SOFTWARE RELIABILITY GROUP

Imperial College London

A Segmented Memory Model for Symbolic Execution

Timotej Kapus Cristian Cadar
Imperial College London

Symbolic Execution

- Program analysis technique
- Active research area
- Used in industry
  - IntelliTest, SAGE
  - KLOVER

Angr

Why symbolic execution?

- *No false-positives!*
  - Every bug found has a concrete input triggering it
- Can interact with the environment
  - I/O, unmodeled libraries
- Only relevant code executed "symbolically", the rest is fast "native" execution

Why (not) symbolic execution?

- Scalability, scalability, scalability
    - Constraint solving is hard
    - Path explosion

This talk

Show a <u>segmented</u> memory model that tackles <u>path explosion</u> due to dereferences of <u>symbolic pointers</u> through the use of static pointer alias analysis

1D symbolic pointers

```c
int i;
make_symbolic(i);
int vector[10] = {1,2,3,4,5,6,7,8,9,10};

if (vector[i] > 8)
   printf("big element\n");
else
   printf("small element");
```

1D Symbolic pointers

```
i = symbolic
```

```
int i;
make_symbolic(i);
int vector[10] = {1,2,3,4,5,6,7,8,9,10};

if(vector[i] > 8)
    printf("big element\n");
else
    printf("small element");
```

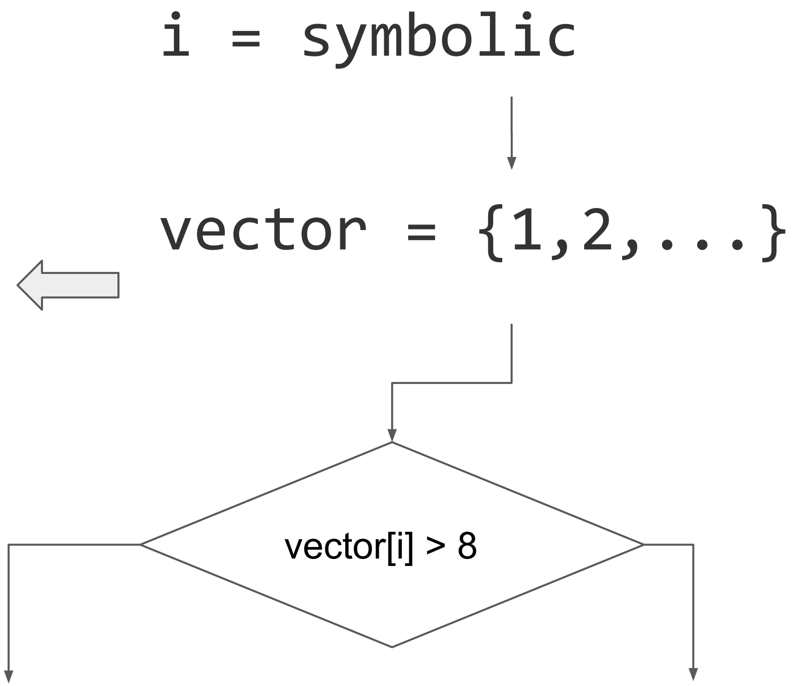
1D Symbolic pointers

```c
int i;
make_symbolic(i);
int vector[10] = {1,2,3,4,5,6,7,8,9,10};

if(vector[i] > 8)
    printf("big element\n");
else
    printf("small element");
```

```
i = symbolic

vector = {1,2,...}
```

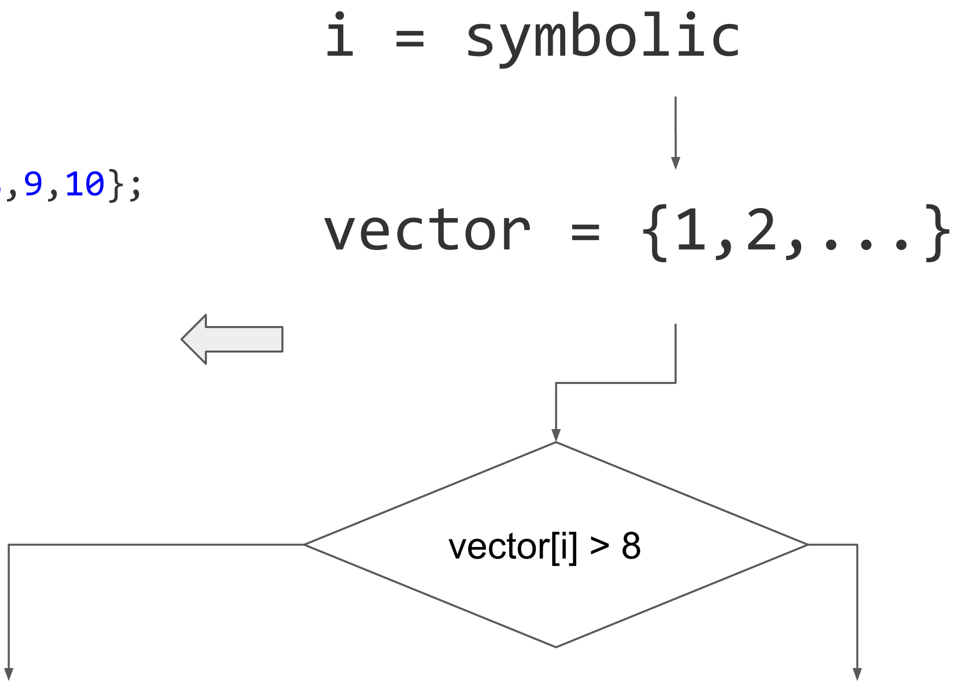
1D Symbolic pointers

```c
int i;
make_symbolic(i);
int vector[10] = {1,2,3,4,5,6,7,8,9,10};

if(vector[i] > 8)
    printf("big element\n");
else
    printf("small element");
```



1D Symbolic pointers

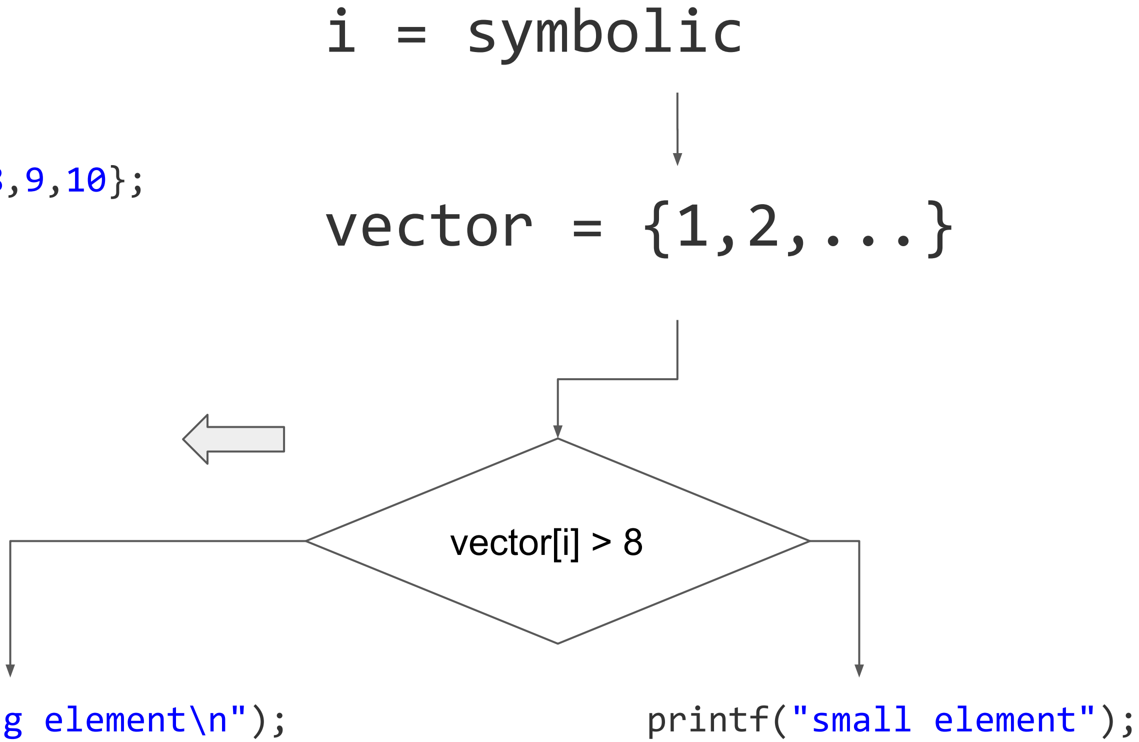
```c
int i;
make_symbolic(i);
int vector[10] = {1,2,3,4,5,6,7,8,9,10};

if(vector[i] > 8)
     printf("big element\n");
else
     printf("small element");
```



printf("big element\n");

1D Symbolic pointers

```c
int i;
make_symbolic(i);
int vector[10] = {1,2,3,4,5,6,7,8,9,10};

if(vector[i] > 8)
    printf("big element\n");
else
    printf("small element");
```

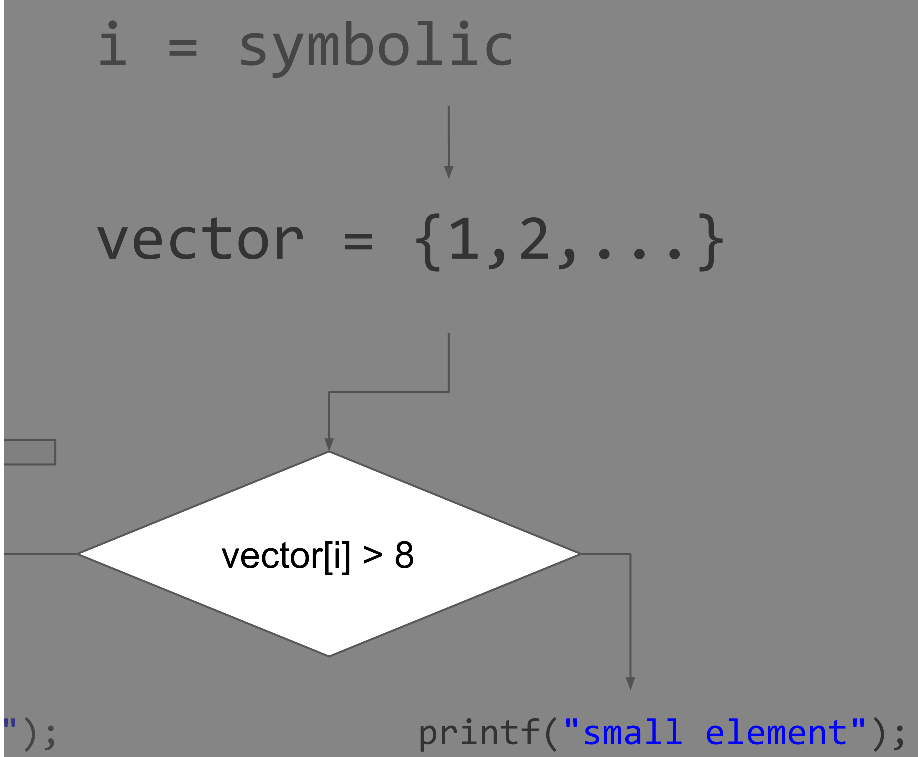


1D Symbolic pointers

- `vector[i]` is a dereference of a symbolic pointer
  - Concrete base address
  - Some symbolic offset `i`

- I.e. if vector is at `0xdeedbeef` `vector[i]` is a

```
load (0xdeedbeef + i)
```



Constraints over memory

- Theory of arrays:
  - `read: array × index → value`
  - `write: array × index × value → array`
  - `read(write(a, p, v), r) = v` **if** p = r
  - `read(write(a, p, v), r) = read(a,r)` **if** p ≠ r

- Simply map C arrays to solver arrays
- Use concrete addresses to resolve C arrays to solver arrays

1D Symbolic pointers: constraints in theory of arrays

```
int i;
make_symbolic(i);
int vector[10] =
{1,2,3,4,5,6,7,8,9,10};


if (vector[i] > 8)
    printf("big element");
else
    printf("small element");
```

i = symbolic

1D Symbolic pointers: constraints in theory of arrays

```
int i;
make_symbolic(i);
int vector[10] =
{1,2,3,4,5,6,7,8,9,10};


if (vector[i] > 8)
    printf("big element");
else
    printf("small element");
```

i = symbolic

array *vector*[10] = [1 2 3 4 5 6 7 8 9 10]

1D Symbolic pointers: constraints in theory of arrays

```
int i;
make_symbolic(i);
int vector[10] =
{1,2,3,4,5,6,7,8,9,10};


if (vector[i] > 8)
    printf("big element");
else
    printf("small element");
```

i = symbolic

array *vector*[10] = [1 2 3 4 5 6 7 8 9 10]

(Read *i vector*)

2D Symbolic pointers

```c
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, sizeof(int));

matrix[1][2] = 42;

if (matrix[i][j] > 8) printf("big element\n");
else printf("zero");
```

2D Symbolic pointers: constraints in theory of arrays

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = symbolic
j = symbolic

2D Symbolic pointers: constraints in theory of arrays

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = symbolic
j = symbolic

array *matrix*[3] = [0xdeedbeef 0xdeedbef0 0xdeedbef1]

2D Symbolic pointers: constraints in theory of arrays

```c
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = symbolic
j = symbolic

array *matrix*[3] = [0xdeedbeef 0xdeedbef0 0xdeedbef1]

array matrix_0[3] = [0 0 0]
array *matrix_1*[3] = [0 0 42]
array *matrix_2*[3] = [0 0 0]

2D Symbolic pointers: constraints in theory of arrays

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    pr
```

(Read *i matrix*)

i = symbolic
j = symbolic

array *matrix*[3] = [0xdeedbeef 0xdeedbef0 0xdeedbef1]

array *matrix_0*[3] = [0 0 0]
array *matrix_1*[3] = [0 0 42]
array *matrix_2*[3] = [0 0 0]

2D Symbolic pointers: constraints in theory of arrays

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    pr
```

i = symbolic
j = symbolic

array *matrix*[3] = [0xdeedbeef 0xdeedbef0 0xdeedbef1]

array *matrix_0*[3] = [0 0 0]
array *matrix_1*[3] = [0 0 42]
array *matrix_2*[3] = [0 0 0]

(Read *j* (Read *i matrix*))

2D Symbolic pointers: constraints in theory of arrays

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    pr
```

i = symbolic
j = symbolic

array *matrix*[3] = [0xdeedbeef 0xdeedbef0 0xdeedbef1]

array *matrix_0*[3] = [0 0 0]
array *matrix_1*[3] = [0 0 42]
array *matrix_2*[3] = [0 0 0]

(Read *j* 0xdeedbeef)

2D Symbolic pointers: constraints in theory of arrays

```
int i, j;
make_symbolic(i, j);
int *matrix[
for (int k =
  matrix[i]

matrix[1][2]

if (matrix[i
    printf("big element\n");
else
    pr
```

i = symbolic
j = symbolic

array *matrix*[3] = [0xdeedbeef 0xdeedbef0 0xdeedbef1]

array *matrix_0*[3] = [0 0 0]
array *matrix_1*[3] = [0 0 42]
array *matrix_2*[3] = [0 0 0]

(Read *j* 0xdeedbeef)

So what now?

- Forking (KLEE)
  - Concretize and fork for each possible value of `matrix[i]`
- State Merging / OR Expression (SAGE)
  - Create a disjunction over all possible values of `matrix[i]`
- Flat Memory (considered by EXE, not implemented)
  - Have the whole memory as a single array

2D Symbolic pointers: Forking

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = 0
j = symbolic

2D Symbolic pointers: Forking

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = 0
j = symbolic

array *matrix_0*[3] = [0 0 0]

2D Symbolic pointers: Forking

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    p
```

(Read *j matrix_0*)

i = 0
j = symbolic

array *matrix_0*[3] = [0 0 0]

2D Symbolic pointers: Forking

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)

else
```

i = 2
j = symbolic

array *matrix_2*[3] = [0 0 0]

(Read j *matrix_2*)

Path explosion

2D Symbolic pointers: State Merging

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

$i = 0 \lor 1 \lor 2$
j = symbolic

2D Symbolic pointers: State Merging

```c
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

$i = 0 \vee 1 \vee 2$
j = symbolic

array *matrix_0*[3] = [0 0 0]
array *matrix_1*[3] = [0 0 42]
array *matrix_2*[3] = [0 0 0]

2D Symbolic pointers: State Merging

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
```

$i = 0 \lor 1 \lor 2$

j = symbolic

array *matrix_0*[3] = [0 0 0]

array *matrix_1*[3] = [0 0 42]

array *matrix_2*[3] = [0 0 0]

(Read *j matrix_0*) $\lor$ (Read *j matrix_1*) $\lor$ (Read *j matrix_2*)

OR expressions are hard(-er) to solve

2D Symbolic pointers: Flat memory

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = symbolic
j = symbolic

2D Symbolic pointers: Flat memory

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = symbolic
j = symbolic

array *memory*[12] = [
3 6 9
0 0 0
0 0 42
0 0 0]

2D Symbolic pointers: Flat memory

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = symbolic
j = symbolic

array *memory*[12] = [

3	6	9
0	0	0
0	0	42
0	0	0]

Note that `calloc` return `3,6,9` as the addresses of the rows now

2D Symbolic pointers: Flat memory

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
   matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
e
```

(Read (3**i* + *j* + 3) *memory*)

i = symbolic
j = symbolic

array *memory*[12] = [

3	6	9
0	0	0
0	0	42
0	0	0]

Unnecessarily large array

Our approach

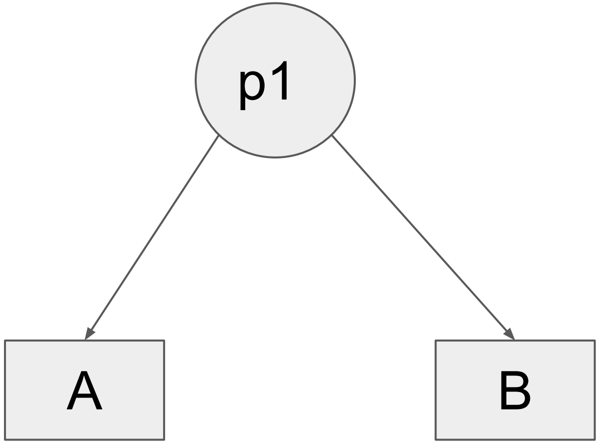
- Use static pointer alias analysis
- Partition memory objects into *segments*
  - *Each pointer only points to a single segment*
- Assign segments to solver arrays

Our approach: partitioning into segments

```
pts(p1) = {A, B}
```

Our approach: partitioning into segments

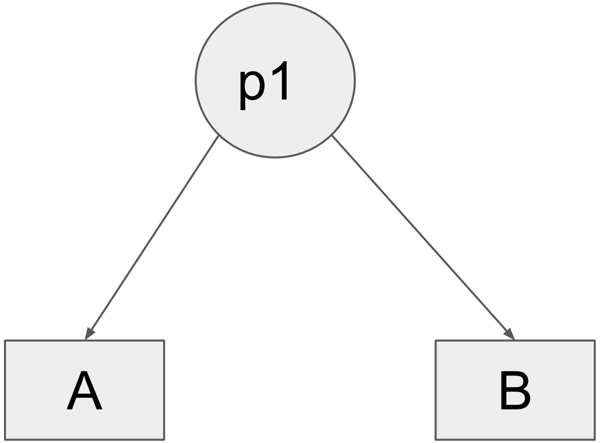
```
pts(p1) = {A, B}
```



Our approach: partitioning into segments

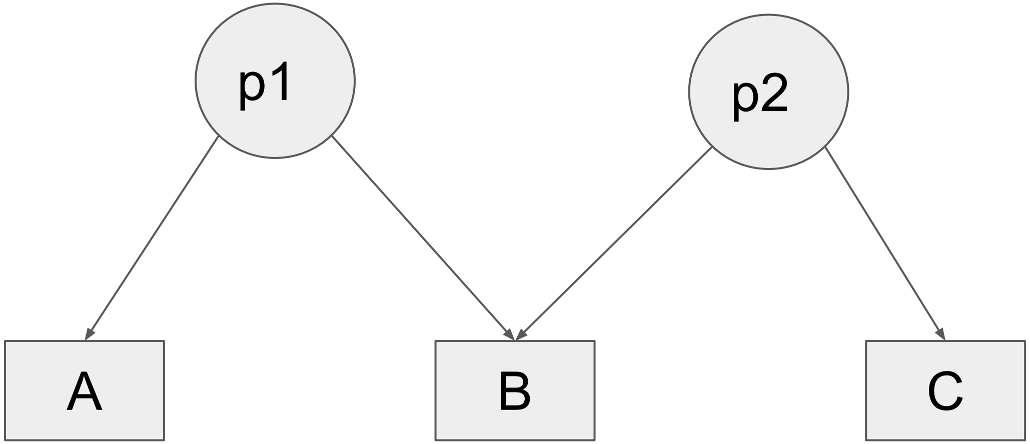
`pts(p1) = {A, B}`

`pts(p2) = {B, C}`



Our approach: partitioning into segments

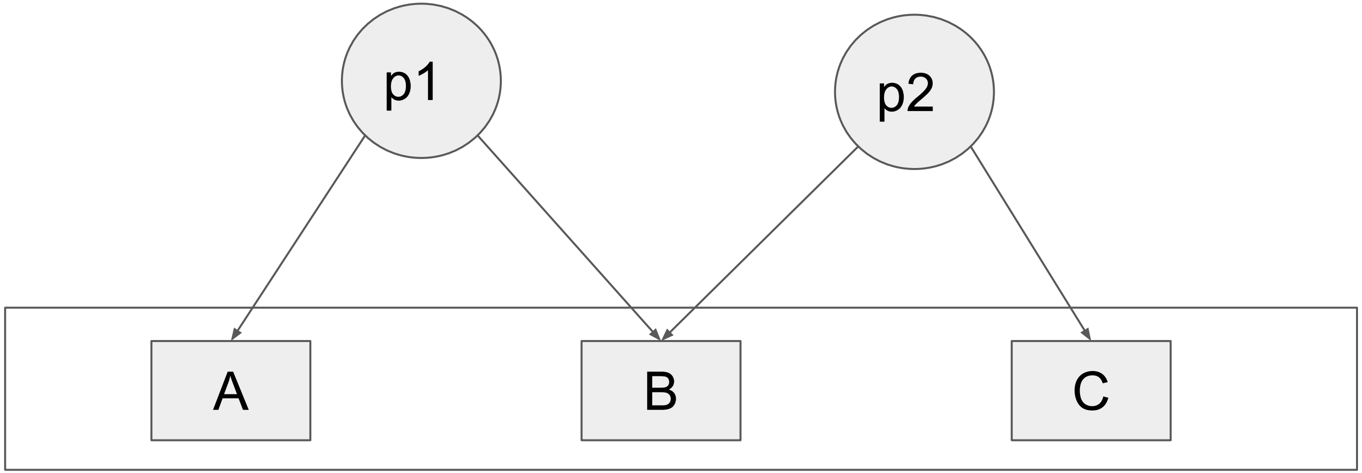
```
pts(p1) = {A, B}                pts(p2) = {B, C}
```



Our approach: partitioning into segments

`pts(p1) = {A, B}`

`pts(p2) = {B, C}`



2D Symbolic pointers: Segmented Memory

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = symbolic
j = symbolic

2D Symbolic pointers: Segmented Memory

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

i = symbolic
j = symbolic

array *segment_0*[3]
= [0xdeedbef0 0xdeedbef3 0xdeedbef6]
array *segment_1*[9]
= [0 0 0
0 0 42
0 0 0]

2D Symbolic pointers: Segmented Memory

```
int i, j;
make_symbolic(i, j);
int *matrix[3];
for (int k = 0; k < 3; k++)
  matrix[i] = calloc(3, 4);

matrix[1][2] = 42;

if (matrix[i][j] > 8)
    p
else
    p
```

(Read (3**i* + *j*) *segment_1*)

i = symbolic
j = symbolic

array *segment_0*[3]
= [0xdeedbef0 0xdeedbef3 0xdeedbef6]
array *segment_1*[9]
= [0 0 0
0 0 42
0 0 0]

Results

- Based on an implementation in KLEE
- Synthetic benchmarks
  - Based on the matrix example
  - Time it takes symbolic execution to explore all paths
  - Increase N - the dimensionality of the matrix
- Real programs

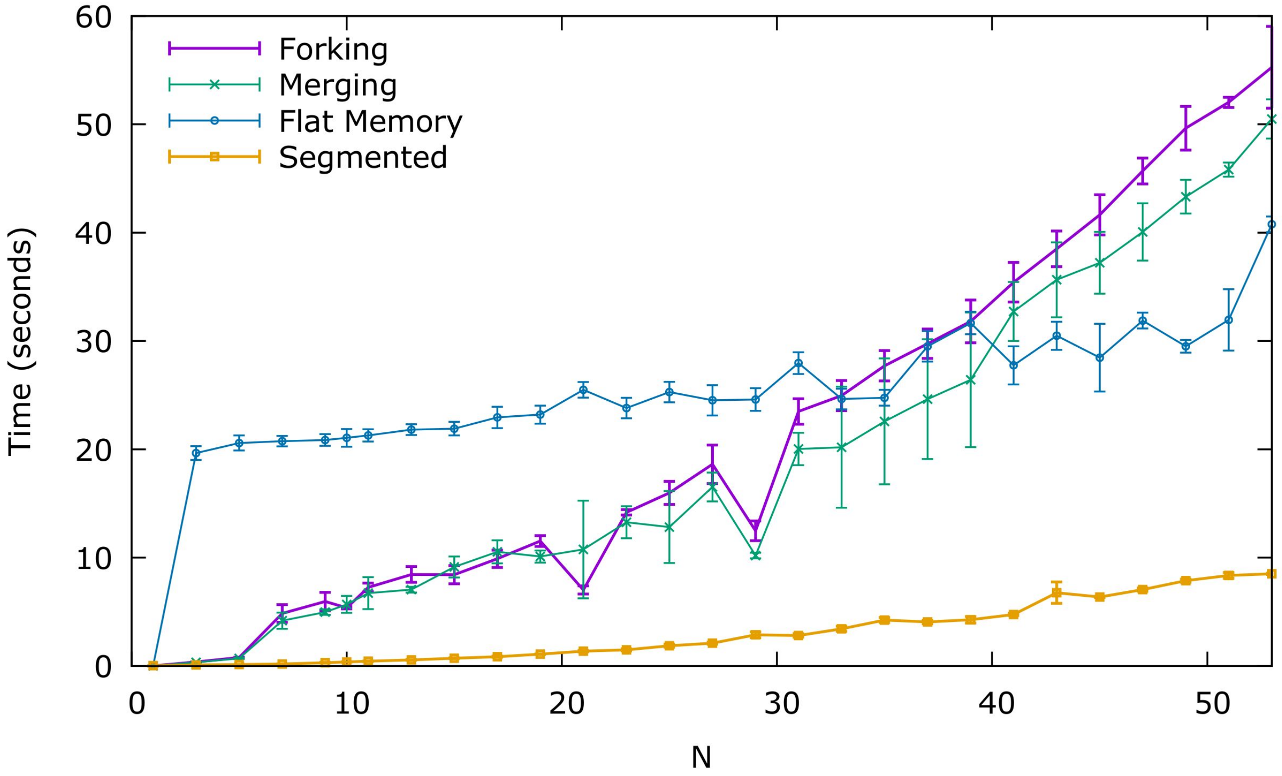
make

m4

NxN matrix: single lookup extra allocation

```
int i, j;
make_symbolic(i, j);
int *matrix[N];
for (int k = 0; k < N; k++)
  matrix[i] = calloc(N, sizeof(int));

matrix[1][2] = 42;
malloc(30000); //extra allocation

if (matrix[i][j] > 8) printf("big element\n");
else printf("zero");
```

NxN matrix: single lookup extra allocation



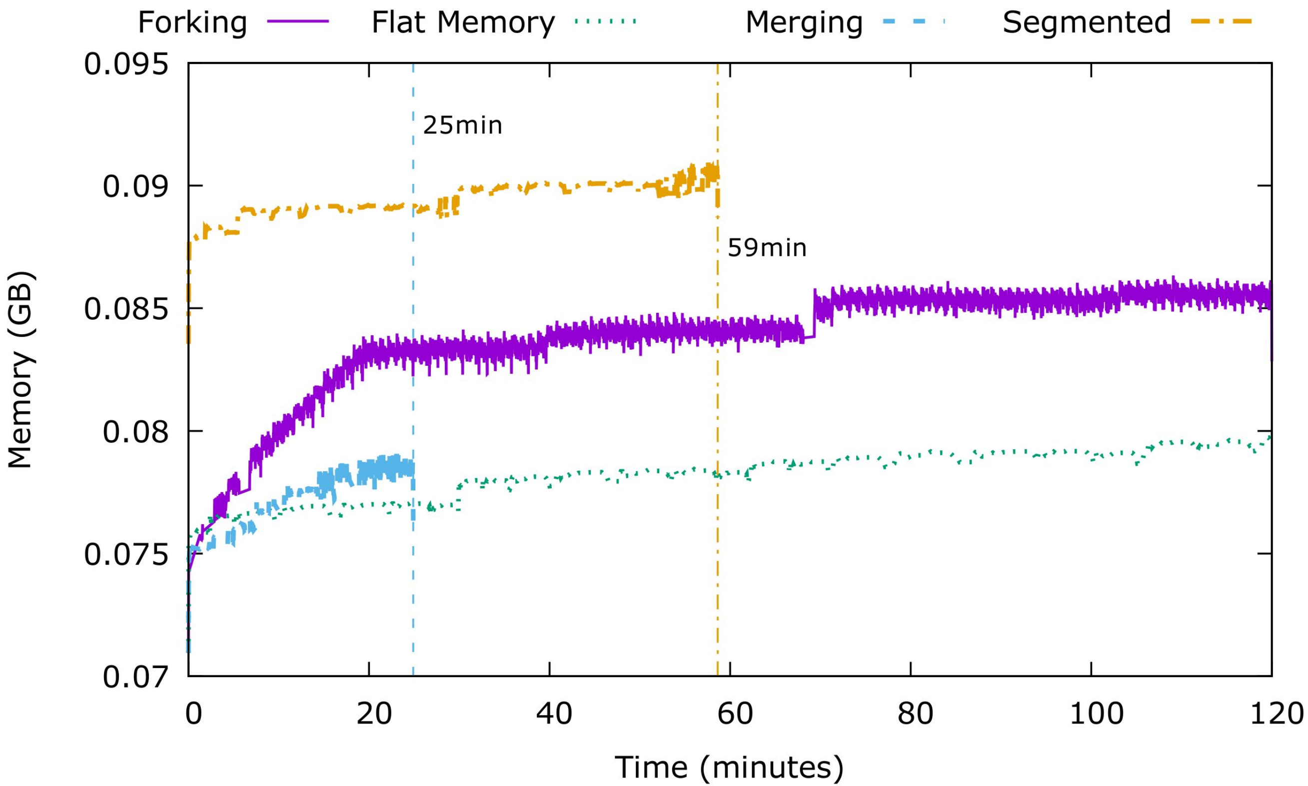
Real programs experiment setup

- We first look at cases that benefit from segmented memory model
  - Hash tables
  - Deep in the search space
- Targeted input files
- 2 hour timeout
- DFS, BFS, default

```
define(`A', `l')
define(`P', 2)
?
?
ifelse(?, P, eval(1 + P))
```

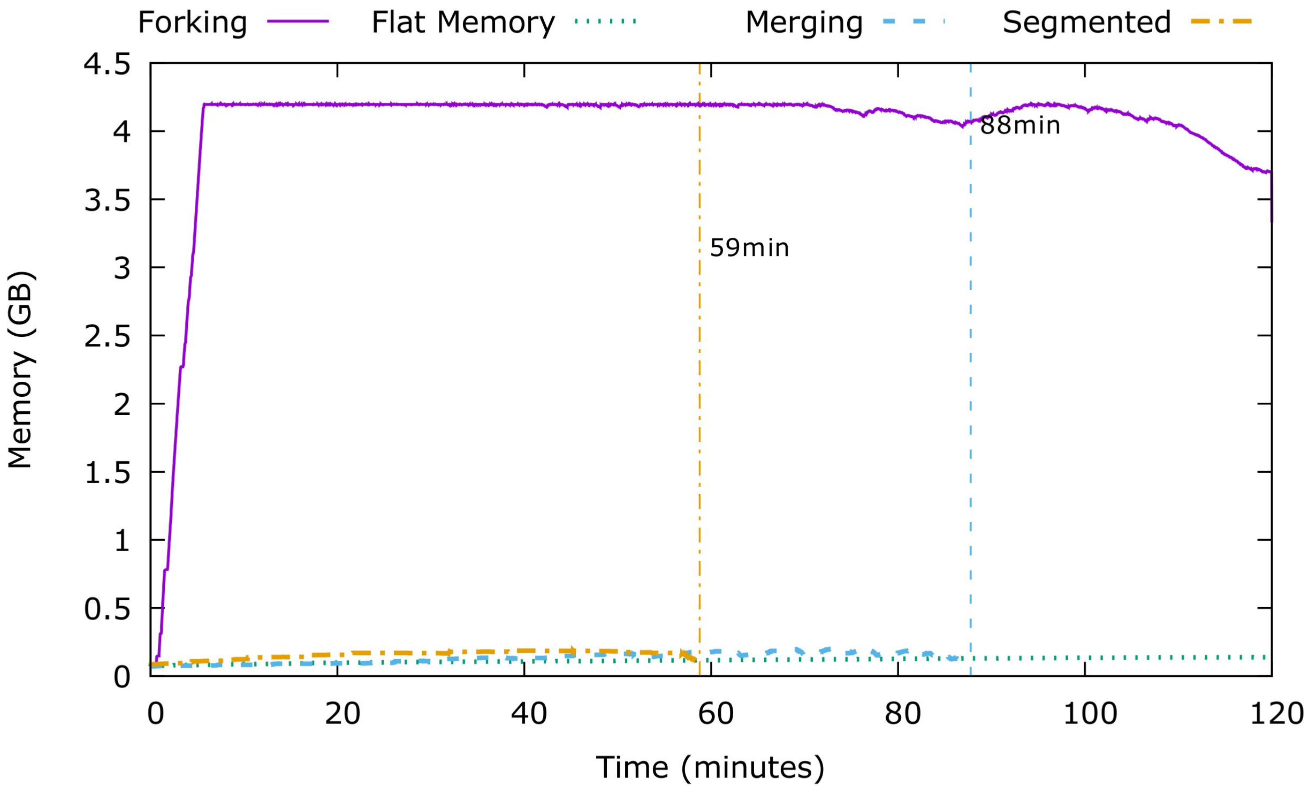
Targeted input file for `m4`

m4 DFS



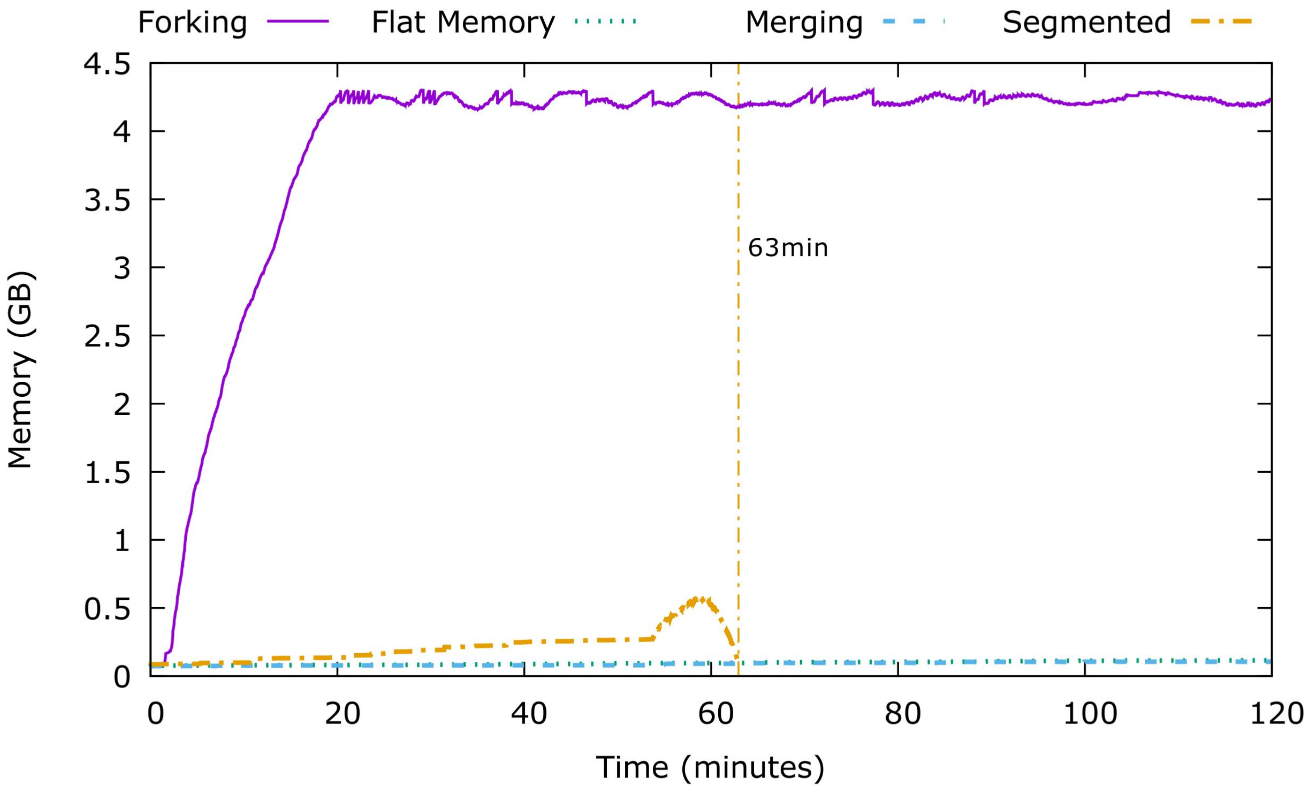
Forking
Flat Memory
Merging
Segmented
0.095
0.09
0.085
0.08
0.075
0.07
Memory (GB)
25min
59min
0
20
40
60
80
100
120
Time (minutes)

m4 BFS



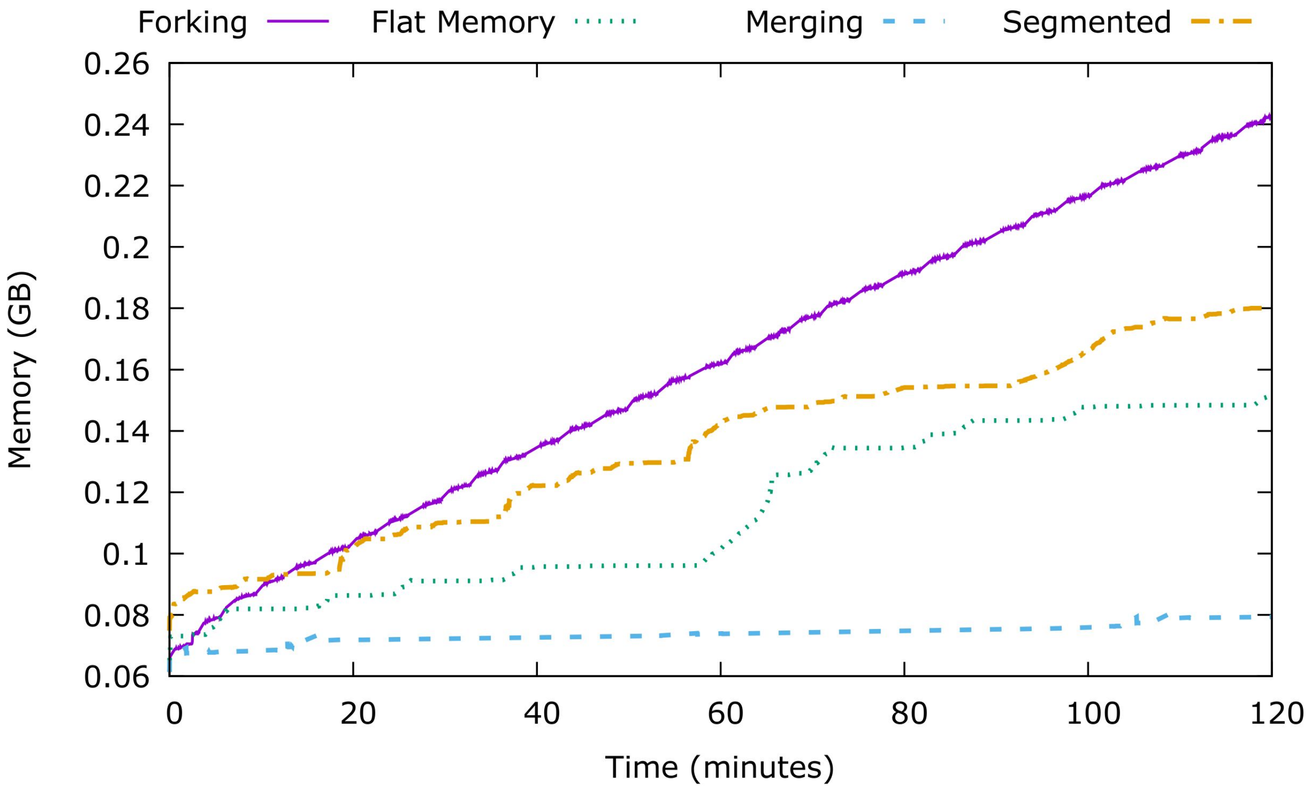
Forking
Flat Memory
Merging
Segmented
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
Memory (GB)
0
20
40
60
80
100
120
Time (minutes)
59min
88min

m4 default



Forking
Flat Memory
Merging
Segmented
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
Memory (GB)
63min
0
20
40
60
80
100
120
Time (minutes)

make DFS



Forking
Flat Memory
Merging
Segmented
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
Memory (GB)
0
20
40
60
80
100
120
Time (minutes)

Segmented memory model without symbolic dereferences

- 105 coreutils
  - No symbolic dereferences
- 1 hour run with DFS and forking model
- Segmented memory model:
  - 18 coreutils timed out in 1h 20min
  - Remaining coreutils on average 4% slower
- We envision using this after running the forking model

Conclusion

- Symbolic pointers are a hard problem
  - 3 existing options: forking, flat memory, merging
- Novel approach: Segmented memory model
  - Builds on flat memory model
  - Uses pointer alias analysis
  - Faster on programs with symbolic pointer dereferences

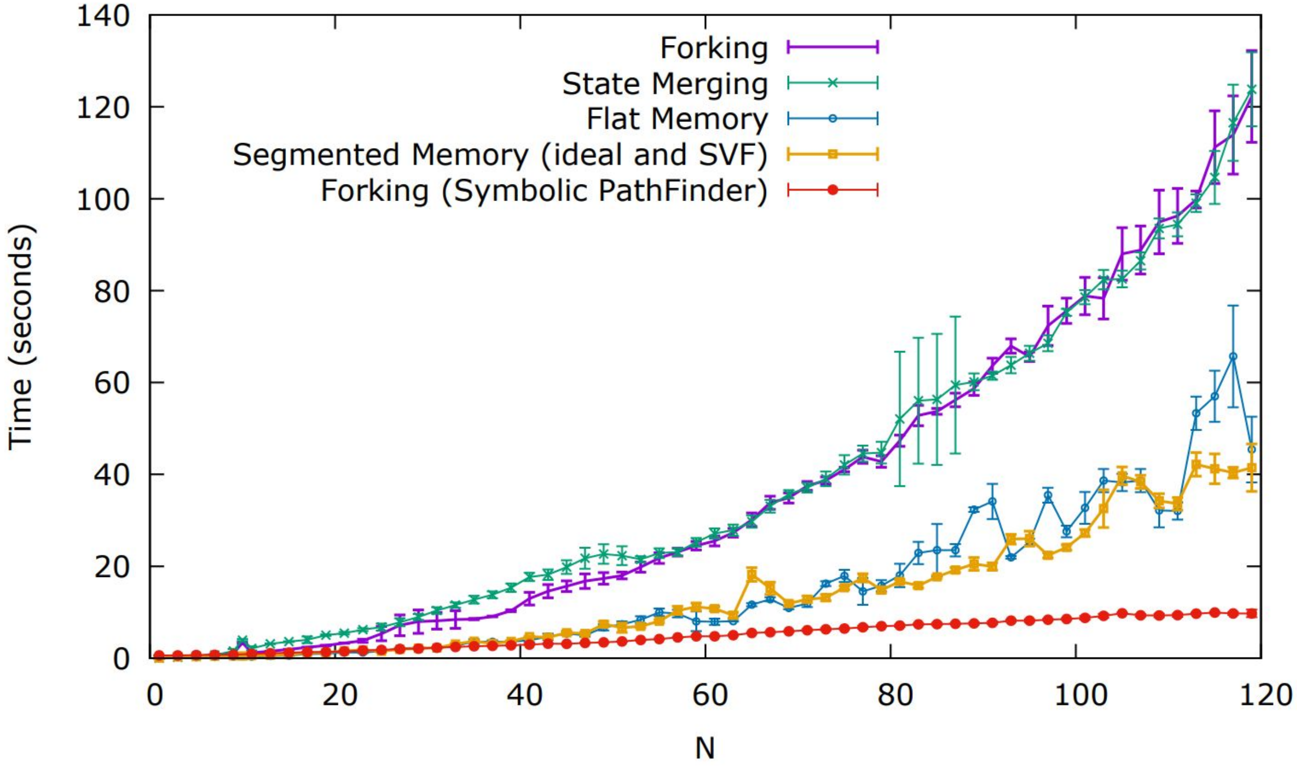
Interested? Looking for a Postdoc?

c.cadar@imperial.ac.uk

srg.doc.ic.ac.uk/vacancies/

SOFTWARE RELIABILITY
GROUP

NxN matrix: single lookup



Symbolic execution example: `get_sign`

```
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```

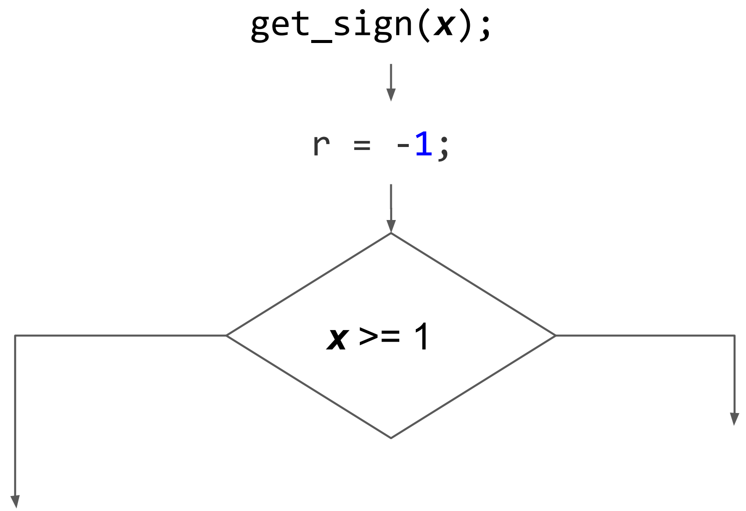
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```

get_sign(***x***);

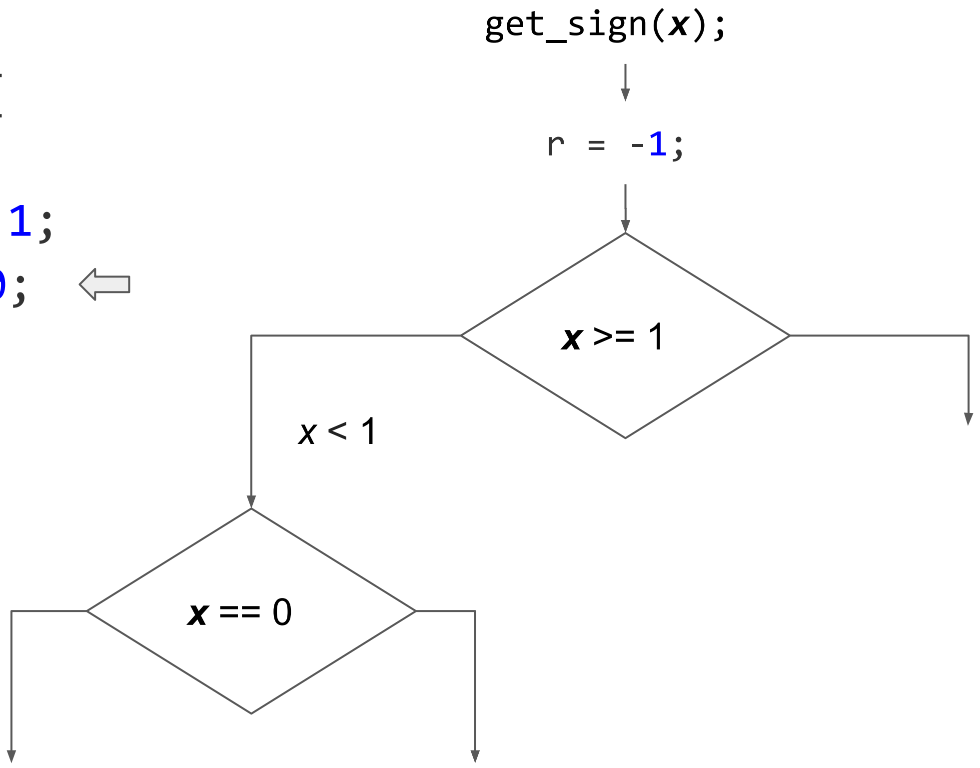
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```

```
get_sign(x);
     ↓
  r = -1;
     ↓
```

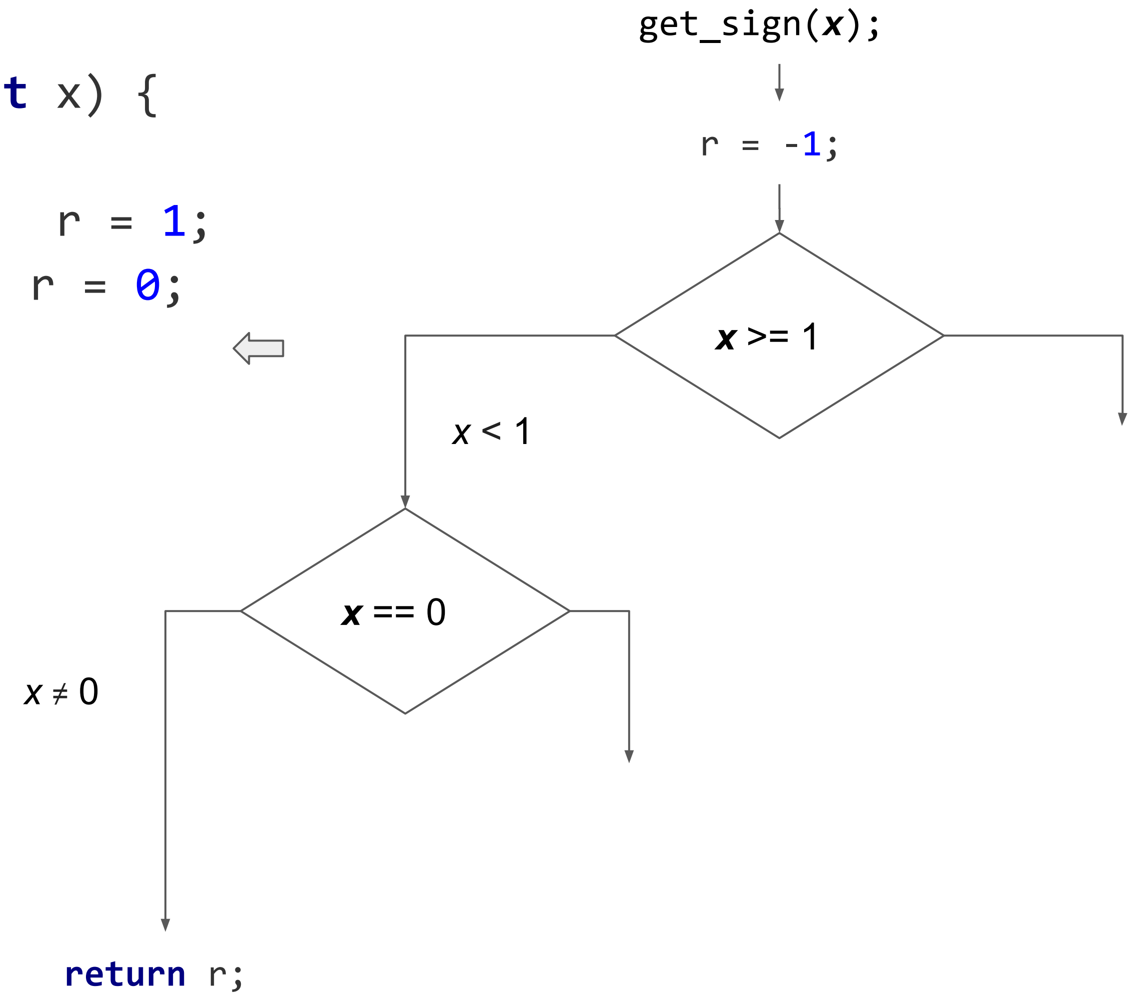
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```



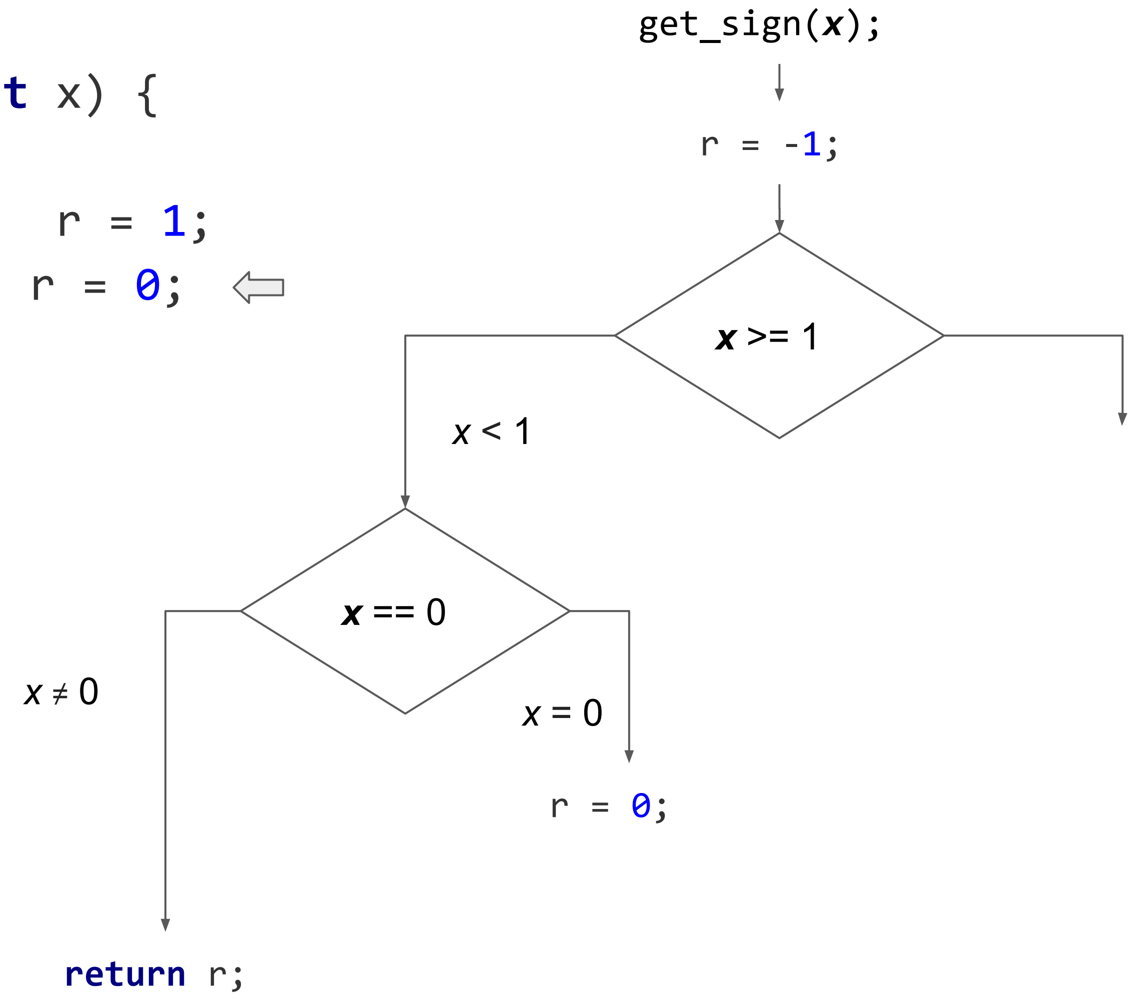
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```



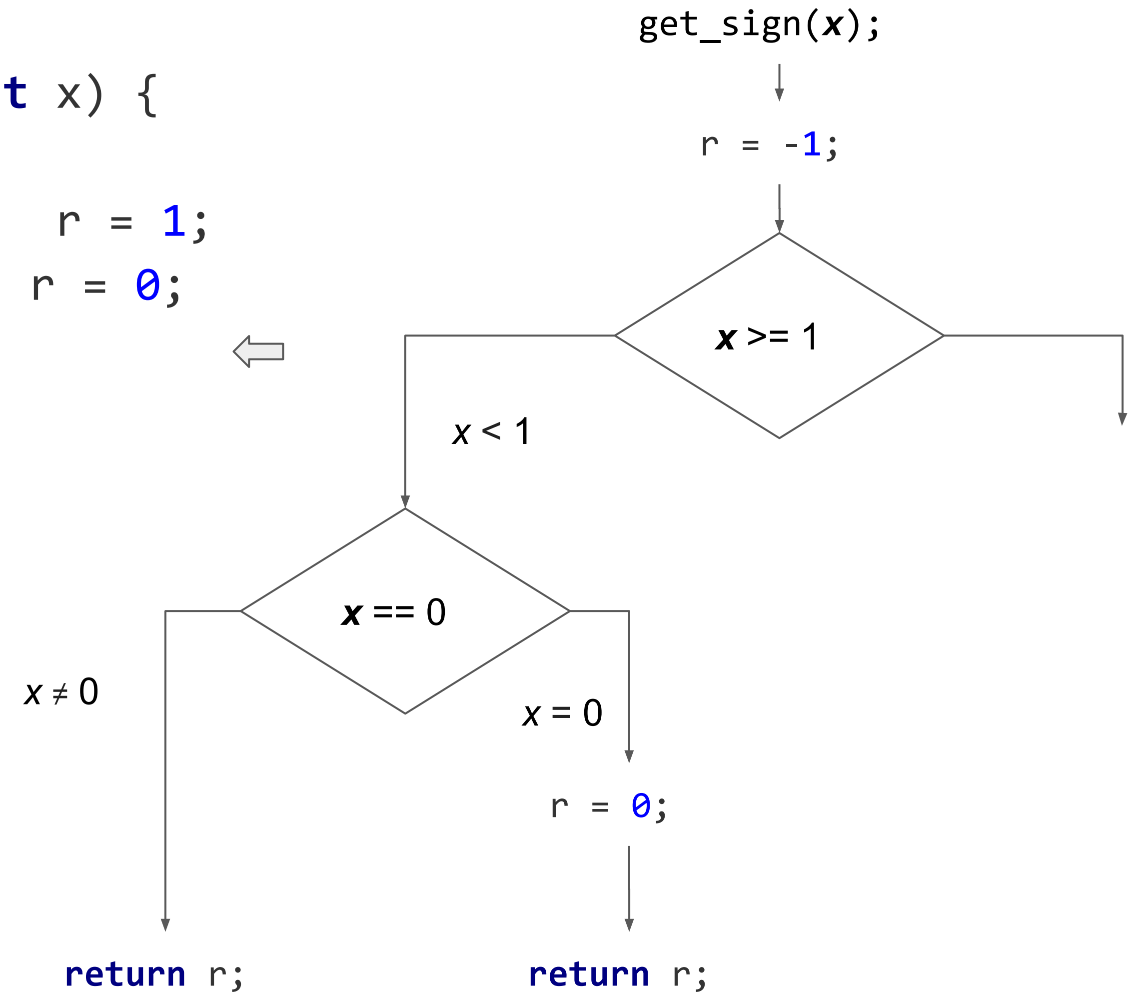
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```



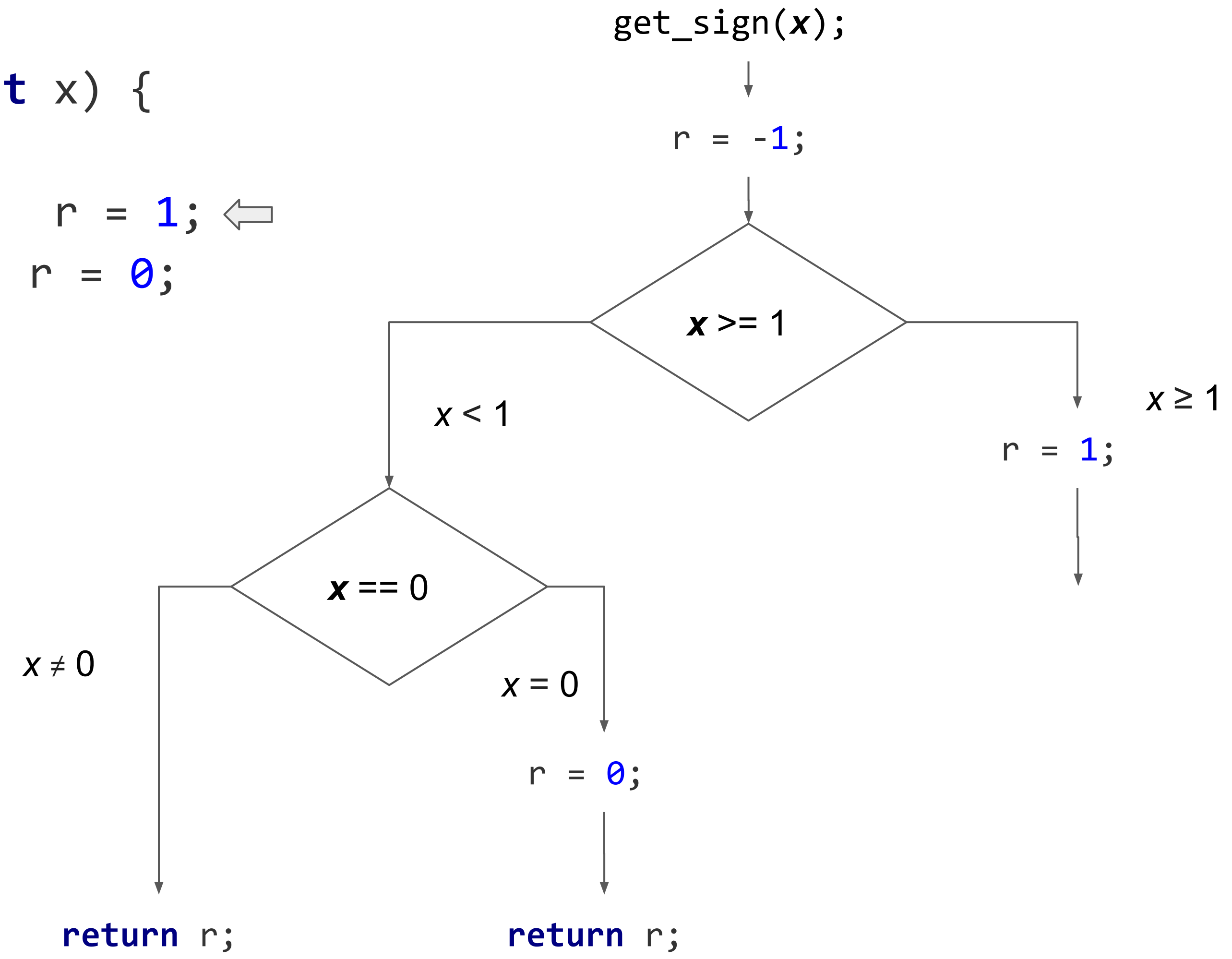
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```



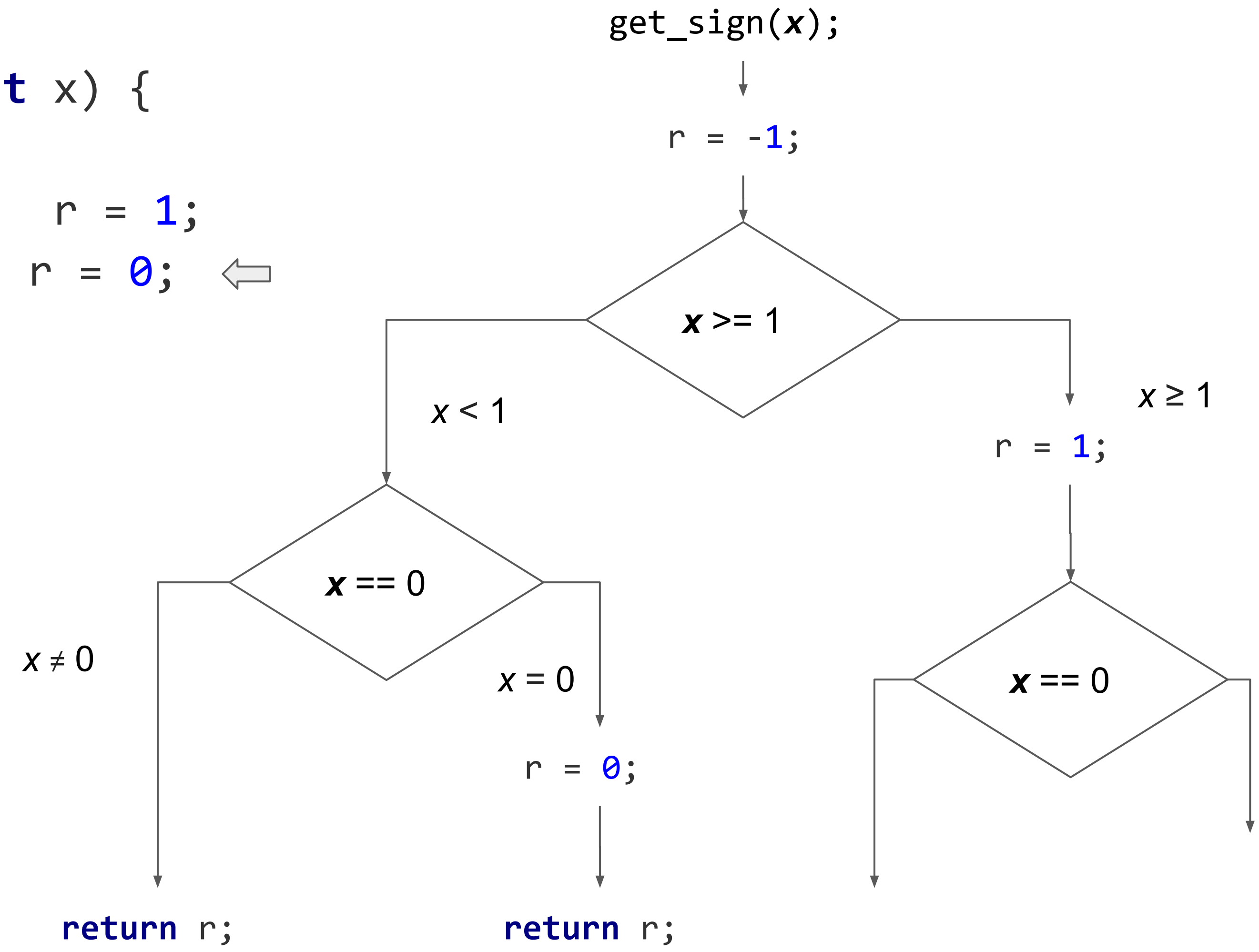
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```



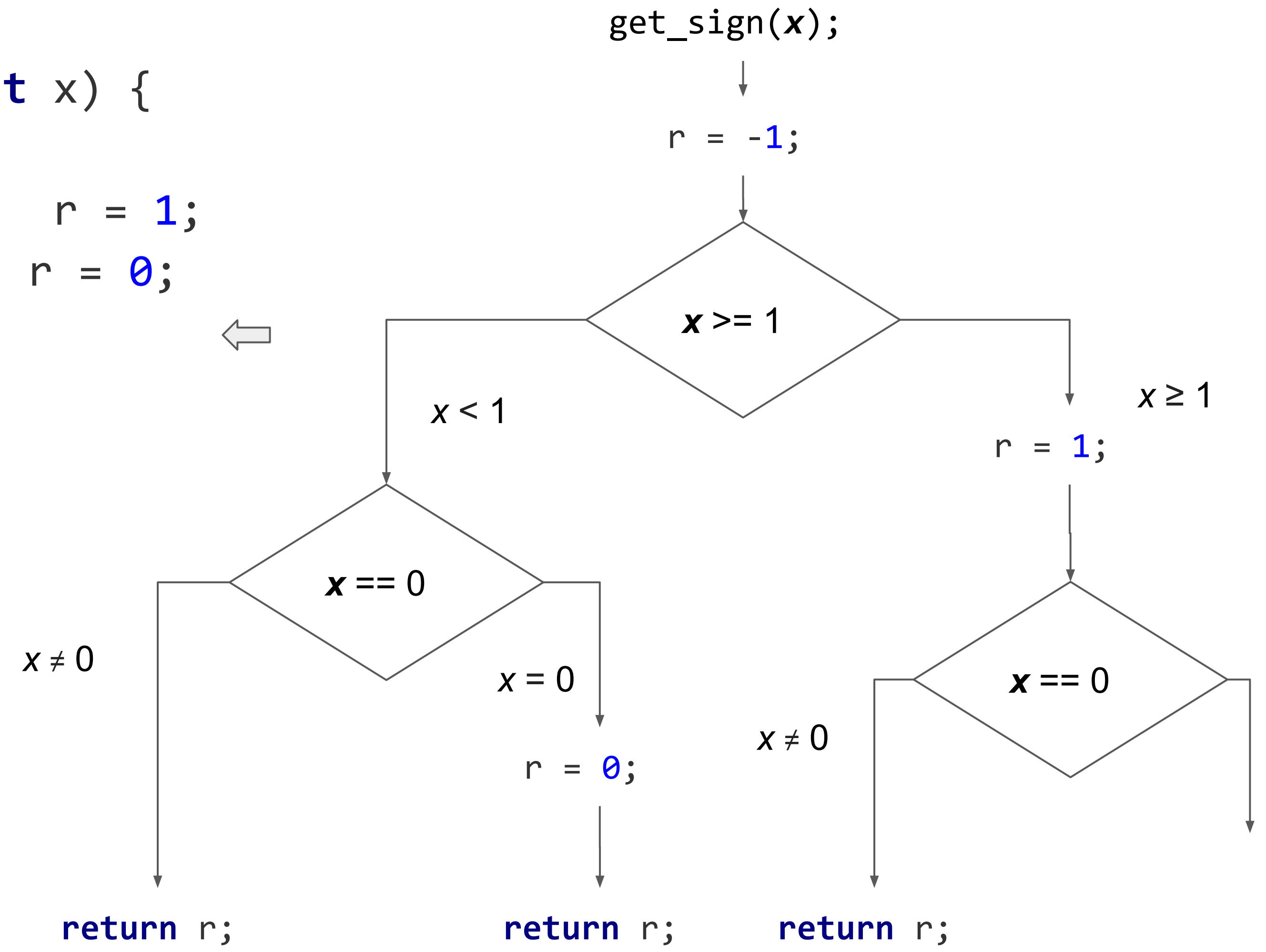
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```



```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```



```
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```



```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1)  r = 1;
    if (x == 0) r = 0;
    return r;
}
```