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
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 segmented memory model that tackles path explosion due to dereferences of symbolic pointers 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

```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
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");
```

```c
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");
```

```
i = symbolic
vector = \{1, 2, ...\}
```

```
vector[i] > 8
```

```c
```
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");
```

```
i = symbolic

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

```
vector[i] > 8

printf("big element\n");
printf("small element");
```
1D Symbolic pointers

- \texttt{vector[i]} is a dereference of a symbolic pointer
  - Concrete base address
  - Some symbolic offset \( i \)

- I.e. if vector is at 0xdeadbeef, \texttt{vector[i]} is a load (0xdeadbeef + i)

\begin{itemize}
  \item \( i \) = symbolic
  \item vector = \{1,2,...\}
\end{itemize}
Constraints over memory

● Theory of arrays:
  ○ read: array × index → value
  ○ write: array × index × value → array
  ○ $\text{read(\text{write}(a, p, v), r)} = v \quad \text{if } p = r$
  ○ $\text{read(\text{write}(a, p, v), r)} = \text{read}(a, r) \quad \text{if } p \neq r$

● 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

```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");
else
    printf("small element");
```
1D Symbolic pointers: constraints in theory of arrays

```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");
else
    printf("small element");
```
1D Symbolic pointers: constraints in theory of arrays

```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");
else
    printf("small element");
```

- $i =$ symbolic
- array vector[10] = [1 2 3 4 5 6 7 8 9 10]
- (Read $i$ vector)
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

```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");
```

\(i = \text{symbolic}\)
\(j = \text{symbolic}\)
2D Symbolic pointers: constraints in theory of arrays

```c
int i, j;
makesymbolic(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");
```

array matrix[3] = [0xdeadbeef 0xdeadbef0 0xdeadbef1]

i = symbolic
j = symbolic
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");
```
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] = [0xdeadbeef 0xdeadbef0 0xdeadbef1]`
- array `matrix_0[3] = [0 0 0]`
- array `matrix_1[3] = [0 0 42]`
- array `matrix_2[3] = [0 0 0]`

(Read i matrix)
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);

array matrix[3] = [0xdeadbeef 0xdeadbef0 0xdeadbef1]
array matrix_0[3] = [0 0 0]
array matrix_1[3] = [0 0 42]
array matrix_2[3] = [0 0 0]

matrix[1][2] = 42;
if (matrix[i][j] > 8)
    printf("big element\n");
else
    printf("zero");
```

Read j (Read i matrix)
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(Read j 0xdeadbeef); // Read j 0xdeadbeef
```

- $i = \text{symbolic}$
- $j = \text{symbolic}$

- array $matrix[3] = [0xdeadbeef 0xdeadbeef0 0xdeadbeef1]$
- 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

```c
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
    printf("zero");
```

- $i =$ symbolic
- $j =$ symbolic

array $matrix[3] = [0xDEEDEEBF 0xDEEDEBEF0 0xDEEDEBEF1]$

- $array\ matrix_0[3] = [0 0 0]$
- $array\ matrix_1[3] = [0 0 42]$
- $array\ matrix_2[3] = [0 0 0]$
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

```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");
```
2D Symbolic pointers: Forking

```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>j = 0
j = symbolic
array matrix_0[3] = [0 0 0]</i>
2D Symbolic pointers: Forking

```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
    (Read j matrix_0)
```

\( i = 0 \)
\( j = \text{symbolic} \)

array matrix_0[3] = [0 0 0]
2D Symbolic pointers: Forking

```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
");
else
    printf("zero\n");
```

```c
i = 2
j = symbolic
```

Array `matrix_2[3] = [0 0 0]`
Path explosion
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");
```

\[ i = 0 \lor 1 \lor 2 \]
\[ j = \text{symbolic} \]
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 ∨ 1 ∨ 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

```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");
```

```
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) V (Read $j$ matrix_1) V (Read $j$ matrix_2)
OR expressions are hard(-er) to solve
2D Symbolic pointers: Flat memory

```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");
```
**2D Symbolic pointers: Flat memory**

```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");
```

array memory[12] = [
    3  6  9
    0  0  0
    0  0  42
    0  0  0
]
2D Symbolic pointers: Flat memory

```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");
```

Note that calloc return 3, 6, 9 as the addresses of the rows now
2D Symbolic pointers: Flat memory

```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");
(Read (3*i + j + 3) memory)
```

```plaintext
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

\[ \text{pts}(p1) = \{A, B\} \]
Our approach: partitioning into segments

\[ \text{pts}(p1) = \{A, B\} \]
Our approach: partitioning into segments

\[ \text{pts}(p1) = \{A, B\} \quad \text{pts}(p2) = \{B, C\} \]
Our approach: partitioning into segments

\[ \text{pts}(p1) = \{A, B\} \quad \text{pts}(p2) = \{B, C\} \]
Our approach: partitioning into segments

pts(p1) = \{A, B\} \quad pts(p2) = \{B, 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");
2D Symbolic pointers: Segmented Memory

```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 segment_0[3]
    = [0xdeedbef0 0xdeedbef3 0xdeedbef6]
array segment_1[9]
    = [ 0 0 0
        0 0 42
        0 0 0 ]
```
```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”);
else
    printf(“zero”);

(Read (3*i + j) segment_1)
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

SQLite

Apache
Portable Runtime Project
NxN matrix: single lookup extra allocation

```c
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

![Graph showing time (seconds) vs N for different methods like Forking, Merging, Flat Memory, and Segmented.](image)
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

1. define(`A', `l')
2. define(`P', 2)
3. ?
4. ?
5. ifelse(? , P, eval(1 + P))

Targeted input file for m4
m4 DFS
m4 BFS
m4 default
make DFS
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/
NxN matrix: single lookup
Symbolic execution example: get_sign

```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;
}

get_sign(x);
```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;
}
```
```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;
}
```
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;
}