Relocatable Addressing Model for Symbolic Execution

David Trabish and Noam Rinetzky
Tel-Aviv University, Israel

ISSTA 2020

TEL AVIV UNIVERSITY

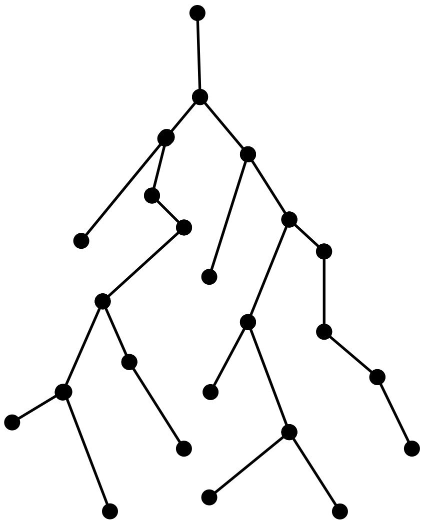
Symbolic Execution

Program analysis technique for path exploration

- Runs the program with **symbolic input**
- Explores only **feasible** paths

Applications:

- Test input generation
- Bug finding



In this talk

We focus on two challenges:

- Path explosion due to symbolic pointers
- Solving array theory constraints

In this talk

We focus on two challenges:

- Path explosion due to symbolic pointers
- Solving array theory constraints

We are going to tackle both challenges using a new addressing model:

Relocatable Addressing Model

Challenge 1:

Symbolic Pointers

Addressing Model

Constraints over memory are encoded using **array theory**

- Every memory object *mo* is backed by an SMT array:
  - Maintains *mo*'s contents
- Every memory object has a **concrete base address**
  - Concrete addresses are used to resolve pointers to SMT arrays

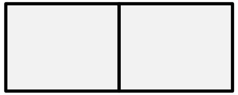
Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```

Symbolic Pointers

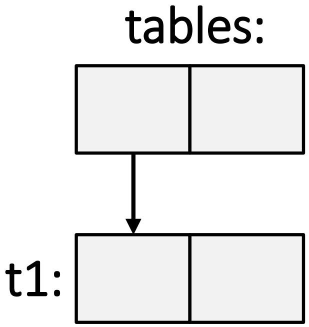
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```

tables:



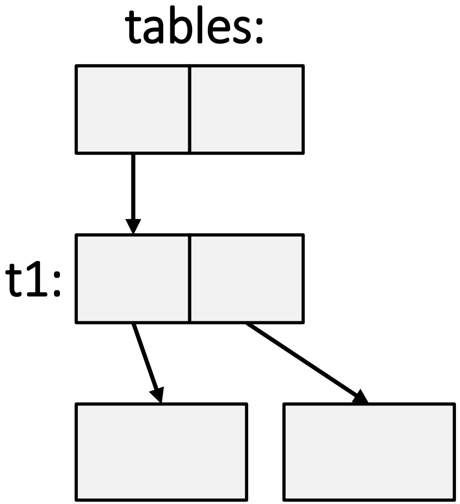
Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



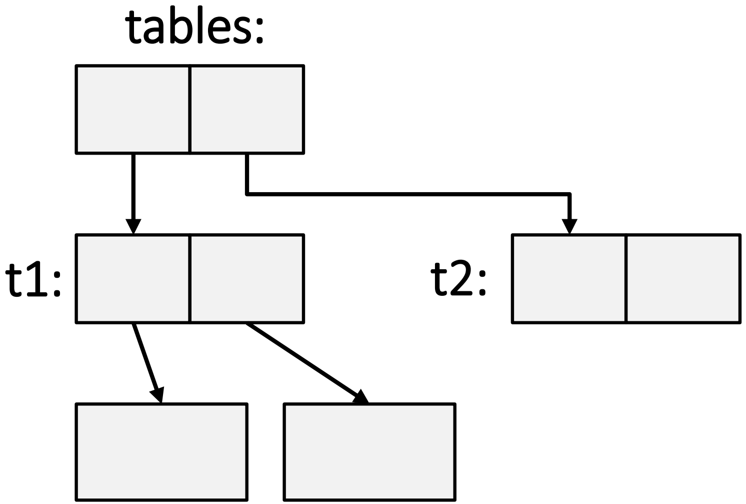
Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



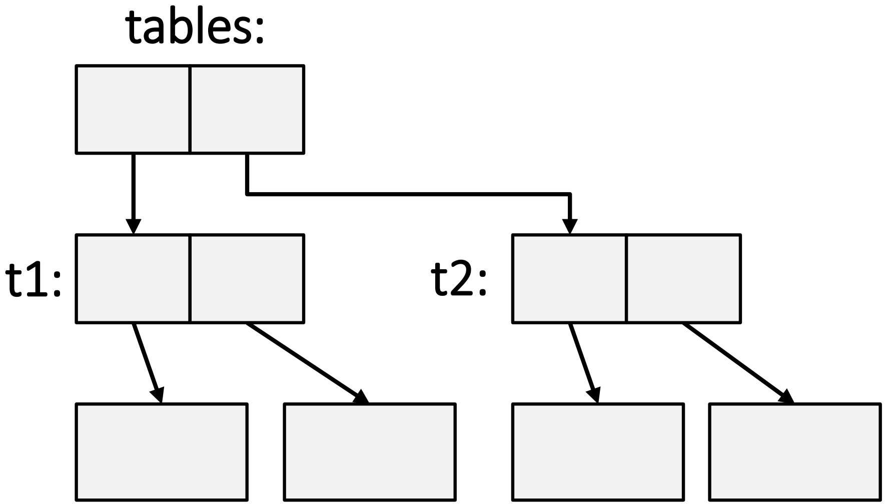
Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



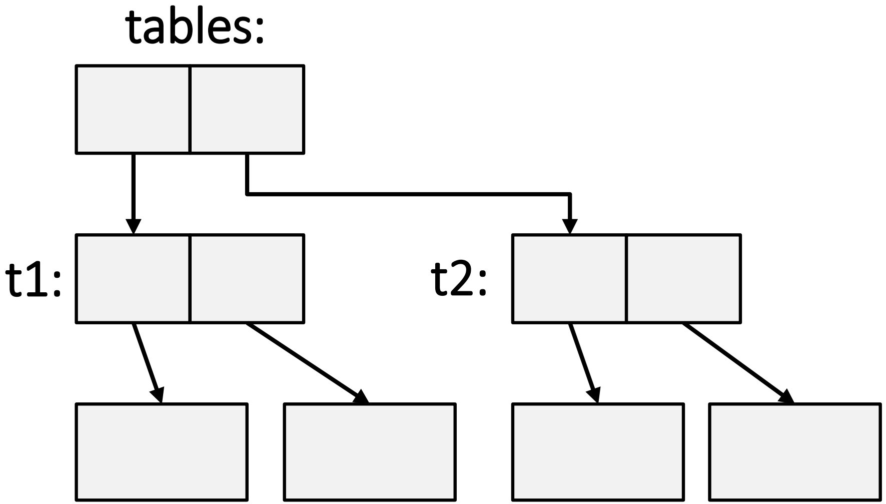
Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



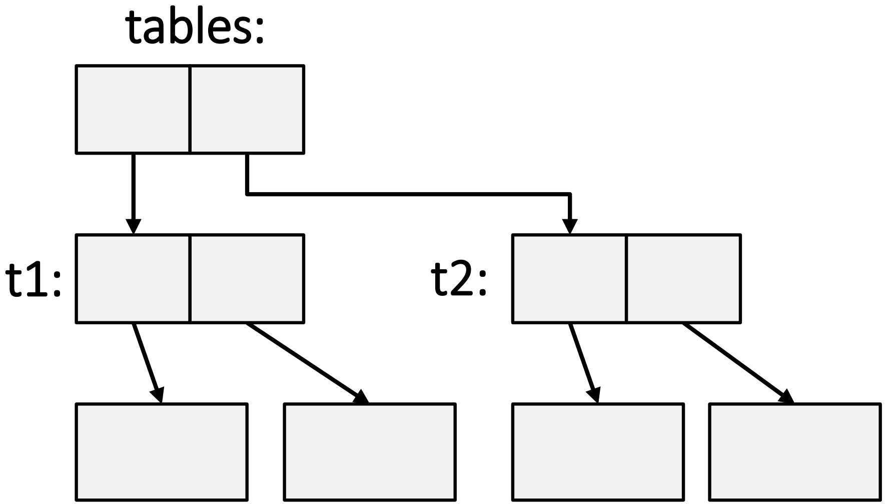
Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



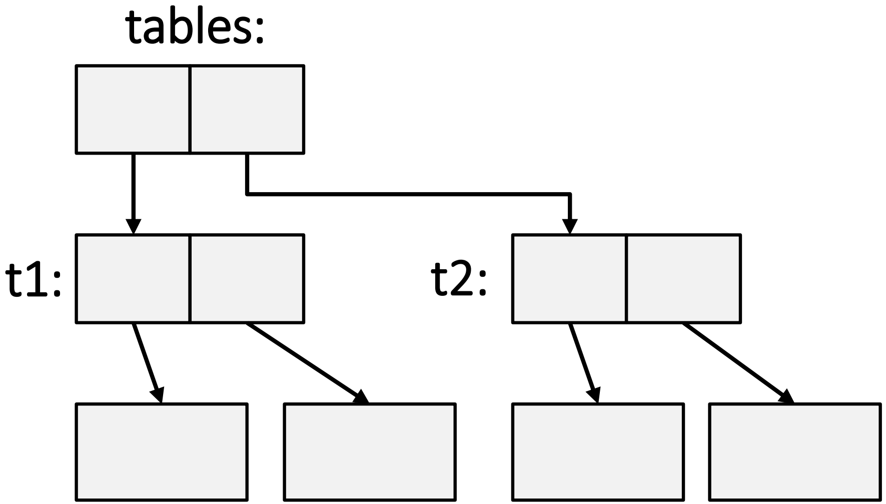
Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



Symbolic Pointers

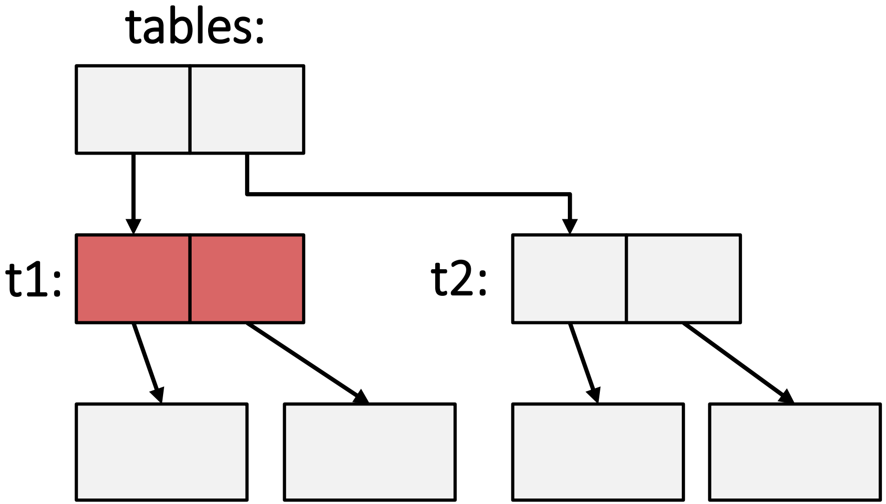
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



$$addr_{t1} + i$$

Symbolic Pointers

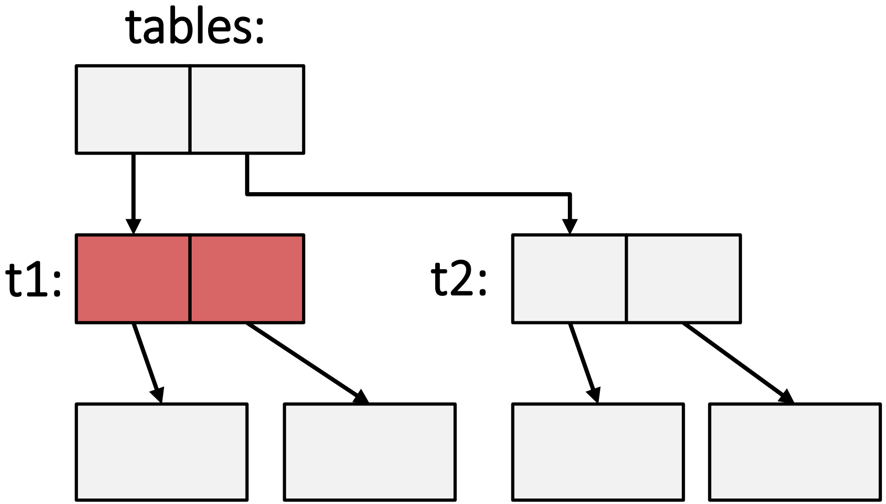
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



$$addr_{t1} + i$$

Symbolic Pointers

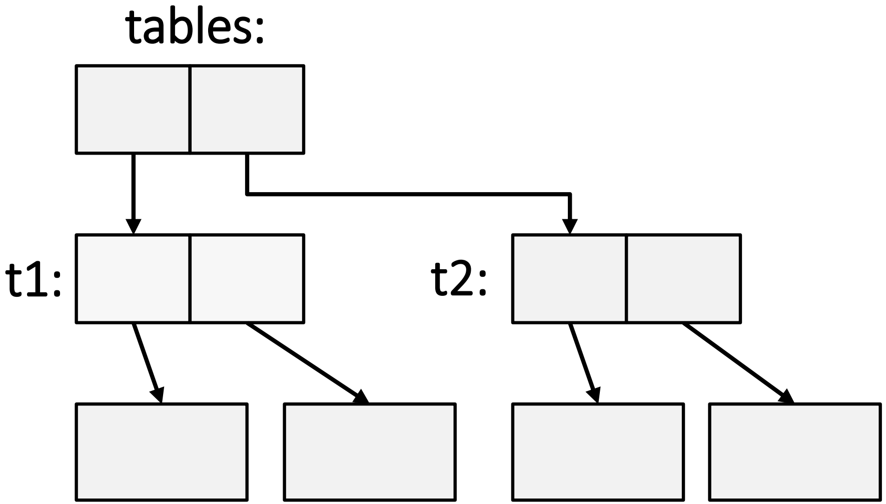
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



$$addr_{t1} + i \quad \rightarrow \quad select(arr_{t1}, i)$$

Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```

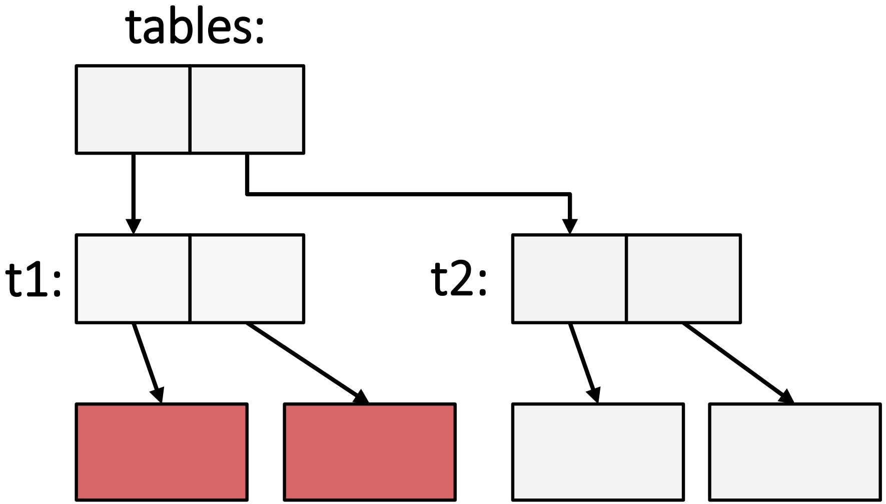


$addr_{t1} + i \rightarrow select(arr_{t1}, i)$

$select(arr_{t1}, i) + j$

Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```

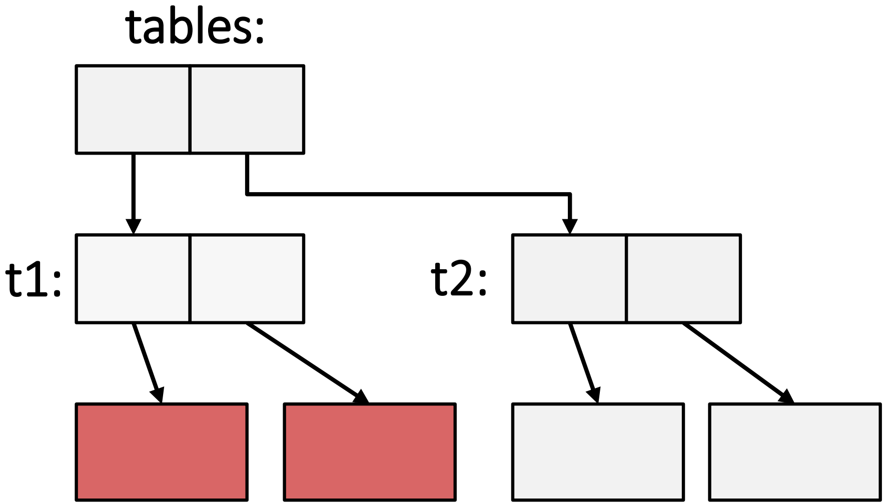


$addr_{t1} + i \rightarrow select(arr_{t1}, i)$

$select(arr_{t1}, i) + j$

Symbolic Pointers

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



$addr_{t1} + i \rightarrow select(arr_{t1}, i)$

$select(arr_{t1}, i) + j \rightarrow$?

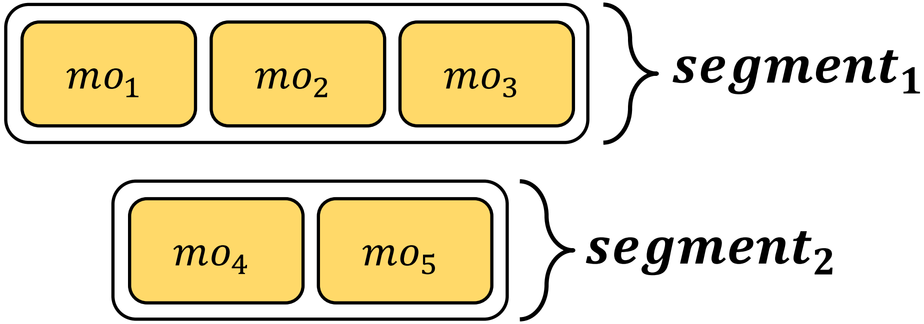
Symbolic Pointers

How can we handle symbolic pointers?

- Forking [KLEE]
- Merging [SAGE]
- Segmented memory model [Kapus et al., FSE'19]

Segmented Memory Model

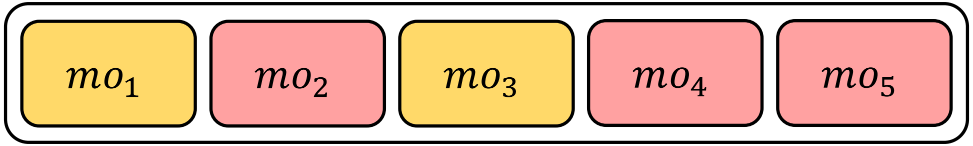
- Partitions the memory into segments using **static pointer analysis**
- Any pointer is guaranteed to be resolved to a single segment
- **Forks are avoided** in the case of multiple resolution



Segmented Memory Model

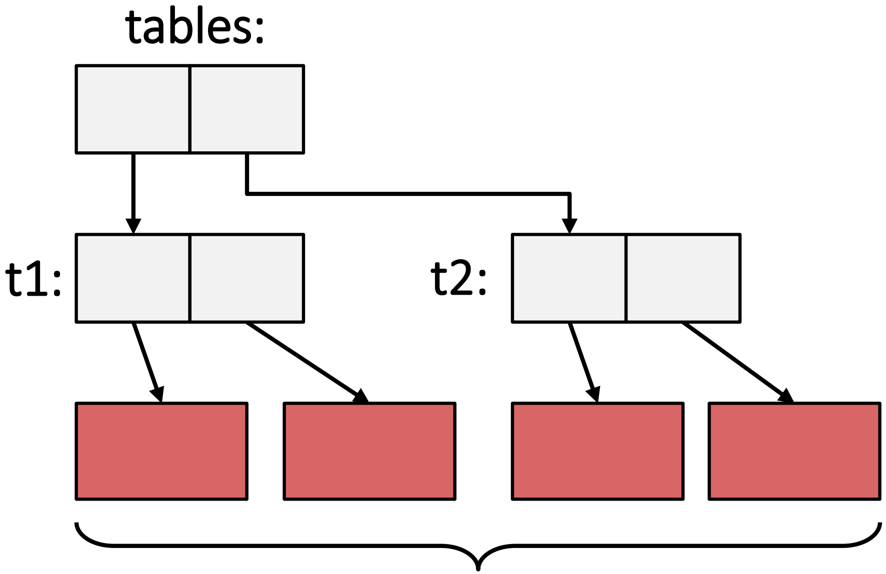
Limitations:

- Based on static pointer analysis that can be **imprecise**
- Segments might contain **redundant** objects
- Array theory constraints become **more complex**



Segmented Memory Model

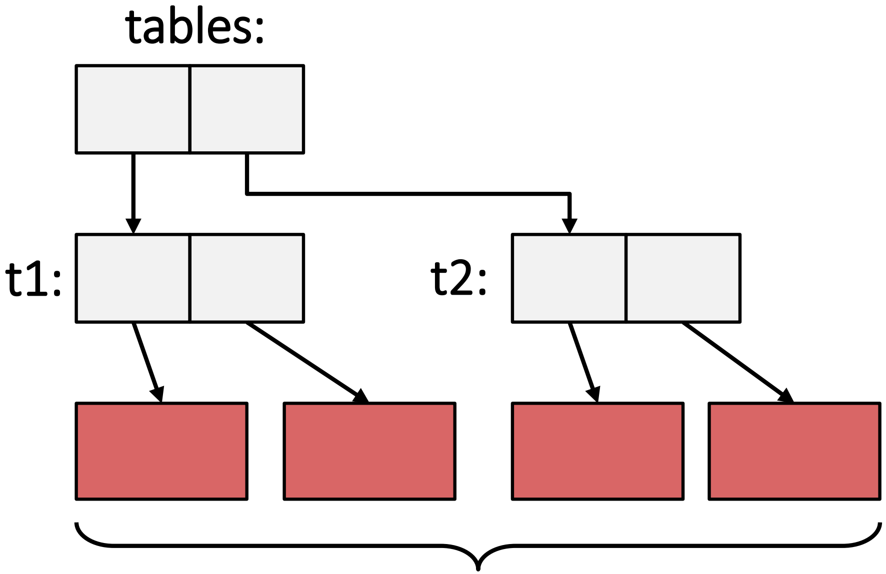
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



pointer analysis can’t distinguish

Segmented Memory Model

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```

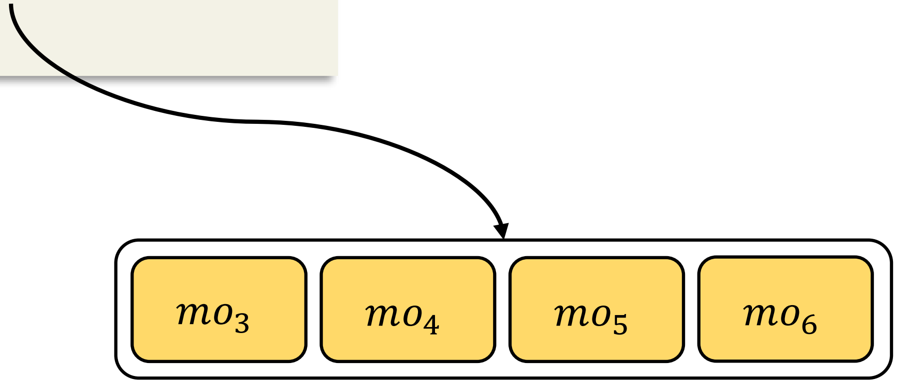


pointer analysis can't distinguish
mapped to the same segment

Segmented Memory Model

```
// i < N, j < 100
unsigned i,j;
if (tables[0][i][j] == 7)
...
```

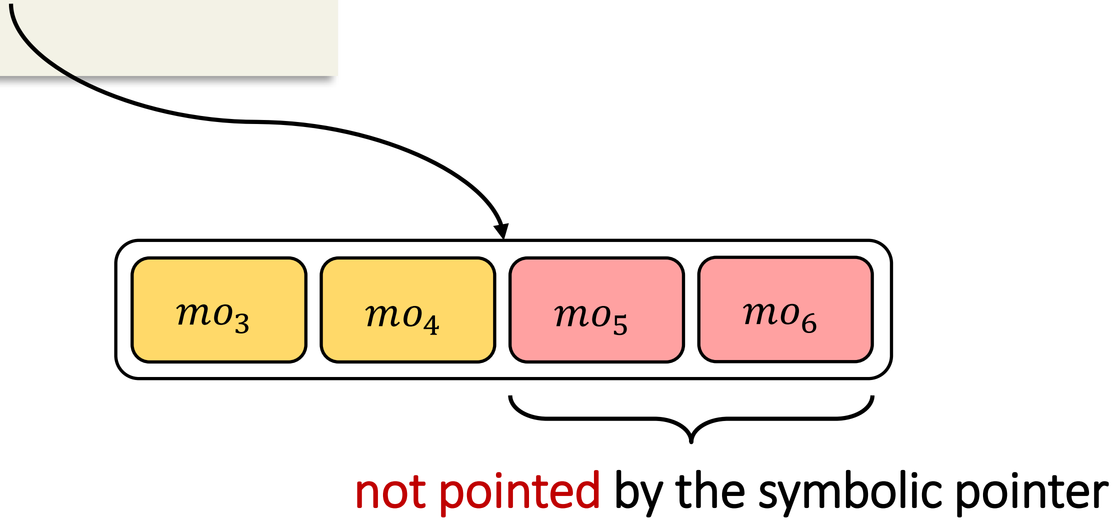
- Forking is avoided ✓



Segmented Memory Model

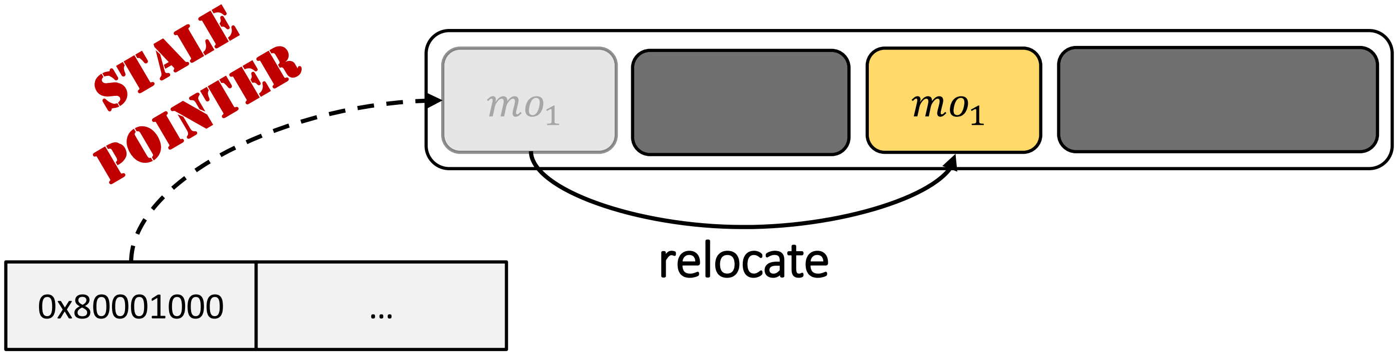
```
// i < N, j < 100
unsigned i,j;
if (tables[0][i][j] == 7)
...
```

- Forking is avoided ✓
- Unnecessarily large segment ✗
- Affects constraint solving ✗



Goal

- It would be nice to create the segments **on-the-fly**
- **Not supported** with the current addressing model
- Relocating an allocated object is **tricky**
  - Requires updating all its references
  - Requires precise type information



Relocatable Addressing Model

We propose a new model:

- Base addresses are **symbolic** values, rather than concrete
- The **non-overlapping property** is preserved using **address constraints**
- The address constraints are substituted when constructing a query

Relocatable Addressing Model

```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```

tables: $[\alpha_1, \alpha_2]$

tables[0]: $[\alpha_3, \alpha_4]$

tables[1]: $[\alpha_5, \alpha_6]$

address constraints

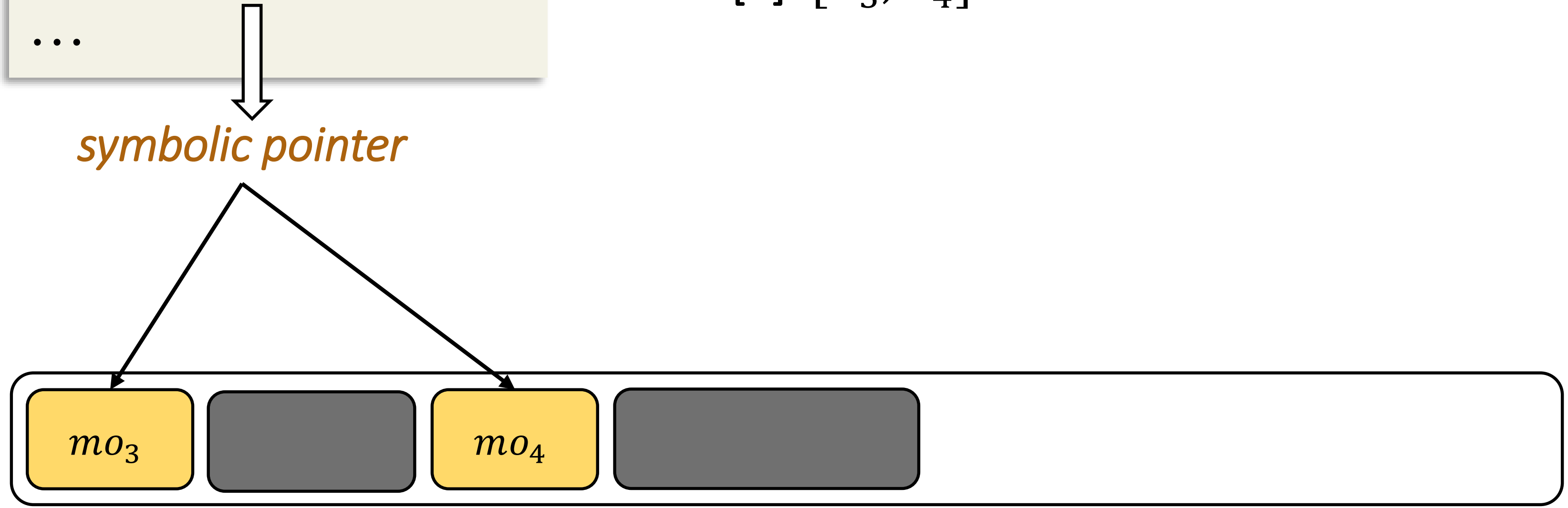
$$\begin{cases} \alpha_1 = 0x80001000 \\ \alpha_2 = 0x80001100 \\ \alpha_3 = 0x80001200 \\ \alpha_4 = 0x80001300 \\ \alpha_5 = \cdots \\ \alpha_6 = \cdots \end{cases}$$

Dynamically Segmented Memory Model

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables: $[\alpha_1, \alpha_2]$

tables[0]: $[\alpha_3, \alpha_4]$



Dynamically Segmented Memory Model

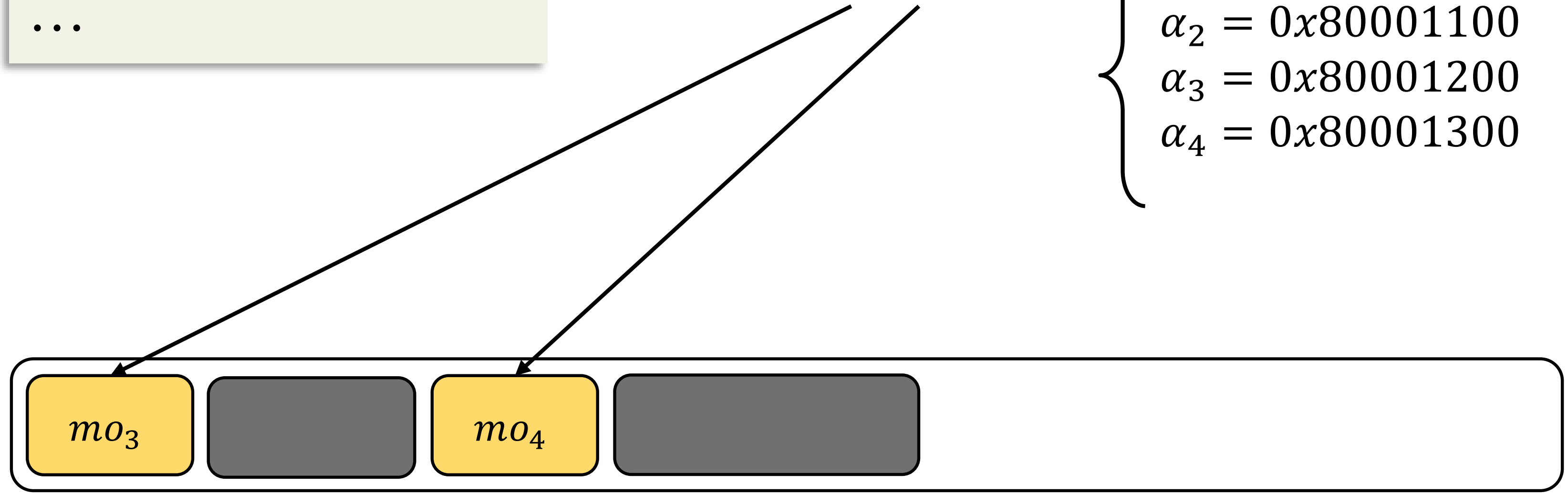
```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables: $[\alpha_1, \alpha_2]$

tables[0]: $[\alpha_3, \alpha_4]$

address constraints

$$\begin{cases} \alpha_1 = 0x80001000 \\ \alpha_2 = 0x80001100 \\ \alpha_3 = 0x80001200 \\ \alpha_4 = 0x80001300 \end{cases}$$



Dynamically Segmented Memory Model

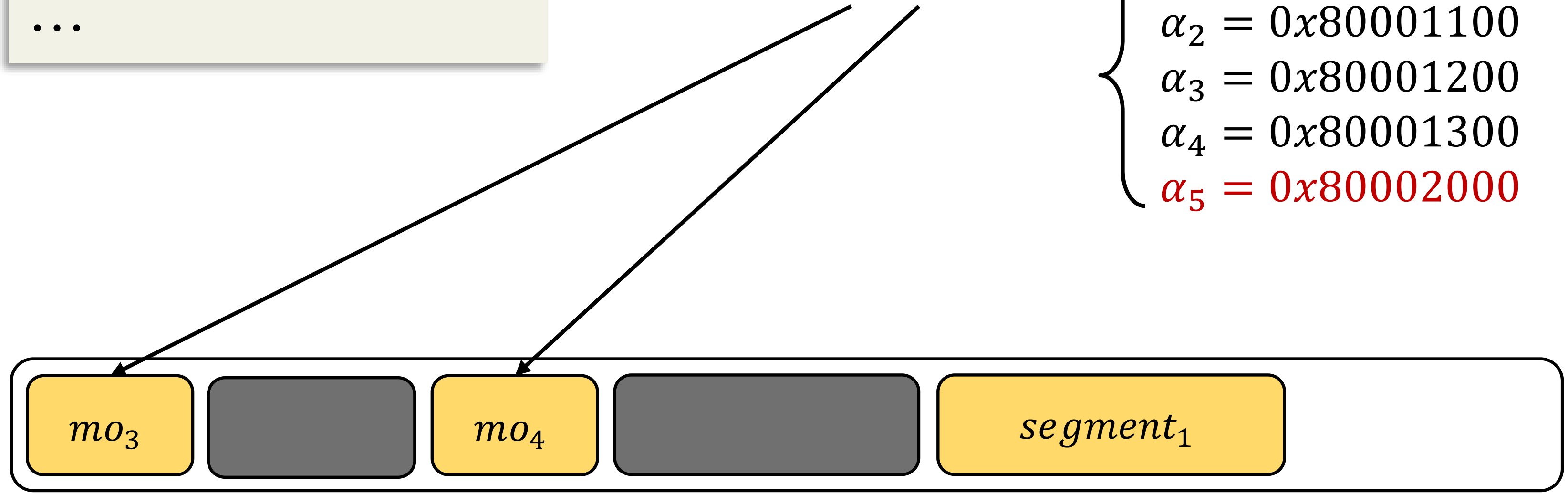
```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables: $[\alpha_1, \alpha_2]$

tables[0]: $[\alpha_3, \alpha_4]$

address constraints

$$\begin{cases} \alpha_1 = 0x80001000 \\ \alpha_2 = 0x80001100 \\ \alpha_3 = 0x80001200 \\ \alpha_4 = 0x80001300 \\ \alpha_5 = 0x80002000 \end{cases}$$



Dynamically Segmented Memory Model

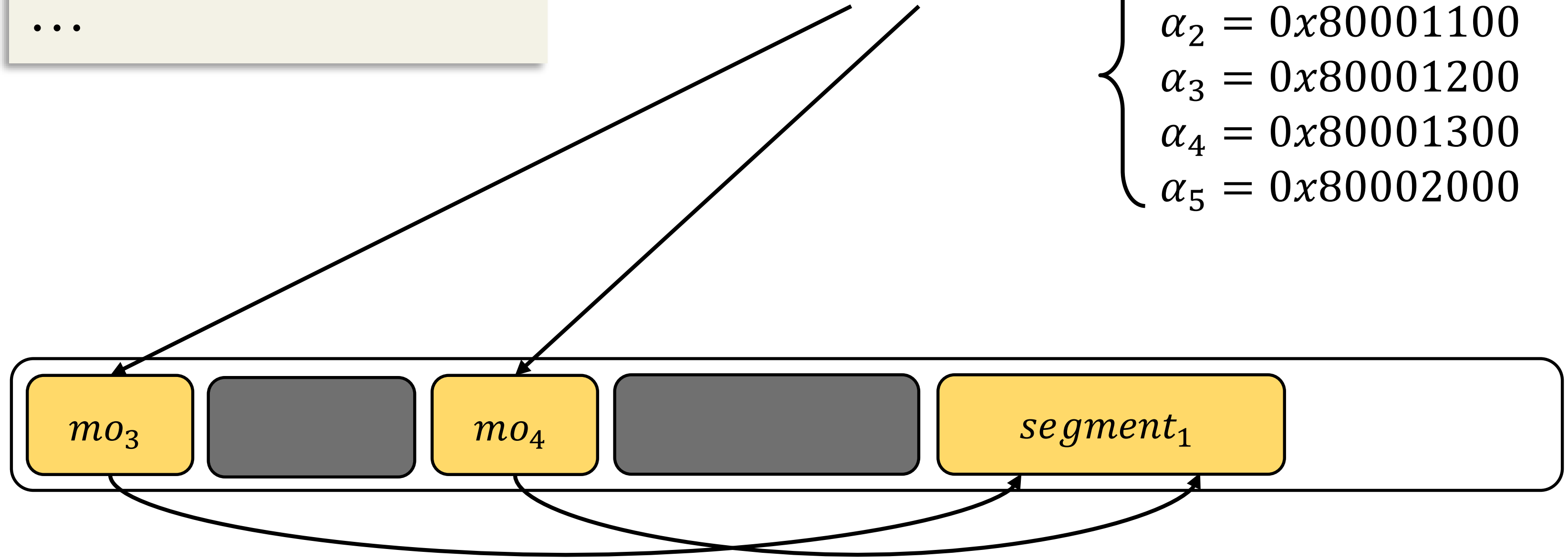
```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables: $[\alpha_1, \alpha_2]$

tables[0]: $[\alpha_3, \alpha_4]$

address constraints

$$\begin{cases} \alpha_1 = 0x80001000 \\ \alpha_2 = 0x80001100 \\ \alpha_3 = 0x80001200 \\ \alpha_4 = 0x80001300 \\ \alpha_5 = 0x80002000 \end{cases}$$



Dynamically Segmented Memory Model

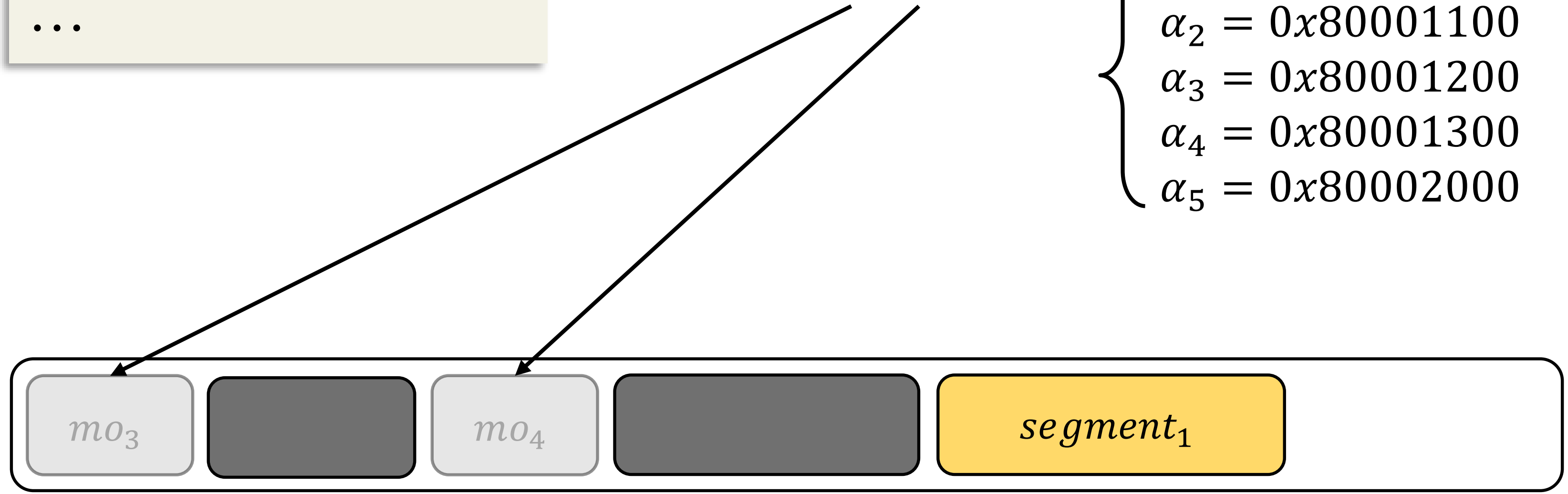
```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables: $[\alpha_1, \alpha_2]$

tables[0]: $[\alpha_3, \alpha_4]$

address constraints

$$\begin{cases} \alpha_1 = 0x80001000 \\ \alpha_2 = 0x80001100 \\ \alpha_3 = 0x80001200 \\ \alpha_4 = 0x80001300 \\ \alpha_5 = 0x80002000 \end{cases}$$



Dynamically Segmented Memory Model

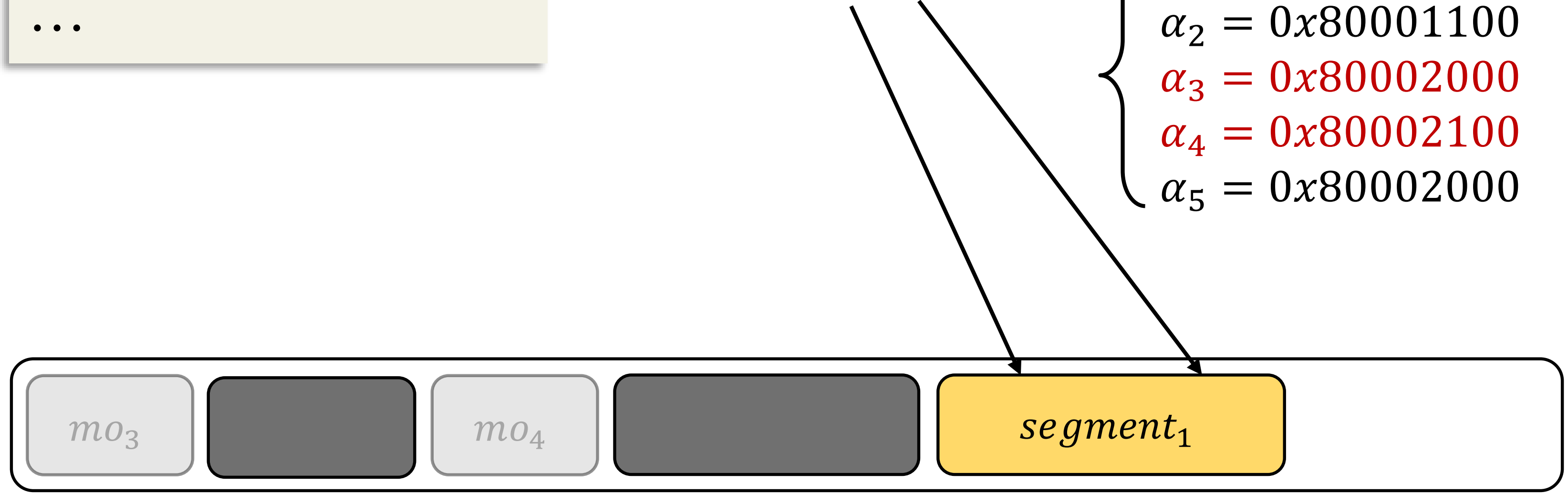
```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables: $[\alpha_1, \alpha_2]$

tables[0]: $[\alpha_3, \alpha_4]$

address constraints

$$\begin{cases} \alpha_1 = 0x80001000 \\ \alpha_2 = 0x80001100 \\ \alpha_3 = 0x80002000 \\ \alpha_4 = 0x80002100 \\ \alpha_5 = 0x80002000 \end{cases}$$



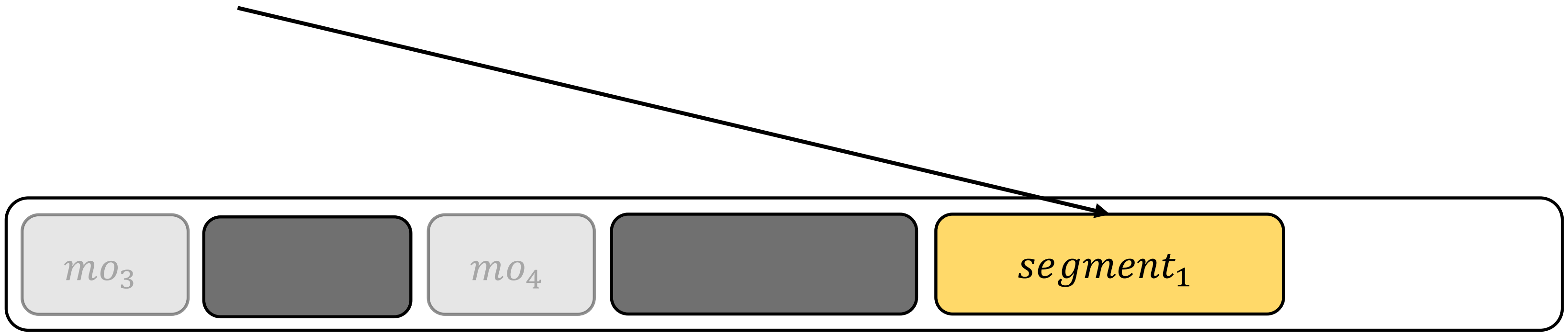
Dynamically Segmented Memory Model

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables: $[\alpha_1, \alpha_2]$

tables[0]: $[\alpha_3, \alpha_4]$

symbolic pointer



Challenge 2:

Constraint Solving

Constraint Solving

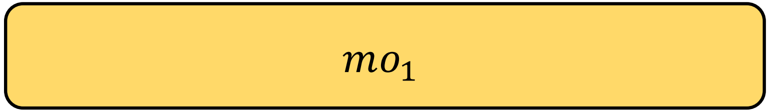
- Solving array theory constraints is **expensive**
- Especially when arrays are big (many *store*'s)

$$select(store\left(store\left(store(\ldots)\right)\right)), x) = y + \cdots$$

Constraint Solving

When a big array is accessed with a symbolic offset:

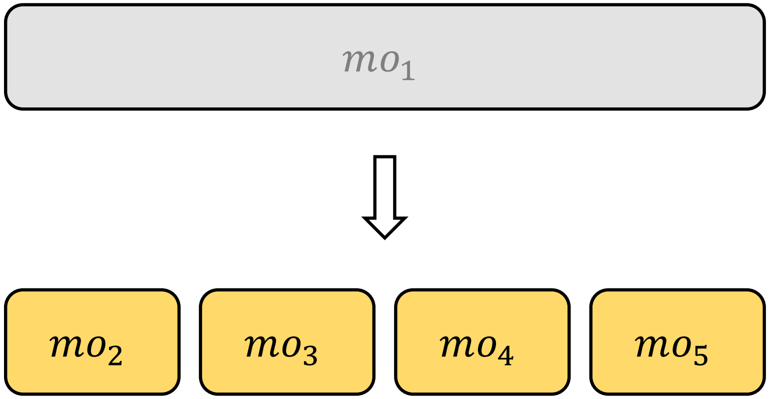
- Split the memory object to smaller adjacent objects
- Different splitting strategies can be applied



Constraint Solving

When a big array is accessed with a symbolic offset:

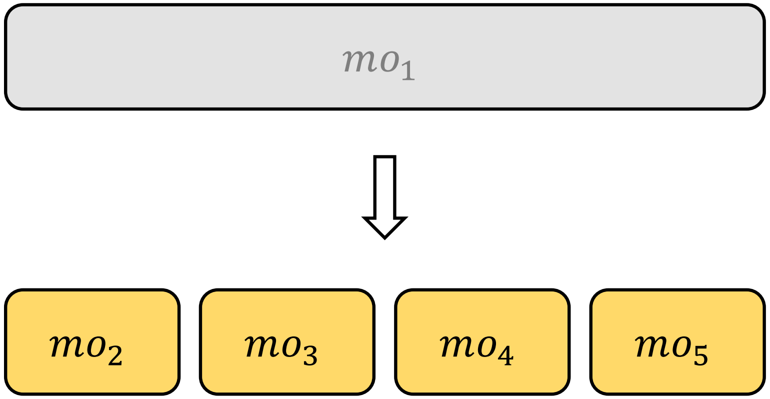
- Split the memory object to smaller adjacent objects
- Different splitting strategies can be applied



Constraint Solving

After the split:

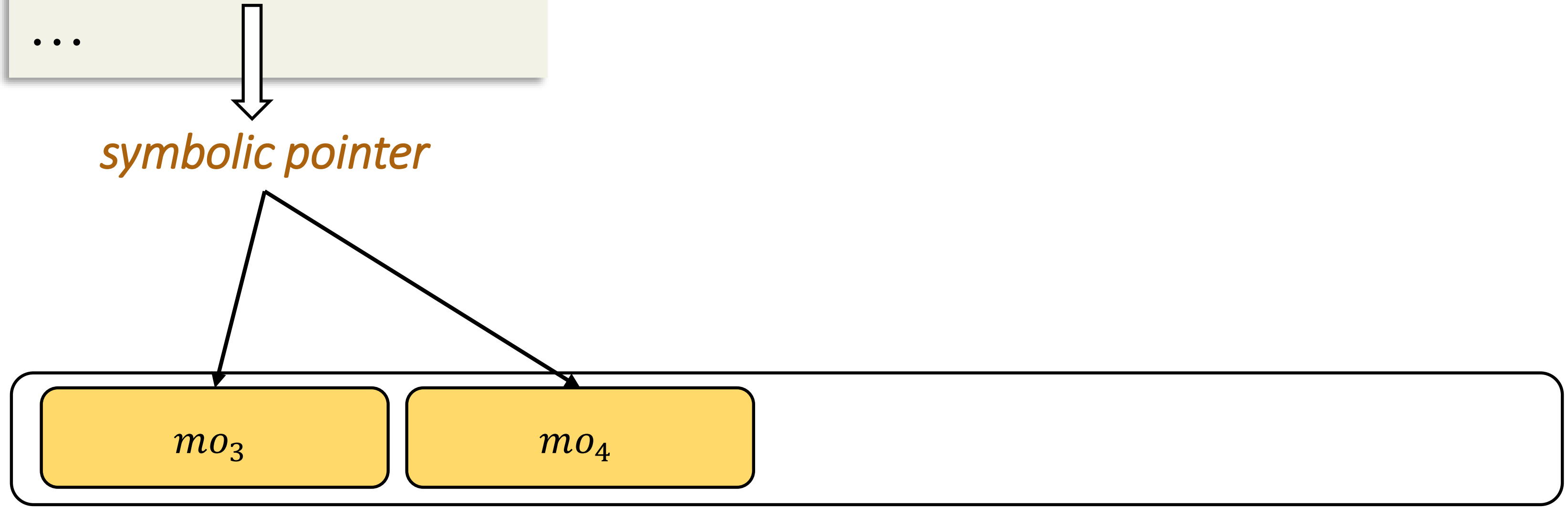
- Potentially **more forks** due to additional multiple resolutions
- But SMT arrays are **smaller**



Dynamically Splitting Objects

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

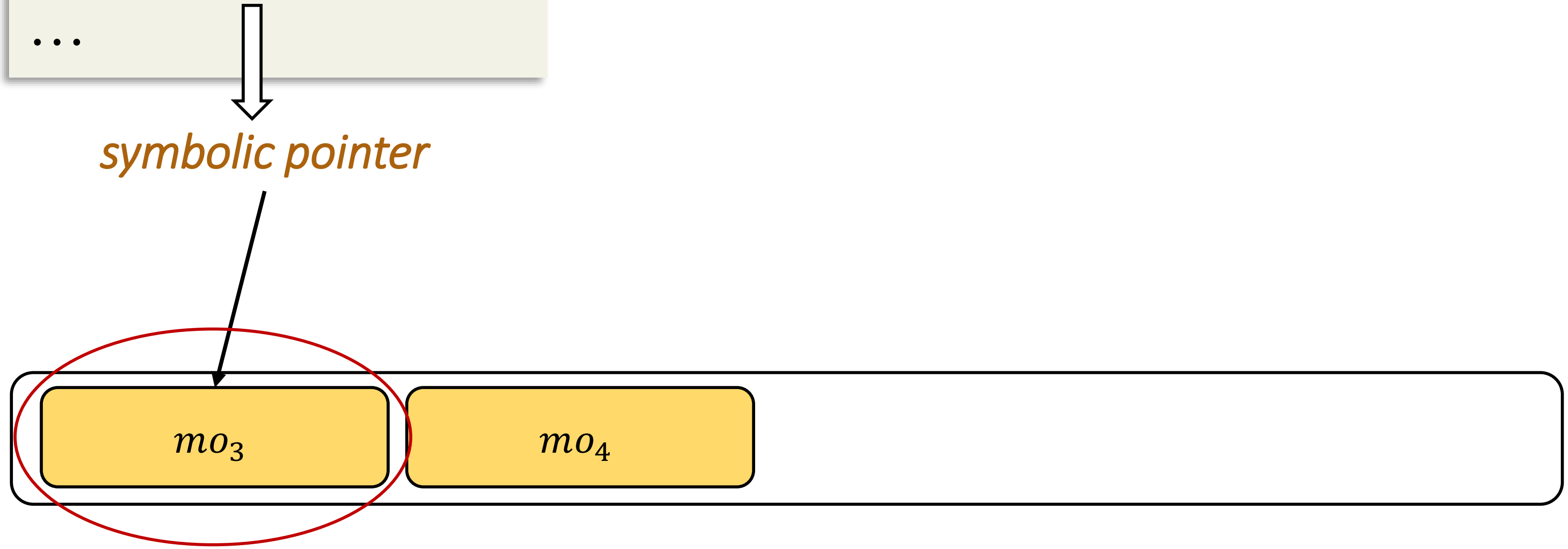
tables[0]: $[\alpha_3, \alpha_4]$



Dynamically Splitting Objects

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables[0]: $[\alpha_3, \alpha_4]$



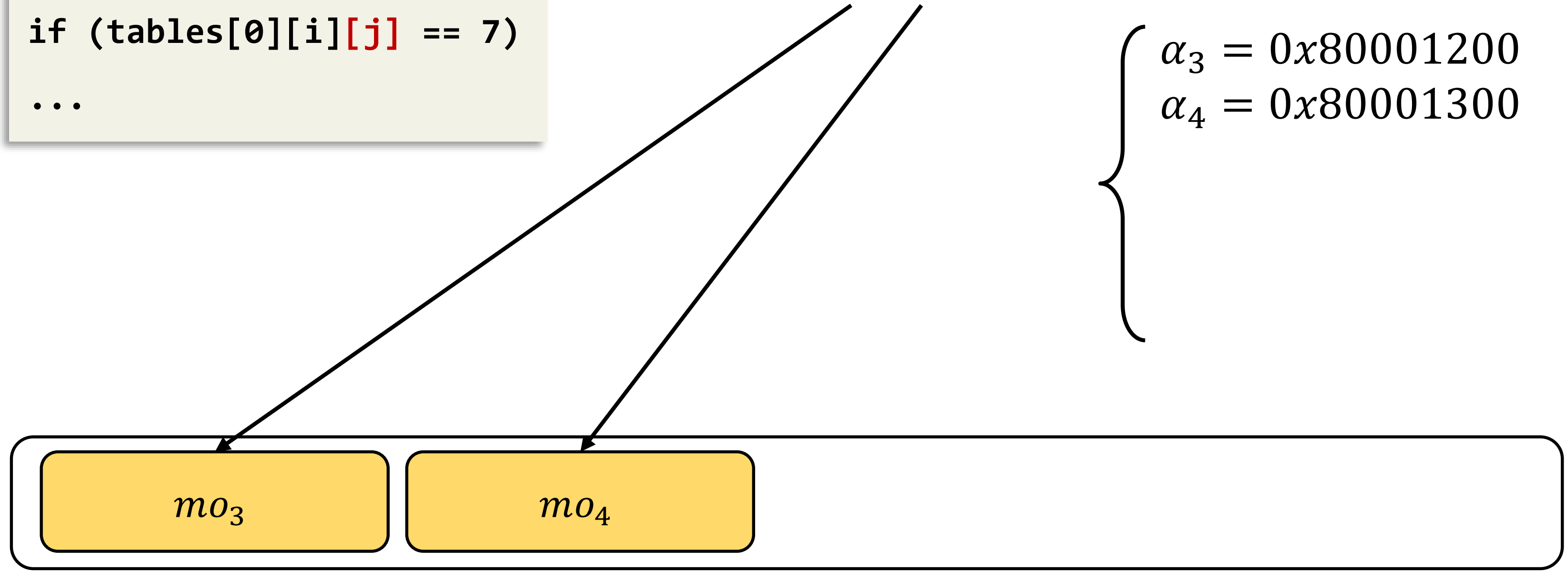
Dynamically Splitting Objects

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables[0]: $[\alpha_3, \alpha_4]$

address constraints

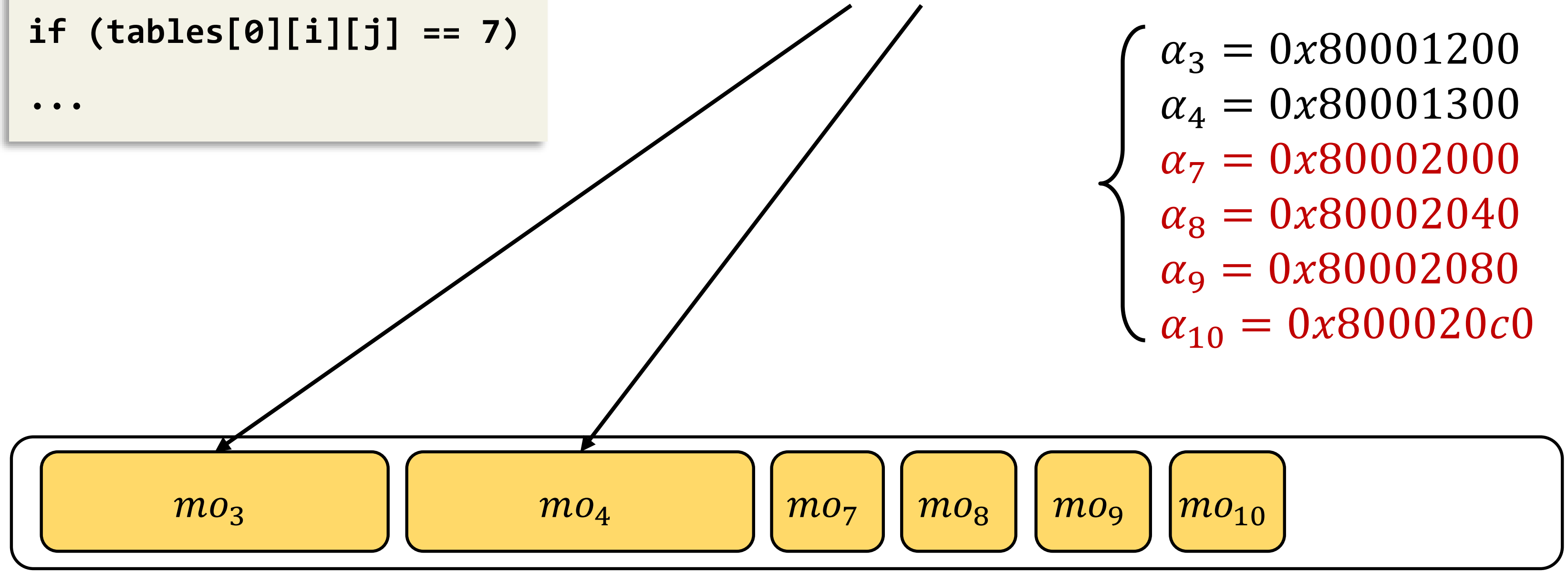
$$\begin{cases} \alpha_3 = 0x80001200 \\ \alpha_4 = 0x80001300 \end{cases}$$



Dynamically Splitting Objects

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables[0]: $[\alpha_3, \alpha_4]$



address constraints

$$
\begin{cases}
\alpha_3 = 0x80001200 \\
\alpha_4 = 0x80001300 \\
\alpha_7 = 0x80002000 \\
\alpha_8 = 0x80002040 \\
\alpha_9 = 0x80002080 \\
\alpha_{10} = 0x800020c0
\end{cases}
$$

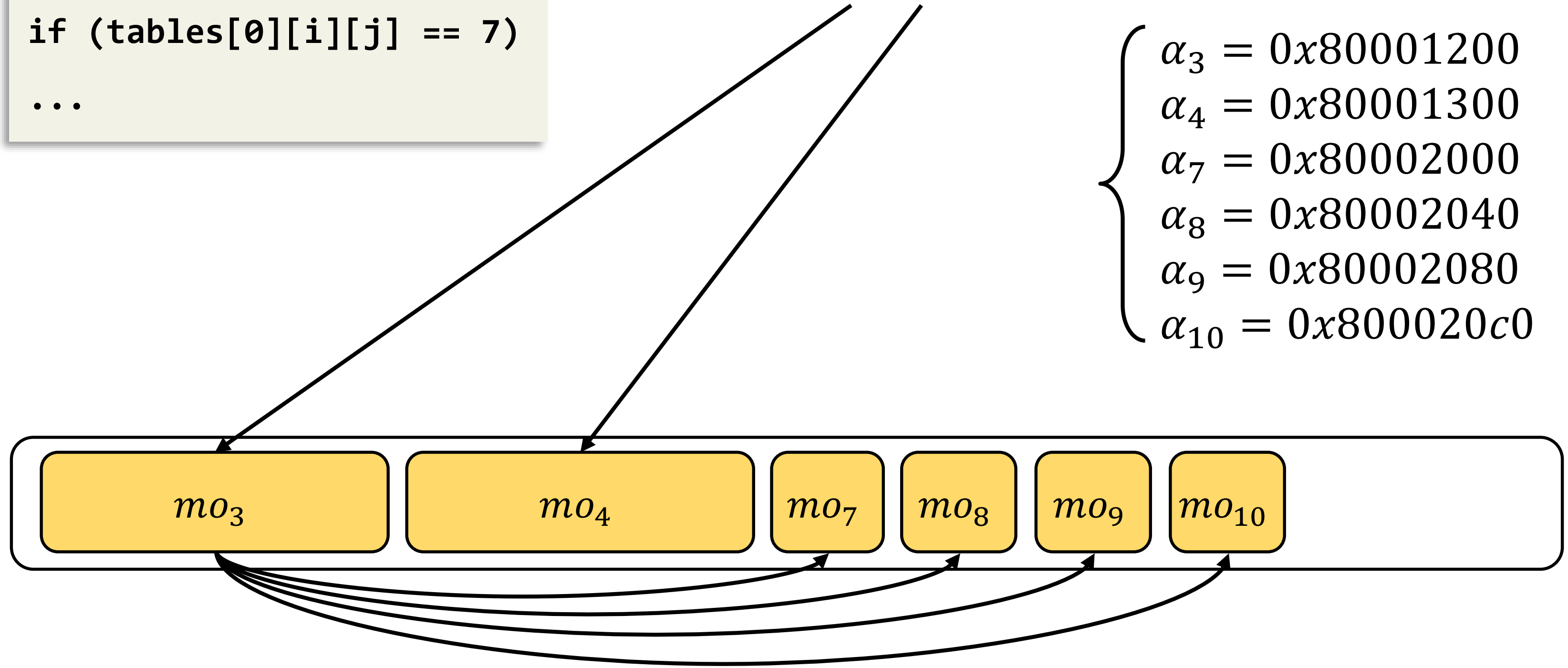
Dynamically Splitting Objects

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables[0]: $[\alpha_3, \alpha_4]$

address constraints

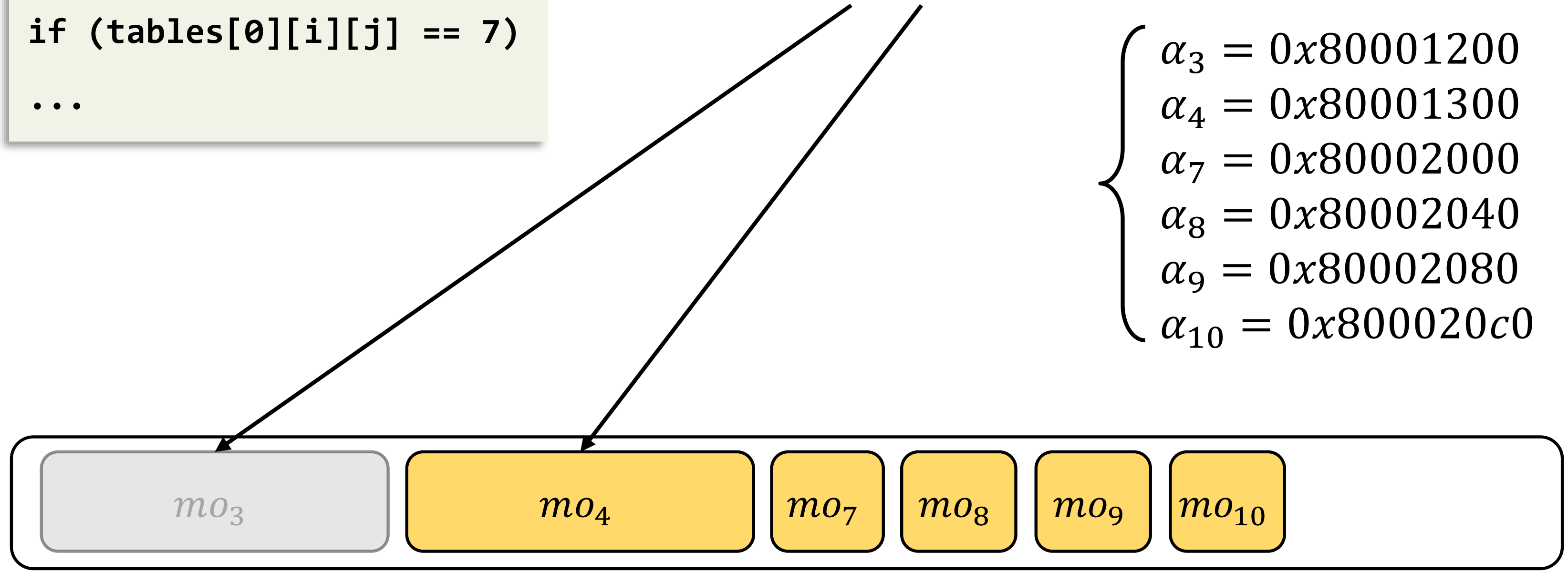
$$
\begin{cases}
\alpha_3 = 0x80001200 \\
\alpha_4 = 0x80001300 \\
\alpha_7 = 0x80002000 \\
\alpha_8 = 0x80002040 \\
\alpha_9 = 0x80002080 \\
\alpha_{10} = 0x800020c0
\end{cases}
$$



Dynamically Splitting Objects

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables[0]: $[\alpha_3, \alpha_4]$

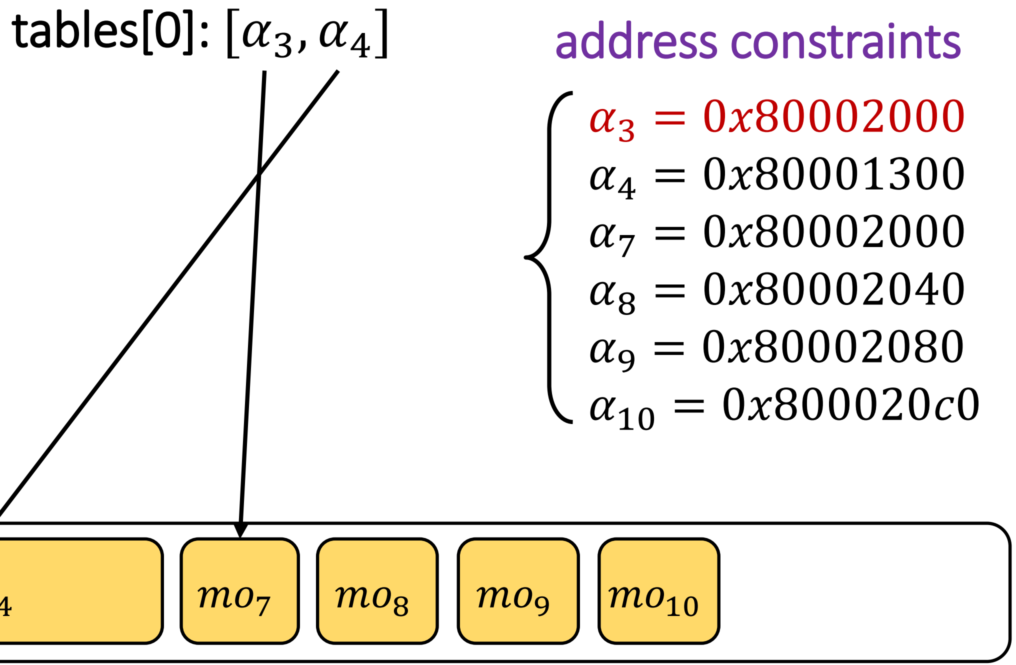


address constraints

$$\begin{cases} \alpha_3 = 0x80001200 \\ \alpha_4 = 0x80001300 \\ \alpha_7 = 0x80002000 \\ \alpha_8 = 0x80002040 \\ \alpha_9 = 0x80002080 \\ \alpha_{10} = 0x800020c0 \end{cases}$$

Dynamically Splitting Objects

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```



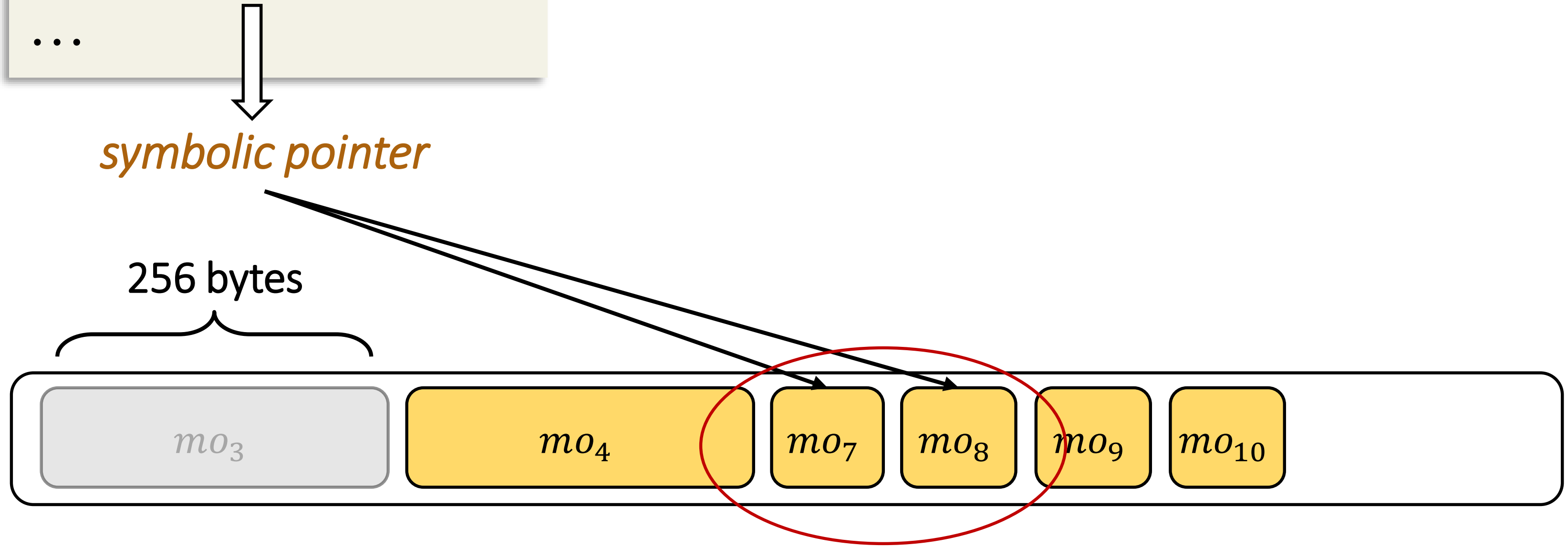
address constraints

$$\begin{cases} \alpha_3 = 0x80002000 \\ \alpha_4 = 0x80001300 \\ \alpha_7 = 0x80002000 \\ \alpha_8 = 0x80002040 \\ \alpha_9 = 0x80002080 \\ \alpha_{10} = 0x800020c0 \end{cases}$$

Dynamically Splitting Objects

```
// i < N, j < 100
if (tables[0][i][j] == 7)
...
```

tables[0]: $[\alpha_3, \alpha_4]$



Implementation

We implemented our addressing model on top of **KLEE**, using:

- LLVM 7.0.0
- STP 2.3.3

Evaluation

We evaluated our model in the context of:

- Inter-object partitioning (merging)
- Intra-object partitioning (splitting)

The benchmarks are:

- m4, make, sqlite, apr, gas, libxml2, coreutils

Evaluation: Merging

We first compare the sizes of the created segments with:

- Segmented memory model (SMM)
- Dynamically segmented memory model (DSMM)

Benchmark	Max. Segment Size (Bytes)	
	SMM	DSMM
m4	2753	1008
make	7574	1776
sqlite	17064	528
apr	8316	240

Evaluation: Merging

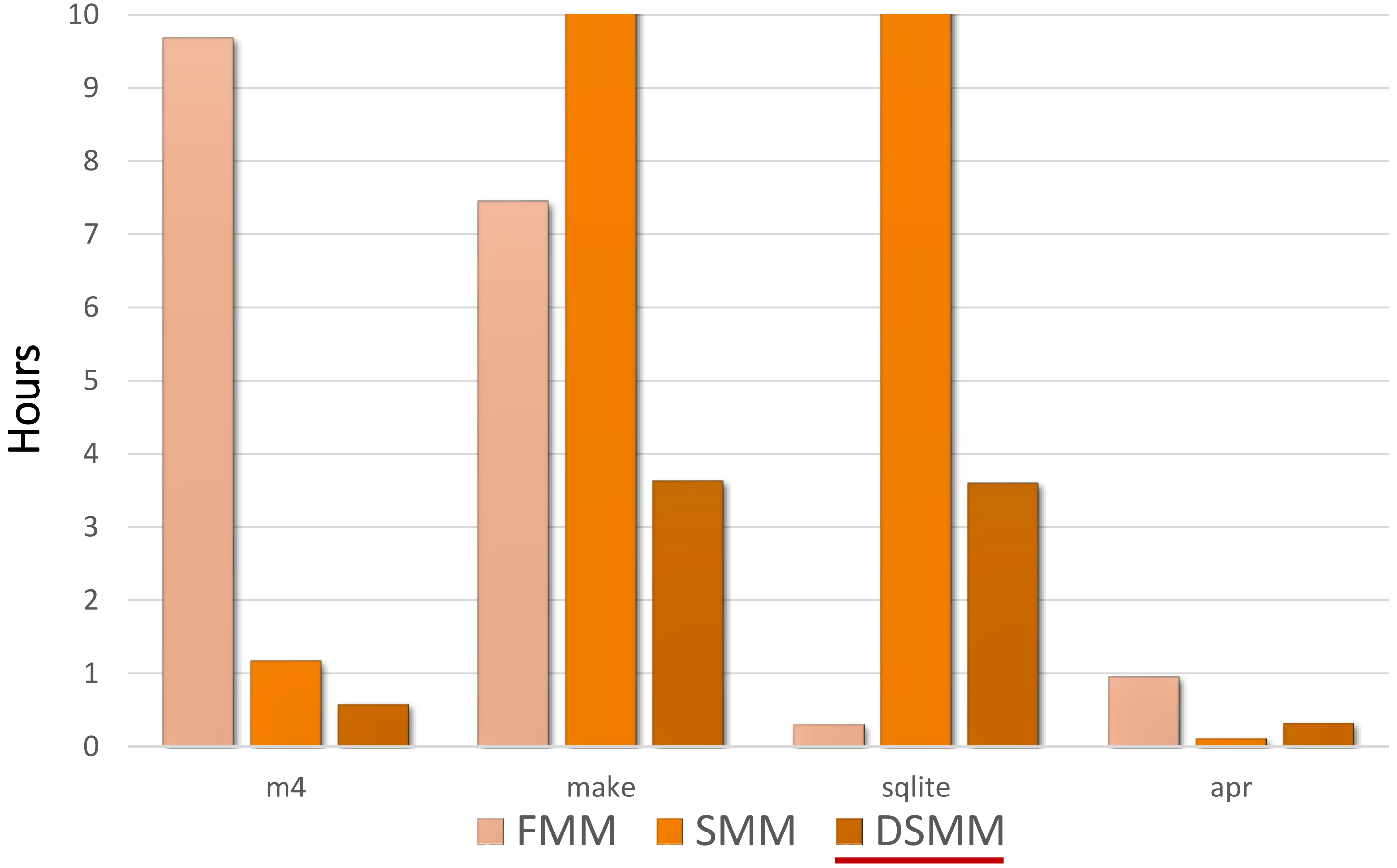
We compare the performance with different models:

- Vanilla KLEE, i.e., forking memory model (FMM)
- Segmented memory model (SMM)
- Dynamically segmented memory model (DSMM)

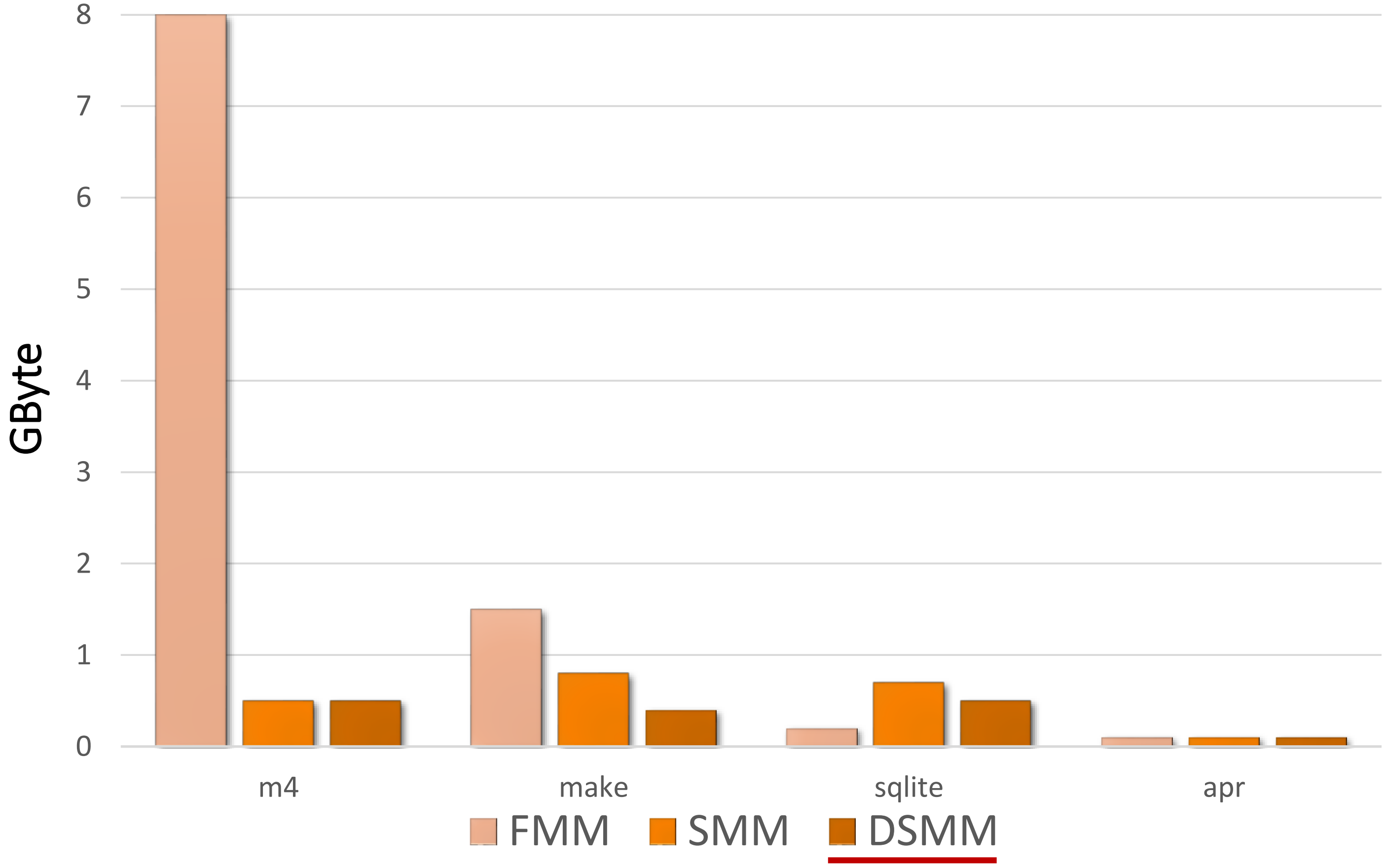
We ran each program with a timeout of 24 hours and recorded:

- Termination time (until full exploration)
- Memory usage

Termination Time



Memory Usage



Evaluation: Splitting

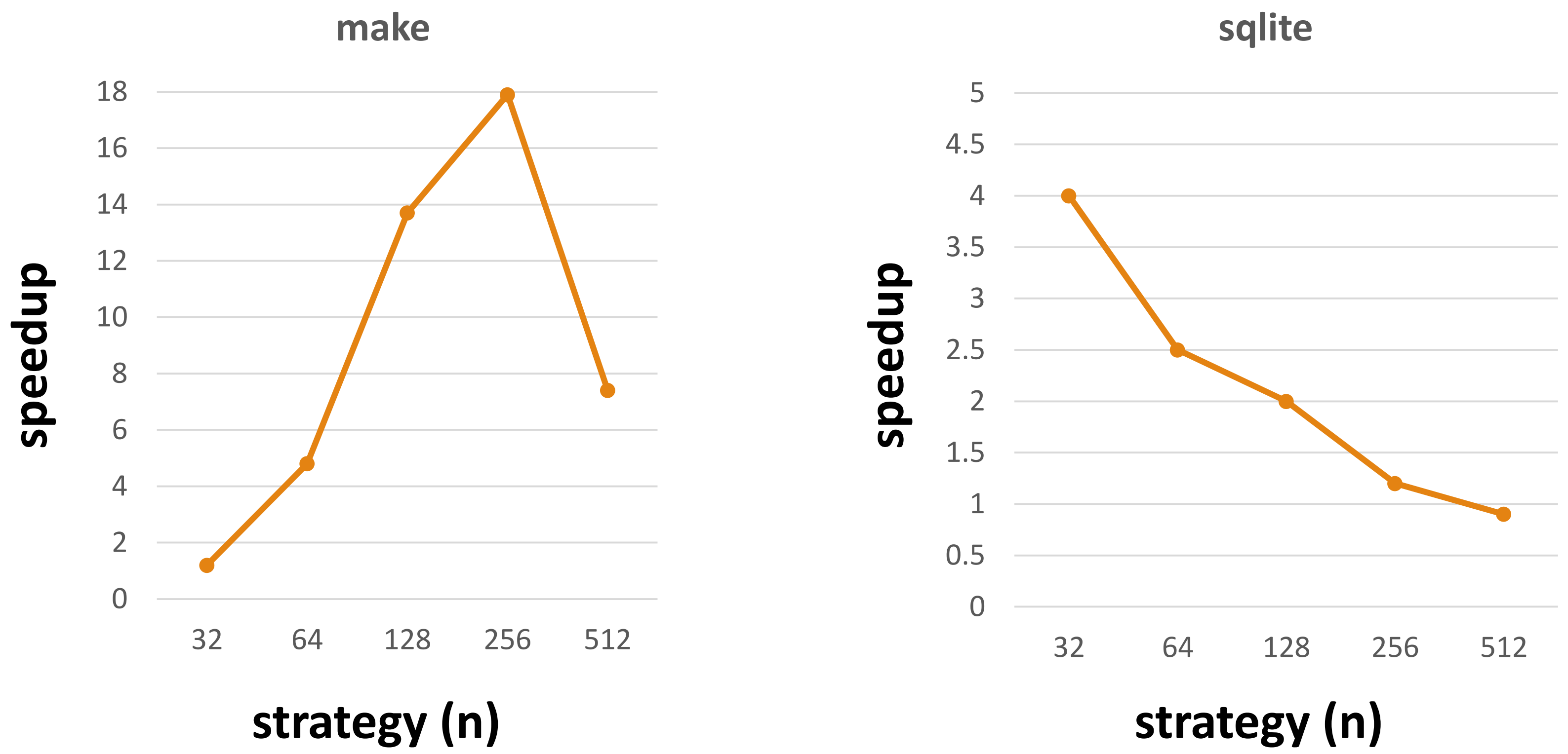
- We use evaluate different splitting strategies
  - $\boldsymbol{S_n}$: a strategy that splits an object to smaller objects of size $\boldsymbol{n}$
  - We use the several values for $\boldsymbol{n}$**:** 32, 64, 128, 256, 512
- We check the speedup in termination time w.r.t. vanilla KLEE

Evaluation: Splitting

- We use evaluate different splitting strategies
  - $\boldsymbol{S_n}$: a strategy that splits an object to smaller objects of size $\boldsymbol{n}$
  - We use the several values for $\boldsymbol{n}$**:** 32, 64, 128, 256, 512
- We check the speedup in termination time w.r.t. vanilla KLEE

Results:

- Significant speedup with most configurations


make
speedup
18
16
14
12
10
8
6
4
2
0
32
64
128
256
512
strategy (n)
sqlite
speedup
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
32
64
128
256
512
strategy (n)

Future Work

- Applying merging and splitting simultaneously
- Predicting when merging or splitting is likely to pay off
- Designing more sophisticated splitting strategies

Questions?

Project page: https://davidtr1037.github.io/ram/
Code available on github: https://github.com/davidtr1037/klee-ram