Relocatable Addressing Model for Symbolic Execution

David Trabish and Noam Rinetzky
Tel-Aviv University, Israel
KLEE Workshop 2021

In this talk

Tackle two challenges:

- **Path explosion** due to symbolic pointers
- **Constraint solving** of array theory constraints

In this talk

Tackle two challenges:

- **Path explosion** due to symbolic pointers
- **Constraint solving** of array theory constraints

Using a new addressing model: **Relocatable Addressing Model**

Challenge 1:
Symbolic Pointers

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...
```

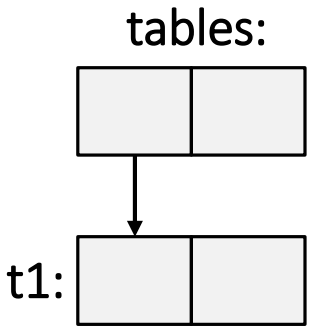
Symbolic Pointers

```
#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:

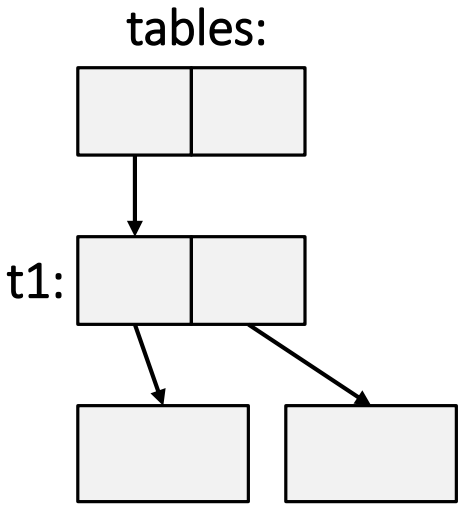
Symbolic Pointers

```
#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...
```



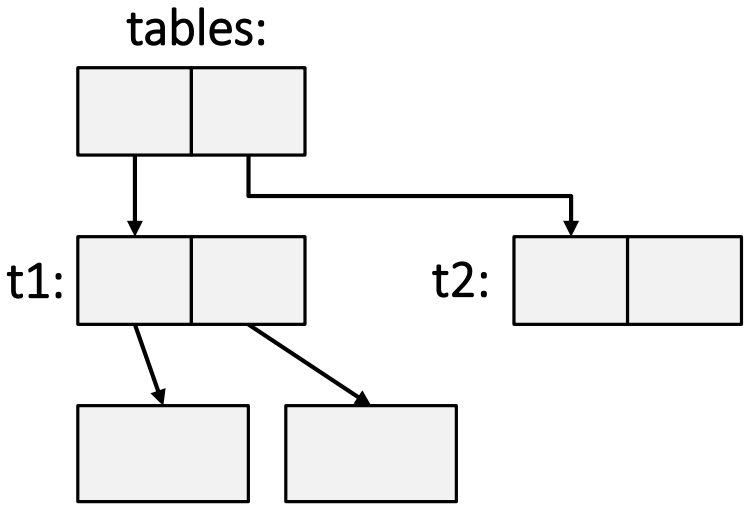
Symbolic Pointers

```
#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...
```



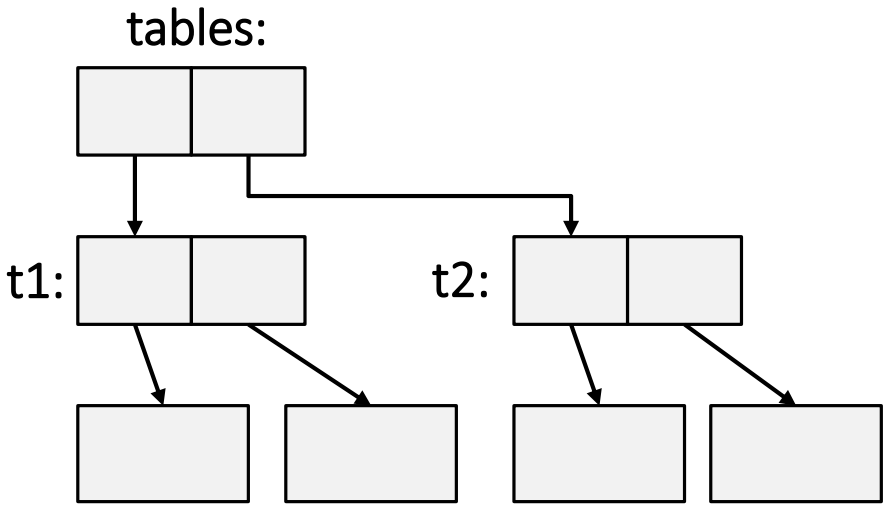
Symbolic Pointers

```
#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...
```



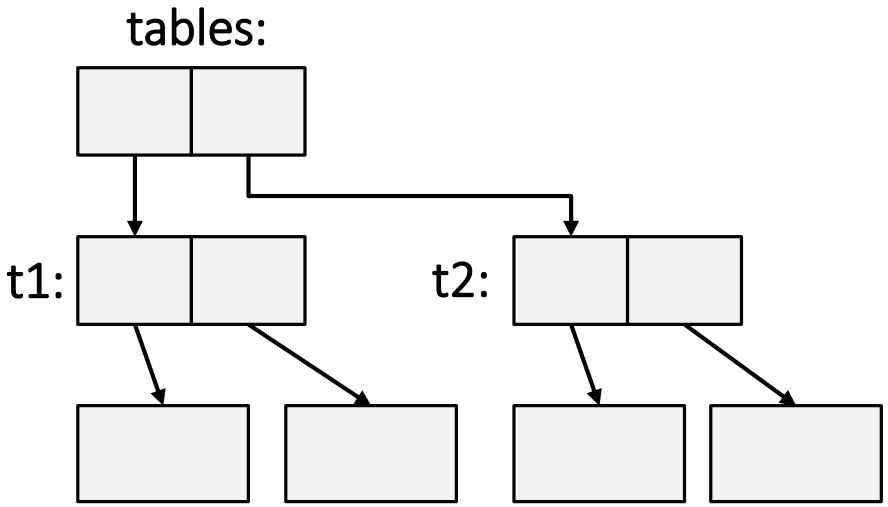
Symbolic Pointers

```
#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...
```



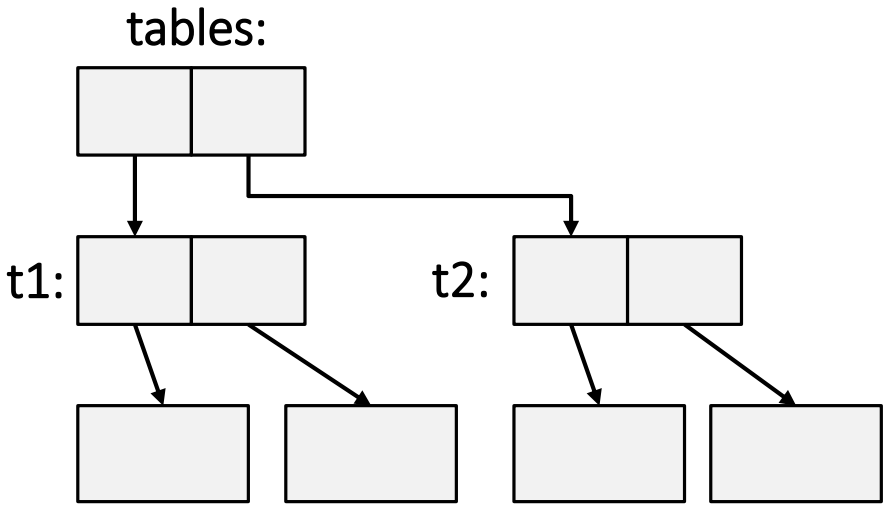
Symbolic Pointers

```
#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...
```



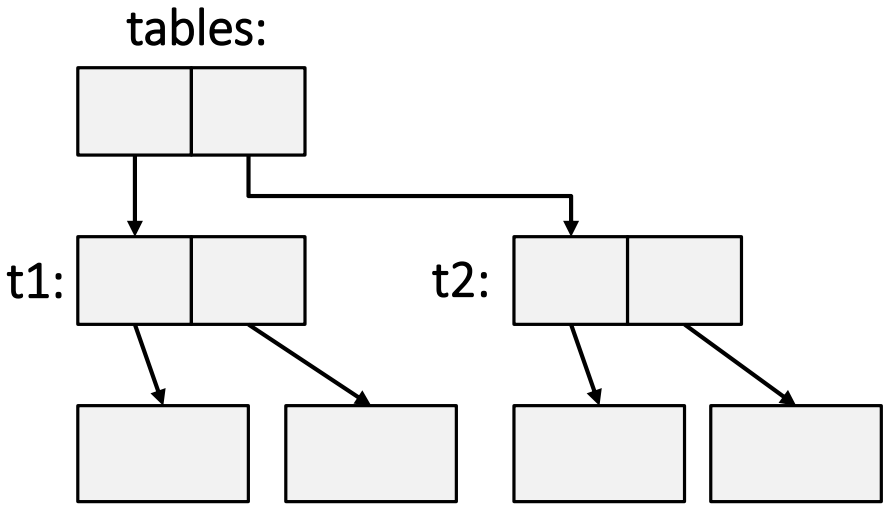
Symbolic Pointers

```
#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

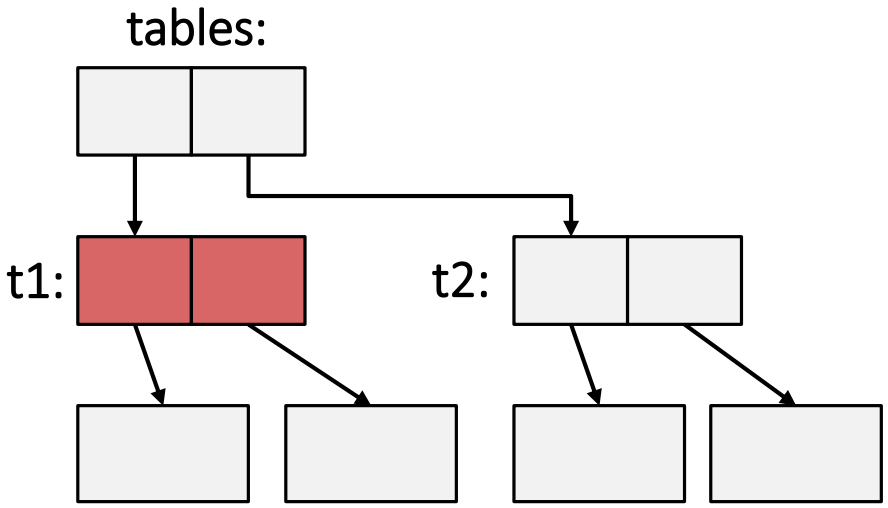
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



$$addr_{t1} + i$$

Symbolic Pointers

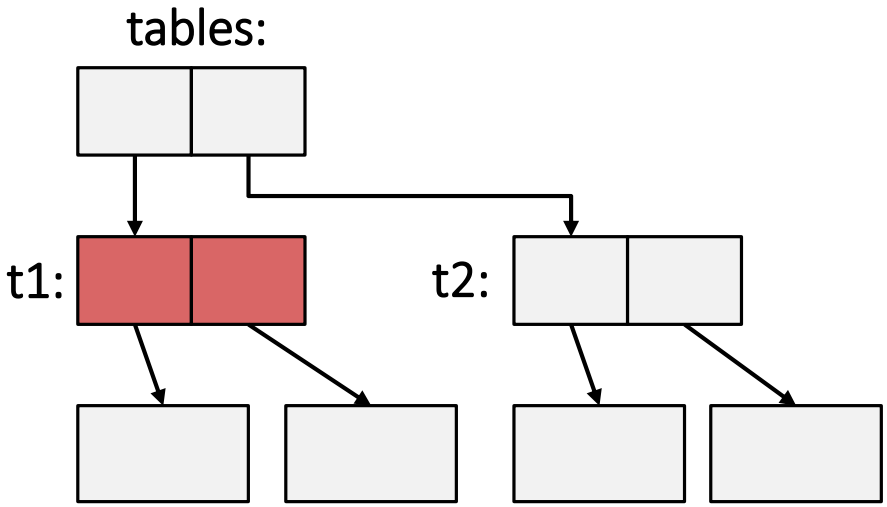
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



$$addr_{t1} + i$$

Symbolic Pointers

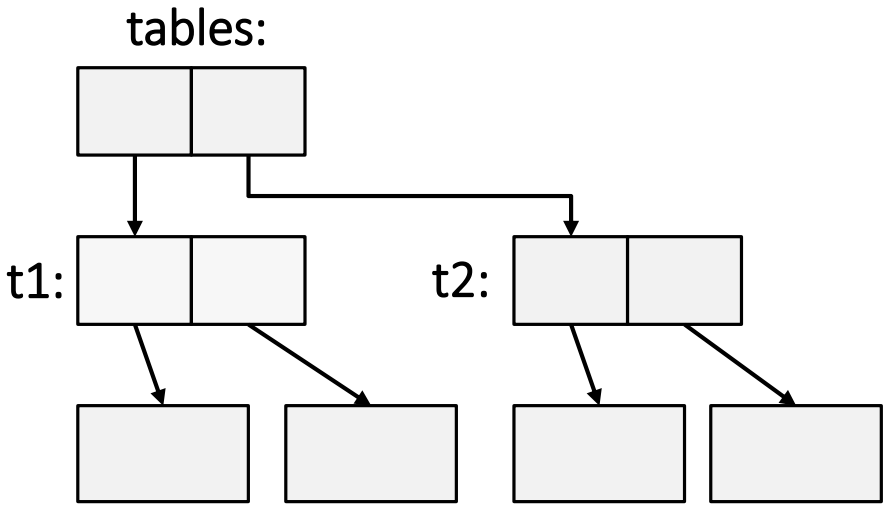
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



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

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...
```

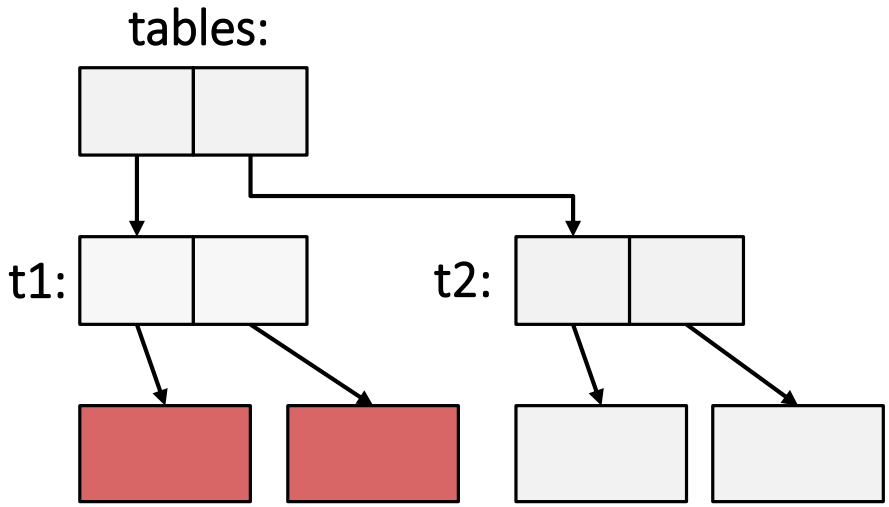


$$addr_{t1} + i \rightarrow select(arr_{t1}, i)$$

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

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...
```

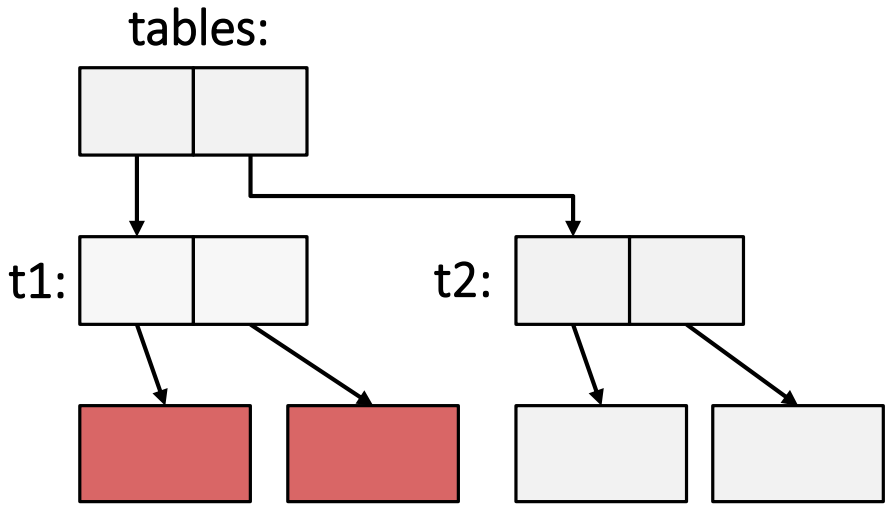


$$addr_{t1} + i \rightarrow select(arr_{t1}, i)$$

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

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...
```



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

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

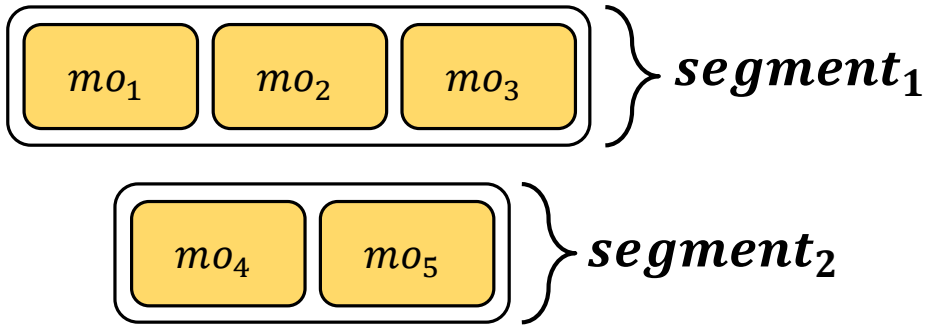
Symbolic Pointers

How can we handle symbolic pointers?

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

Segmented Memory Model

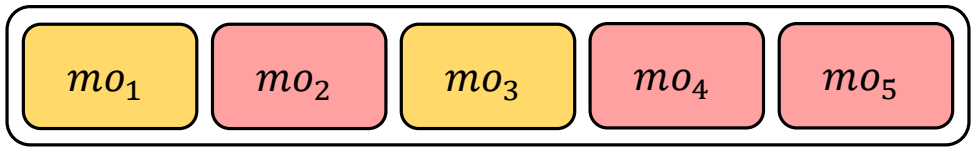
- Partitions the memory into segments using **static pointer analysis**
- Any pointer is guaranteed to be resolved to a single segment
- **Forks are avoided** on symbolic pointer resolution



Segmented Memory Model

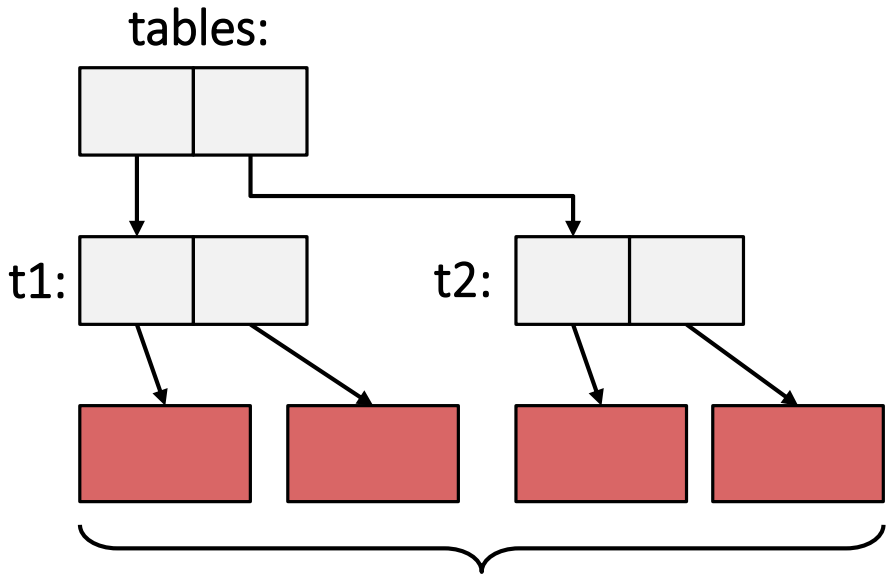
Limitations:

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



Segmented Memory Model

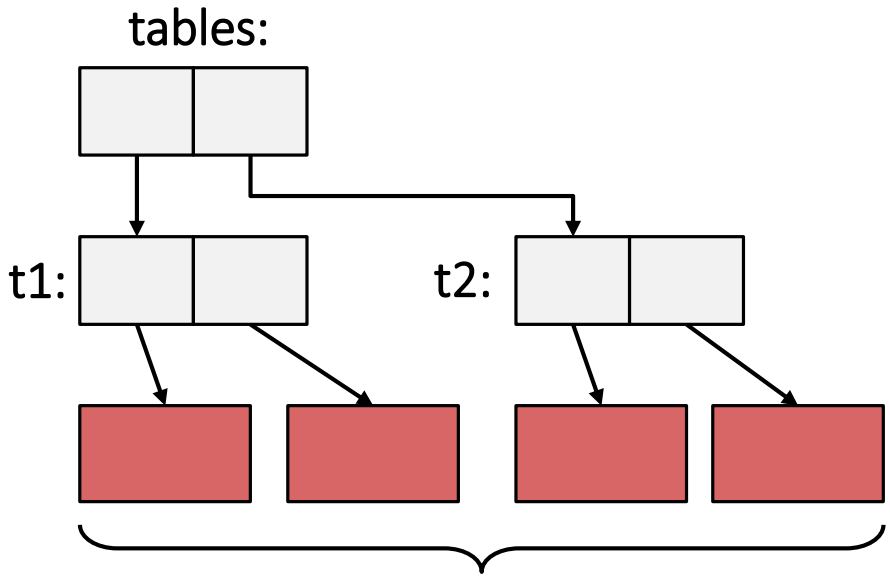
```
#define N 2
#define T 2
char **tables[T];
for (t = 0; t < T; t++) {
 tables[t] = calloc(N, PTR_SIZE);
 for (k = 0; k < N; k++)
  tables[t][k] = calloc(256, 1);
}
unsigned i,j; // i < N, j < 100
if (tables[0][i][j] == 7)
 // do something...
```



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...
```

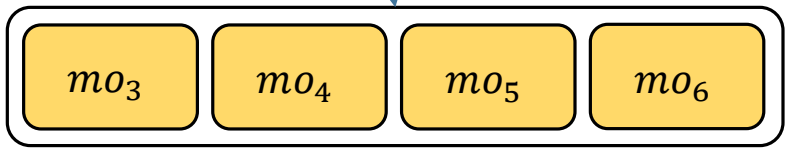


pointer analysis can't distinguish
mapped to the same segment

Segmented Memory Model

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

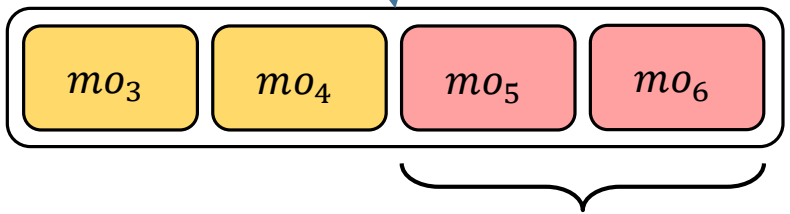
- Forking is avoided ✓



Segmented Memory Model

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

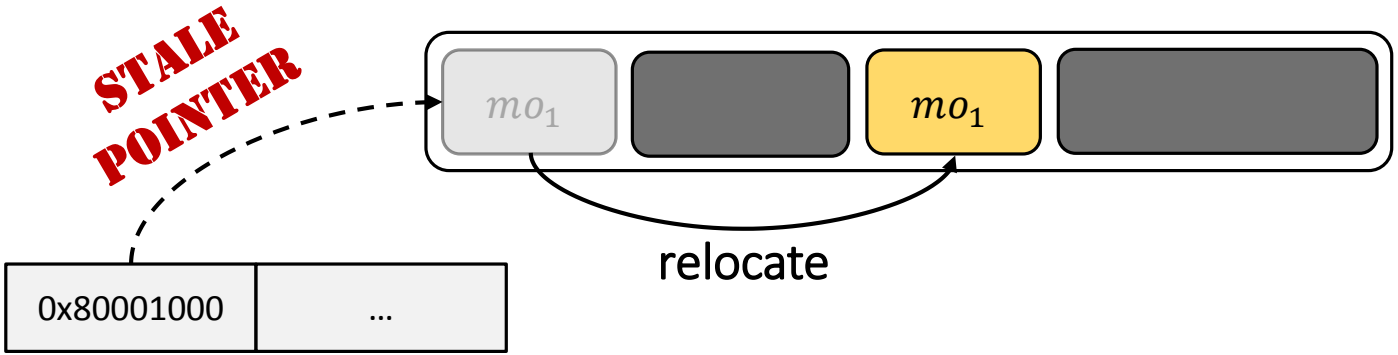
- Forking is avoided ✓
- Unnecessarily large segment ✘
- Affects constraint solving ✘



not pointed by the symbolic pointer

Goal

- Create the segments **on-the-fly**
- **Not supported** with the current addressing model
- Relocating an object is **tricky**, requires:
  - Updating all its references
  - Precise type information



Relocatable Addressing Model

A new model:

- Base addresses are **symbolic** (and not concrete) values
- Use additional **address constraints**
  - Preserves the *non-overlapping* property

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...
```

memory

- tables: $[\alpha_1, \alpha_2]$
- tables[0]: $[\alpha_3, \alpha_4]$
- tables[1]: $[\alpha_5, \alpha_6]$

address constraints

- $\alpha_1 = 0x80001000$
- $\alpha_2 = 0x80001100$
- $\alpha_3 = 0x80001200$
- $\alpha_4 = 0x80001300$
- $\alpha_5 = \cdots$
- $\alpha_6 = \cdots$

Dynamically Segmented Memory Model

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

memory

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)
...
```

memory

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)
...
```

memory

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)
...
```

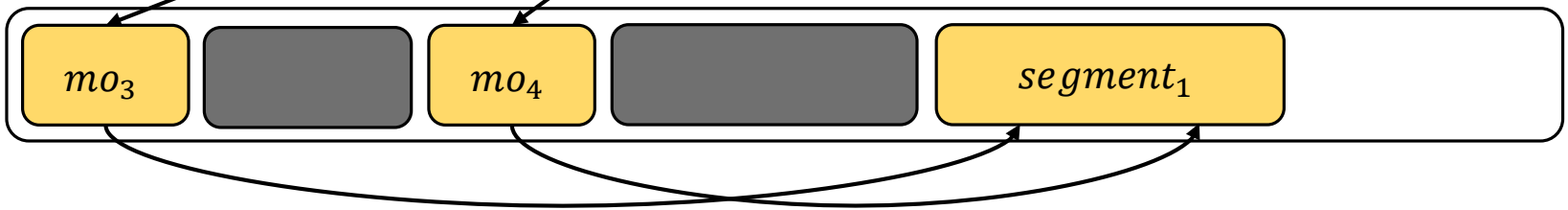
memory

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)
...
```

memory

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)
...
```

memory

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)
...
```

- Avoiding forks ✓
- Smaller segments ✓

Challenge 2:
Constraint Solving

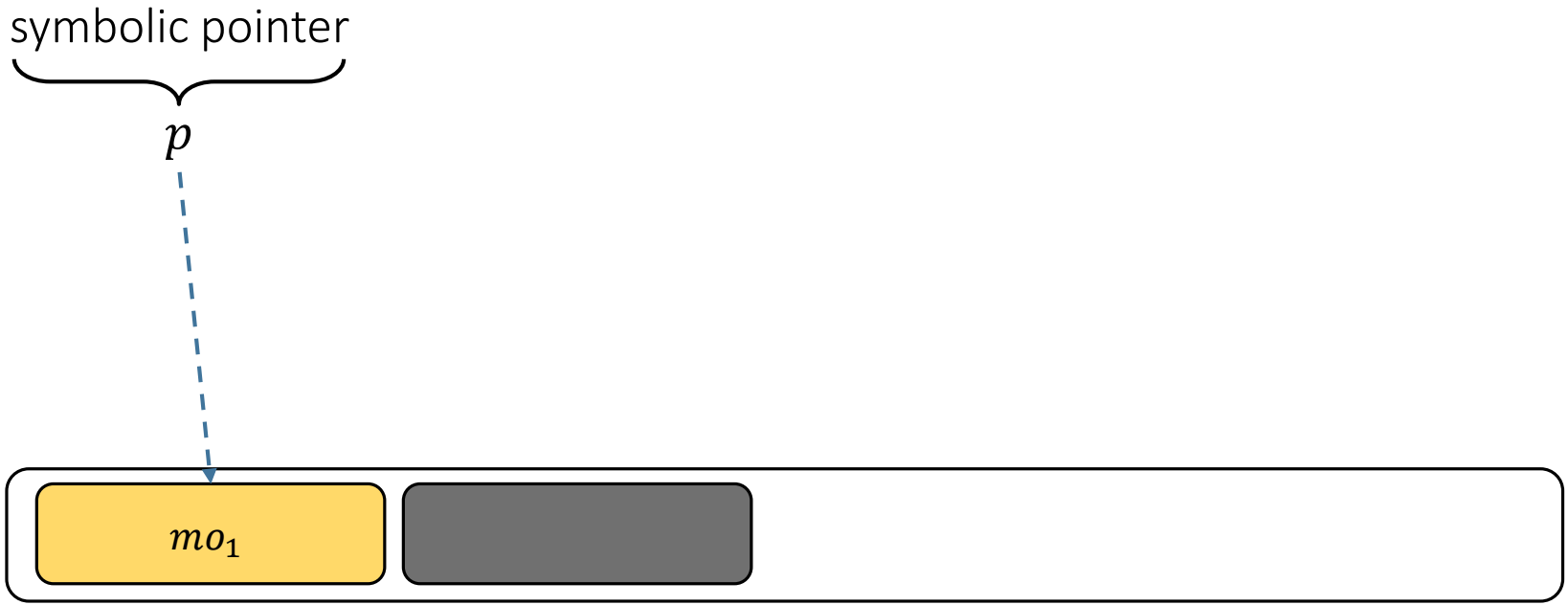
Constraint Solving

Solving array theory constraints is **expensive**

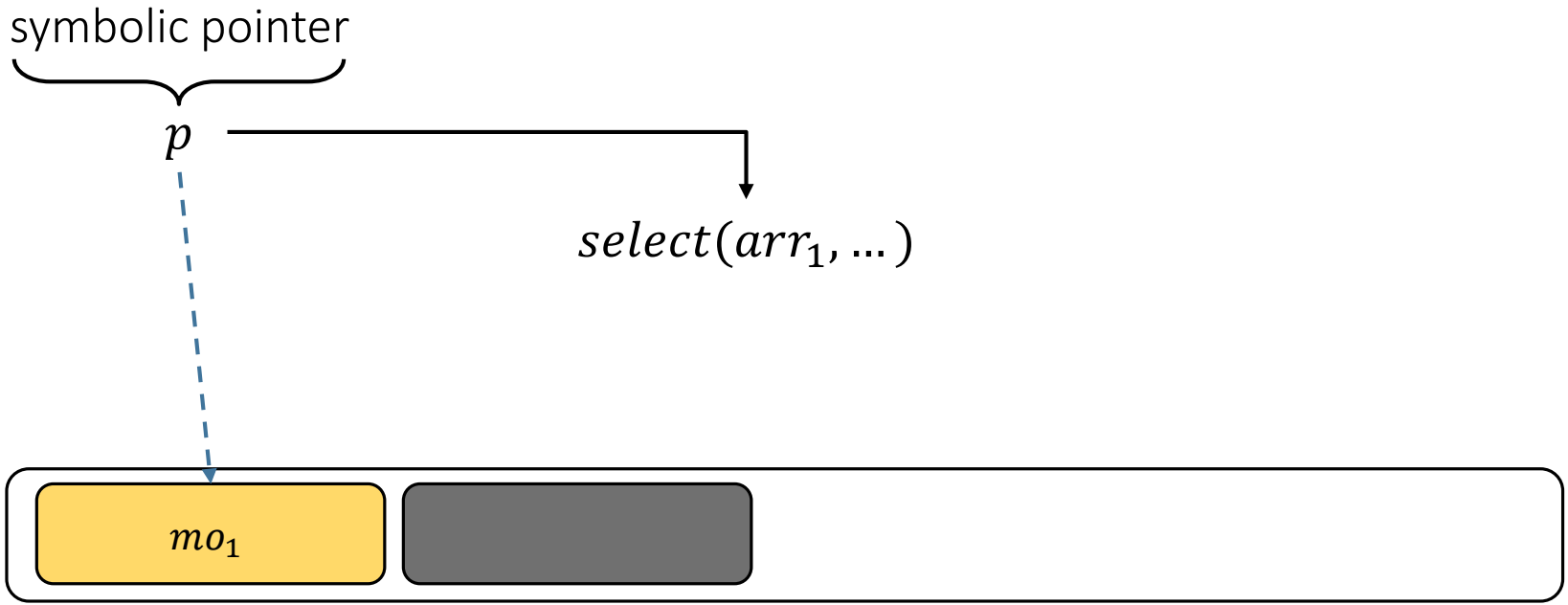
- Especially when arrays are big (many *store*'s)

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

Dynamically Splitting Objects



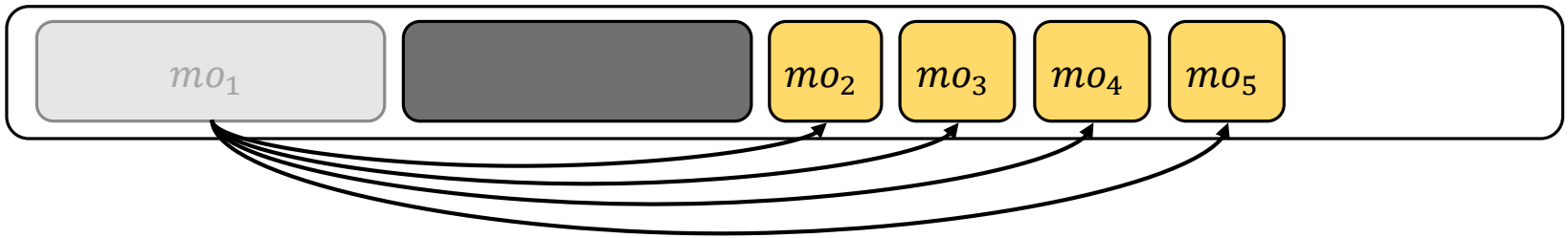
Dynamically Splitting Objects



Dynamically Splitting Objects

symbolic pointer

p

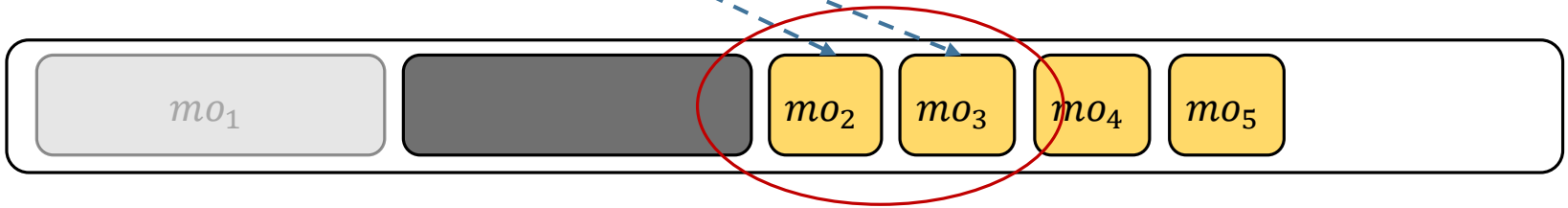


Dynamically Splitting Objects

symbolic pointer

p

- More forks ✘
- Smaller SMT arrays ✓



Evaluation

Evaluated in the context of:

- Merging (inter-object partitioning)
- Splitting (intra-object partitioning)

Evaluation: Merging

Compare the sizes of created segments between:

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

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

Evaluation: Merging

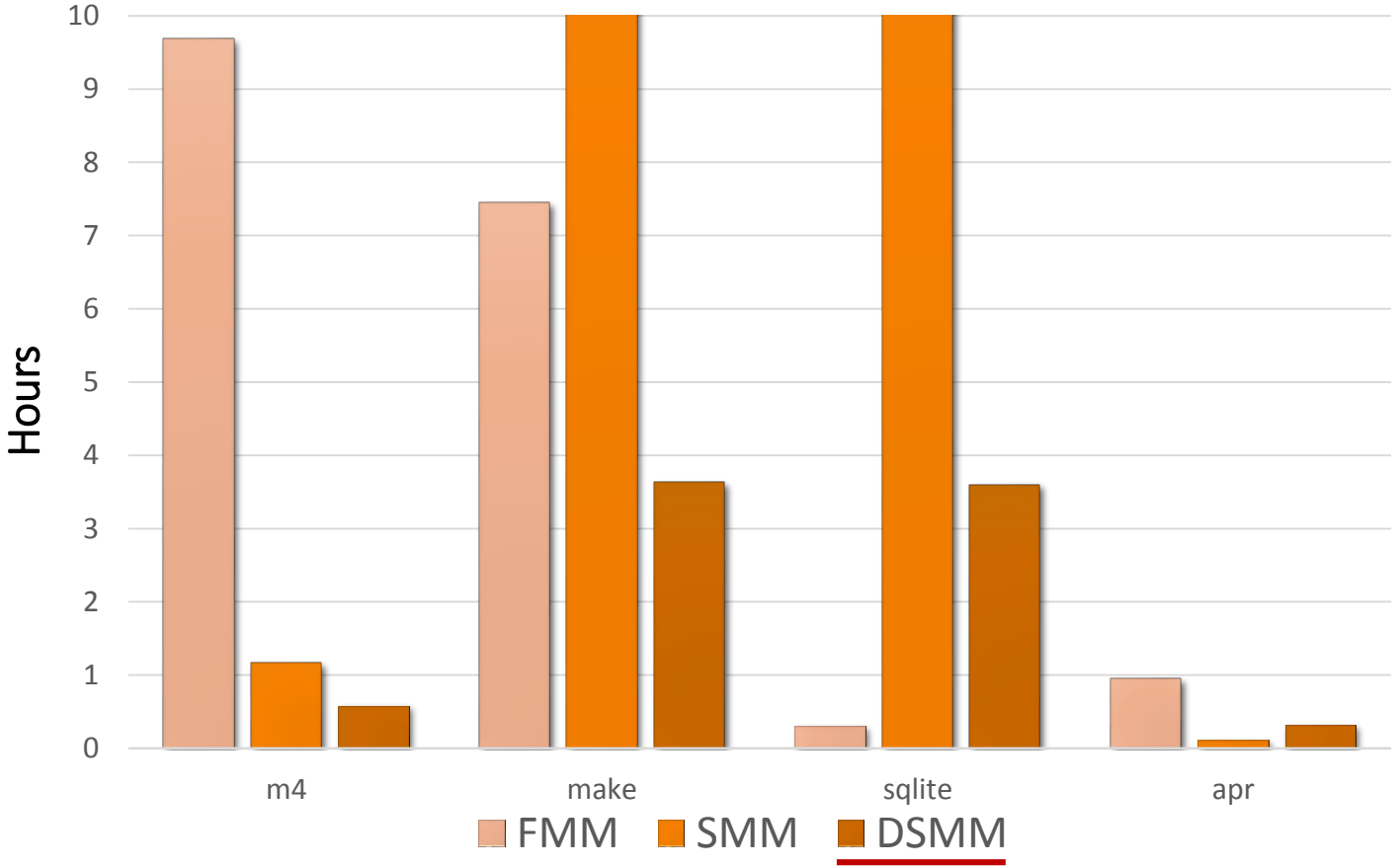
Compare the performance between:

- FMM *(Forking memory model of vanilla KLEE)*
- SMM *(Segmented memory model)*
- DSMM *(Dynamically segmented memory model)*

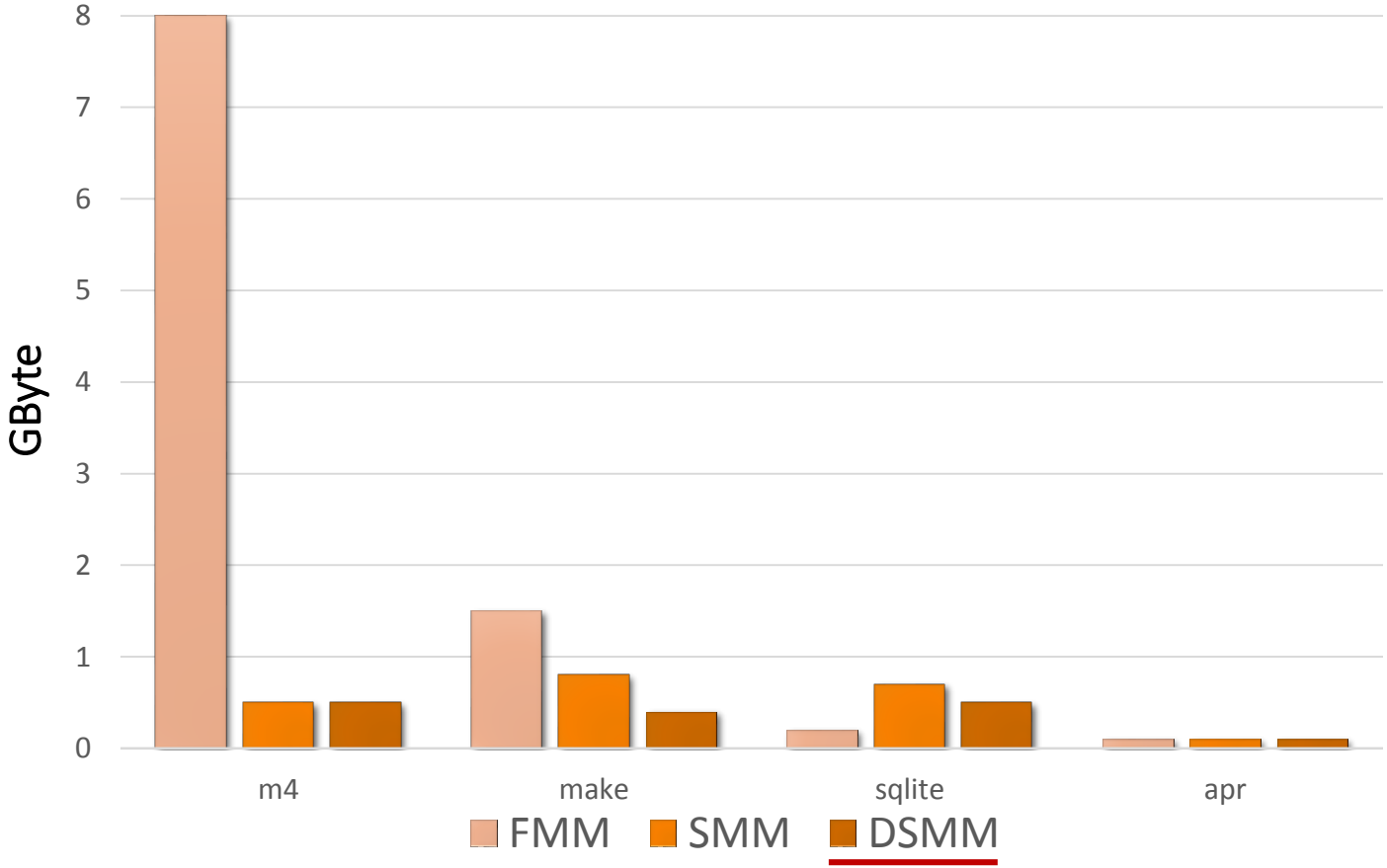
Run with a timeout of 24 hours and check:

- Termination time (until full exploration)
- Memory usage

Termination Time

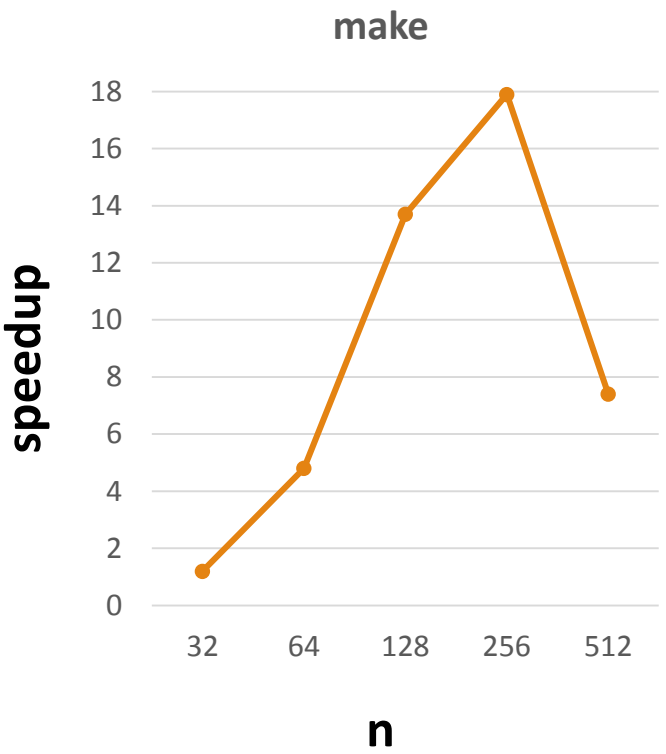


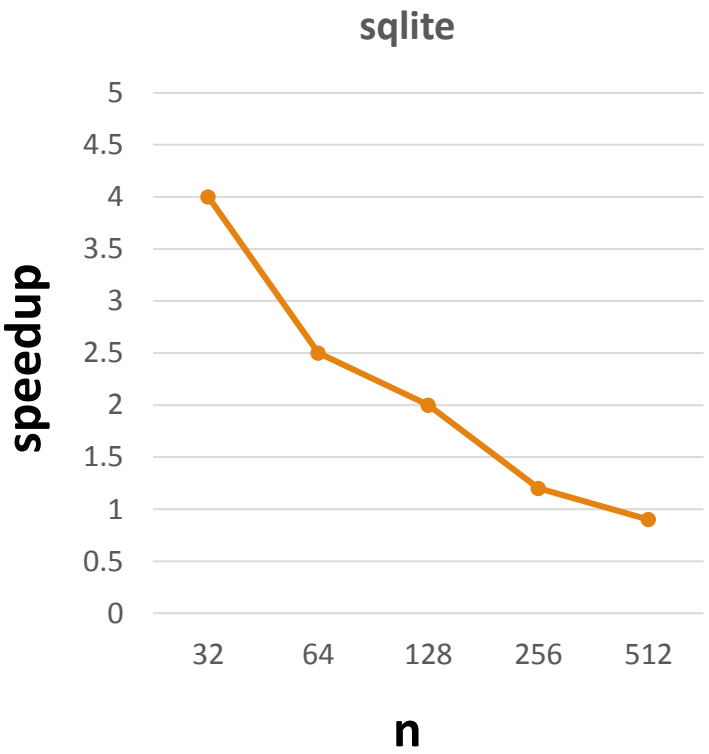
Memory Usage



Evaluation: Splitting

- Split an object to smaller objects of size $\boldsymbol{n}$
  - Use several values for $\boldsymbol{n}$**:** 32, 64, 128, 256, 512
- Check the termination time w.r.t. vanilla KLEE


make
speedup
18
16
14
12
10
8
6
4
2
0
32
64
128
256
512
n


sqlite
speedup
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
32
64
128
256
512
n

Applications & Future Work

Applications

- Improving query caching
  - Address-Aware Query Caching for Symbolic Execution (ICST 2021)

Future work:

- Predicting when merging or splitting is likely to pay off
- Hybrid segmented memory model

Questions?

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