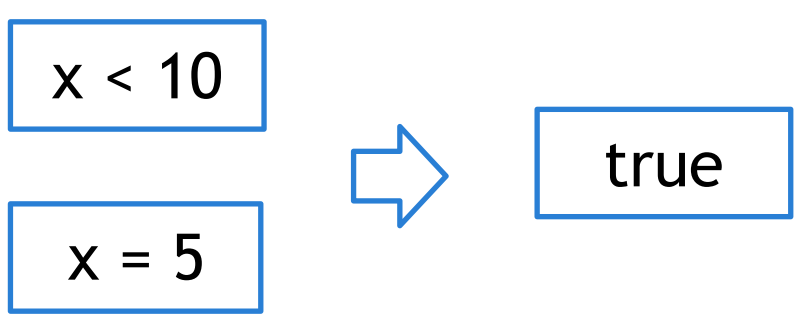
$\boxed{x = 5}$

Optimizations

2. Simplifying queries

   b. Constraint Set Simplification

   - Constraints on same variables tend to become more specific
   - Rewrites previous constraints when new, equality constraints, are added to the set



Optimizations

- Simplifying queries

  c. Implied Value Concretization

  - The value of a variable effectively becomes concrete
  - Concrete value is written back to memory

$$\boxed{x + 1 = 10} \Rightarrow \boxed{x = 9}$$

Optimizations

- Simplifying queries

  d. Constraint Independence

  - As in EXE

Optimizations

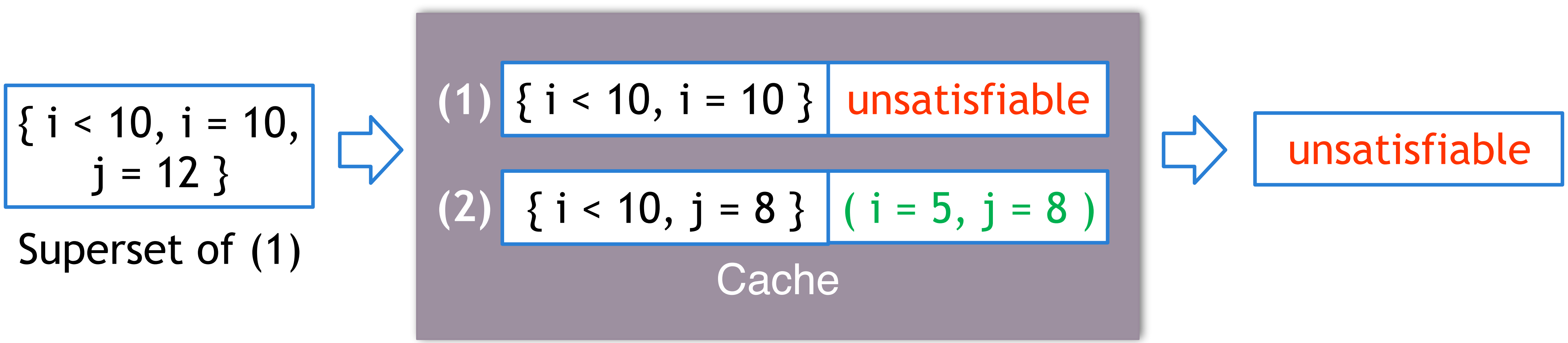
3. Counter-Example Cache
   - More sophisticated than in EXE
   - Allows efficient searching for cache entries for both subsets and supersets of a given set

(1)	{ i < 10, i = 10 }	unsatisfiable
(2)	{ i < 10, j = 8 }	(i = 5, j = 8)

Cache

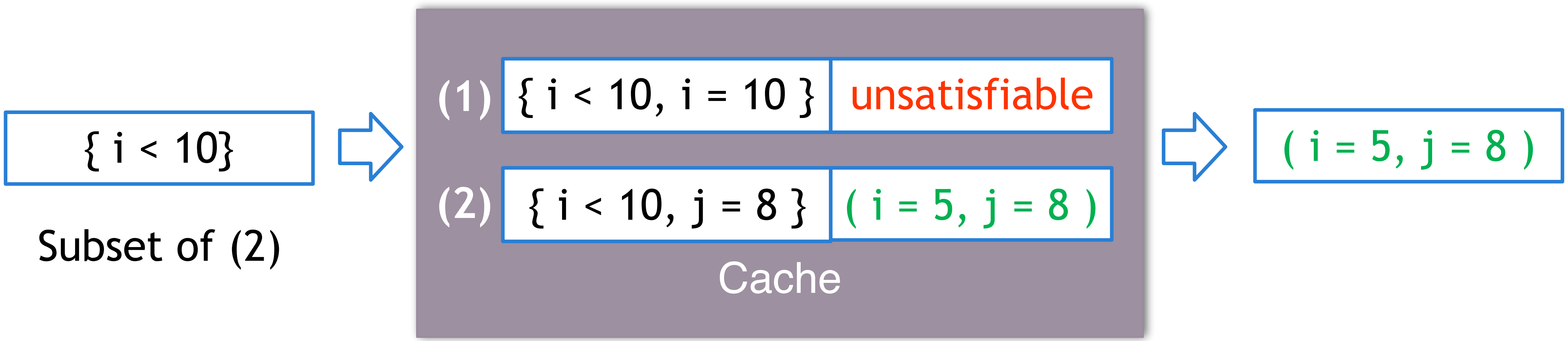
Optimizations

3. Counter-Example Cache
   - More sophisticated than in EXE
   - Allows efficient searching for cache entries for both subsets and supersets of a given set



Optimizations

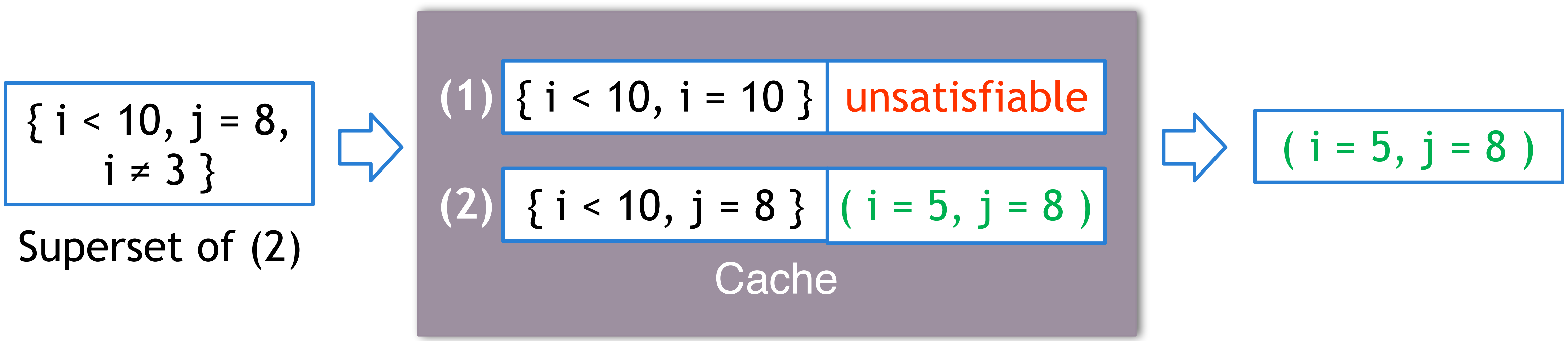
3. Counter-Example Cache
   - More sophisticated than in EXE
   - Allows efficient searching for cache entries for both subsets and supersets of a given set



Optimizations

3. Counter-Example Cache

- More sophisticated than in EXE
- Allows efficient searching for cache entries for both subsets and supersets of a given set



Optimizations

4. Search Heuristics – State Scheduling
   - The state to run at each instruction is selected by interleaving 2 strategies
   - Each is used in a Round-Robin fashion
   - Each state is run for a “time slice”
   - Ensures a state which frequently executes expensive instructions will not dominate execution time

Optimizations

4. Search Heuristics – State Scheduling
    a. Random Path Selection
    - Traverses tree of paths from root to leaves
      (internal nodes – forks, leaves – states)
    - At branch points – randomly selects path to follow
    - States in each subtree have equal probability of being selected
    - Favors states higher in the tree – less constraints, freedom to reach uncovered code
    - Avoids starvation (loop + symbolic condition = “forks bomb”)

Optimizations

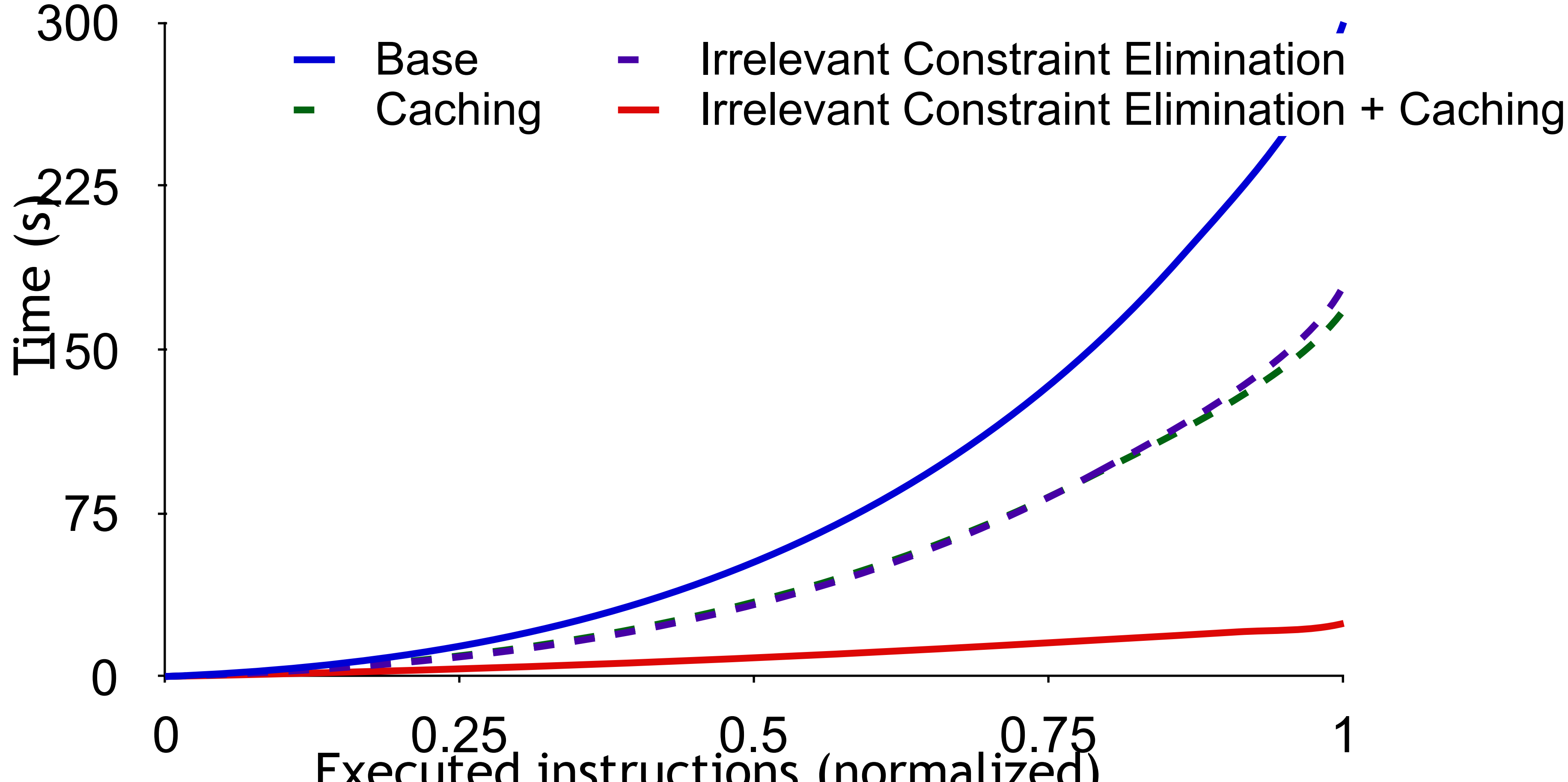
4. Search Heuristics – State Scheduling
   b. Coverage-Optimized Search
      - Tries to select states more likely to cover new code
      - Computes min. distance to uncovered instruction, call stack size & whether state recently covered new code
      - Randomly selects a state according to these weights

Optimizations

- Experimental Performance
  - Used to generate tests in
    - GNU COREUTILS Suite (89 programs)
    - BUSYBOX (72 programs)
    - Both have variety of functions, intensive interaction with the environment
    - Heavily tested, mature code
  - Used to find bugs in
    - Total of 450 applications

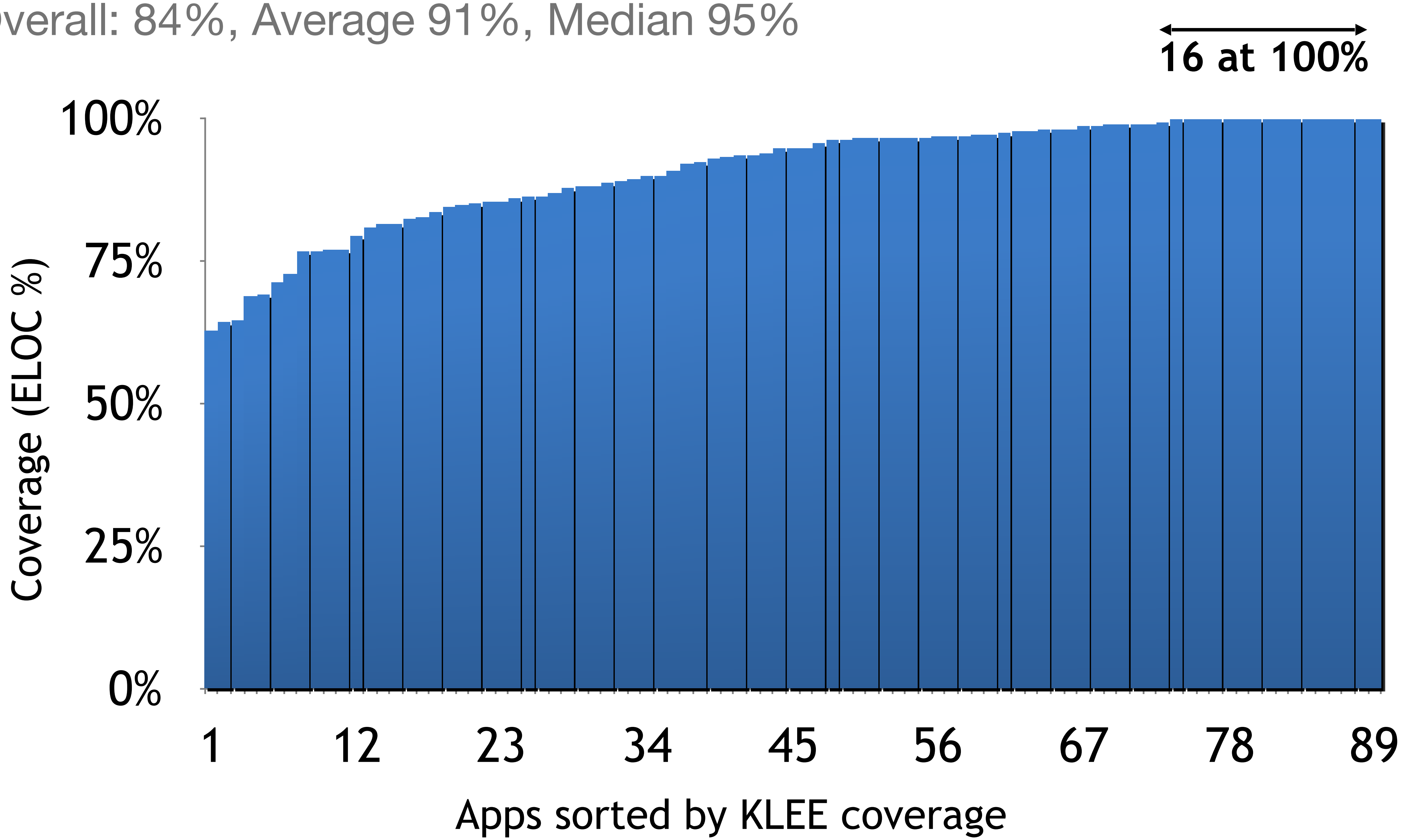
Optimizations

- Experimental Performance
  - Query simplification + caching
    - Number of STP queries reduced to 5% (!) of original
    - Time spent solving queries to STP reduced from 92% of overall time to 41% of overall time
    - Speedup



Results – Line Coverage

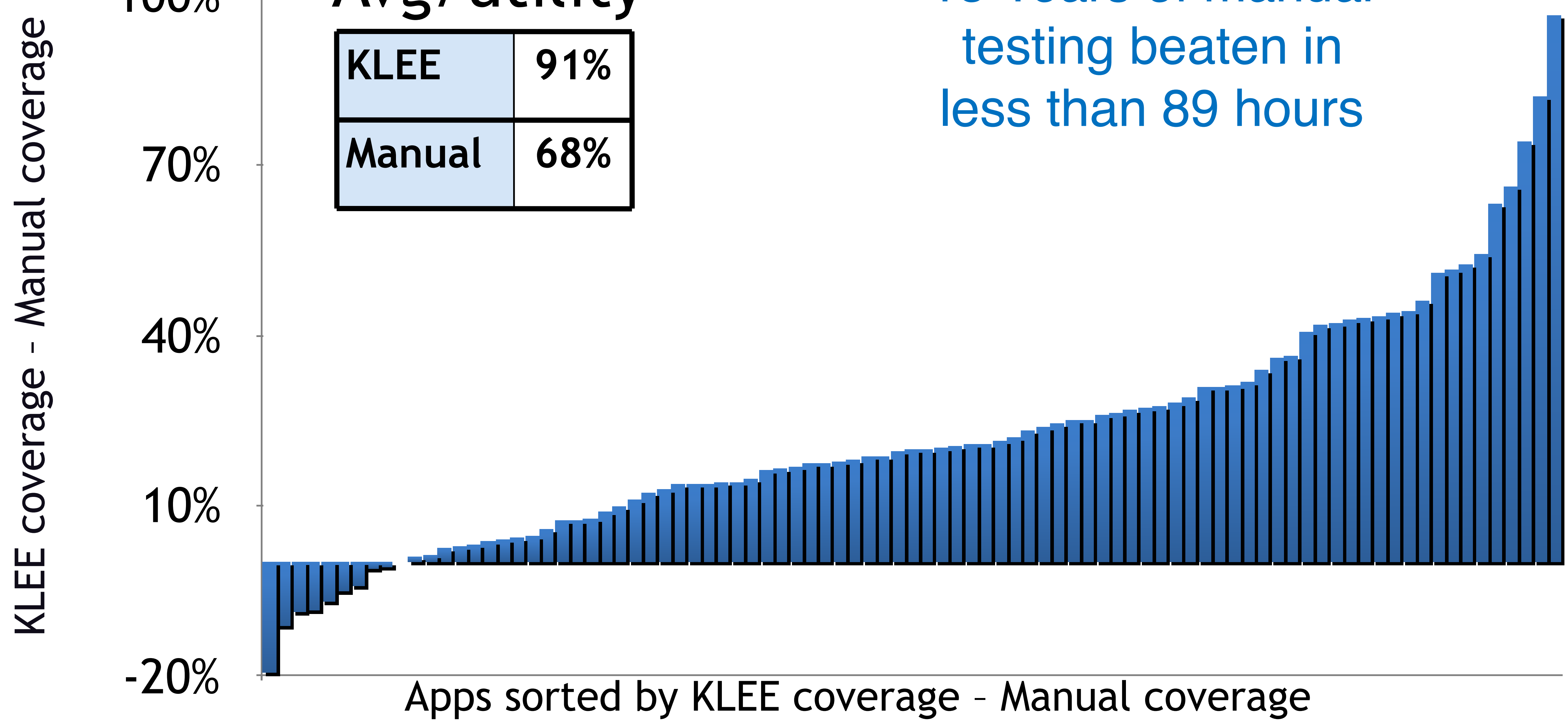
- GNU COREUTILS
- Overall: 84%, Average 91%, Median 95%



 [Leibowitz'13]

Results – Line Coverage

- GNU COREUTILS



Avg/utility

KLEE	91%
Manual	68%

Results – Line Coverage

- High coverage with few test cases
  - Average of 37 tests per tool in GNU COREUTILS
- “Out of the box” – utilities unaltered
- Entire tool suite (no focus on particular apps)
- However
  - Checks only low-level errors and violations
  - Developer tests also validate output to be as expected

Results – Bugs found

- 10 memory error crashes in GNU COREUTILS
  - More than found in previous 3 years combined
  - Generates actual command lines exposing crashes

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

```
t1.txt: "\t \tMD5("
t2.txt: "\b\b\b\b\b\b\b\t"
t3.txt: "\n"
t4.txt: "a"
```

Under-Constrained Symbolic Execution: Correctness Checking for Real Code, David A. Ramos and Dawson Engler, Usenix Security 2015

Contributions

- Technique + tool for finding deep bugs in real, open source C/C++ code
  - No manual testcases
  - No functional specification
- Bugs reported may have security implications; exploitability must be determined manually
  - Memory access, heap management, assertion failures, division-by-zero
- Found 77 new bugs in BIND, OpenSSL, Linux kernel
  - 2 OpenSSL DoS vulnerabilities: CVE-2014-0198, CVE-2015-0292
  - 14 Linux kernel vulnerabilities (mostly minor DoS issues)

Problem: Scalability

- Path explosion
  - |paths| ~ 2 ^ if-statements
- Path length and complexity
  - Undecidable: infinite-length paths (halting problem)
- SMT query complexity (NP-complete)

Solution: Under-Constrained

- Directly execute individual functions within a program
  - Less code = Fewer paths
  - Function calls executed (inter-procedural)
  - Able to test previously-unreachable code
- Challenges
  - Complex inputs (e.g., pointer-rich data structures)
  - Under-constrained: inputs have unknown preconditions
    - False positives

UC-KLEE tool

- Extends KLEE tool (OSDI 2008)
- Runs LLVM bitcode compiled from C/C++ source
- Automatically synthesizes complex inputs
    - Based on lazy initialization (Java PathFinder)
    - Supports pointer manipulation and casting in C/C++ (no type safety)
    - User-specified input depth (k-bound) [Deng 2006]

Lazy Initialization

- Symbolic (input) pointers initially unbound
- On first dereference:
  - New object allocated
  - Symbolic pointer bound to new object's address
- On subsequent dereferences:
  - Pointer resolves to object allocated above

Example

Unbound Symbolic Input

```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
       sum += n->val;
       n = n->next;
   }
   return sum;
}
```

Example

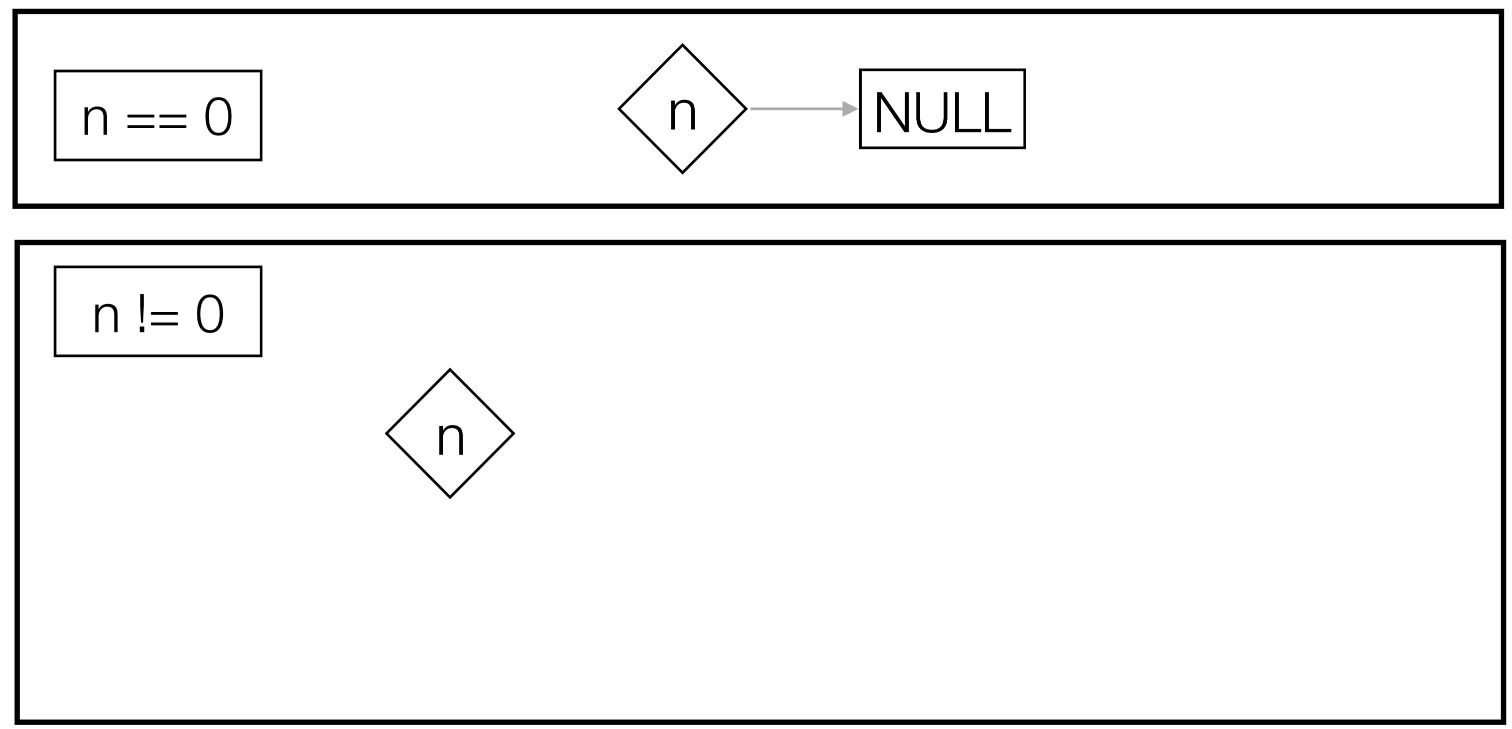
```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
      sum += n->val;
      n = n->next;
   }
   return sum;
}
```

•

Example

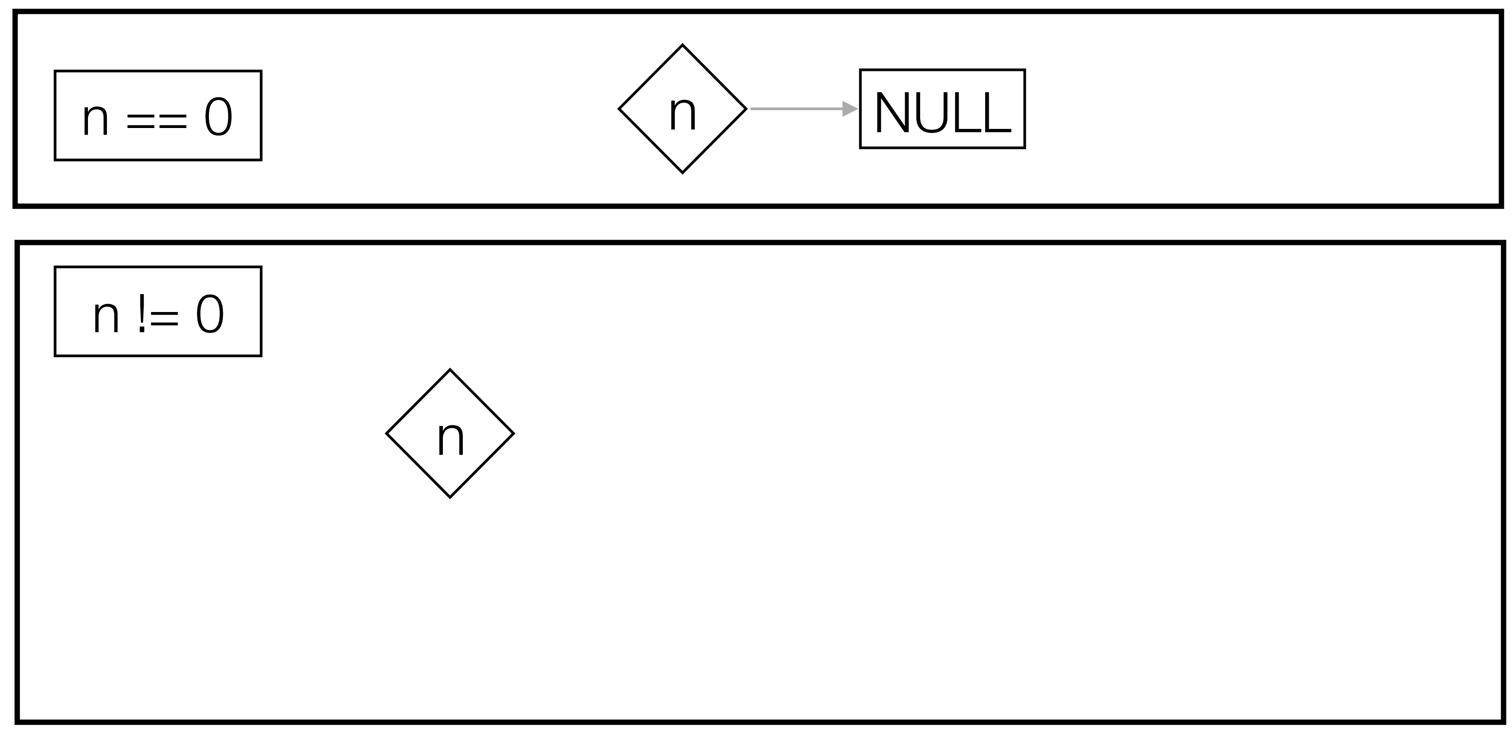
```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
       sum += n->val;
       n = n->next;
   }
   return sum;
}
```

•



Example

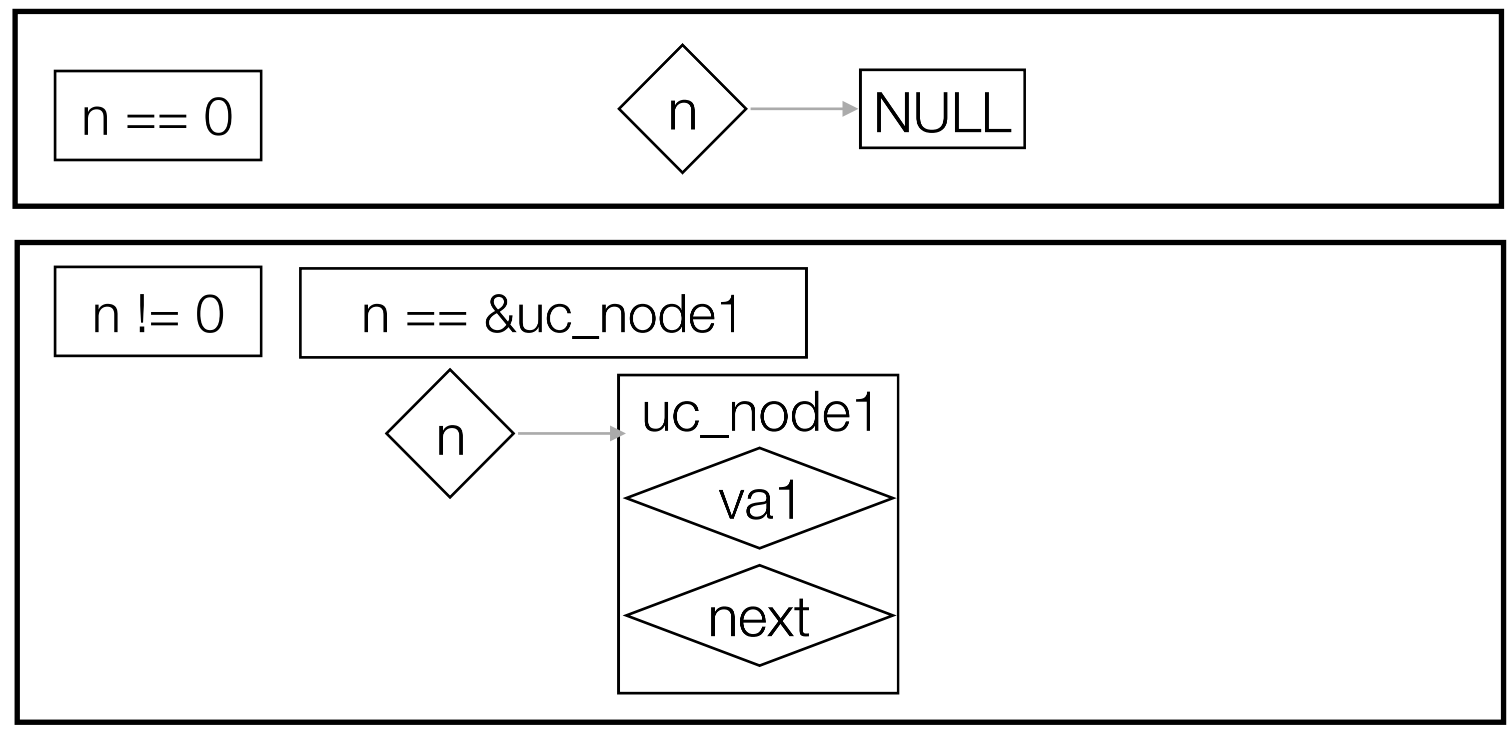
```
int   listSum(node *n) {
    int  sum = 0;
    while (n) {
       sum += n->val;
       n = n->next;
    }
    return sum;
}
```



Example

```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
      sum += n->val;
      n = n->next;
   }
   return sum;
}
```

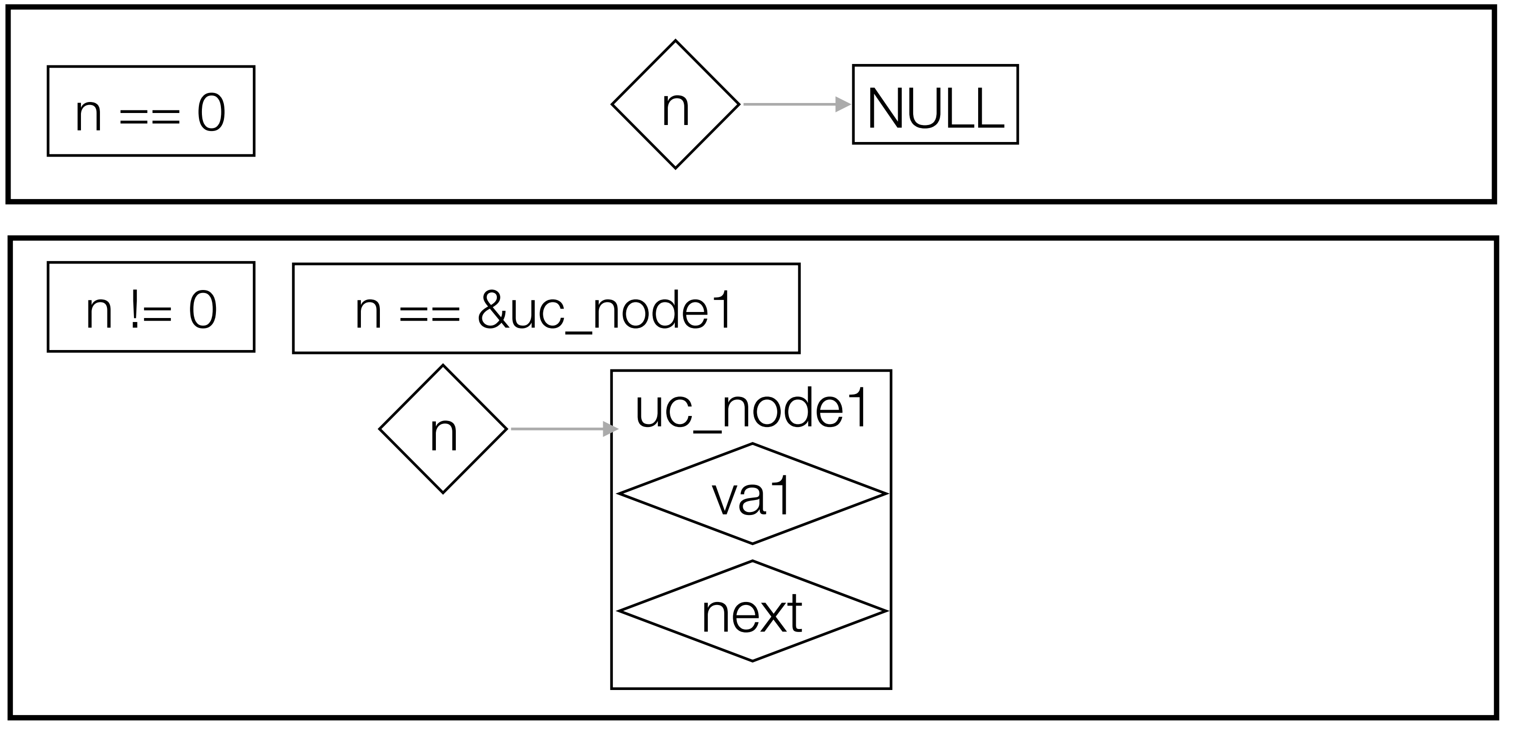
•



Example

```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
      sum += n->val;
      n = n->next;
   }
   return sum;
}
```

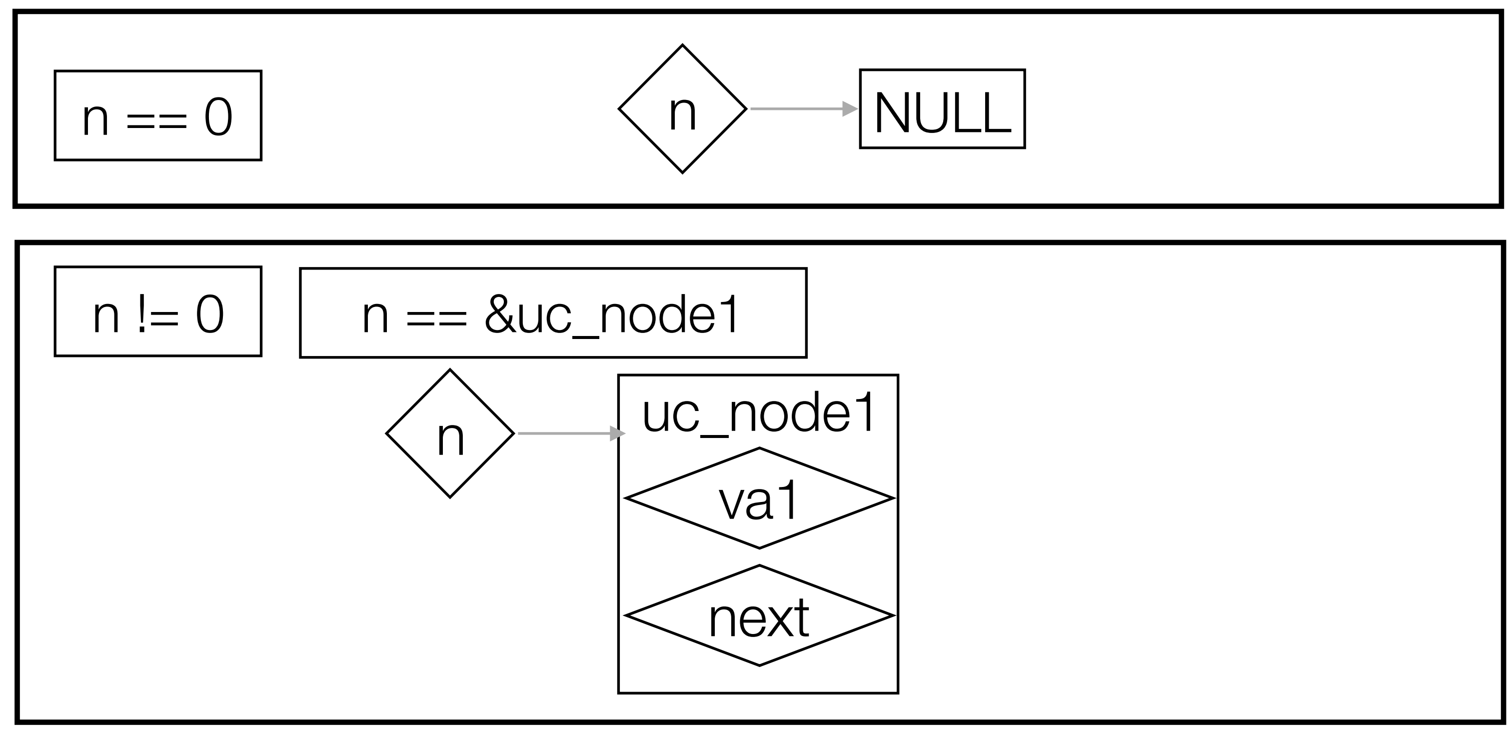
•



Example

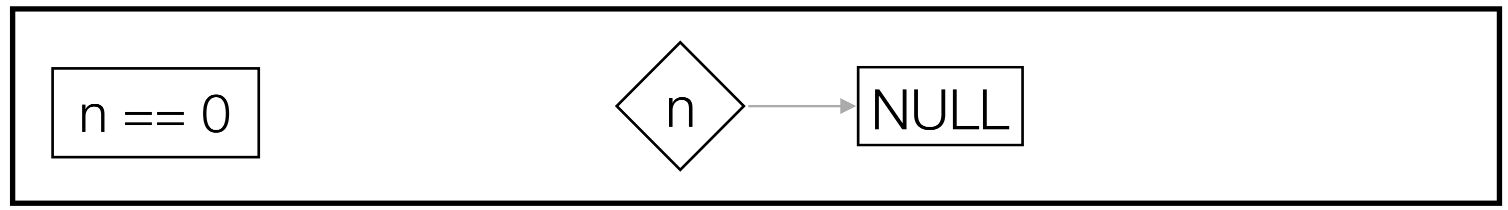
```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
      sum += n->val;
      n = n->next;
   }
   return sum;
}
```

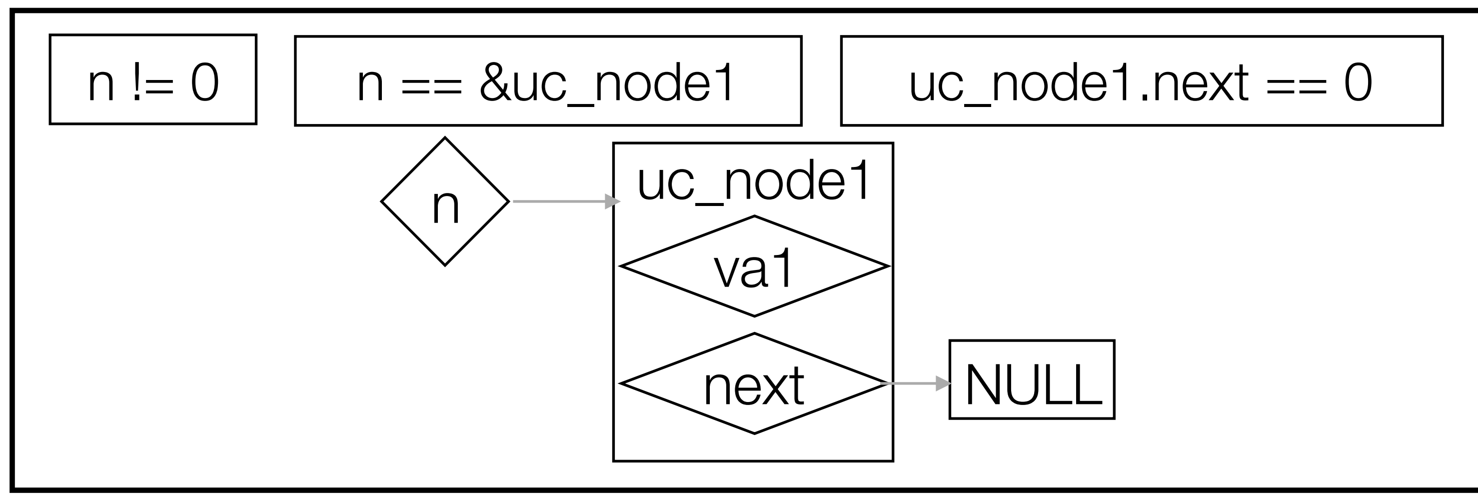
•

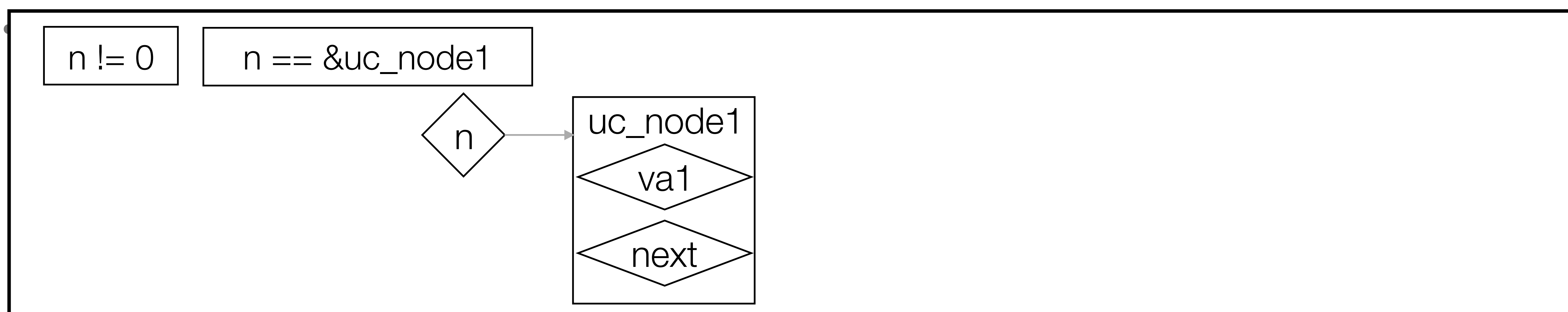


Example

```
int   listSum(node *n) {
    int  sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

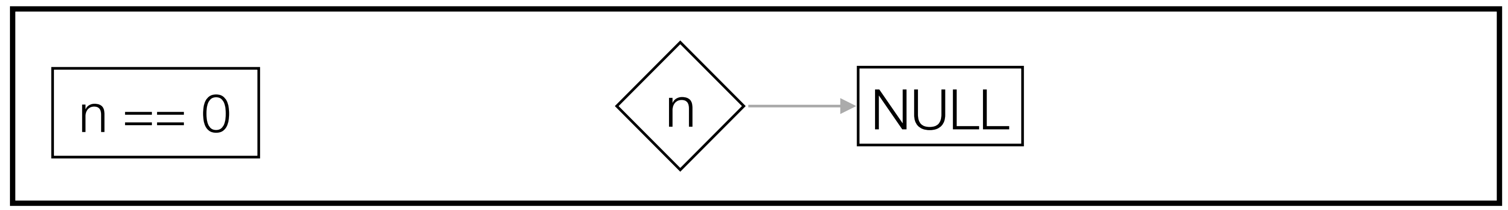


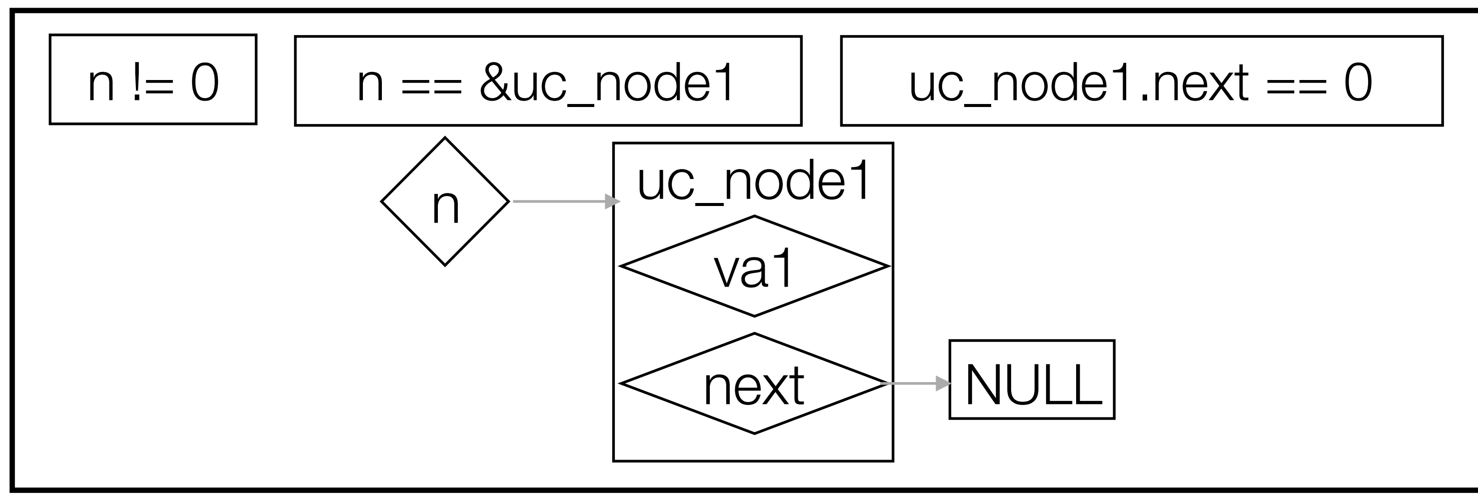


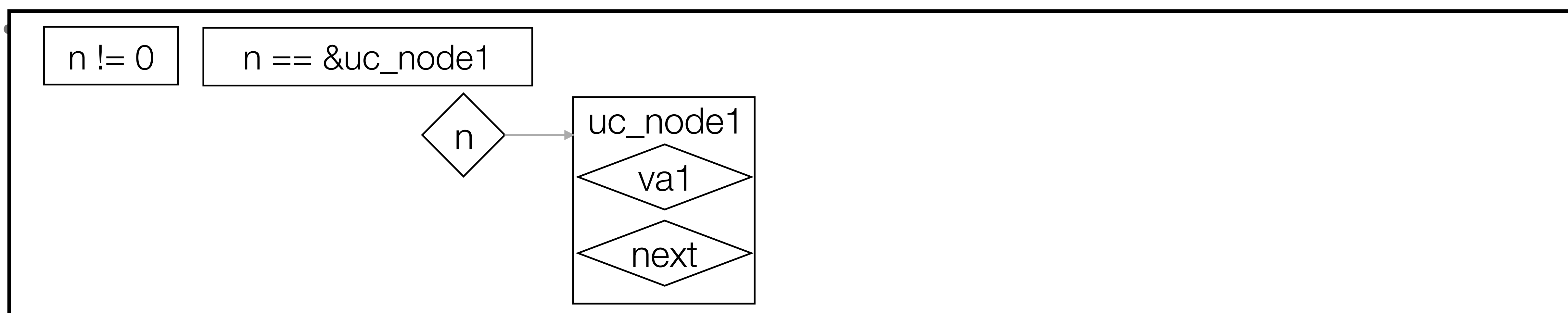


Example

```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
      sum += n->val;
      n = n->next;
   }
   return sum;
}
```

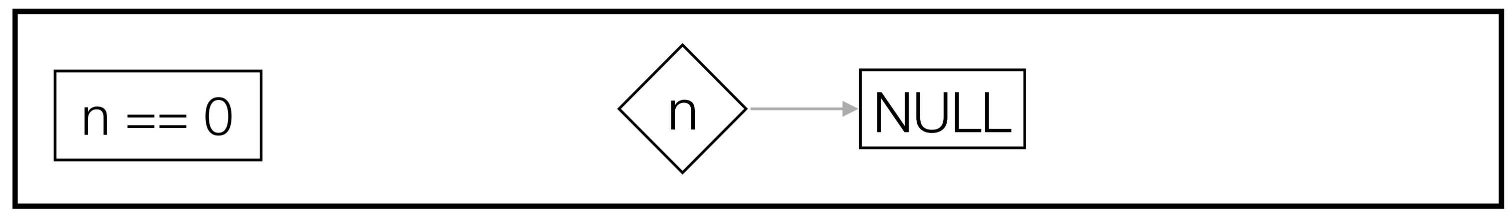


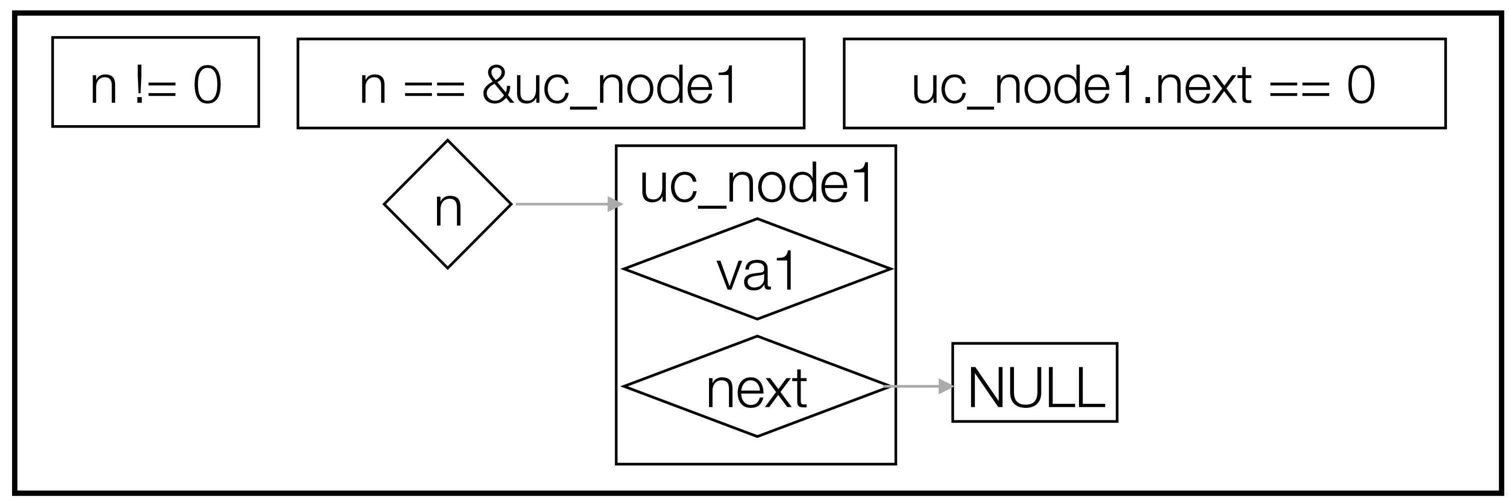


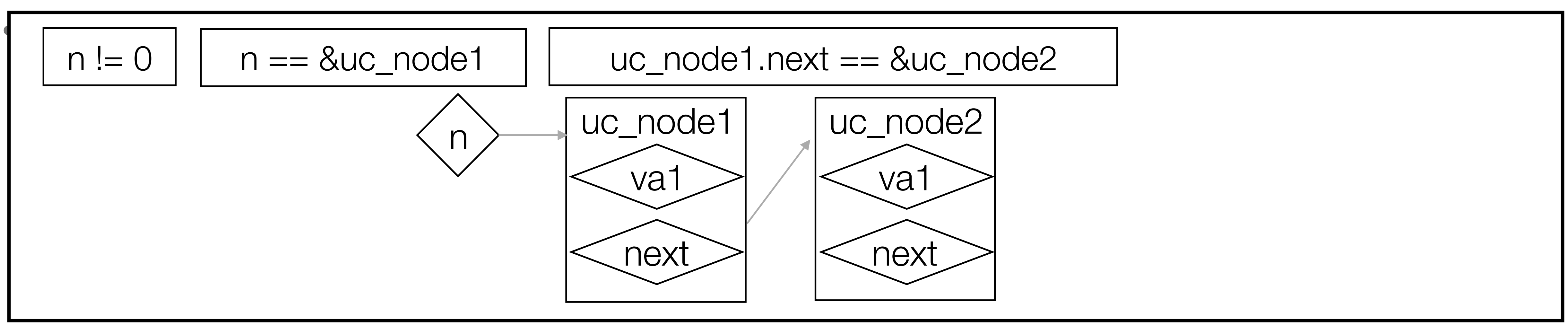


Example

```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
     sum += n->val;
     n = n->next;
   }
   return sum;
}
```

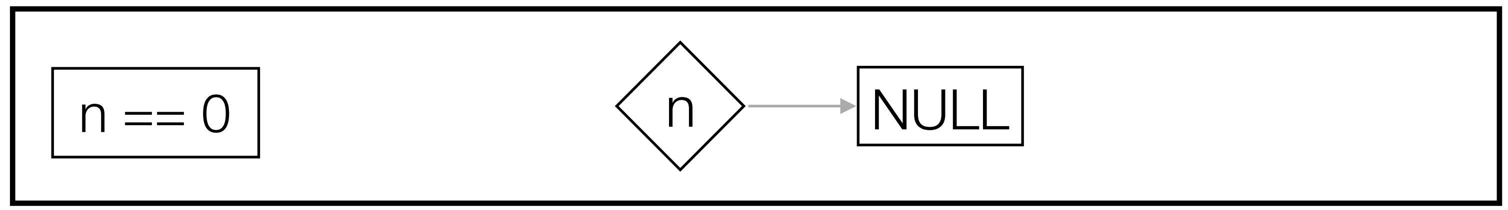


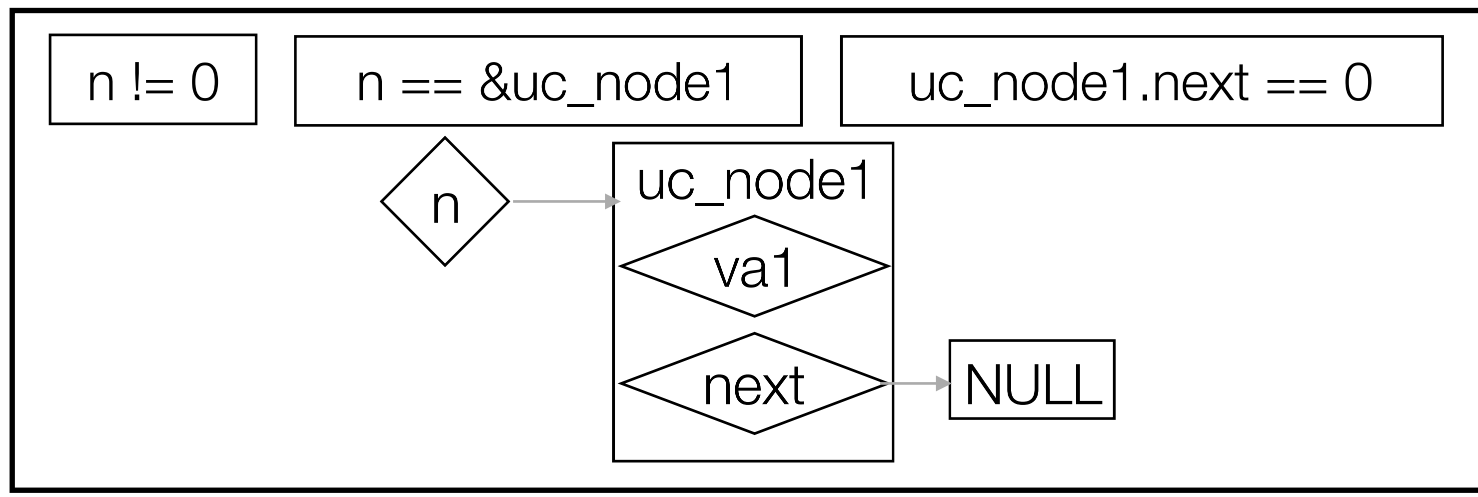


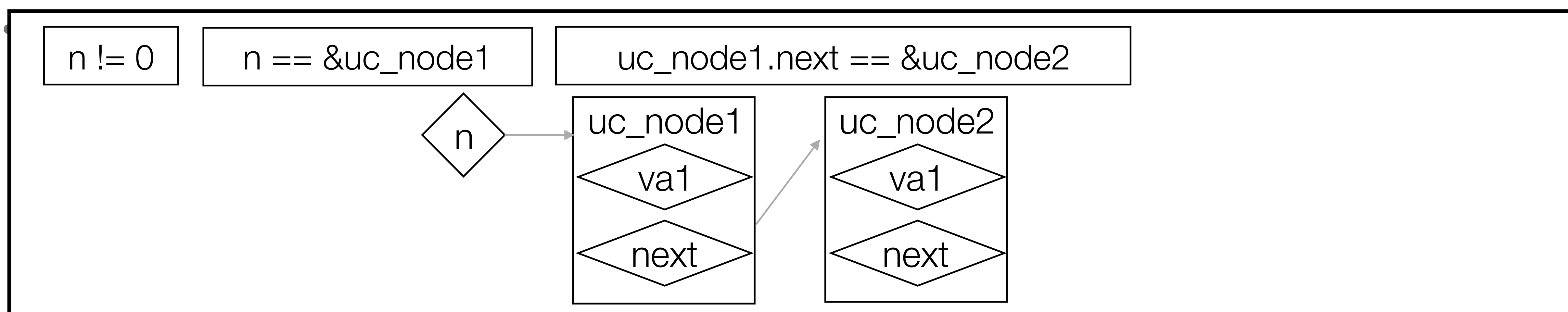


Example

```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
      sum += n->val;
      n = n->next;
   }
   return sum;
}
```

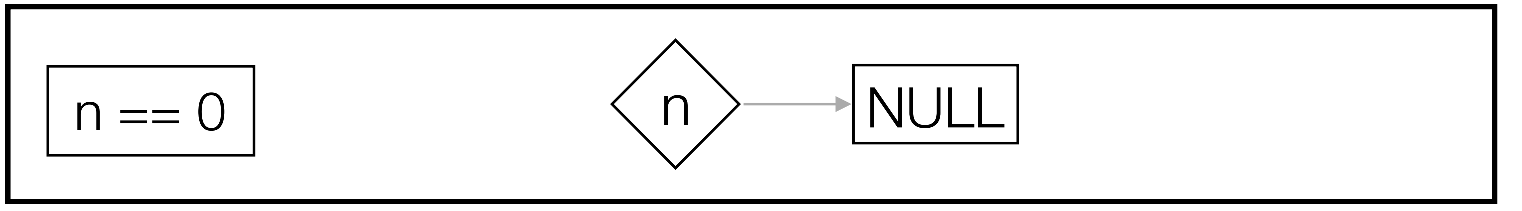


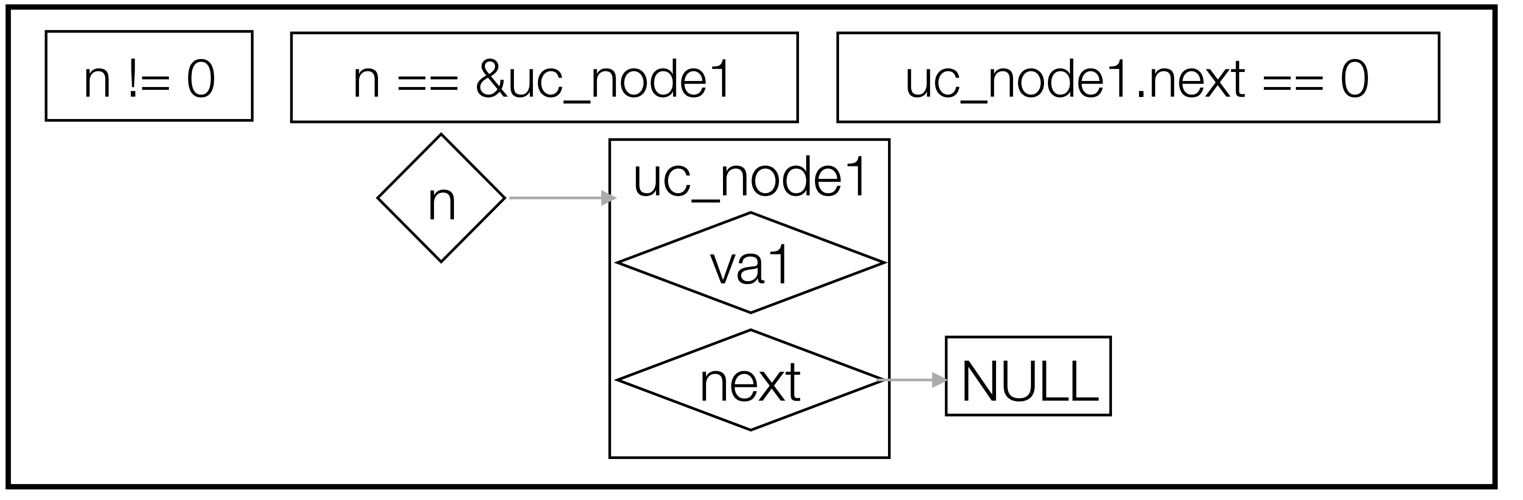


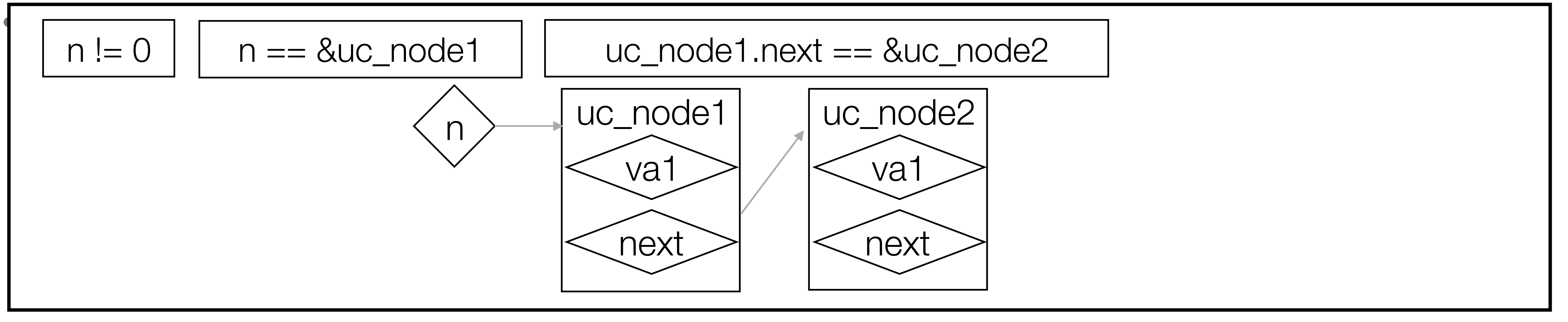


Example

```
int  listSum(node *n) {
  int  sum = 0;
  while (n) {
     sum += n->val;
     n = n->next;
  }
  return sum;
}
```

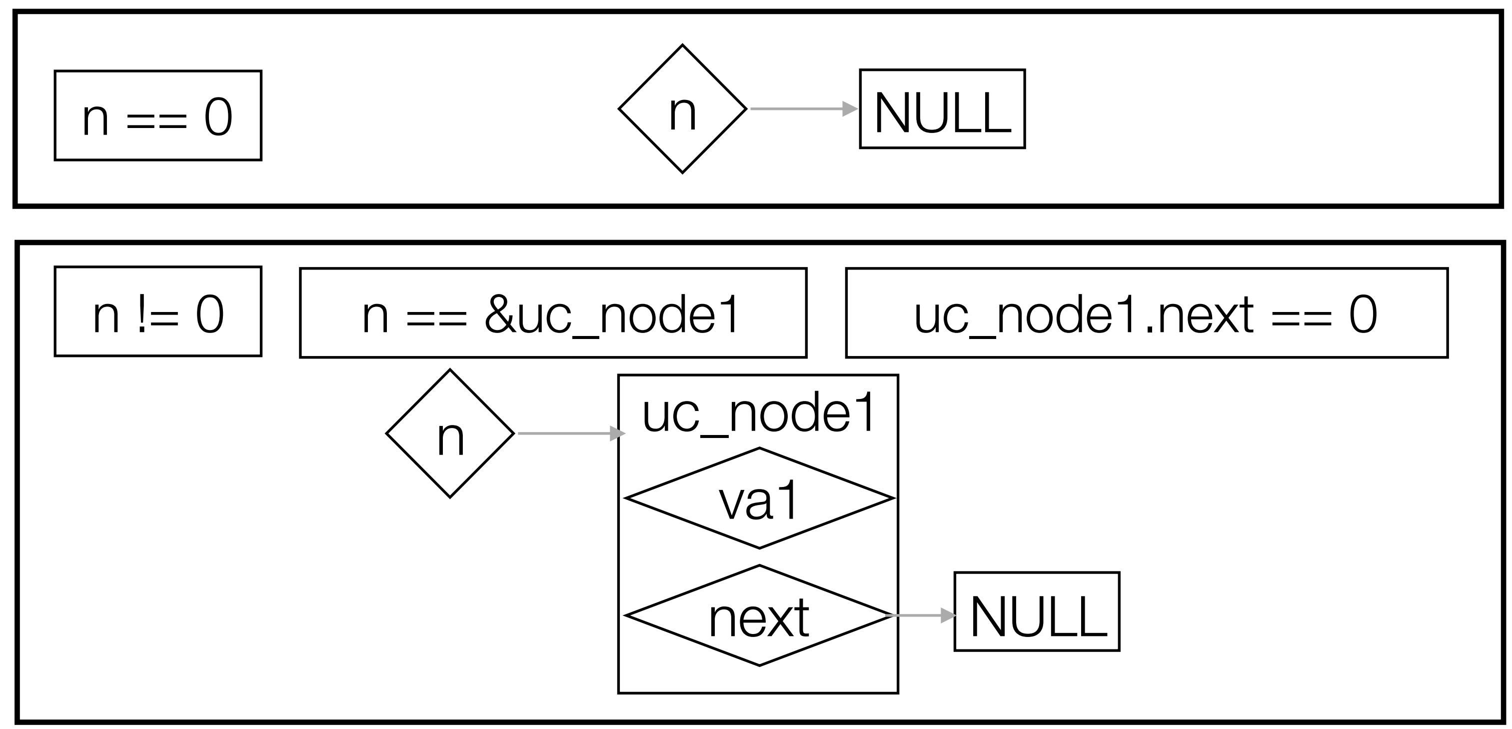


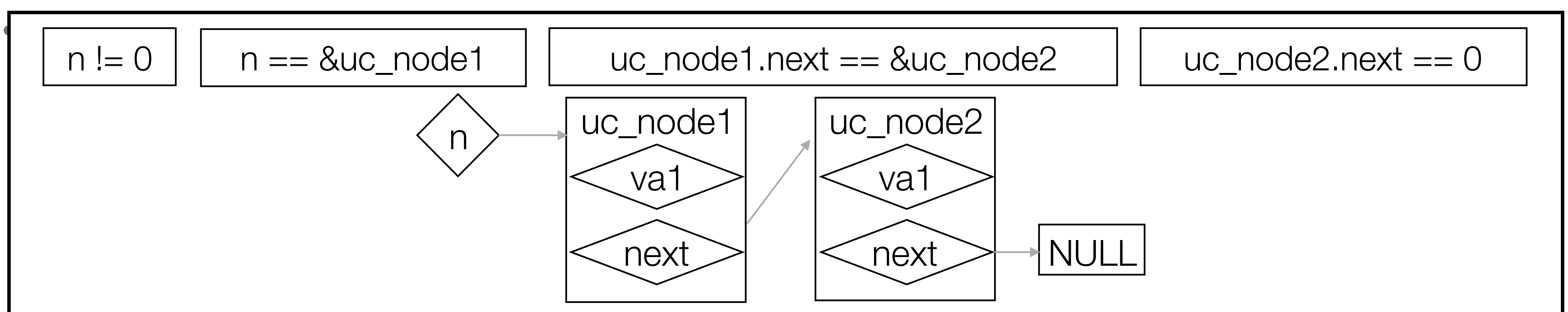




Example

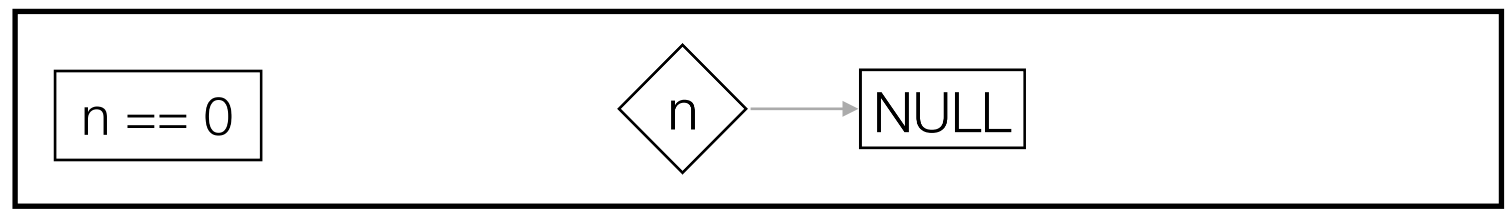
```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
      sum += n->val;
      n = n->next;
   }
   return sum;
}
```

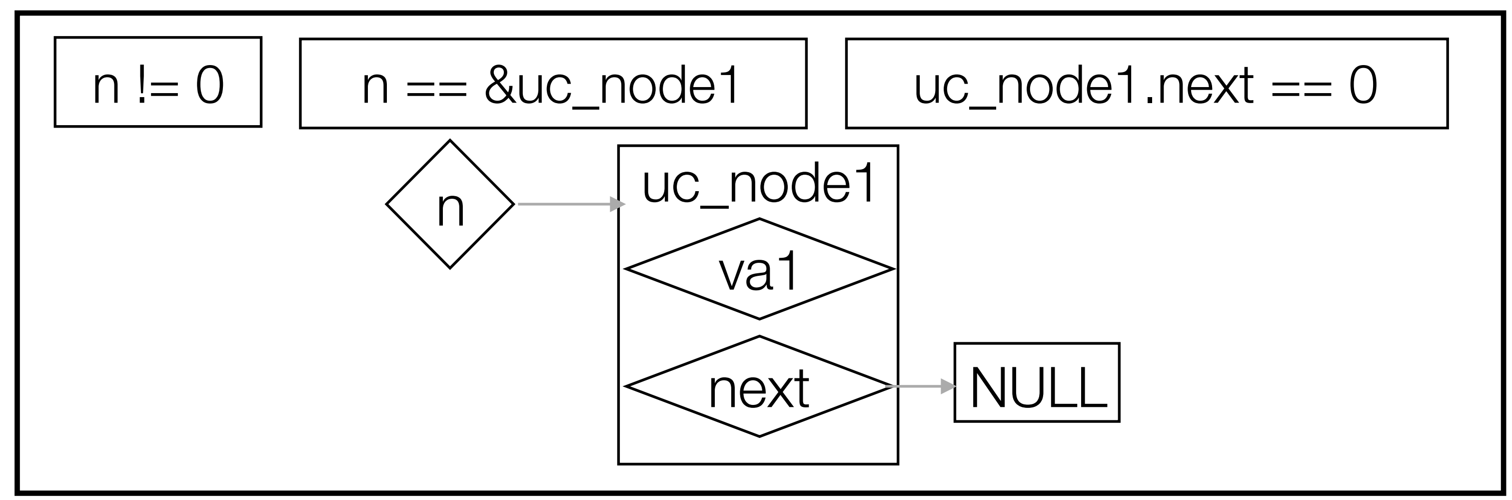


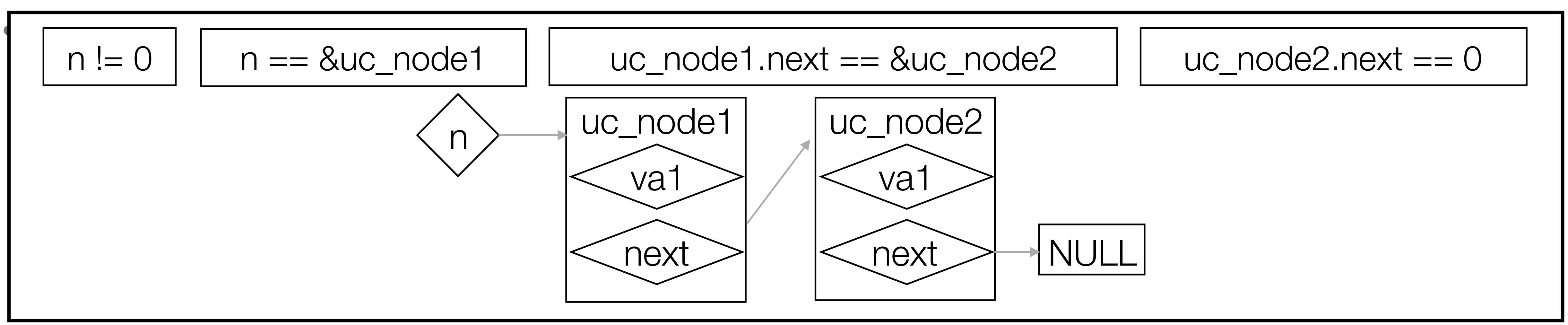


Example

```
int   listSum(node *n) {
   int  sum = 0;
   while (n) {
      sum += n->val;
      n = n->next;
   }
   return sum;
}
```







Use Cases

- Equivalence checking: patches
  - Yesterday’s code vs. today’s code (i.e., fewer bugs today)
  - Goal: detect (and prevent!) new crashes introduced by patches
  - Other uses discussed in CAV 2011 paper
- General bug-finding: rule-based checkers
  - Single version of a function; under-constrained + additional checker rules
  - Memory leaks, uninitialized data, unsafe user input
  - Simple interface for adding new checkers

Equivalence Checking

- Value equivalence
  - Return value
  - Arguments passed by reference
  - Global/static variables
  - System call effects (modeled)
- Error (crash) equivalence
  - Both versions typically have the same same (unknown) preconditions!
  - Neither version crashes on an input
  - Both versions crash on an input

USE CASE: whether patches introduce crashes

Equivalence Checking

- Check per path equivalence of two functions
- If all paths exhausted, equivalence verified (up to input bound)

Evaluation

- BIND, OpenSSL
  - Mature, security-critical codebases (~400 KLOC each)
- Patches
  - BIND: 487 patches to 9.9 stable (14 months)
  - OpenSSL: 324 patches to 1.0.1 stable (27 months)
- Ran UC-KLEE for 1 hour on each patched function

Evaluation: Patches

- Discovered 10 new bugs (4 in BIND, 6 in OpenSSL)
  - 2 OpenSSL DoS vulnerabilities:
    - CVE-2014-0198: NULL pointer dereference
    - CVE-2015-0292: Out-of-bounds memcpy read
- Verified (w/ caveats) that patches do not introduce crashes
  - 67 (13.8%) for BIND, 48 (14.8%) for OpenSSL

- More results available in the publication

Acknowledgments/References

- [Naik’18] IS 700: Software Analysis and Testing, Mayur Naik, Upenn Fall 2018.
- [Chowdhury’15] Information Security, CS 526, Omar Chowdhury, University of Iowa, 2015
- [Leibowitz’13] Presented by Yoni Leibowitz, EECS 395/495: Programming Languages and Analysis for Security , Northwestern University, 2013
- [Ramos’15] Under-Constrained Symbolic Execution: Correctness Checking for Real Code, David A. Ramos and Dawson Engler, Slidesm, Usenix Security 2015