Novel Memory Models for Symbolic Execution

by David Trabish
Supervisor: Prof. Noam Rinetzky

TEL AVIV UNIVERSITY

Symbolic Execution

Program analysis technique

- Systematically explores paths
- Checks feasibility using SMT

Applications:

- Test case generation
- Bug finding
- …

Today’s Talk

Challenges	Path explosion Constraint solving False negatives
Our Attack	Novel memory models

Outline

- Background
    - Symbolic execution
    - Memory model
- Symbolic base addresses
    - Relocatable memory model
    - Address-aware query caching
- Symbolic-size allocations
    - Bounded symbolic-size model
    - State merging with quantifiers
- Conclusions and future work

Outline

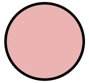
- Background
    - Symbolic execution
    - Memory model
- Symbolic base addresses
    - Relocatable memory model
    - Address-aware query caching
- Symbolic-size allocations
    - Bounded symbolic-size model
    - State merging with quantifiers
- Conclusions and future work

Symbolic Execution: Example

```
int get_sign(int x) {
  if (x == 0) {
    return 0;
  }

  if (x < 0) {
    return -1;
  } else {
    return 1;
  }
}
```

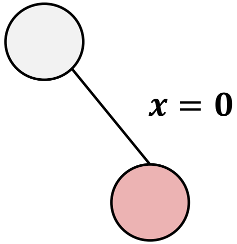
Symbolic Execution: Example

```
int get_sign(int x) {
  if (x == 0) {
    return 0;
  }

  if (x < 0) {
    return -1;
  } else {
    return 1;
  }
}
```



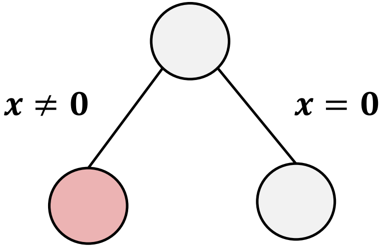
Symbolic Execution: Example

```
int get_sign(int x) {
  if (x == 0) {
    return 0;
  }

  if (x < 0) {
    return -1;
  } else {
    return 1;
  }
}
```



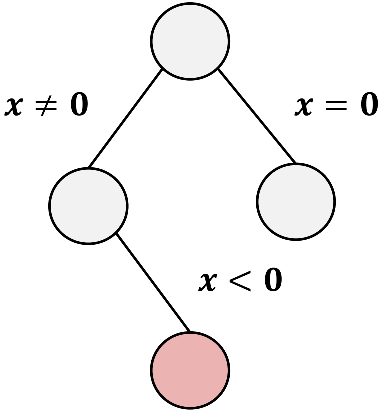
Symbolic Execution: Example

```
int get_sign(int x) {
  if (x == 0) {
    return 0;
  }

  if (x < 0) {
    return -1;
  } else {
    return 1;
  }
}
```



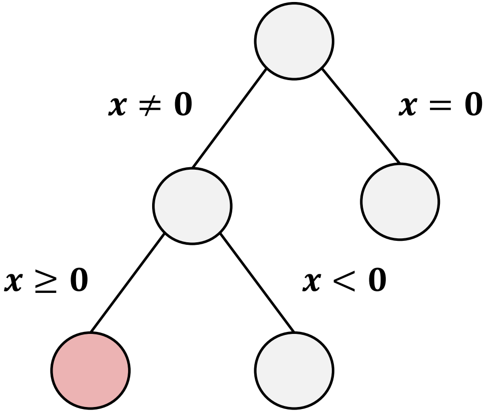
Symbolic Execution: Example

```
int get_sign(int x) {
  if (x == 0) {
    return 0;
  }

  if (x < 0) {
    return -1;
  } else {
    return 1;
  }
}
```



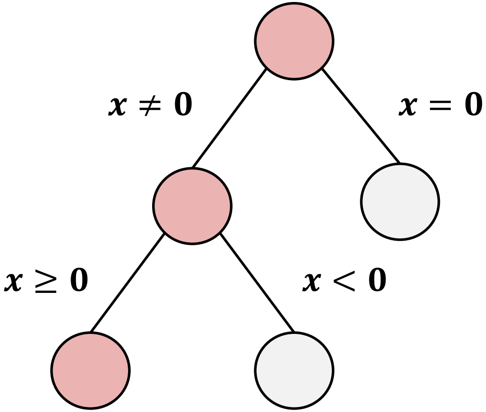
Symbolic Execution: Example

```
int get_sign(int x) {
  if (x == 0) {
    return 0;
  }

  if (x < 0) {
    return -1;
  } else {
    return 1;
  }
}
```



Symbolic Execution: Example

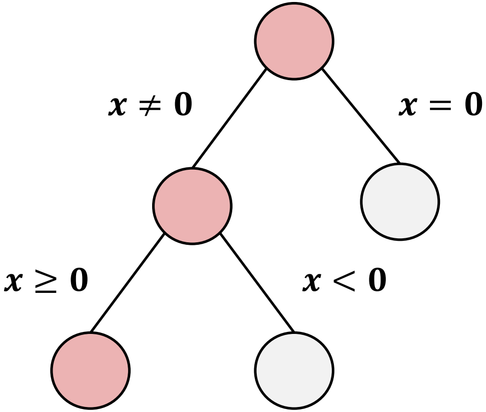
```
int get_sign(int x) {
  if (x == 0) {
    return 0;
  }

  if (x < 0) {
    return -1;
  } else {
    return 1;
  }
}
```



$x \neq 0 \land x \geq 0$

Symbolic Execution: Example

```
int get_sign(int x) {
  if (x == 0) {
    return 0;
  }

  if (x < 0) {
    return -1;
  } else {
    return 1;
  }
}
```



$x \neq 0 \wedge x \geq 0$

$\Downarrow$ SMT

$x \mapsto 7$

Outline

- Background
  - Symbolic execution
  - Memory model
- Symbolic base addresses
  - Relocatable memory model
  - Address-aware query caching
- Symbolic-size allocations
  - Bounded symbolic-size model
  - State merging with quantifiers
- Conclusions and future work

Standard Memory Model

Two main components:

- Memory objects
  - Integers, arrays, heap allocations, etc.
- Address space
  - Location of memory objects

Memory Objects

Defined by a tuple (b, s, a):

- Concrete base address
- Concrete size
- SMT array

Memory Objects

Reading at offset i from (b, s, a):

$$(i < 0 \lor i \geq s) \quad \xRightarrow{SAT} \quad \text{out-of-bounds error}$$

Memory Objects

Reading at offset i from (b, s, a):

$$(i < 0 \lor i \geq s) \xRightarrow{\mathit{UNSAT}} \underbrace{select(a, i)}_{\text{read value}}$$

Memory Objects

Writing v at offset i in (b, s, a):

$$(i < 0 \ \lor \ i \geq s) \quad \xRightarrow{\mathit{SAT}} \quad \text{out-of-bounds error}$$

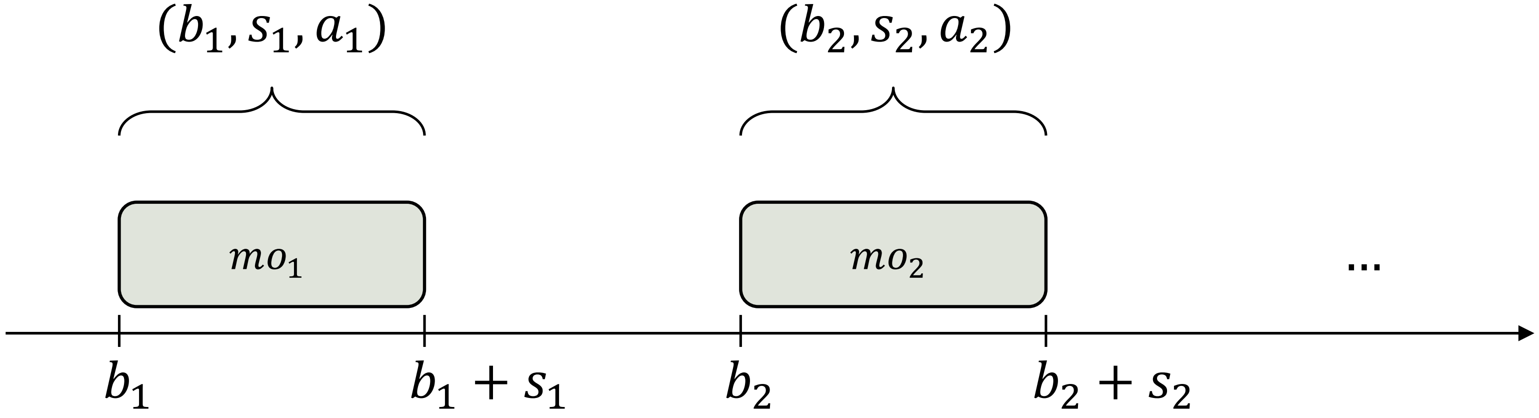
Memory Objects

Writing v at offset i in (b, s, a):

$$(i < 0 \lor i \geq s) \xRightarrow{\mathit{UNSAT}} \underbrace{a[i \mapsto v]}_{\text{updated SMT array}}$$

Address Space

- Linear space
- Disjoint intervals

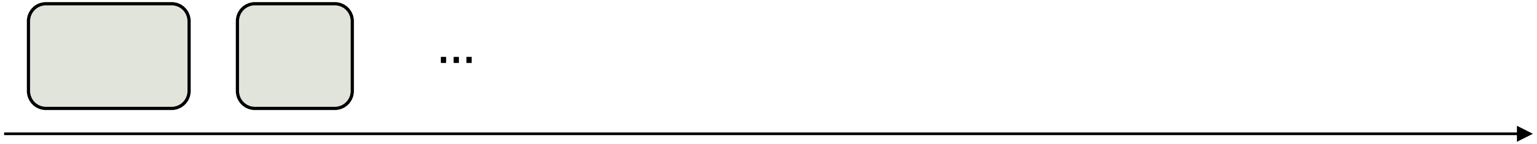


Memory Operations

- Allocation
- Dereference
- Deallocation

Memory Operations

Allocate n bytes



Memory Operations

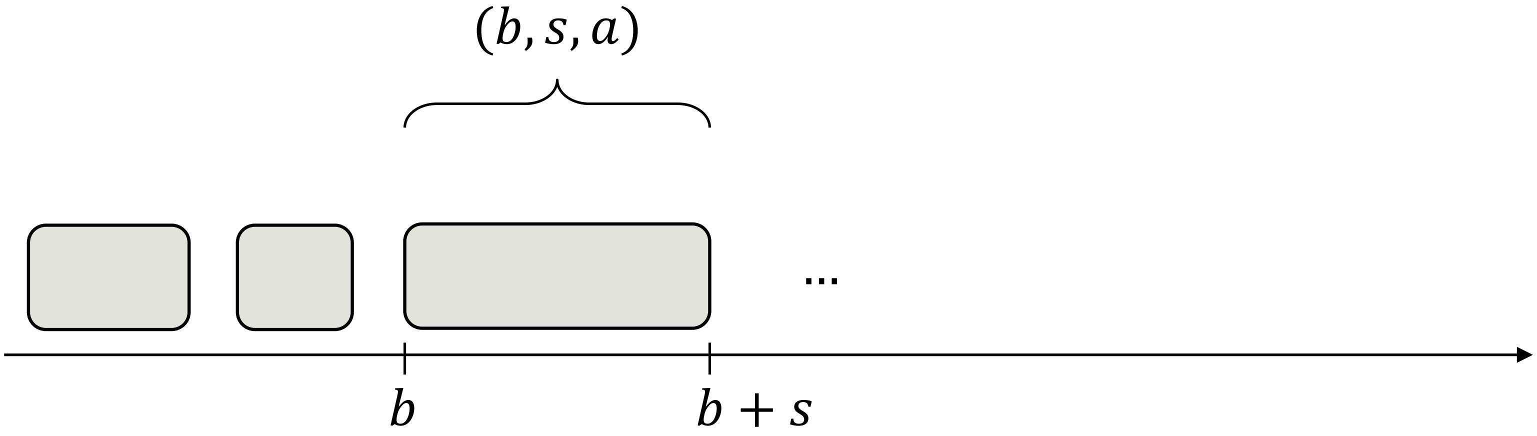
Allocate n bytes

concretize n to s

Memory Operations

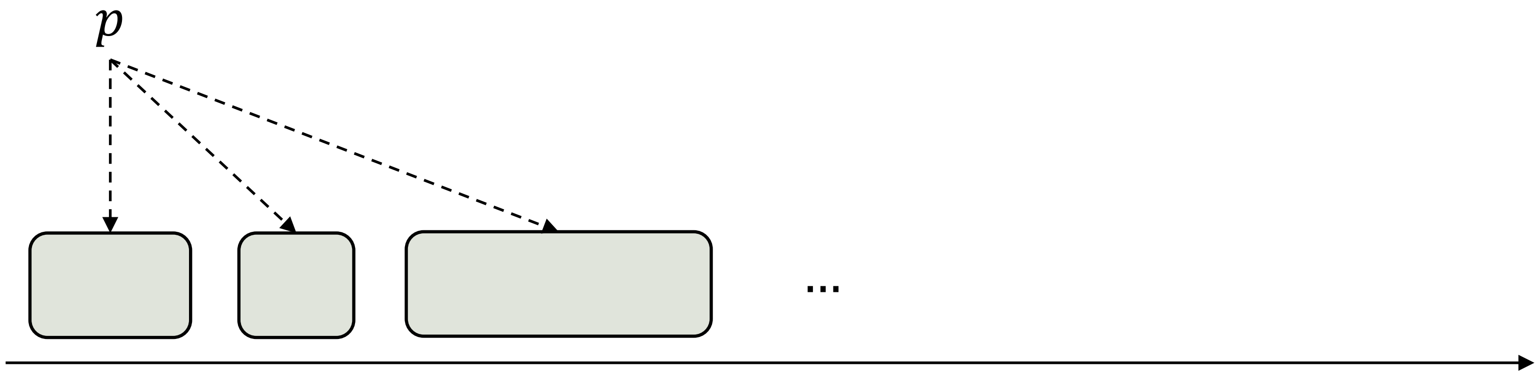
Allocate n bytes

concretize n to s



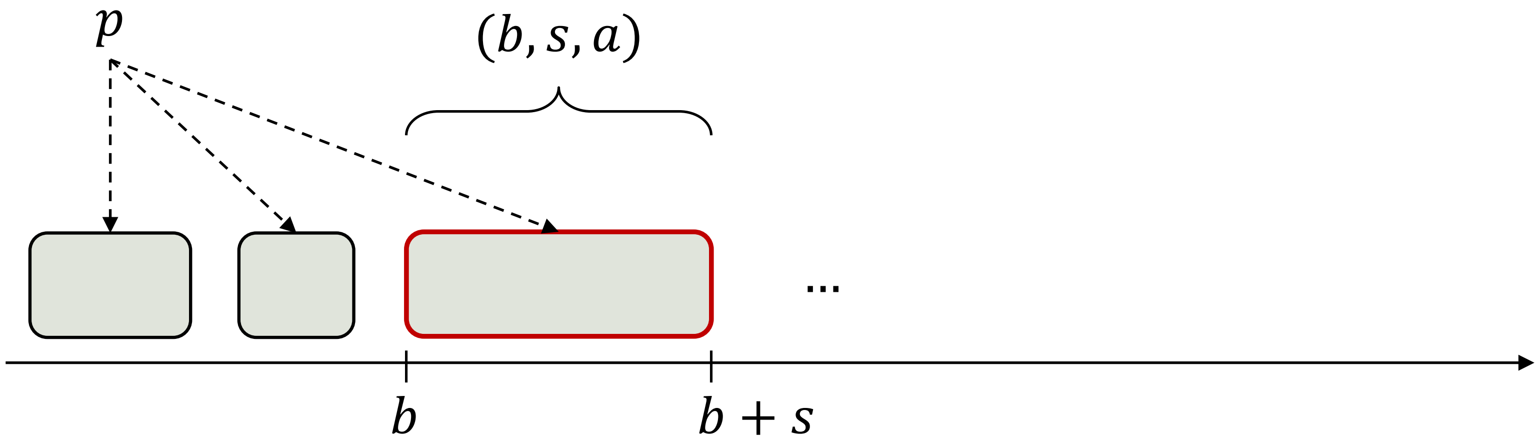
Memory Operations

Dereference p



Memory Operations

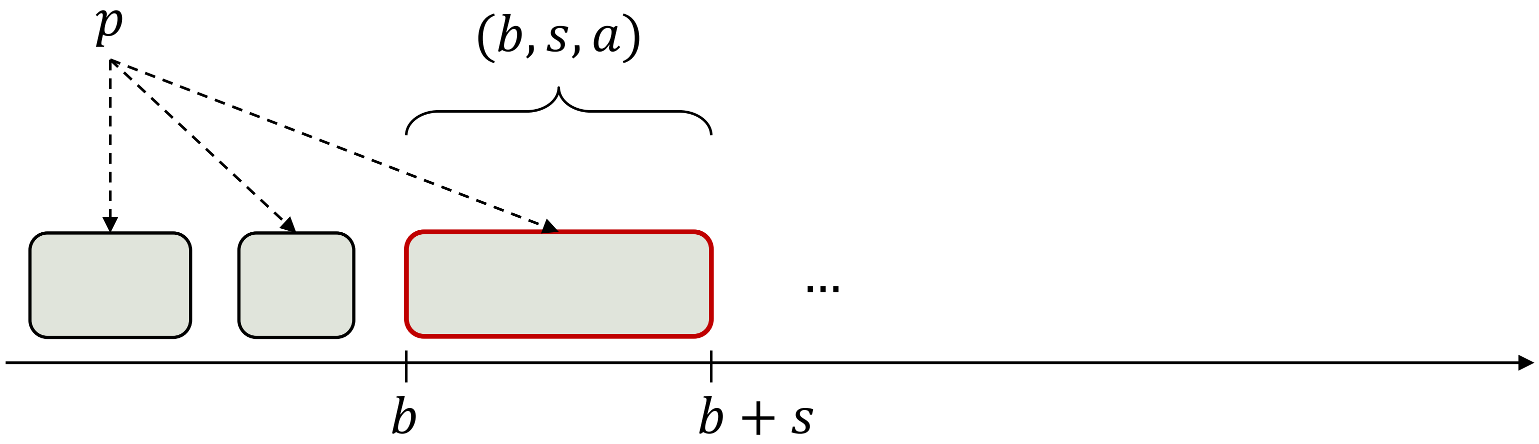
Dereference p



Memory Operations

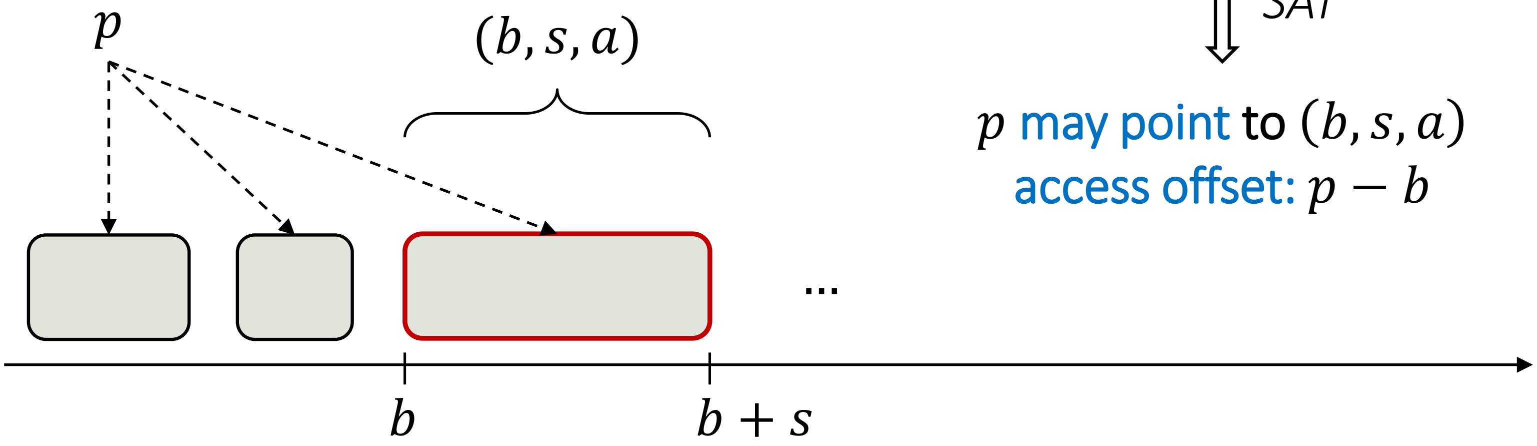
Dereference p

$$p \geq b \ \wedge \ p < b + s$$



Memory Operations

Dereference p



$$p \geq b \ \land \ p < b + s$$

$$\Big\Downarrow \textit{SAT}$$

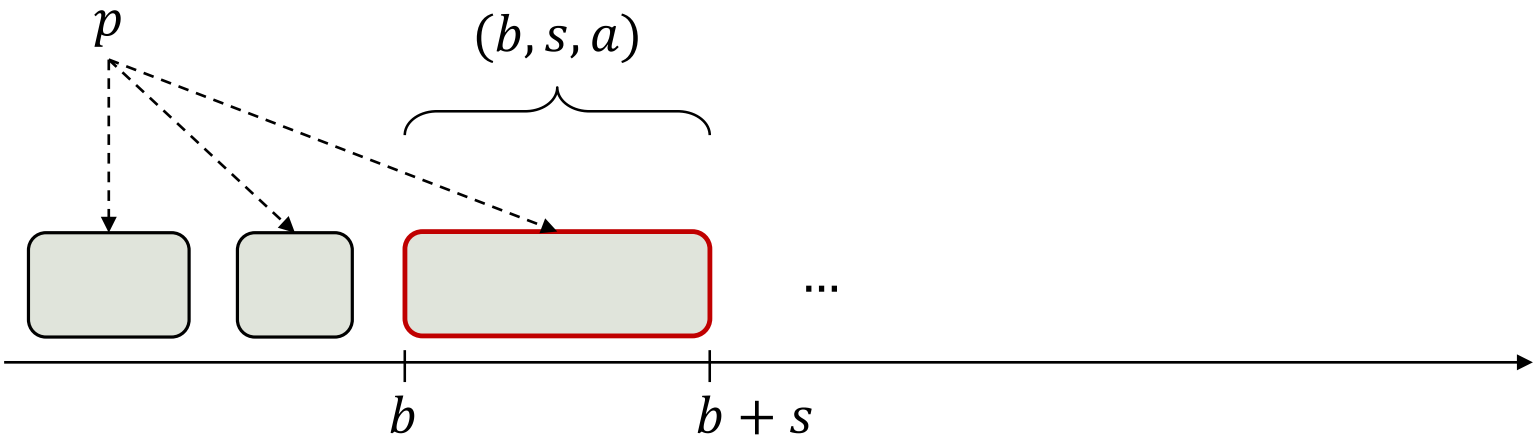
p may point to (b, s, a)

access offset: $p - b$

Memory Operations

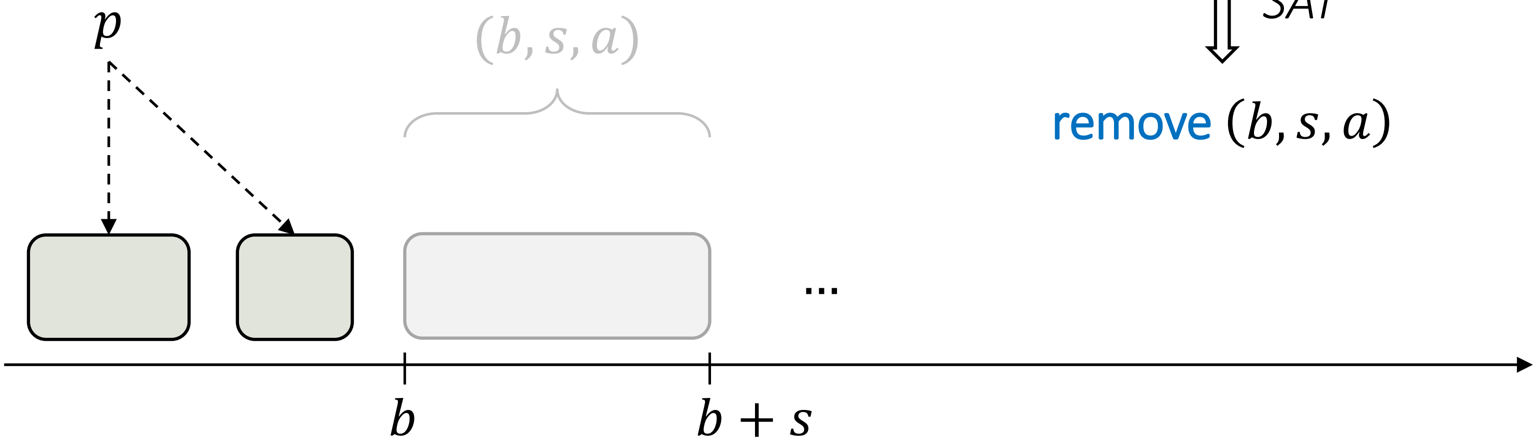
Deallocate p

$p = b$



Memory Operations

Deallocate p



$p = b$

$\Downarrow$ *SAT*

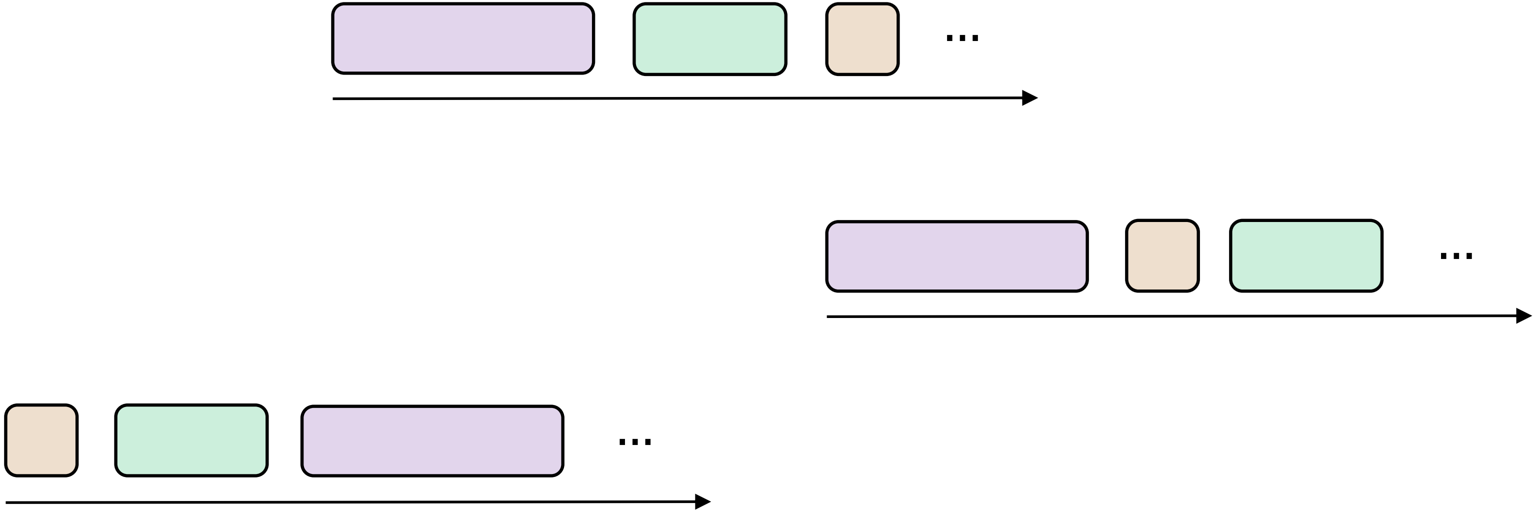
remove (b, s, a)

Outline

- Background
  - Symbolic execution
  - Memory model
- Symbolic base addresses
  - Relocatable memory model
  - Address-aware query caching
- Symbolic-size allocations
  - Bounded symbolic-size model
  - State merging with quantifiers
- Conclusions and future work

Observation

Specific address values don’t matter



Outline

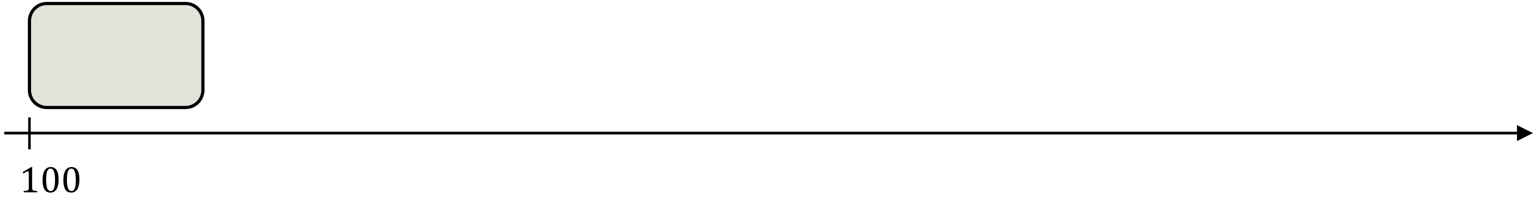
- Background
    - Symbolic execution
    - Memory model
- Symbolic base addresses
    - Relocatable memory model
    - Address-aware query caching
- Symbolic-size allocations
    - Bounded symbolic-size model
    - State merging with quantifiers
- Conclusions and future work

Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

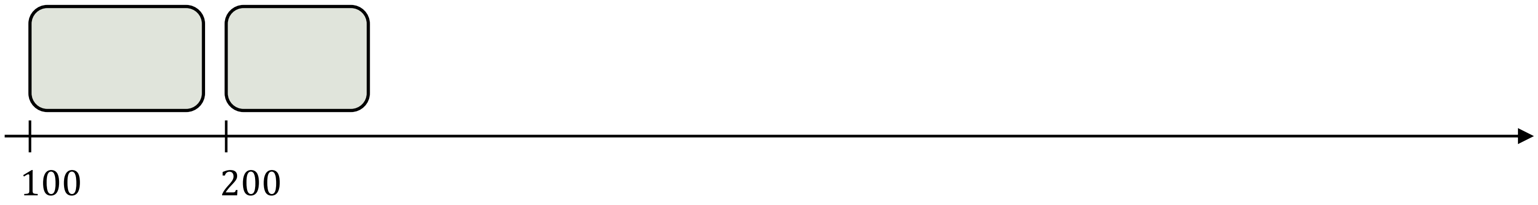
Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```



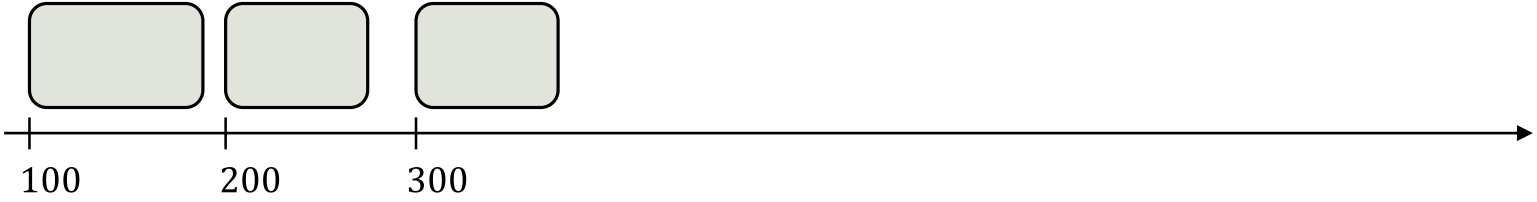
Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```



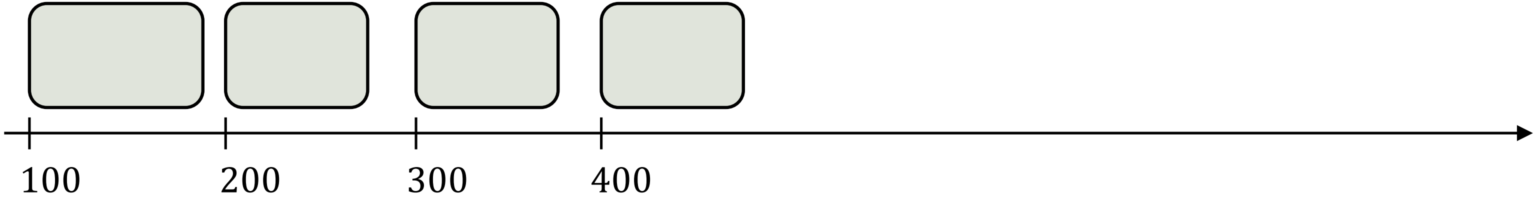
Symbolic Pointers

```c
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```



Symbolic Pointers

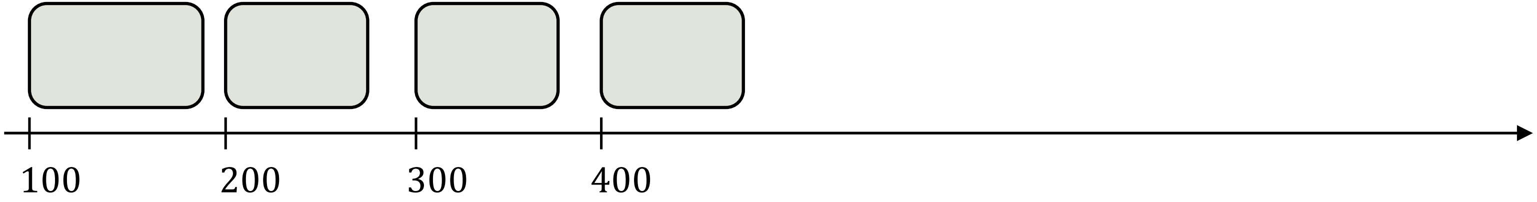
```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```



Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$p \overset{\text{def}}{=} 100 + i * 4$



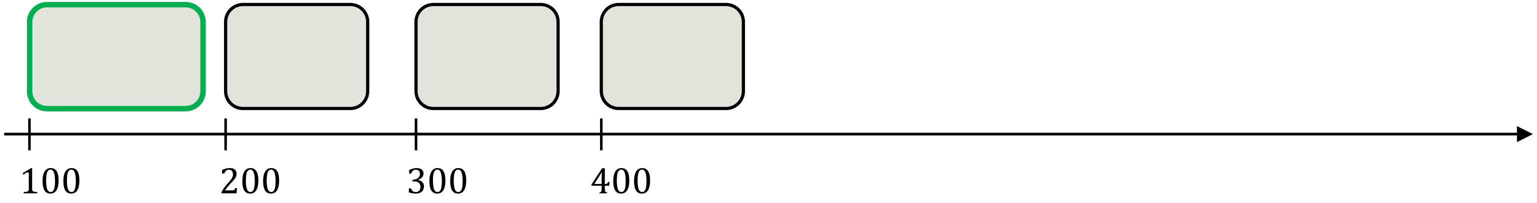
Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$p \overset{\text{def}}{=} 100 + i * 4$

$$\overbrace{i < 2 \ \land\ j < 10 \ \land\ 100 \leq p < 112}^{\text{resolution query}}$$

SAT



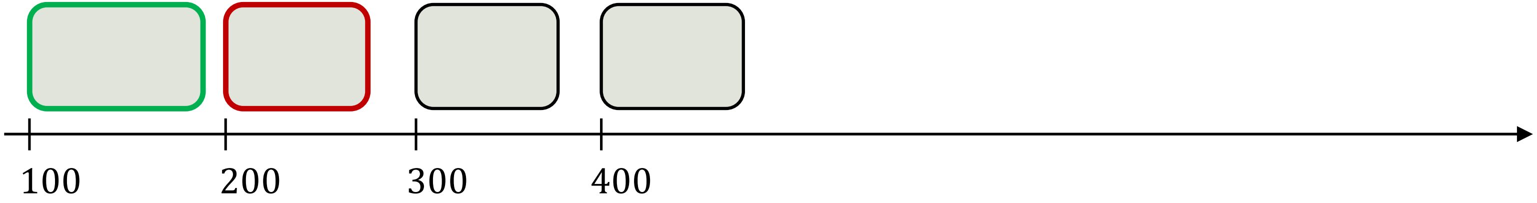
Symbolic Pointers

```c
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$p \overset{\text{def}}{=} 100 + i * 4$

$$\overbrace{i < 2 \ \wedge\ j < 10 \ \wedge\ 200 \leq p < 210}^{\text{resolution query}}$$

UNSAT



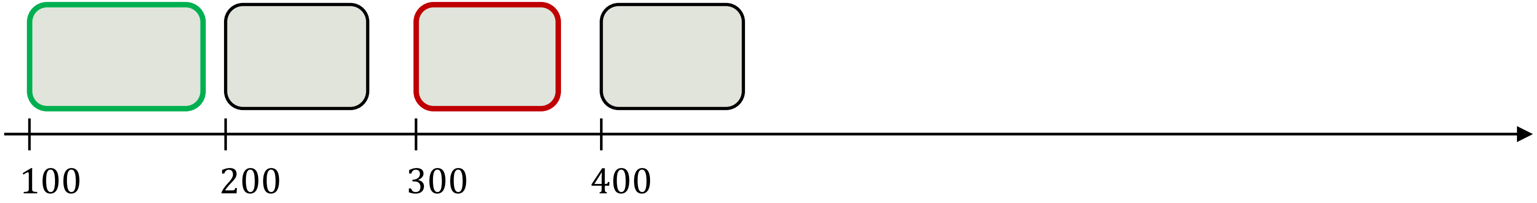
Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} 100 + i * 4$$

$$\overbrace{i < 2 \ \wedge \ j < 10 \ \wedge \ 300 \le p < 310}^{\text{resolution query}}$$

UNSAT



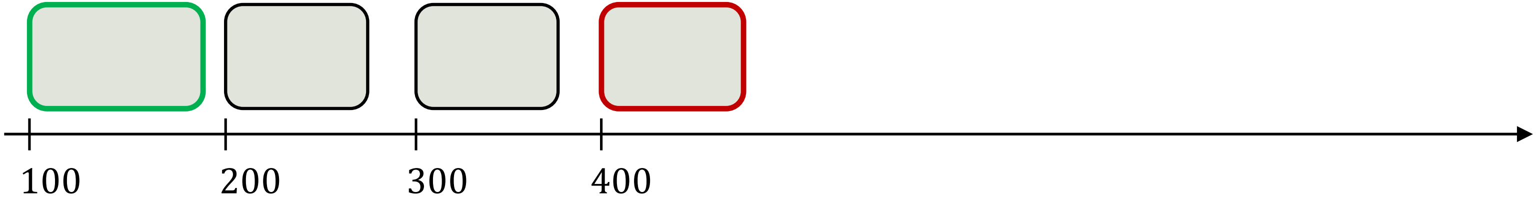
Symbolic Pointers

```c
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$p \overset{\text{def}}{=} 100 + i * 4$

$$\overbrace{i < 2 \ \wedge\ j < 10 \ \wedge\ 400 \leq p < 410}^{\text{resolution query}}$$

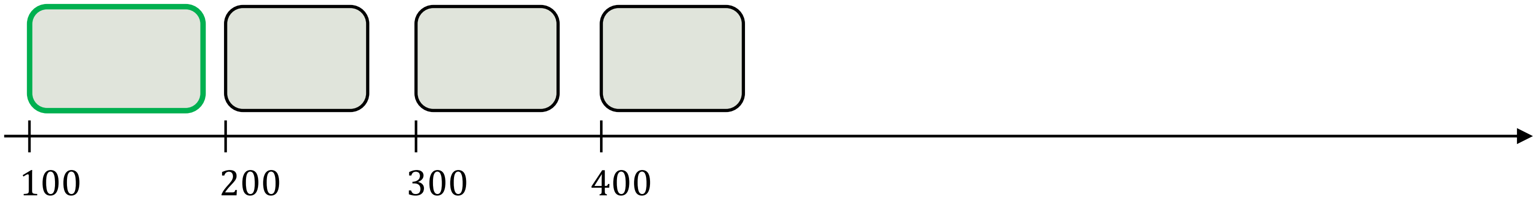
UNSAT



Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} 100 + i * 4$$

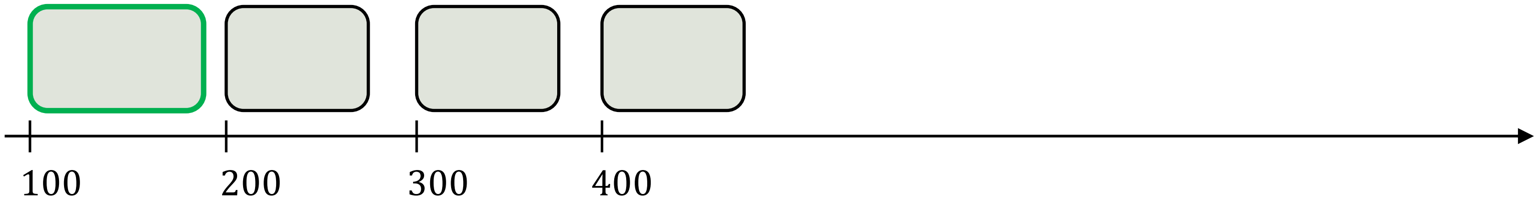


Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} 100 + i * 4$$

$$select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400\,], p - 100)$$



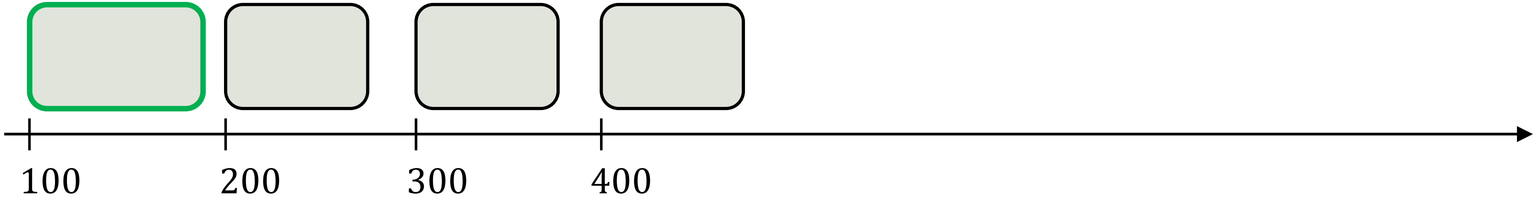
Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} 100 + i * 4$$

$$select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], p - 100)$$

$$select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4)$$



Symbolic Pointers

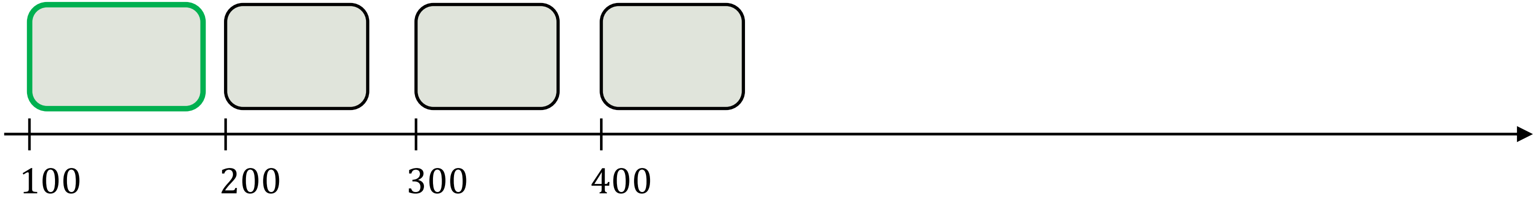
```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$p \overset{\text{def}}{=} 100 + i * 4$

$select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400\ ], p - 100)$

$select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4)$

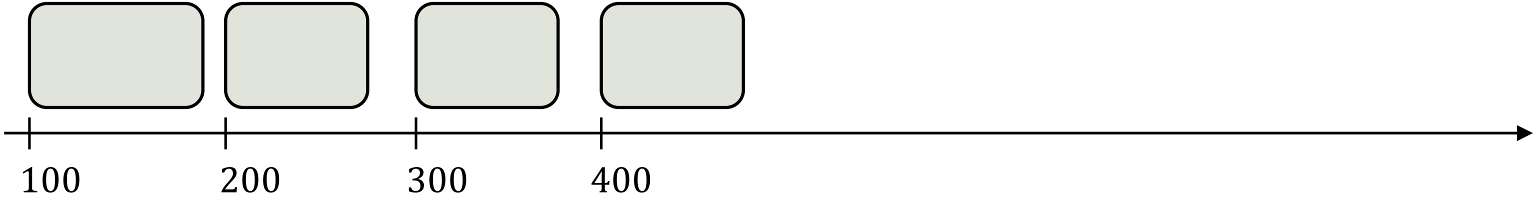
$select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$



Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

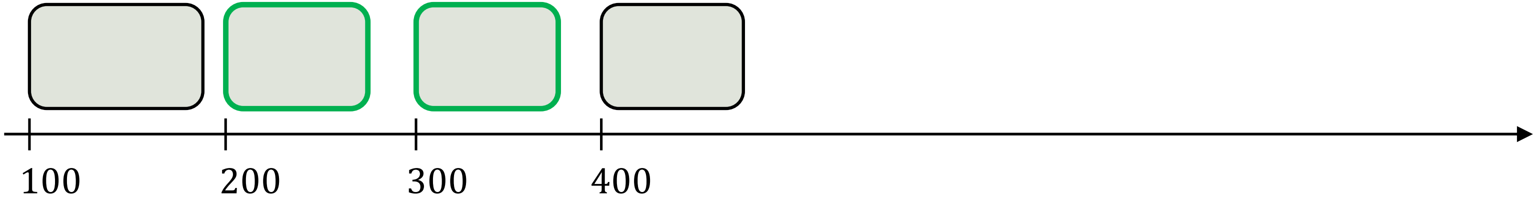
$$p \overset{\text{def}}{=} select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$



Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

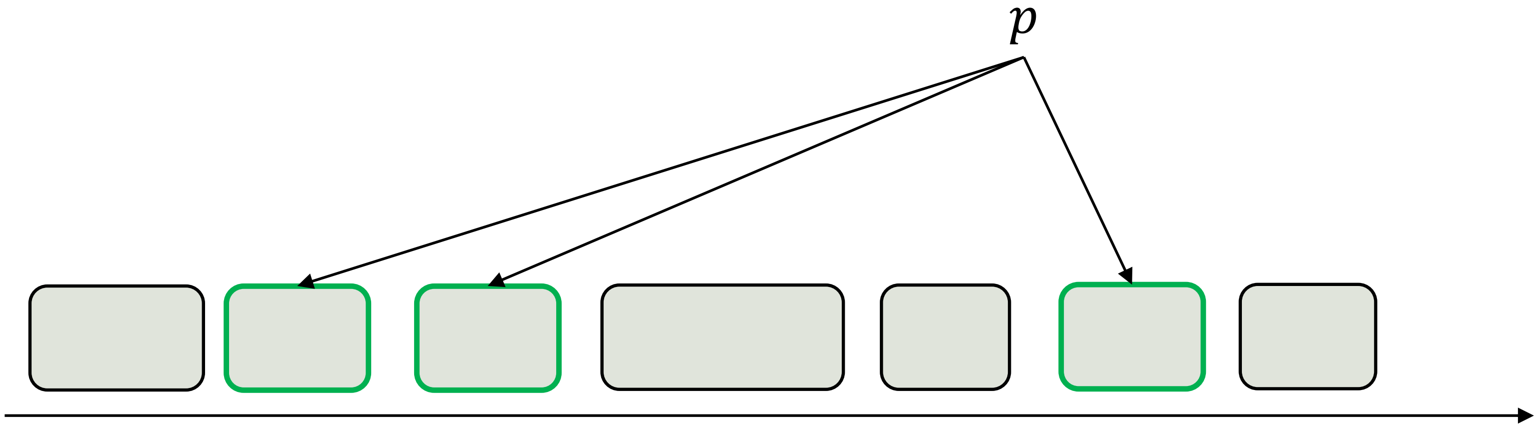
$$p \overset{\text{def}}{=} select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$



Multiple Resolutions

Approaches:

- Forking
- Merging

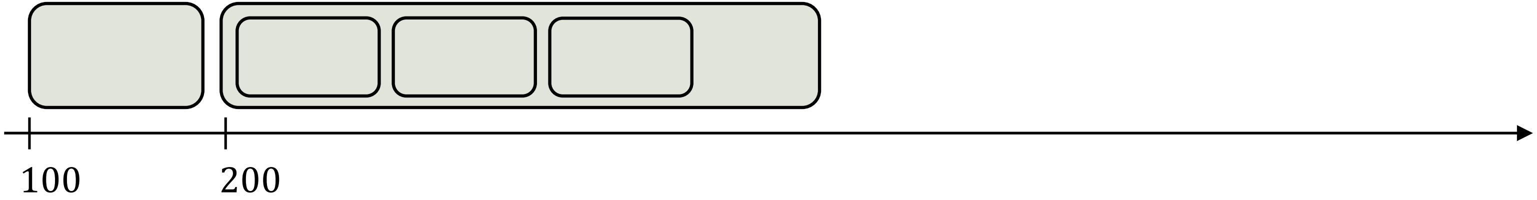


Segmented Memory Model

- Introduced by *Kapus et al.* (FSE 2019)
- Partitions the memory into segments using pointer analysis
- Pointer dereference without forking
    - Any pointer is resolved to at most one segment

Segmented Memory Model

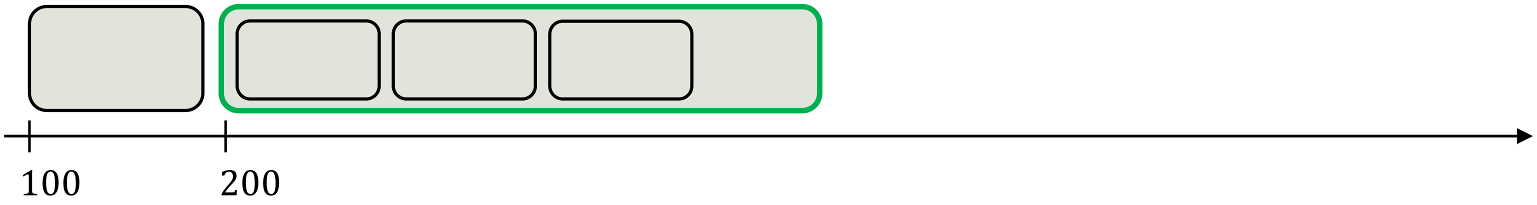
```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```



Segmented Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} \mathit{select}(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$

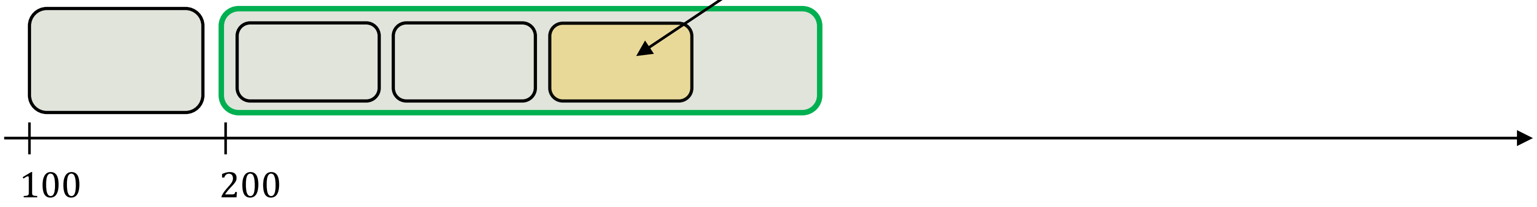


Segmented Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$

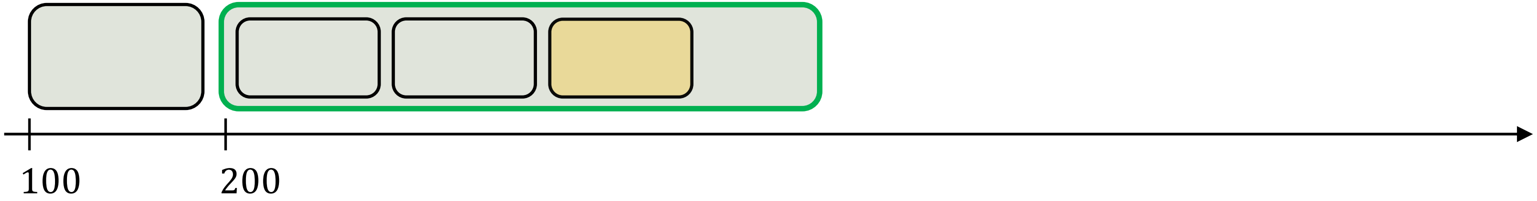
redundant



Segmented Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

- Avoids forking
- Unnecessarily large segments
- Slower constraint solving



Relocatable Memory Model

Memory objects:

- Defined by a tuple (β, s, a)
- Base addresses are symbolic

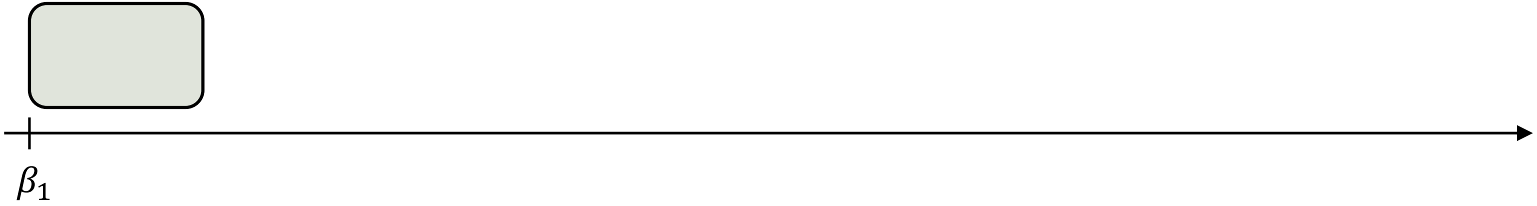
Address space:

- Maintain address constraints
- Preserves the *non-overlapping* property

Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

address constraints:
$\beta_1 = 100$

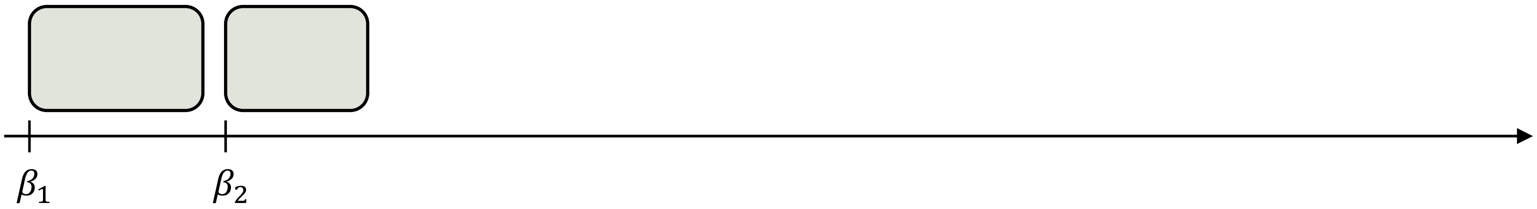


Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

address constraints:

$\beta_1 = 100 \wedge \beta_2 = 200$

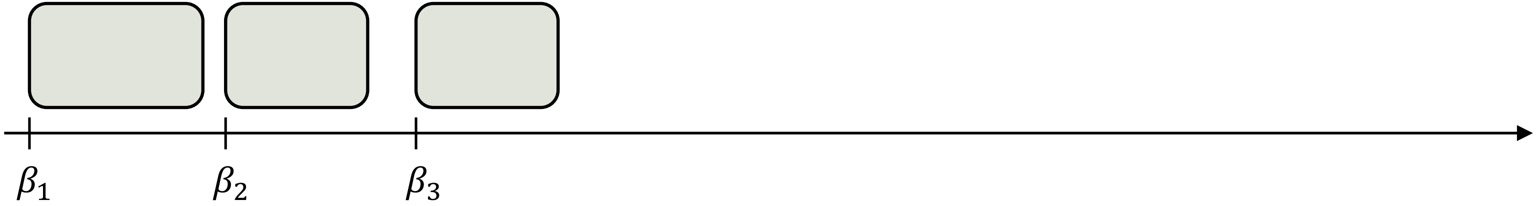


Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

address constraints:

$\beta_1 = 100 \land \beta_2 = 200 \land \beta_3 = 300$

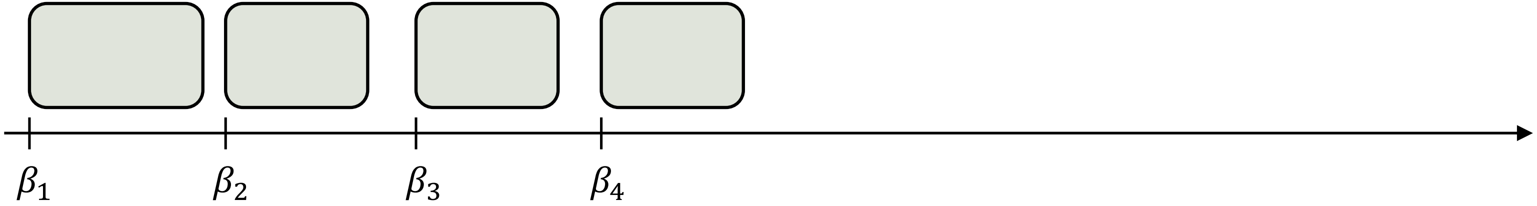


Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

address constraints:

$\beta_1 = 100 \wedge \beta_2 = 200 \wedge \beta_3 = 300 \wedge \beta_4 = 400$



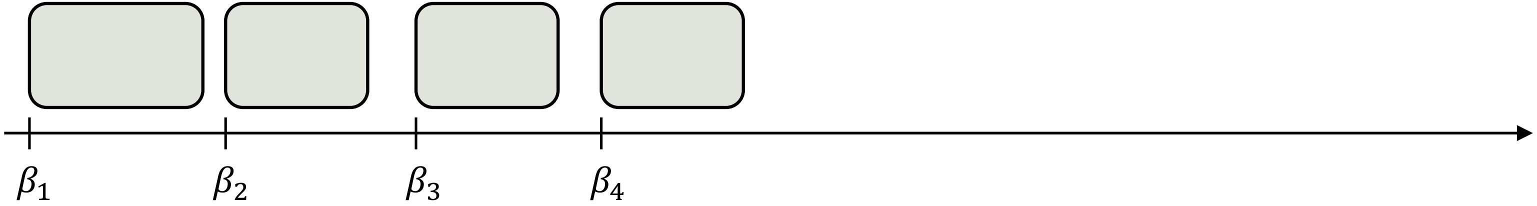
Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$$

address constraints:

$\beta_1 = 100 \wedge \beta_2 = 200 \wedge \beta_3 = 300 \wedge \beta_4 = 400$



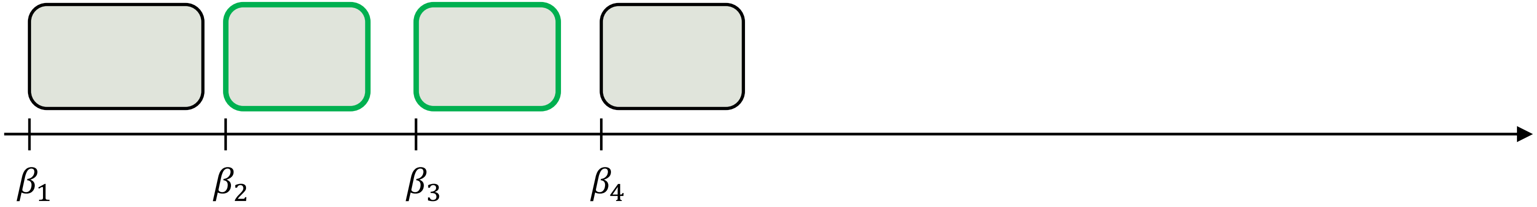
Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$$

address constraints:

$$\beta_1 = 100 \land \beta_2 = 200 \land \beta_3 = 300 \land \beta_4 = 400$$



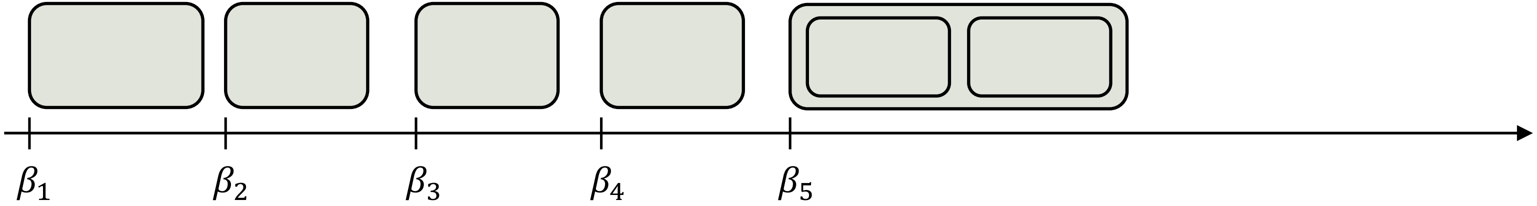
Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$$

address constraints:

$$\beta_1 = 100 \wedge \beta_2 = 200 \wedge \beta_3 = 300 \wedge \beta_4 = 400 \wedge \beta_5 = 500$$



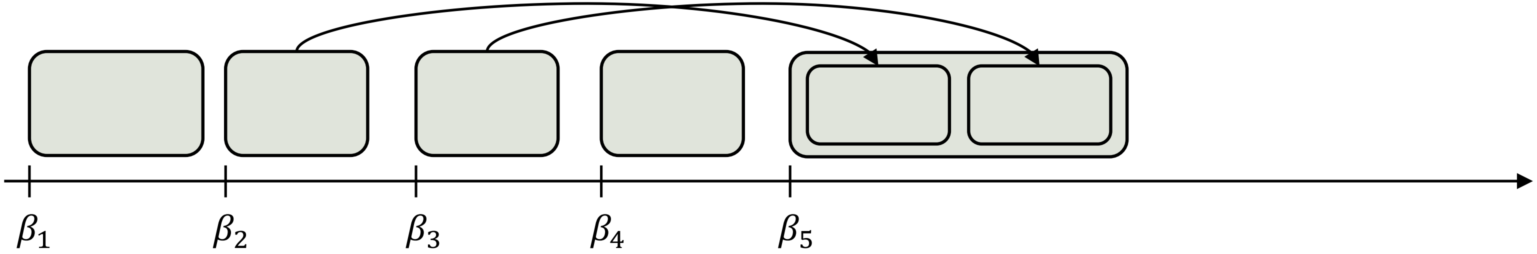
Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$$

address constraints:

$\beta_1 = 100 \wedge \beta_2 = 200 \wedge \beta_3 = 300 \wedge \beta_4 = 400 \wedge \beta_5 = 500$



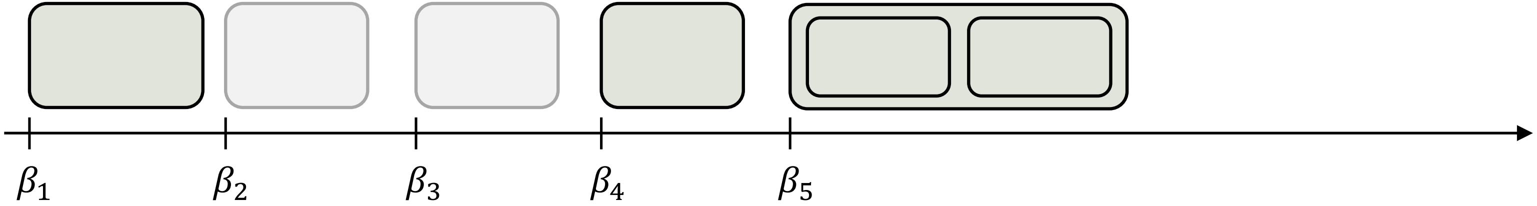
Relocatable Memory Model

```c
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$$

address constraints:

$$\beta_1 = 100 \land \beta_2 = 200 \land \beta_3 = 300 \land \beta_4 = 400 \land \beta_5 = 500$$



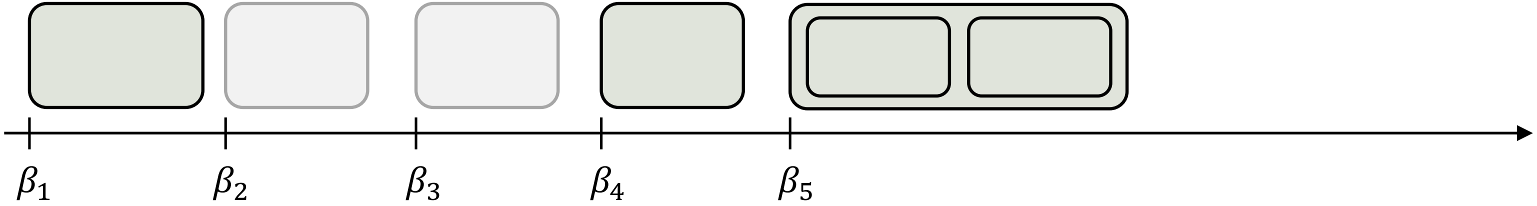
Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$$

address constraints:

$$\beta_1 = 100 \wedge \beta_2 = 500 \wedge \beta_3 = 510 \wedge \beta_4 = 400 \wedge \beta_5 = 500$$



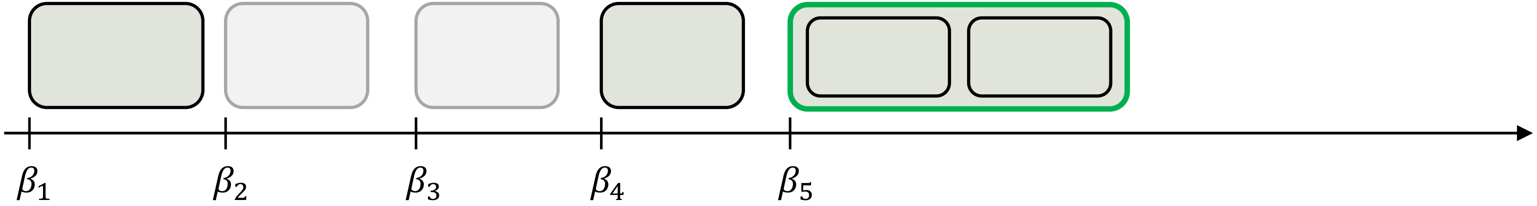
Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$$

address constraints:

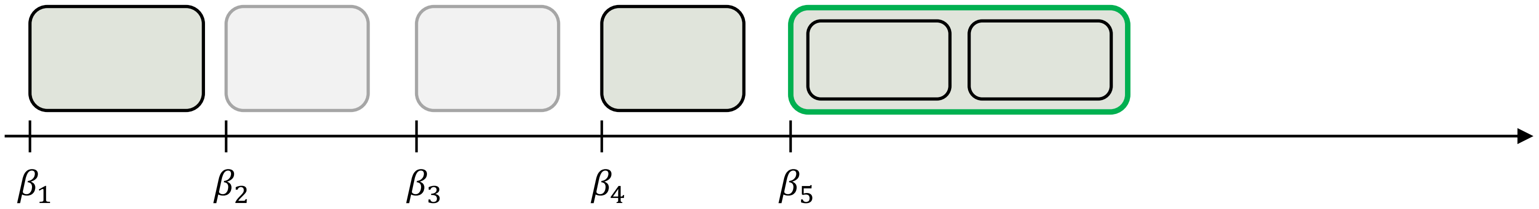
$$\beta_1 = 100 \land \beta_2 = 500 \land \beta_3 = 510 \land \beta_4 = 400 \land \beta_5 = 500$$



Relocatable Memory Model

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

- Avoids forking
- Smaller segments
- Faster constraint solving



Evaluation

Implemented on top of *KLEE*

Benchmarks:

- m4, make, sqlite, apr

Segment size:

- Average reduction: 83%

Average speedup in analysis time:

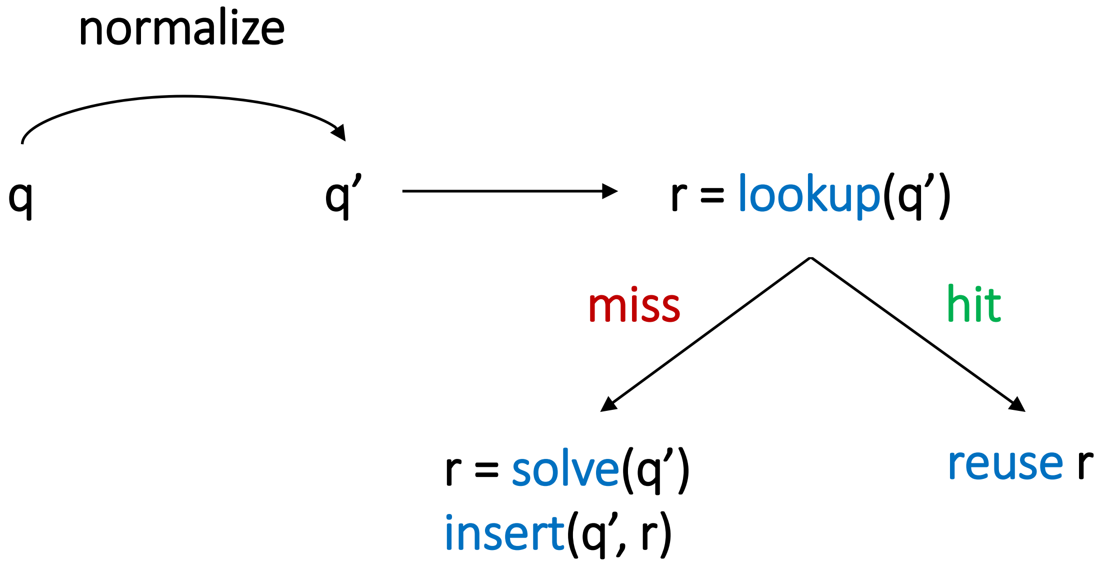
- Relocatable vs. Forking: 2.7X
- Relocatable vs. Segmented: 3.0X

Outline

- Background
  - Symbolic execution
  - Memory model
- Symbolic base addresses
  - Relocatable memory model
  - Address-aware query caching
- Symbolic-size allocations
  - Bounded symbolic-size model
  - State merging with quantifiers
- Conclusions and future work

Query Caching

A common technique for accelerating constraint solving



normalize

q

q’

r = lookup(q’)

miss

hit

r = solve(q’)
insert(q’, r)

reuse r

```c



char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

```diff
+ int z; // symbolic
+ if (z == 0) allocate_objects();
+
  char **array = calloc(3, PTR_SIZE);
  for (int i = 0; i < 3; i++) {
    array[i] = calloc(10, 1);
  }

  // symbolic: i < 2, j < 10
  unsigned i, j;
  if (array[i][j] == 7) {
    //...
  }
```

Address-Dependent Queries

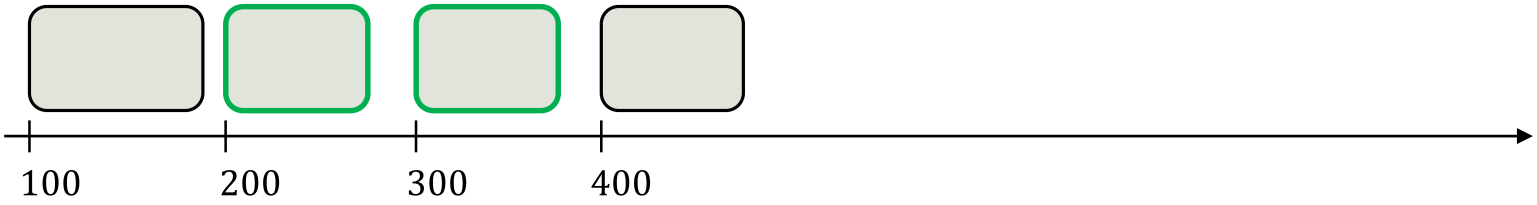
```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

What happens when $z \neq 0$?

Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

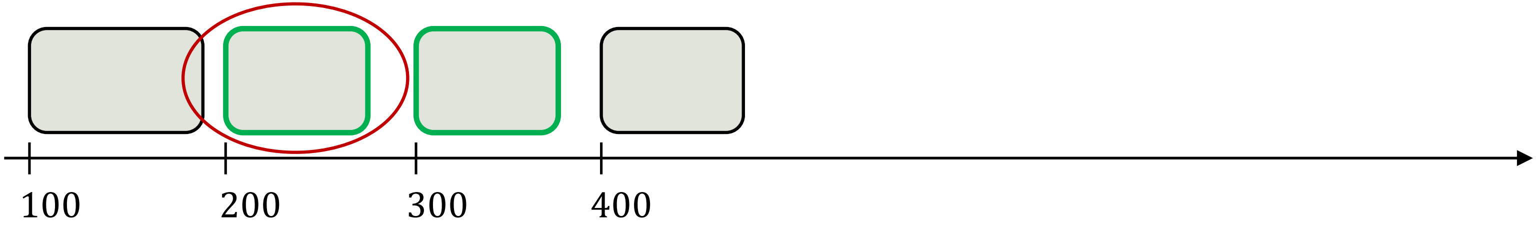
$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$



Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$



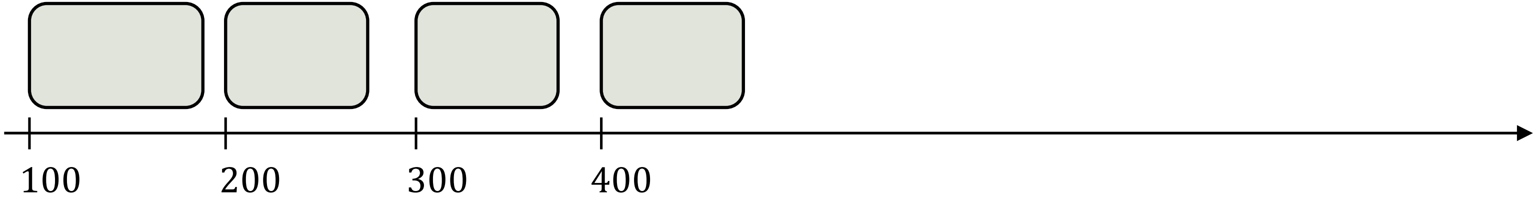
Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$

path constraints:

$$pc \overset{\text{def}}{=} z \neq 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 200 \leq p < 210$$



Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

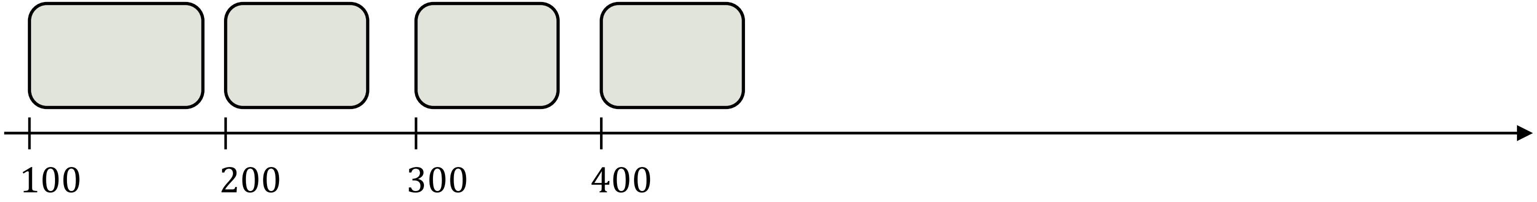
$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$

path constraints:

$$pc \overset{\text{def}}{=} z \neq 0 \ \land \ i < 2 \ \land \ j < 10 \ \land \ 200 \leq p < 210$$

query:

$$pc \ \land \ select(a_2, p - 200) = 7$$



Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

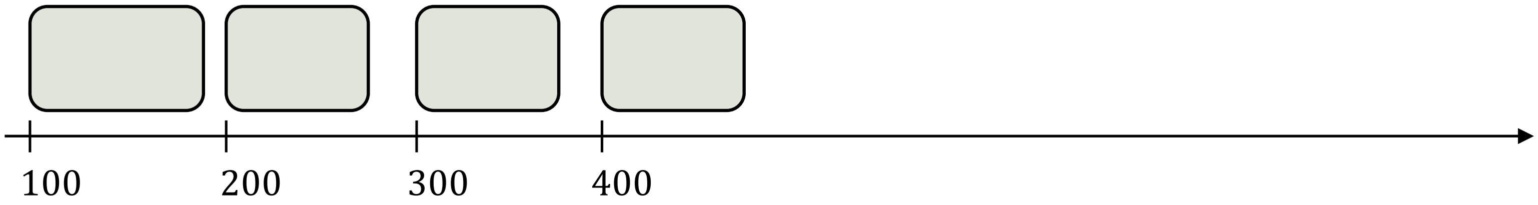
$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$

path constraints:

$$pc \overset{\text{def}}{=} z \neq 0 \ \wedge\ i < 2 \ \wedge\ j < 10 \ \wedge\ 200 \leq p < 210$$

query:

$$pc \ \wedge\ select(a_2, p - 200) = 7$$



Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

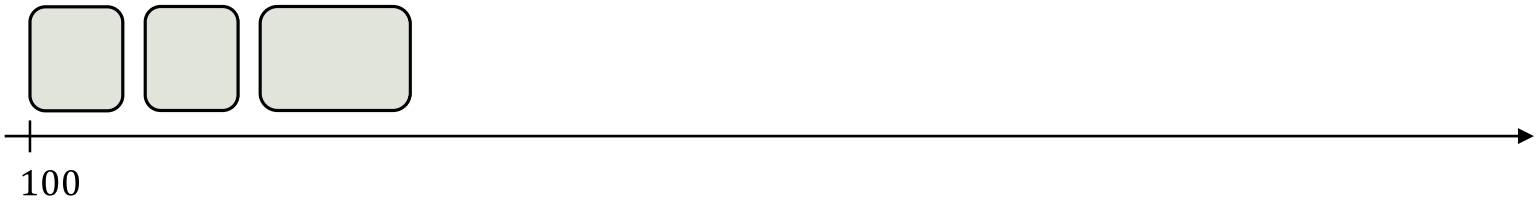
What happens when $z = 0$?

Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

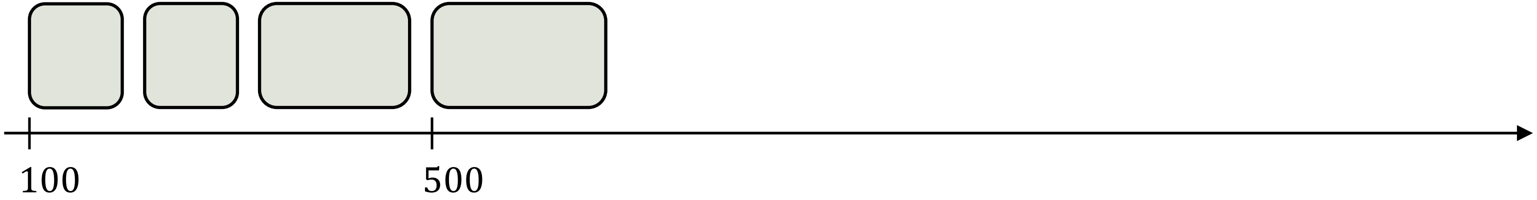
Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```



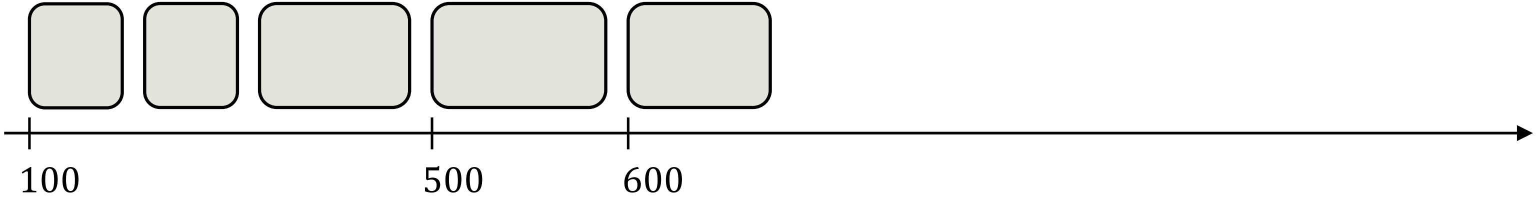
Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```



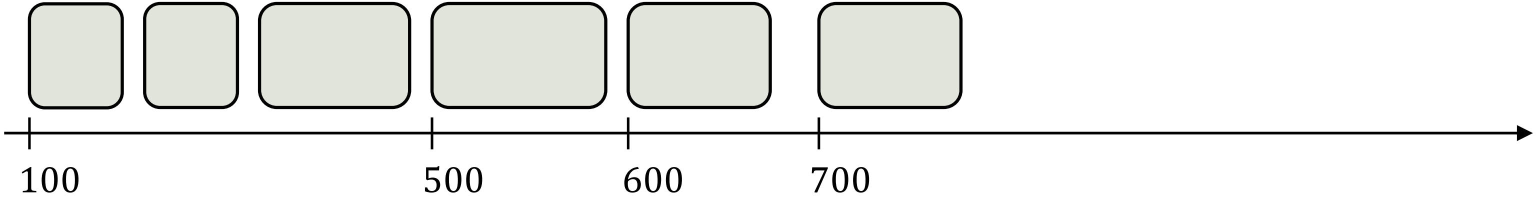
Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```



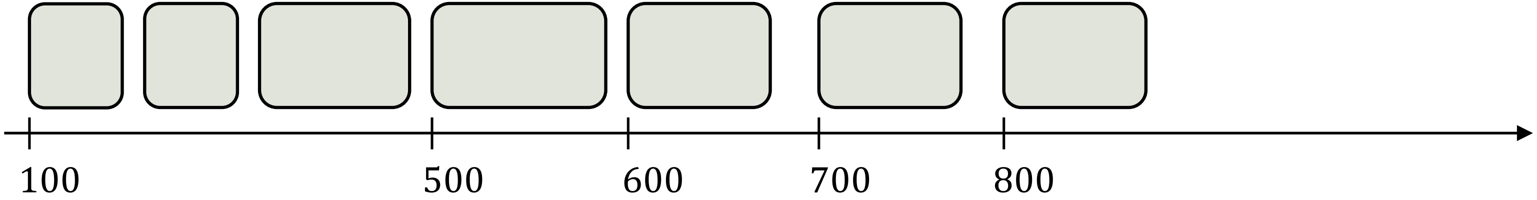
Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```



Address-Dependent Queries

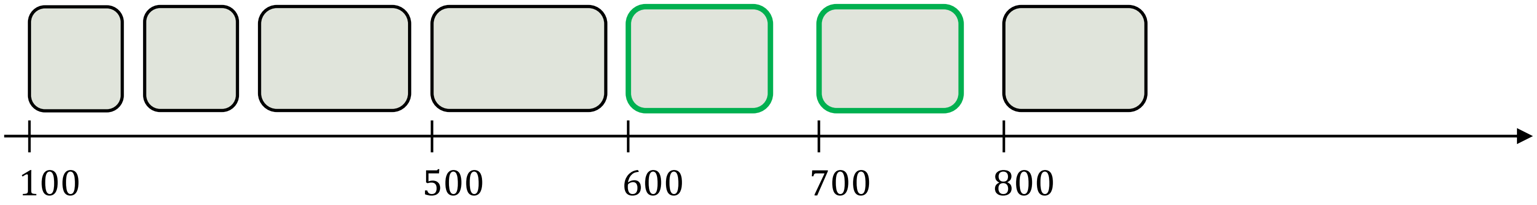
```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```



Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

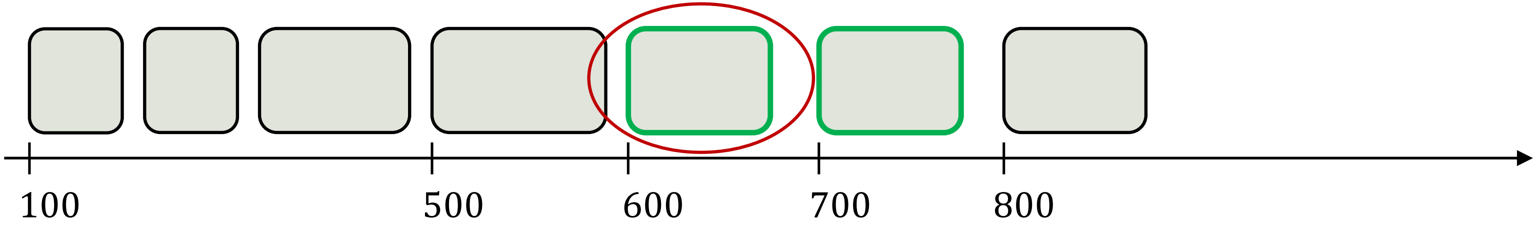
$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$$



Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$$



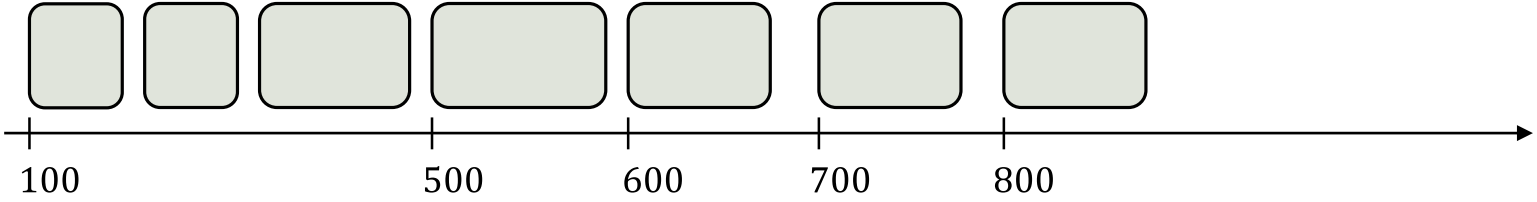
Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$$

path constraints:

$$pc \overset{\text{def}}{=} z = 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$$



Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

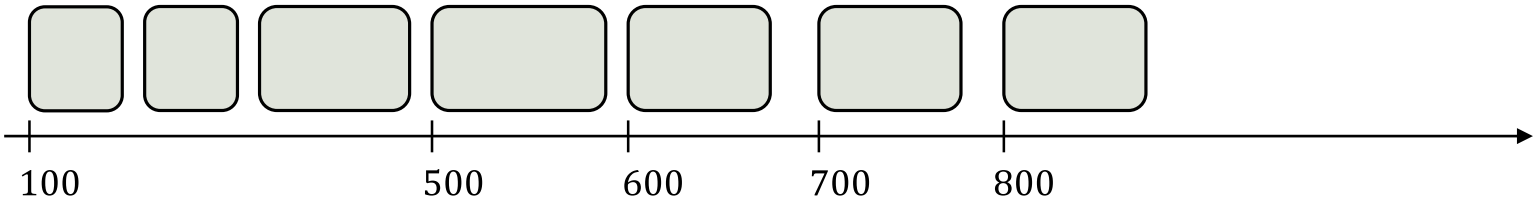
$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$$

path constraints:

$$pc \overset{\text{def}}{=} z = 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$$

query:

$$pc \ \wedge \ select(a_2, p - 600) = 7$$



Address-Dependent Queries

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

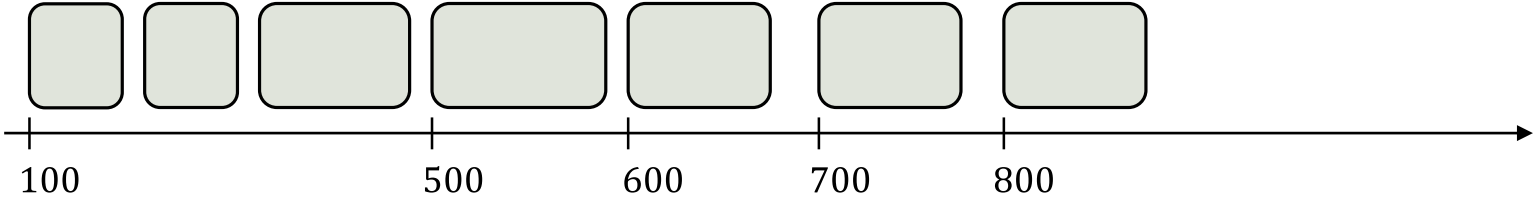
$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$$

path constraints:

$$pc \overset{\text{def}}{=} z = 0 \wedge i < 2 \wedge j < 10 \wedge 600 \leq p < 610$$

query:

$$pc \wedge select(a_2, p - 600) = 7$$



$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$

$$pc \overset{\text{def}}{=} z \neq 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 200 \leq p < 210$$

query:

$$pc \ \wedge \ select(a_2, p - 200) = 7$$

$$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$$

$$pc \overset{\text{def}}{=} z = 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$$

query:

$$pc \ \wedge \ select(a_2, p - 600) = 7$$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$

$pc \overset{\text{def}}{=} z \neq 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 200 \leq p < 210$

query:

$pc \ \wedge \ select(a_2, p - 200) = 7$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$

$pc \overset{\text{def}}{=} z = 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$

query:

$pc \ \wedge \ select(a_2, p - 600) = 7$

$$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$$

$$pc \stackrel{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \ 200 \leq p < 210$$

query:

$$pc \ \wedge \ select(a_2, p - 200) = 7$$

$$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$$

$$pc \stackrel{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$$

query:

$$pc \ \wedge \ select(a_2, p - 600) = 7$$

$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i * 4) + j$

$pc \stackrel{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \ 200 \leq p < 210$

query:

$pc \ \wedge \ select(a_2, p - 200) = 7$

$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i * 4) + j$

$pc \stackrel{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$

query:

$pc \ \wedge \ select(a_2, p - 600) = 7$

- Equisatisfiable
- Query caching **fails** (No common normal form)

Solution: Relocatable Memory Model

- Base addresses are **symbolic**
  - Distinguish between **integer** and **address** values
- Determine **equisatisfiability** by checking:
  - Expression isomorphism (equality up to renaming)
  - Address space isomorphism

***Assuming** that the analyzed program has **no undefined behavior**.*

Solution: Relocatable Memory Model

```
int z; // symbolic
if (z == 0) allocate_objects();

char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  //...
}
```

$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$

path constraints:

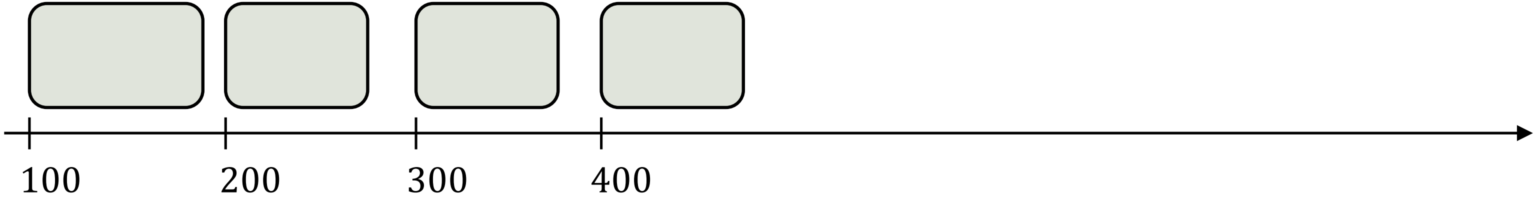
$pc \overset{\text{def}}{=} z \neq 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \beta_2 \leq p < \beta_2 + 10$

query:

$pc \ \wedge \ select(a_2, p - \beta_2) = 7$

address constraints:

$\beta_1 = 100 \ \wedge \beta_2 = 200 \wedge \beta_3 = 300 \wedge \beta_4 = 400$



$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \beta_2 \leq p < \beta_2 + 10$

query:

$pc \ \wedge \ select(a_2, p - \beta_2) = 7$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_6, 1 \mapsto \beta_7, 2 \mapsto \beta_8], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \beta_6 \leq p < \beta_6 + 10$

query:

$pc \ \wedge \ select(a_2, p - \beta_6) = 7$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \beta_2 \leq p < \beta_2 + 10$

query:
$pc \ \wedge \ select(a_2, p - \beta_2) = 7$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_6, 1 \mapsto \beta_7, 2 \mapsto \beta_8], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \beta_6 \leq p < \beta_6 + 10$

query:
$pc \ \wedge \ select(a_2, p - \beta_6) = 7$

$\beta_2 \leftrightarrow \beta_6 \qquad \beta_3 \leftrightarrow \beta_7 \qquad \beta_4 \leftrightarrow \beta_8$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \beta_2 \leq p < \beta_2 + 10$

query:

$pc \ \wedge \ select(a_2, p - \beta_2) = 7$

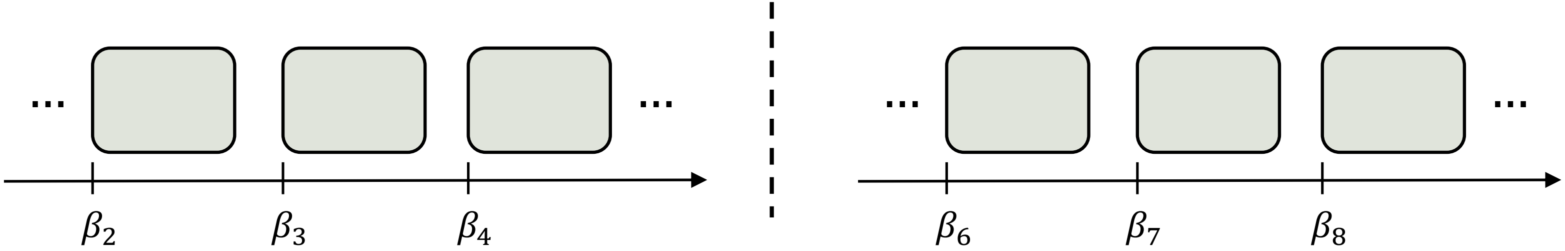
$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_6, 1 \mapsto \beta_7, 2 \mapsto \beta_8], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \beta_6 \leq p < \beta_6 + 10$

query:

$pc \ \wedge \ select(a_2, p - \beta_6) = 7$

$\beta_2 \leftrightarrow \beta_6 \qquad \beta_3 \leftrightarrow \beta_7 \qquad \beta_4 \leftrightarrow \beta_8$



$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$

$pc \stackrel{\text{def}}{=} i < 2 \ \land \ j < 10 \ \land \beta_2 \leq p < \beta_2 + 10$

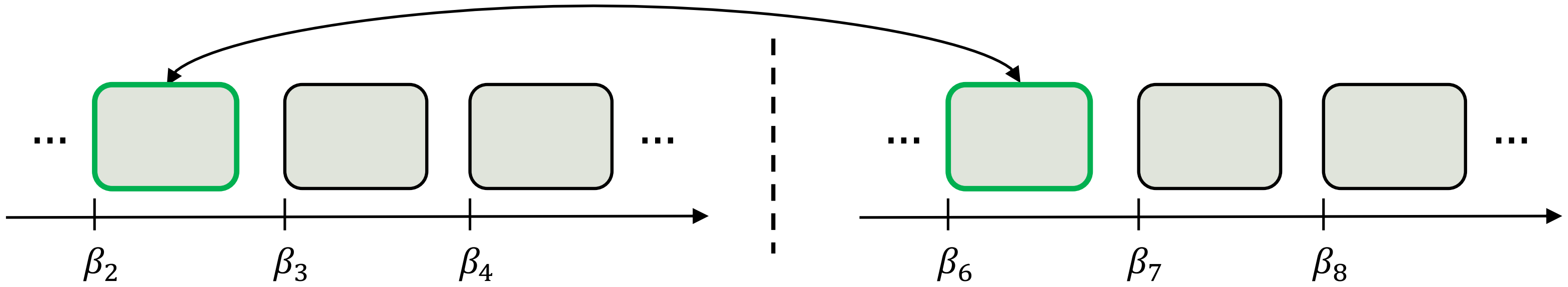
query:
$pc \ \land \ select(a_2, p - \beta_2) = 7$

$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto \beta_6, 1 \mapsto \beta_7, 2 \mapsto \beta_8], i * 4) + j$

$pc \stackrel{\text{def}}{=} i < 2 \ \land \ j < 10 \ \land \beta_6 \leq p < \beta_6 + 10$

query:
$pc \ \land \ select(a_2, p - \beta_6) = 7$

$\beta_2 \leftrightarrow \beta_6 \qquad \beta_3 \leftrightarrow \beta_7 \qquad \beta_4 \leftrightarrow \beta_8$



$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \land \ j < 10 \ \land \beta_2 \leq p < \beta_2 + 10$

query:

$pc \ \land \ select(a_2, p - \beta_2) = 7$

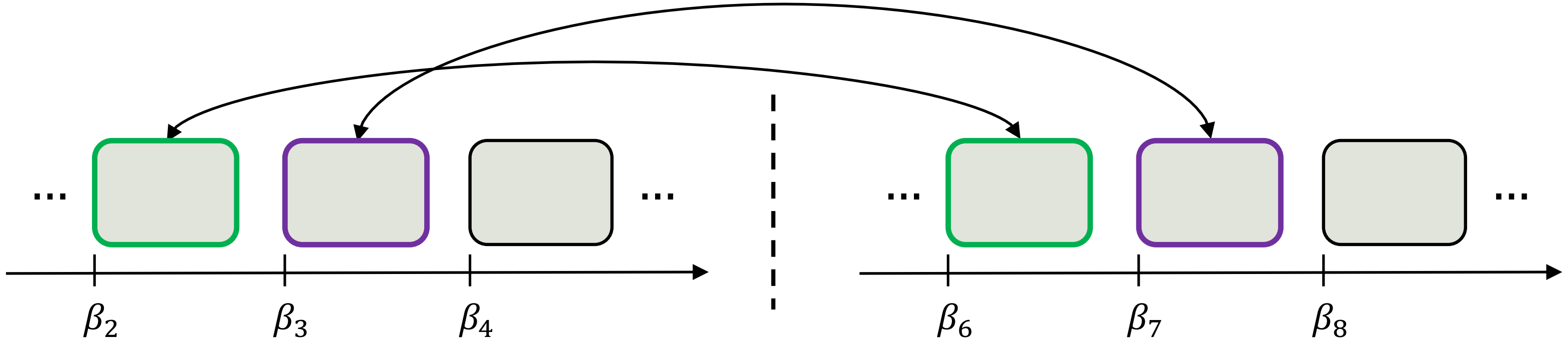
$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_6, 1 \mapsto \beta_7, 2 \mapsto \beta_8], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \land \ j < 10 \ \land \beta_6 \leq p < \beta_6 + 10$

query:

$pc \ \land \ select(a_2, p - \beta_6) = 7$

$\beta_2 \leftrightarrow \beta_6 \qquad \beta_3 \leftrightarrow \beta_7 \qquad \beta_4 \leftrightarrow \beta_8$



$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_2, 1 \mapsto \beta_3, 2 \mapsto \beta_4], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \beta_2 \leq p < \beta_2 + 10$

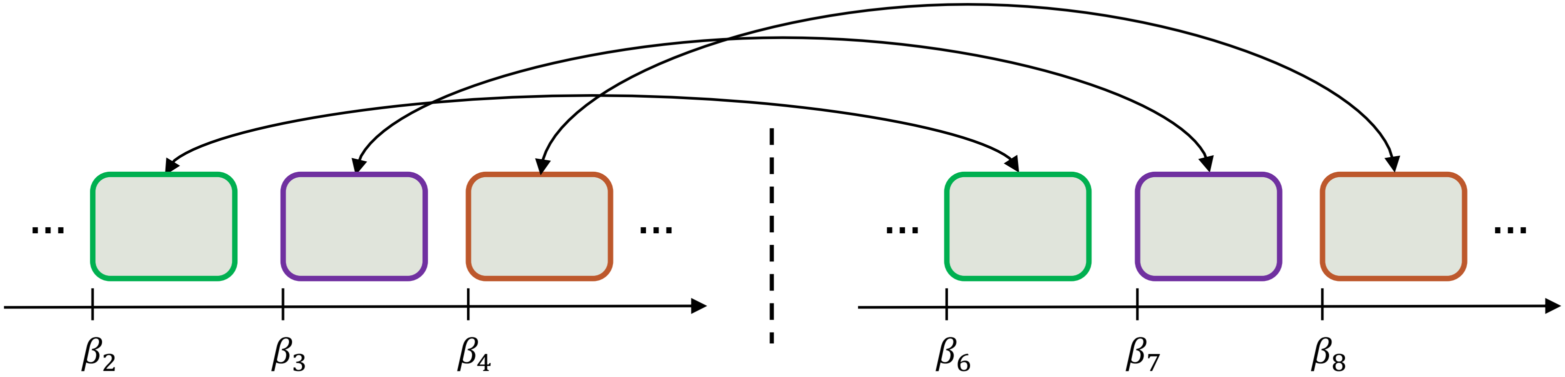
query:
$pc \ \wedge \ select(a_2, p - \beta_2) = 7$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto \beta_6, 1 \mapsto \beta_7, 2 \mapsto \beta_8], i * 4) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \beta_6 \leq p < \beta_6 + 10$

query:
$pc \ \wedge \ select(a_2, p - \beta_6) = 7$

$\beta_2 \leftrightarrow \beta_6 \qquad \beta_3 \leftrightarrow \beta_7 \qquad \beta_4 \leftrightarrow \beta_8$



Evaluation

Implemented on top of *KLEE*

Benchmarks:

- m4, make, sqlite, apr, libxml2, expat, bash, json-c

Cache misses (number of queries passed to SMT solver):

- Average reduction: 58%

Analysis speedup in analysis time: 2.2X

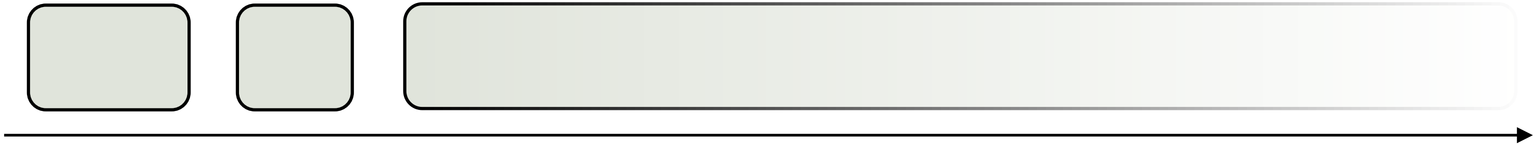
Outline

- Background
  - Symbolic execution
  - Memory model
- Symbolic base addresses
  - Relocatable memory model
  - Address-aware query caching
- Symbolic-size allocations
  - Bounded symbolic-size model
  - State merging with quantifiers
- Conclusions and future work

Observation

Modeling symbolic-size objects is hard:

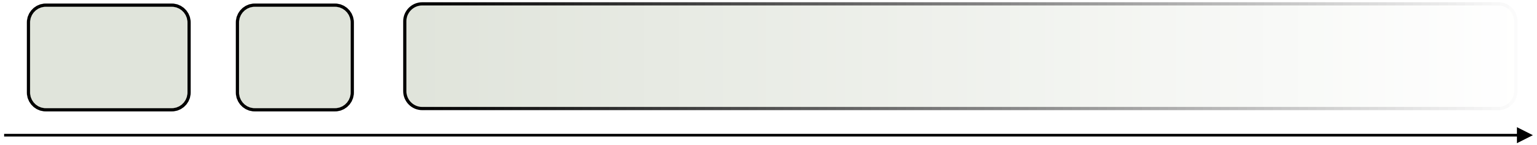
- **Fixed**
  - Limited exploration



Observation

Modeling symbolic-size objects is hard:

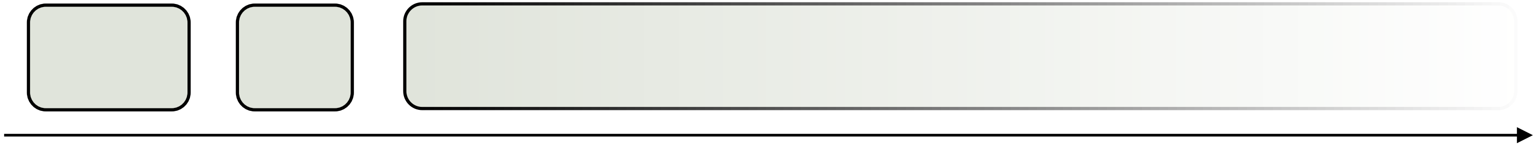
- Fixed
  - Limited exploration
- Unbounded
  - Overlapping in a linear address space
  - High memory consumption



Observation

Modeling symbolic-size objects is hard:

- Fixed
  - Limited exploration
- Unbounded
  - Overlapping in a linear address space
  - High memory consumption
- Bounded
  - Integrates with a linear address space
  - Controllable memory consumption



Outline

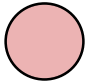
- Background
  - Symbolic execution
  - Memory model
- Symbolic base addresses
  - Relocatable memory model
  - Address-aware query caching
- Symbolic-size allocations
  - Bounded symbolic-size model
  - State merging with quantifiers
- Conclusions and future work

Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

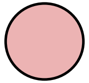
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

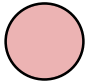


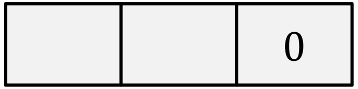
concretize $k + 1$ to 3

		0

Example

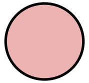
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

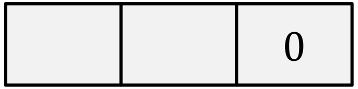




Example

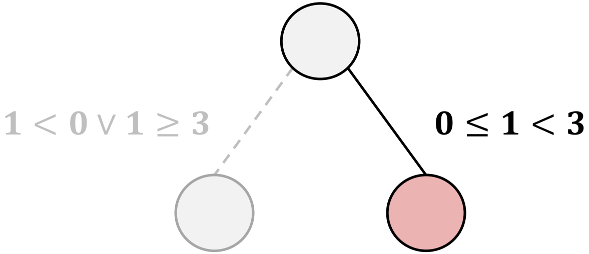
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

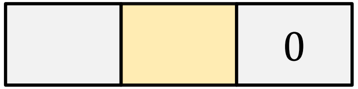




Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

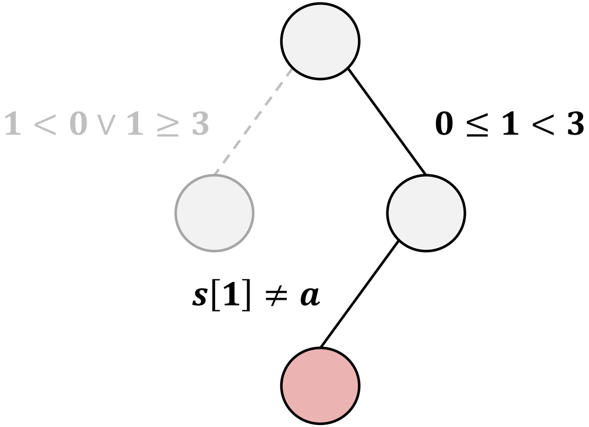




Example

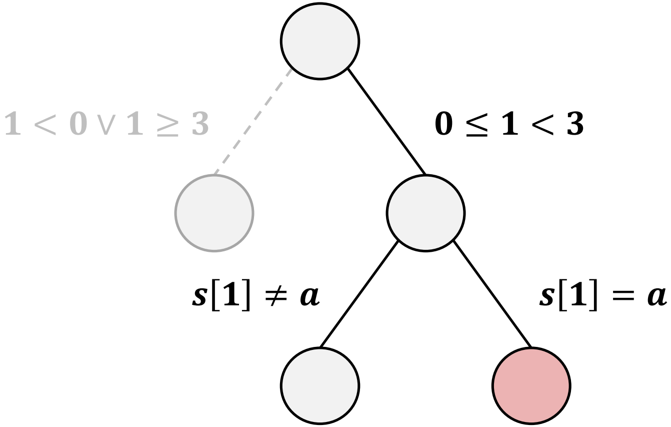
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

		0



Example

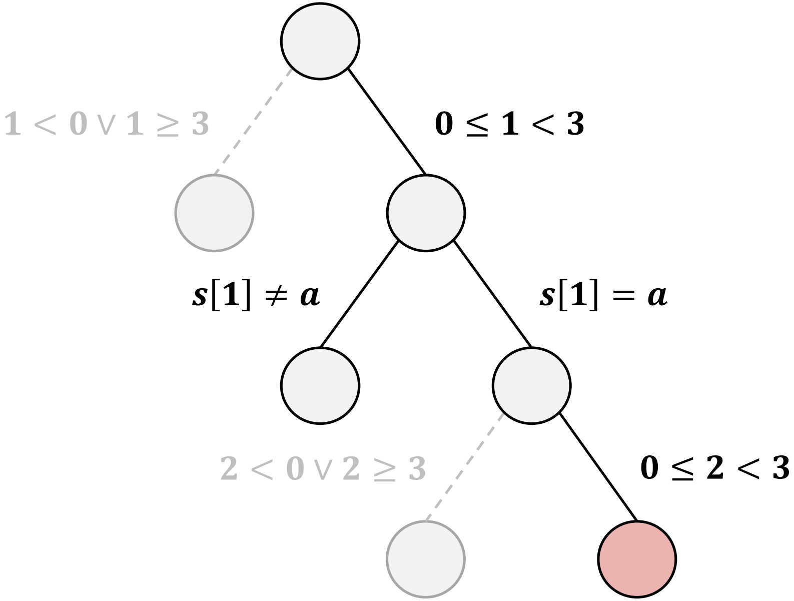
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



		0

Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

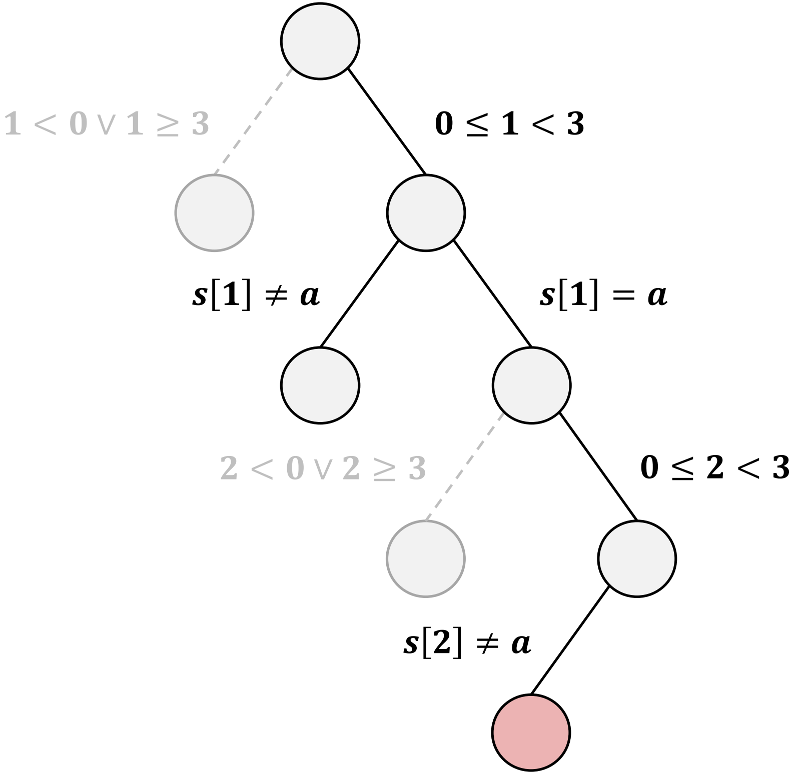


		0

Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

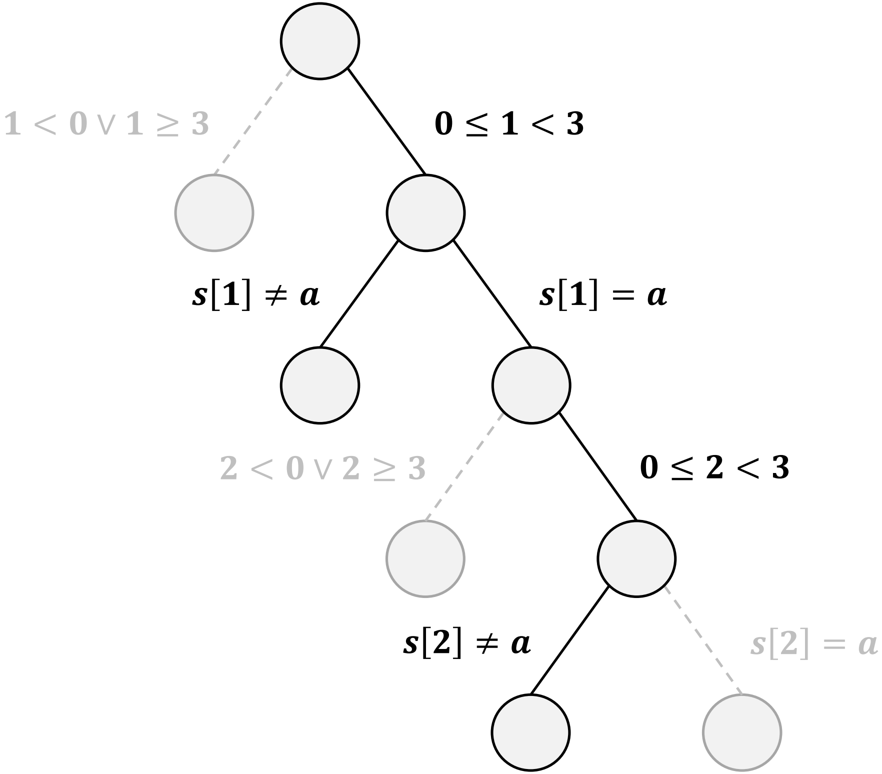
		0



Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

		0



Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

concretize $k + 1$ to 3

⇓

missed bug!

out-of-bounds access if $k = 0$

Bounded Symbolic-Size Model

Defined by a tuple (b, σ, c, a):

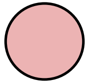
- Concrete base address
- Symbolic size
- Concrete capacity: $0 < \sigma \leq c$
- SMT array

Easily integrated with a linear address space

Controllable memory consumption

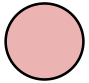
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



Example

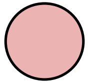
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



capacity constraint: $0 < k + 1 \leq 3$

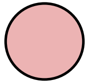
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



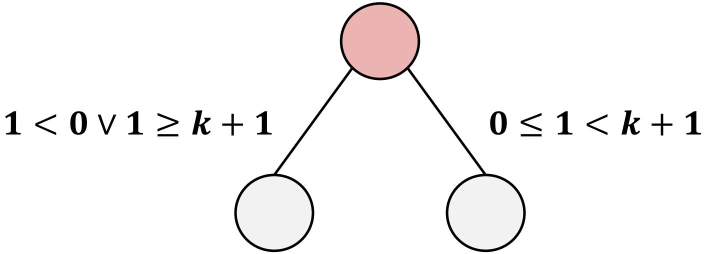
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



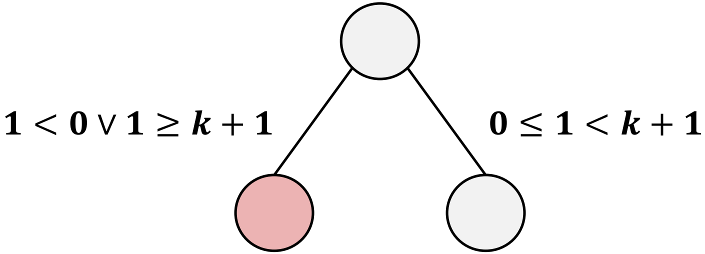
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



Example

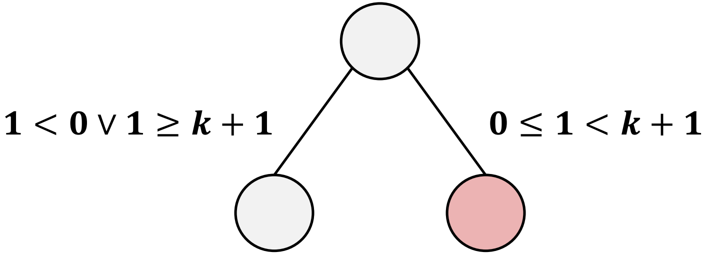
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



memory error

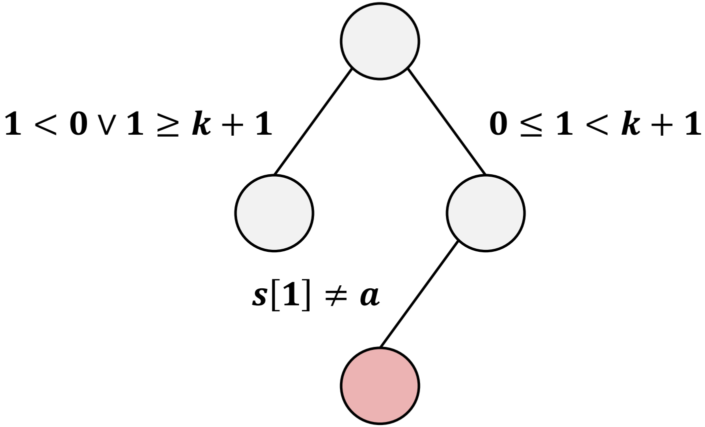
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



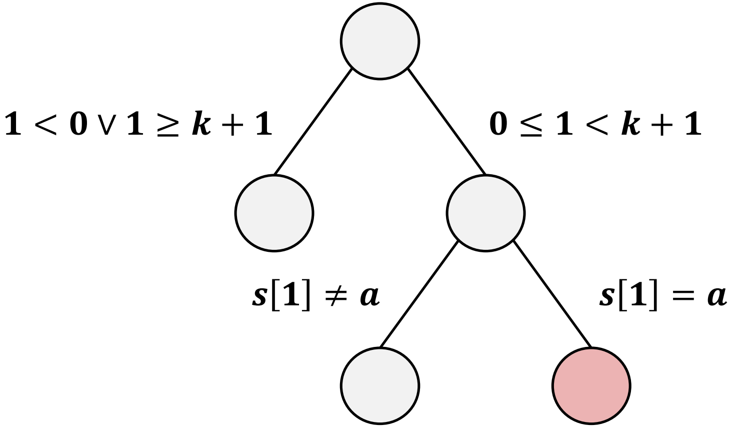
$1 < 0 \lor 1 \geq k + 1$

$0 \leq 1 < k + 1$

$s[1] \neq a$

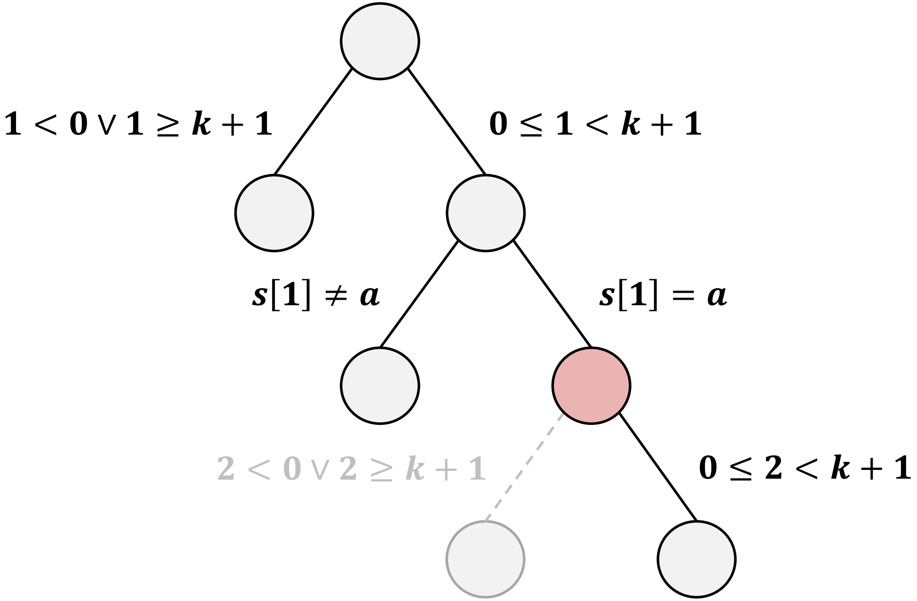
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



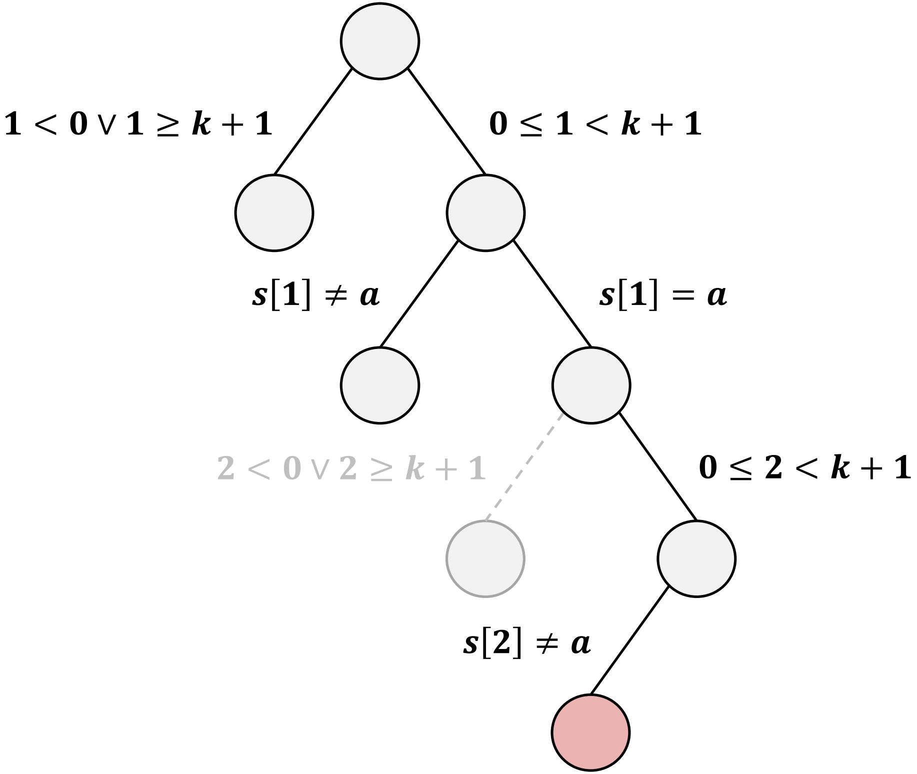
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



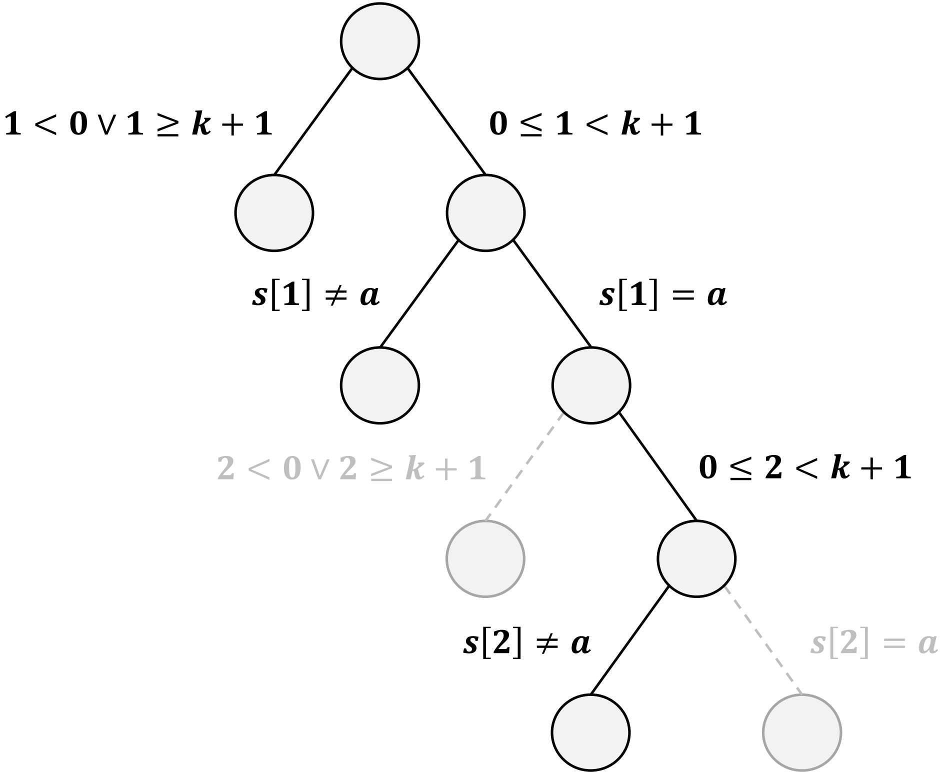
Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```



Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s+1, ‘a’);
```

capacity constraint: $0 < k + 1 \leq 3$

⇓

detected bug!

Arising Challenges

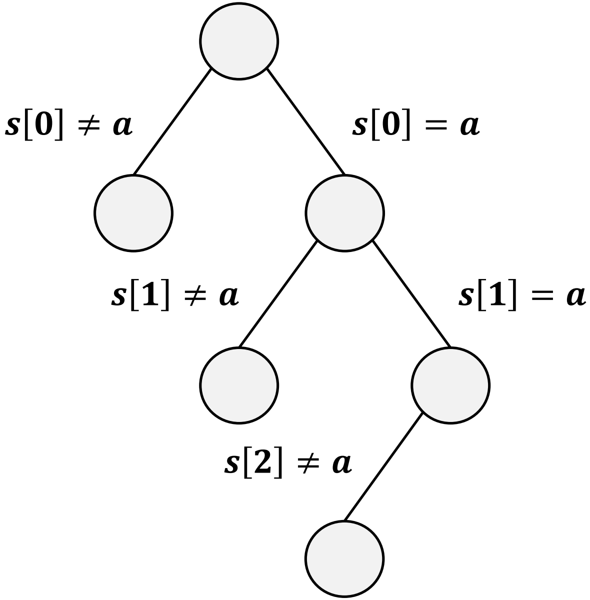
- Additional symbolic-size expressions
- Amplifies path explosion
  - Especially with **size-dependent loops**

State Merging Approach

- Detect **symbolic-size dependent** loops
- Execute the loop till **full exploration**
- **Merge** the resulting states

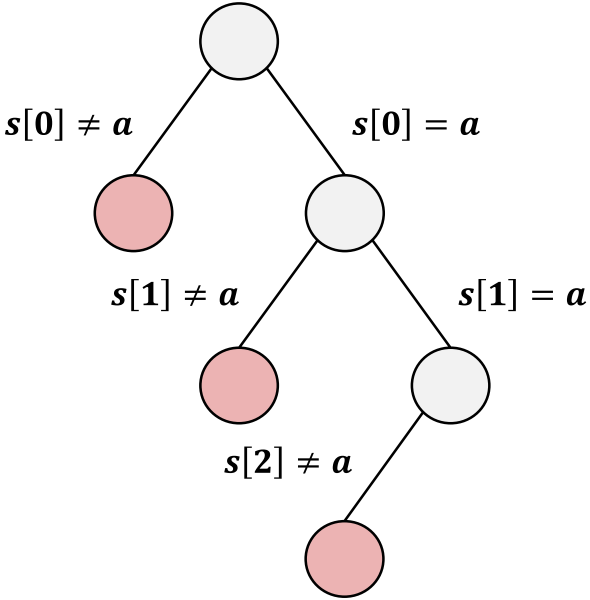
State Merging Approach

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, ‘a’);
```



State Merging Approach

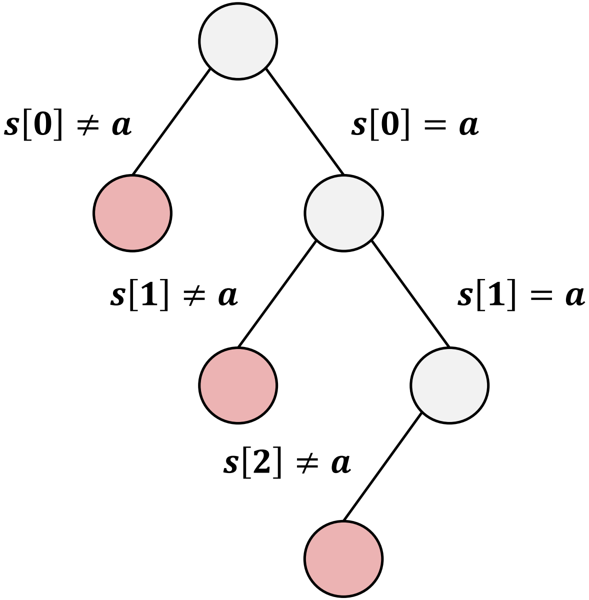
$$\overbrace{\underbrace{(0 < k + 1 \leq 3)}_{\text{prefix}} \;\wedge\; \underbrace{\left(\begin{array}{l}(s[0] \neq a) \vee \\ (s[0] = a \wedge s[1] \neq a) \vee \\ (s[0] = a \wedge s[1] = a \wedge s[2] \neq a)\end{array}\right)}_{\text{suffix}}}^{\text{merged constraint}}$$



State Merging Approach

$$\underbrace{(\mathbf{0} < \boldsymbol{k} + \mathbf{1} \leq \mathbf{3}) \ \wedge \ \begin{pmatrix} (\boldsymbol{s}[0] \neq \boldsymbol{a}) \vee \\ (\boldsymbol{s}[0] = \boldsymbol{a} \wedge \boldsymbol{s}[1] \neq \boldsymbol{a}) \vee \\ (\boldsymbol{s}[0] = \boldsymbol{a} \wedge \boldsymbol{s}[1] = \boldsymbol{a} \wedge \boldsymbol{s}[2] \neq \boldsymbol{a}) \end{pmatrix}}_{}$$

merged constraint

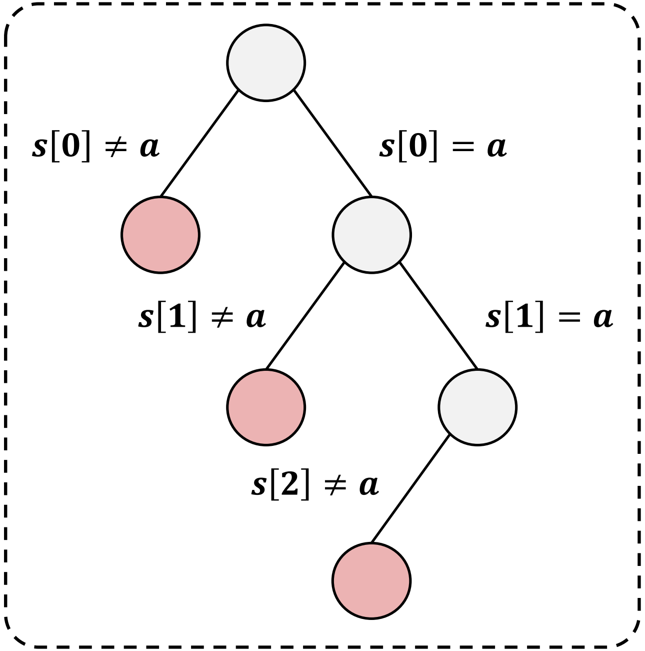


State Merging Approach

$$\overbrace{(0 < k+1 \leq 3) \;\; \wedge \;\; \left(\begin{array}{l} (s[0] \neq a) \vee \\ (s[0] = a \wedge s[1] \neq a) \vee \\ (s[0] = a \wedge s[1] = a \wedge s[2] \neq a) \end{array}\right)}^{\text{merged constraint}}$$

$$\Big\Downarrow \; \text{rewrite}$$

$$(0 < k + 1 \leq 3 \wedge (s[0] \neq a \vee (s[0] = a \wedge (s[1] \neq a \vee (s[1] = a \wedge s[2] \neq a)))))$$



Evaluation

Implemented on top of *KLEE*

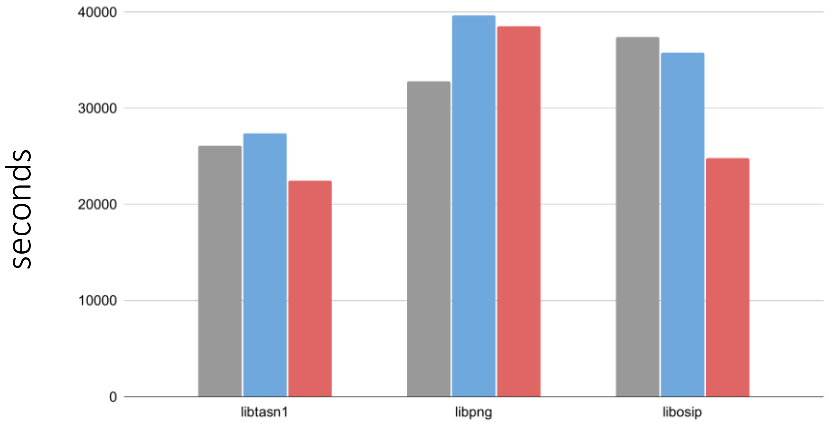
Benchmarks:

- libtasn1, libpng, libosip

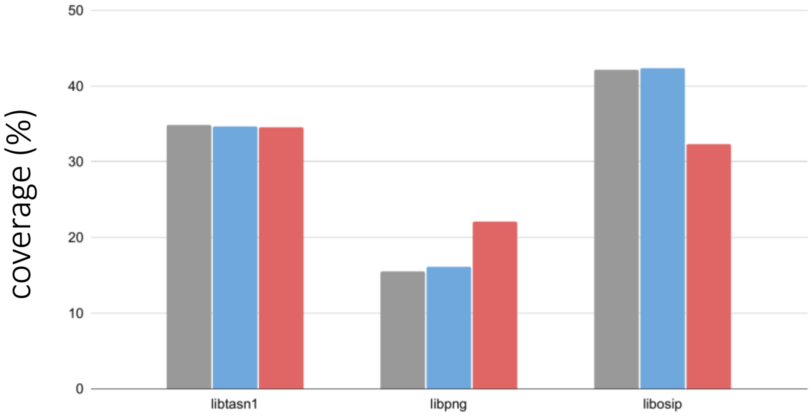
- Concrete size
- Symbolic size (forking)
- Symbolic size (merging)

modes

Total analysis time



Total coverage



Evaluation

Found bugs:

- libtasn1
  - one *out-of-bound-read*
- oSIP
  - three *out-of-bound-read's*
  - one *integer-underflow*

All the bugs were **confirmed** and **fixed**.

Outline

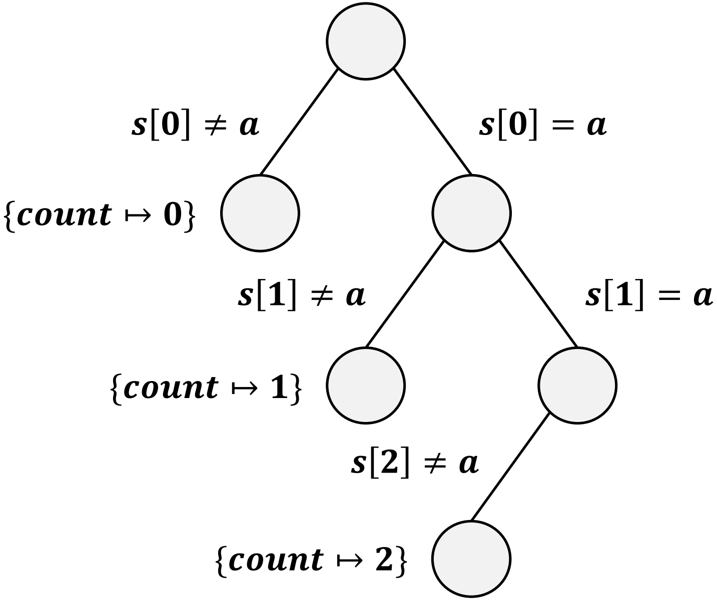
- Background
  - Symbolic execution
  - Memory model
- Symbolic base addresses
  - Relocatable memory model
  - Address-aware query caching
- Symbolic-size allocations
  - Bounded symbolic-size model
  - State merging with quantifiers
- Conclusions and future work

```
  int strspn(char *s, char c) {
    int count = 0;
    while (s[count] == c) {
      count++;
    }
    return count;
  }

  unsigned k; // symbolic
  char *s = malloc(k+1); // symbolic
  s[k] = 0;
- int n = strspn(s, ‘a’);
```

```
  int strspn(char *s, char c) {
    int count = 0;
    while (s[count] == c) {
      count++;
    }
    return count;
  }

  unsigned k; // symbolic
  char *s = malloc(k+1); // symbolic
  s[k] = 0;
+ int n = strspn(s, ‘a’);
+ int m = strspn(s + n, ‘b’);
```

Standard State Merging

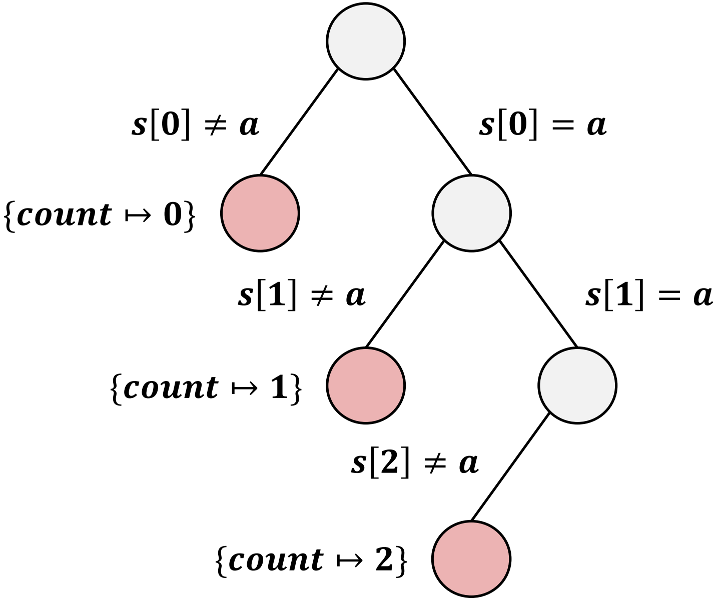
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
```



Standard State Merging

Merging the **path constraints**

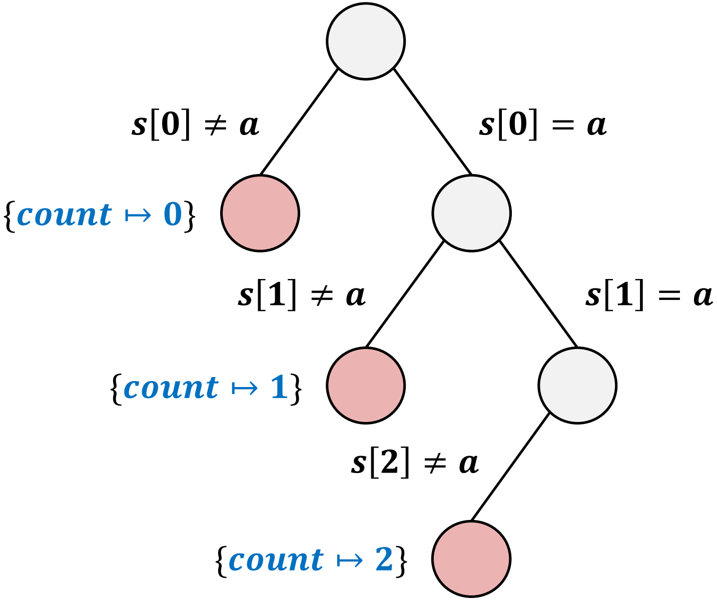
$(s[0] \neq a) \vee$
$(s[0] = a \wedge s[1] \neq a) \vee$
$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$



Standard State Merging

Merging the **memory**

$$\underbrace{\begin{array}{l} \boldsymbol{ite}(\\ \quad s[0] \neq a, \\ \quad 0, \\ \quad \boldsymbol{ite}(\\ \quad\quad s[0] = a \wedge s[1] \neq a, \\ \quad\quad 1, \\ \quad\quad 2 \\ \quad) \\ ) \end{array}}_{\text{merged value of count}}$$

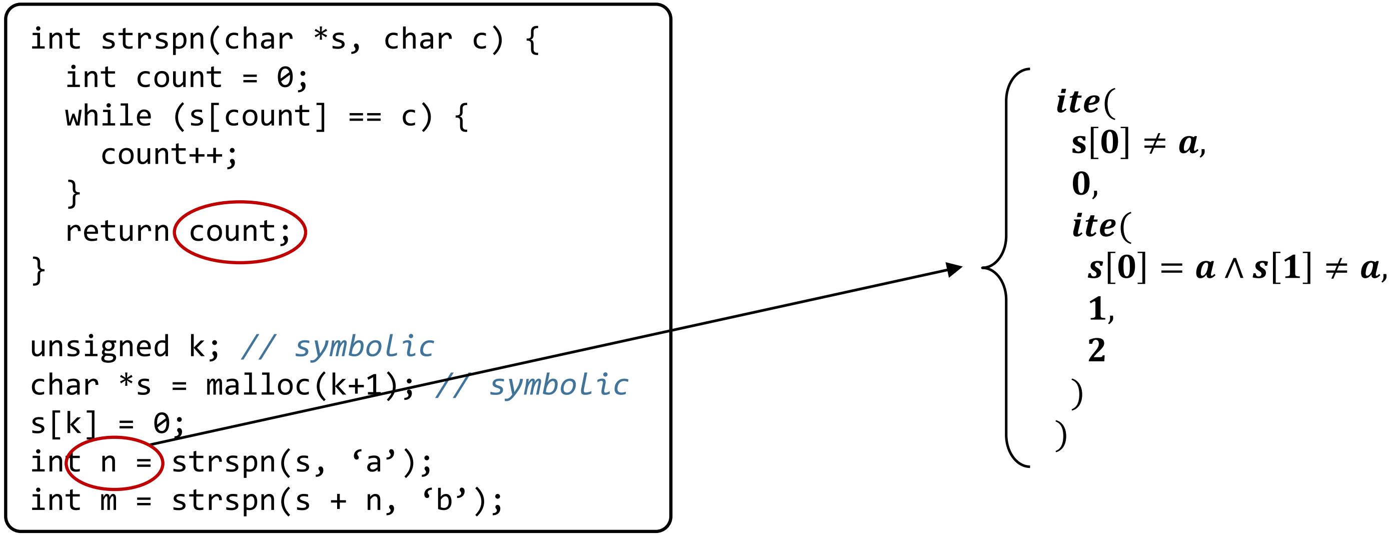


Standard State Merging

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
```

$$\left\{\begin{array}{l} \boldsymbol{ite}(\\ \quad \mathbf{s}[\mathbf{0}] \neq \boldsymbol{a}, \\ \quad \mathbf{0}, \\ \quad \boldsymbol{ite}(\\ \quad\quad \boldsymbol{s}[\mathbf{0}] = \boldsymbol{a} \wedge \boldsymbol{s}[\mathbf{1}] \neq \boldsymbol{a}, \\ \quad\quad \mathbf{1}, \\ \quad\quad \mathbf{2} \\ \quad) \\ ) \end{array}\right.$$

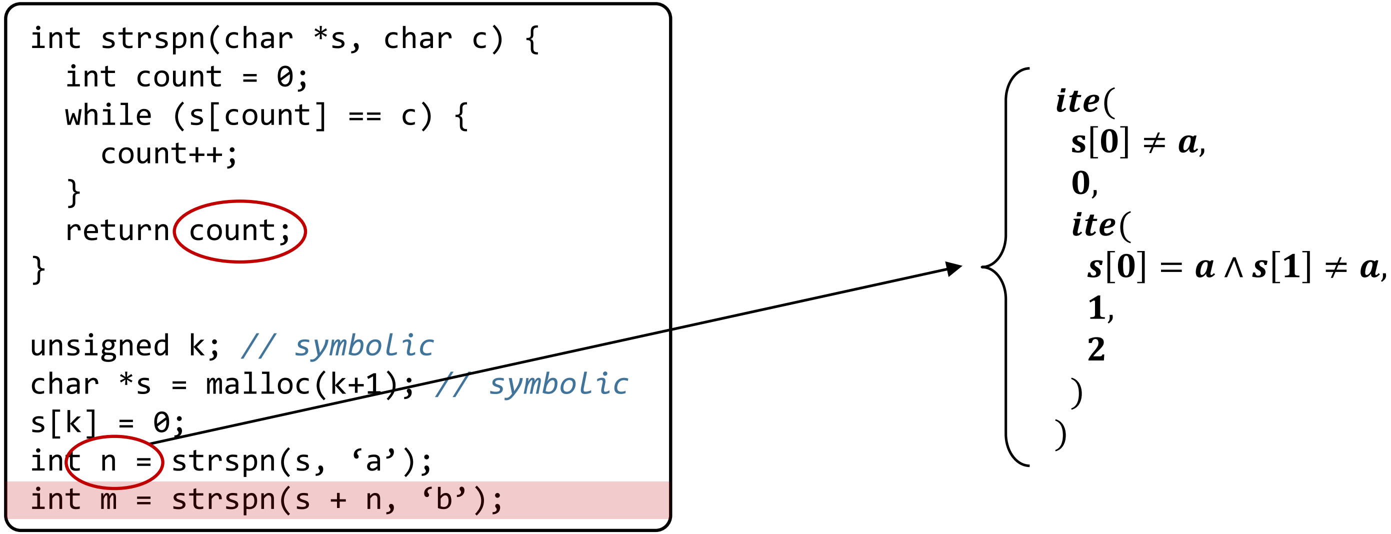
Standard State Merging



```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
```

$$\left\{\begin{array}{l}
\boldsymbol{ite}(\\
\quad \mathbf{s}[0] \neq \boldsymbol{a}, \\
\quad \mathbf{0}, \\
\quad \boldsymbol{ite}(\\
\quad\quad \boldsymbol{s}[0] = \boldsymbol{a} \wedge \boldsymbol{s}[1] \neq \boldsymbol{a}, \\
\quad\quad \mathbf{1}, \\
\quad\quad \mathbf{2} \\
\quad) \\
)
\end{array}\right.$$

Standard State Merging



```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, 'a');
int m = strspn(s + n, 'b');
```

$$
\begin{cases}
ite(\\
\quad s[0] \neq a, \\
\quad 0, \\
\quad ite(\\
\quad\quad s[0] = a \wedge s[1] \neq a, \\
\quad\quad 1, \\
\quad\quad 2 \\
\quad) \\
)
\end{cases}
$$

Standard State Merging

Path constraints

$\ldots \wedge$
$(s[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] \neq a) \vee$
$(s[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] = a \wedge s[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 1] \neq a) \vee$
$(s[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] = a \wedge s[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 1] = a \wedge s[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 2] \neq a)$

Value of **m**

```
$\mathit{ite}($
  $\mathrm{s}[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] \neq a,$
  $0,$
  $\mathit{ite}($
    $s[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] = a \wedge s[\mathit{ite}(\mathrm{s}[0] \neq a, 0, \mathit{ite}(s[0] = a \wedge s[1] \neq a, 1,2)) + 1] \neq a,$
    $1,$
    $2$
  $)$
$)$
```

State Merging with Quantifiers

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
```

State Merging with Quantifiers

Merging the path constraints

$$
\begin{aligned}
&(s[0] \neq a) \vee \\
&(s[0] = a \wedge s[1] \neq a) \vee \\
&(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)
\end{aligned}
$$

State Merging with Quantifiers

Merging the path constraints

$$\begin{aligned}&(s[0] \neq a) \lor \\ &(s[0] = a \land s[1] \neq a) \lor \\ &(s[0] = a \land s[1] = a \land s[2] \neq a)\end{aligned}$$

$$s[0] = a \land \cdots \land s[i-1] = a \land s[i] \neq a$$

$$\Updownarrow$$

State Merging with Quantifiers

Merging the path constraints

$$(s[0] \neq a) \lor$$
$$(s[0] = a \land s[1] \neq a) \lor$$
$$(s[0] = a \land s[1] = a \land s[2] \neq a)$$

$$s[0] = a \land \cdots \land s[i-1] = a \land s[i] \neq a$$

$$\Updownarrow$$

$$(\forall x. 1 \leq x \leq i \rightarrow s[x-1] = a) \land s[i] \neq a$$

bound variable →

State Merging with Quantifiers

Merging the path constraints

$$\begin{gathered}(s[0] \neq a) \lor \\ (s[0] = a \land s[1] \neq a) \lor \\ (s[0] = a \land s[1] = a \land s[2] \neq a)\end{gathered}$$

$$\Updownarrow$$

$$\begin{gathered}\big((\forall x.\, 1 \leq x \leq 0 \rightarrow s[x-1] = a) \land s[0] \neq a\big) \lor \\ \big((\forall x.\, 1 \leq x \leq 1 \rightarrow s[x-1] = a) \land s[1] \neq a\big) \lor \\ \big((\forall x.\, 1 \leq x \leq 2 \rightarrow s[x-1] = a) \land s[2] \neq a\big)\end{gathered}$$

State Merging with Quantifiers

Merging the path constraints

$$
\begin{array}{l}
(s[0] \neq a) \lor \\
(s[0] = a \land s[1] \neq a) \lor \\
(s[0] = a \land s[1] = a \land s[2] \neq a)
\end{array}
$$

$$\Updownarrow$$

$$
\begin{array}{l}
\big((\forall x.\, 1 \leq x \leq 0 \rightarrow s[x-1] = a) \land s[0] \neq a\big) \lor \\
\big((\forall x.\, 1 \leq x \leq 1 \rightarrow s[x-1] = a) \land s[1] \neq a\big) \lor \\
\big((\forall x.\, 1 \leq x \leq 2 \rightarrow s[x-1] = a) \land s[2] \neq a\big)
\end{array}
$$

$$\Updownarrow$$

$$0 \leq i \leq 2 \land (\forall x.\, 1 \leq x \leq i \rightarrow s[x-1] = a) \land s[i] \neq a$$

fresh free variable

State Merging with Quantifiers

Merging memory

$0 \leq i \leq 2 \wedge (\forall x. 1 \leq x \leq i \rightarrow s[x-1] = a) \wedge s[i] \neq a$

merged value of **n** $$\left\{ \begin{array}{l} ite(\\ \quad \mathrm{s}[0] \neq a, \\ \quad 0, \\ \quad ite(\\ \quad\quad s[0] = a \wedge s[1] \neq a, \\ \quad\quad 1, \\ \quad\quad 2 \\ \quad) \\ ) \end{array} \right.$$

State Merging with Quantifiers

Merging memory

$0 \leq i \leq 2 \land (\forall x. 1 \leq x \leq i \rightarrow s[x-1] = a) \land s[i] \neq a$

$$\text{merged value of } \mathbf{n} \left\{ \begin{array}{l} ite(\\ \quad s[0] \neq a, \\ \quad 0, \\ \quad ite(\\ \qquad s[0] = a \land s[1] \neq a, \\ \qquad 1, \\ \qquad 2 \\ \quad) \\ ) \end{array} \right. \qquad \Longrightarrow \qquad i$$

State Merging with Quantifiers

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
```

State Merging with Quantifiers

Path constraints

$\ldots \wedge 0 \le j \le 2 \wedge (\forall x.\, 1 \le x \le j \rightarrow s[i + x - 1] = b) \wedge s[i + j] \ne b$

Value of **m**

j

Synthesizing Quantified Constraints

path constrains

$(s[0] \neq a)$
$(s[0] = a \wedge s[1] \neq a)$
$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$

$\Downarrow$

abstraction

$$\begin{array}{ll} \beta & \alpha^0\beta \\ \alpha\beta & \alpha^1\beta \\ \alpha\alpha\beta & \alpha^2\beta \end{array} \left.\vphantom{\begin{array}{l}\beta\\\beta\\\beta\end{array}}\right\} \alpha^*\beta \quad \Rightarrow$$

synthesis constraints

$\varphi_\alpha(1) \overset{\text{def}}{=} s[0] = a$
$\varphi_\alpha(2) \overset{\text{def}}{=} s[1] = a$
$\Rightarrow \quad \varphi_\alpha(x) \overset{\text{def}}{=} s[x-1] = a$

$\varphi_\beta(0) \overset{\text{def}}{=} s[0] \neq a$
$\varphi_\beta(1) \overset{\text{def}}{=} s[1] \neq a$
$\varphi_\beta(2) \overset{\text{def}}{=} s[2] \neq a$
$\Rightarrow \quad \varphi_\beta(x) \overset{\text{def}}{=} s[x] \neq a$

$\Uparrow$

quantified path constraints

$0 \leq i \leq 2 \wedge (\forall x.1 \leq x \leq i \rightarrow \varphi_\alpha[x]) \wedge \varphi_\beta[i]$

Synthesizing Quantified Constraints

path constrains

$(s[0] \neq a)$
$(s[0] = a \land s[1] \neq a)$
$(s[0] = a \land s[1] = a \land s[2] \neq a)$

$\Longleftrightarrow$

quantified path constraints

$0 \leq i \leq 2 \land (\forall x. 1 \leq x \leq i \rightarrow \varphi_\alpha[x]) \land \varphi_\beta[i]$

$\Downarrow$

abstraction

$$\begin{array}{ll} \beta & \alpha^0\beta \\ \alpha\beta & \alpha^1\beta \\ \alpha\alpha\beta & \alpha^2\beta \end{array} \left.\vphantom{\begin{array}{l} a \\ a \\ a \end{array}}\right\} \alpha^*\beta$$

$\Rightarrow$

$$\overbrace{\hphantom{xxxxxxxxxxxxxxxxxxxx}}^{\Uparrow}$$

synthesis constraints

$\varphi_\alpha(1) \overset{\text{def}}{=} s[0] = a$
$\varphi_\alpha(2) \overset{\text{def}}{=} s[1] = a$
$\Rightarrow$ $\varphi_\alpha(x) \overset{\text{def}}{=} s[x-1] = a$

$\varphi_\beta(0) \overset{\text{def}}{=} s[0] \neq a$
$\varphi_\beta(1) \overset{\text{def}}{=} s[1] \neq a$
$\varphi_\beta(2) \overset{\text{def}}{=} s[2] \neq a$
$\Rightarrow$ $\varphi_\beta(x) \overset{\text{def}}{=} s[x] \neq a$

Additional Contributions

Specialized solving procedure

- Efficiently solving quantified formulas

Incremental state merging

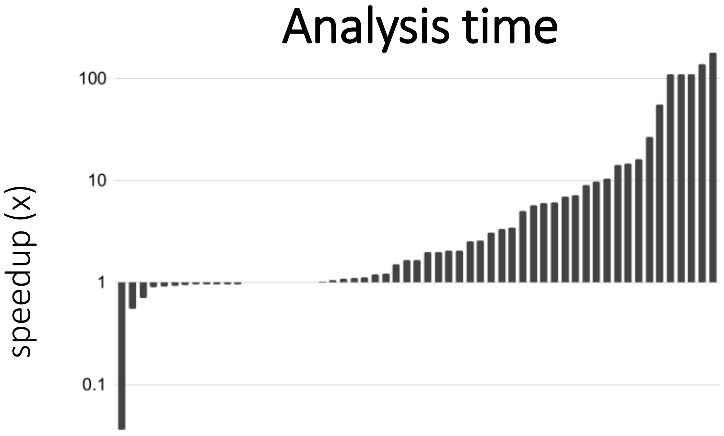
- Handling complex loops (exponential execution trees)

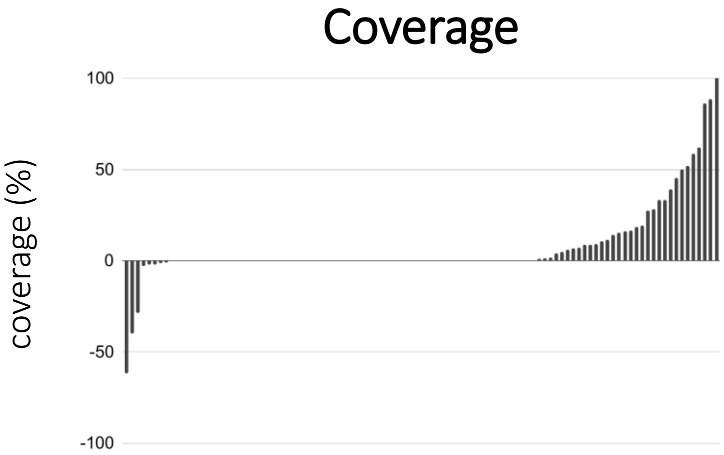
Evaluation

Implemented on top of ***KLEE***

Benchmarks:

- oSIP *(35 subjects)*
- wget *(31 subjects)*
- libtasn1 *(13 subjects)*
- libpng *(12 subjects)*
- apr *(20 subjects)*
- json-c *(5 subjects)*
- busybox *(30 subjects)*





Evaluation

Found bugs in ***klee-uclibc*** in the experiments with ***busybox***

- Two *memory out-of-bound's*

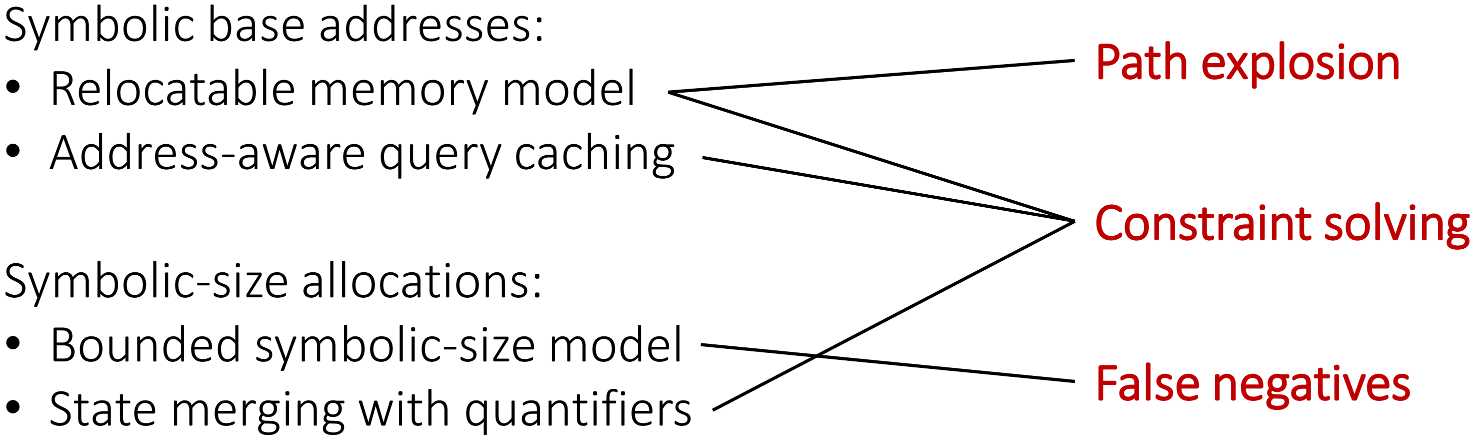
All the bugs were **confirmed** and **fixed**.

Outline

- Background
  - Symbolic execution
  - Memory model
- Symbolic base addresses
  - Relocatable memory model
  - Address-aware query caching
- Symbolic-size allocations
  - Bounded symbolic-size model
  - State merging with quantifiers
- Conclusions and future work

Summary

Tackle the challenges of **symbolic execution** using novel memory models



Symbolic base addresses:

- Relocatable memory model
- Address-aware query caching

Symbolic-size allocations:

- Bounded symbolic-size model
- State merging with quantifiers

Path explosion

Constraint solving

False negatives

Publications

Past-Sensitive Pointer Analysis for Symbolic Execution (FSE 2020)

- *D. Trabish, T. Kapus, N. Rinetzky, C. Cadar*

Relocatable Addressing Model for Symbolic Execution (ISSTA 2020)

- *D. Trabish, N. Rinetzky*

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

- *D. Trabish, S. Itzhaky, N. Rinetzky*

A Bounded Symbolic-Size Model for Symbolic Execution (FSE 2021)

- *D. Trabish, S. Itzhaky, N. Rinetzky*

State Merging with Quantifiers in Symbolic Execution (FSE 2023)

- *D. Trabish, N. Rinetzky, S. Shoham, V. Sharma*

Implementations

Past-Sensitive Pointer Analysis

- *https://github.com/davidtr1037/klee-pspa*

Relocatable Memory Model

- *https://github.com/davidtr1037/klee-ram*

Address-Aware Query Caching

- *https://github.com/davidtr1037/klee-aaqc*

Bounded Symbolic-Size Model

- *https://github.com/davidtr1037/klee-symsize*

State Merging with Quantifiers

- *https://github.com/davidtr1037/klee-quantifiers*

Future Work

- Generalizing the relocatable memory model
- Modeling unbounded objects
- More applications with quantified encoding
- Generalizing the solving procedure for quantified constraints

Thanks!

Backup

Publications

TODO

Symbolic State

TODO

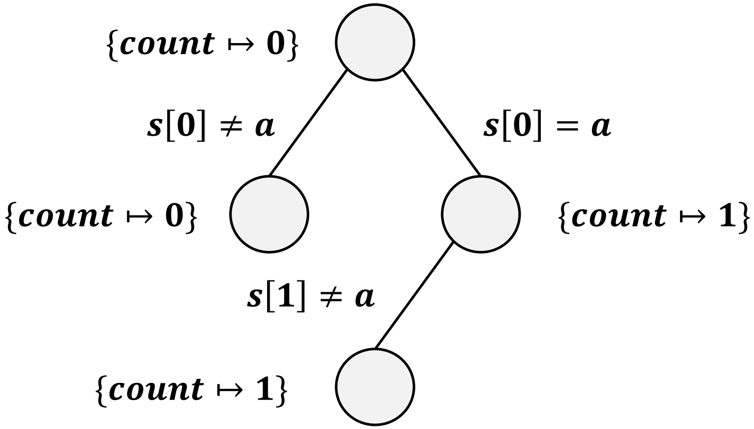
Logic / SMT Theories

TODO

Example

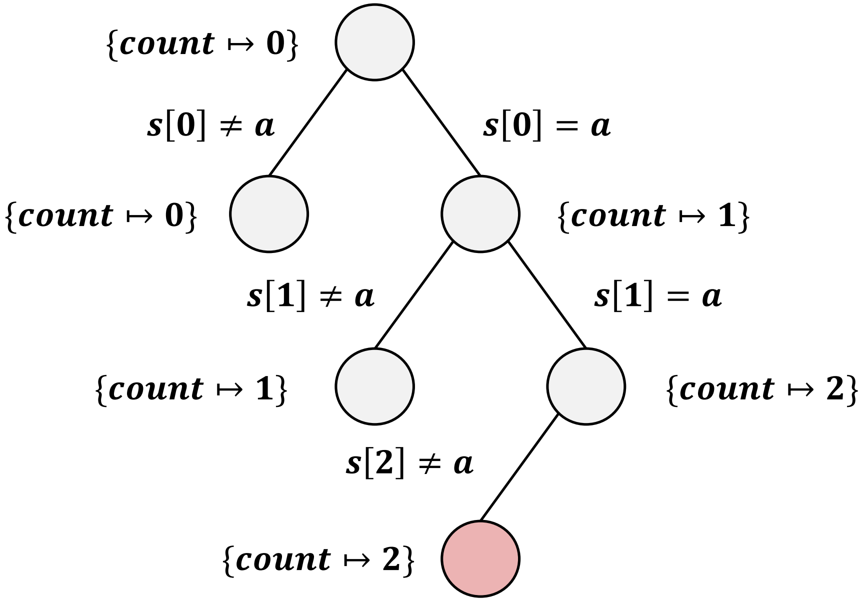
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, ‘a’);
if (n > 1) {
  // do something...
}
```

UNSAT



Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

unsigned k; // symbolic
char *s = malloc(k+1); // symbolic
s[k] = 0;
int n = strspn(s, ‘a’);
if (n > 1) {
  // ...
}
```



SAT

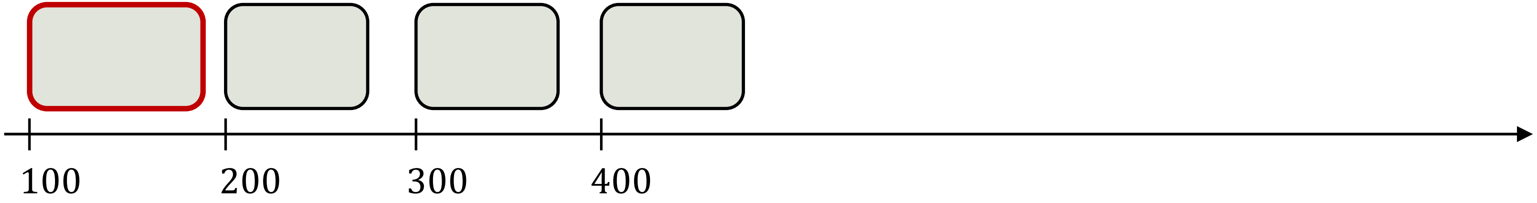
Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \stackrel{\text{def}}{=} select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i) + j$$

$$\overbrace{i < 2 \,\wedge\, j < 10 \,\wedge\, 100 \leq p < 116}^{\text{resolution query}}$$

UNSAT



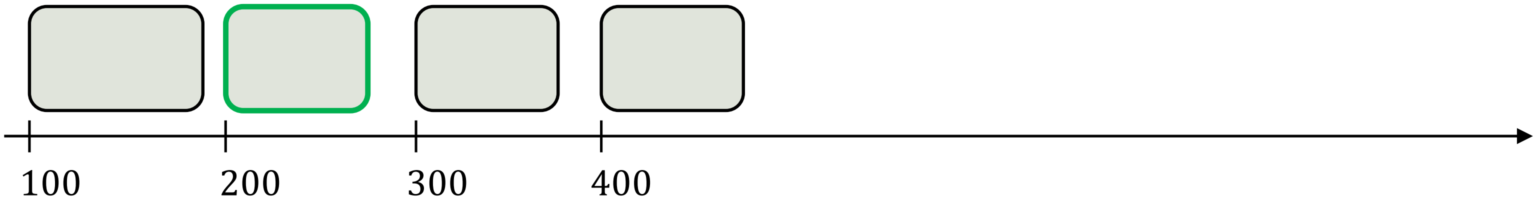
Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i) + j$$

$$\overbrace{i < 2 \;\wedge\; j < 10 \;\wedge\; 200 \le p < 210}^{\text{resolution query}}$$

SAT



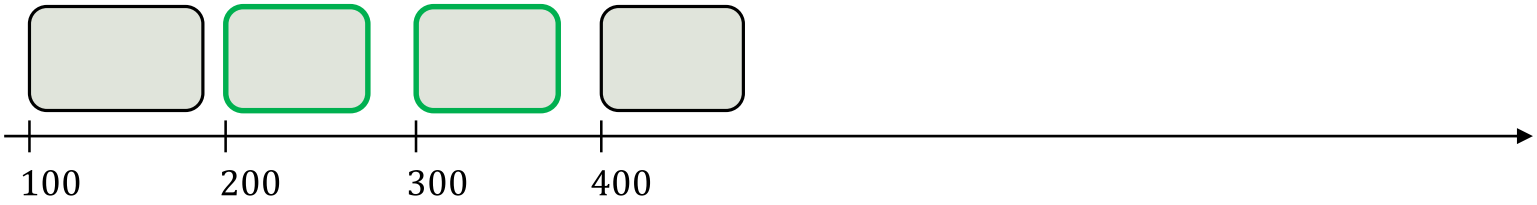
Symbolic Pointers

```
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i) + j$$

$$\overbrace{i < 2 \ \wedge\ j < 10 \ \wedge\ 300 \le p < 310}^{\text{resolution query}}$$

SAT



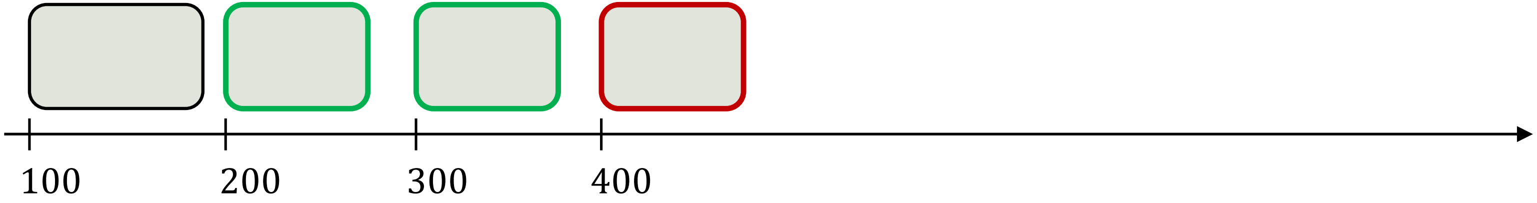
Symbolic Pointers

```c
char **array = calloc(3, PTR_SIZE);
for (int i = 0; i < 3; i++) {
  array[i] = calloc(10, 1);
}

// symbolic: i < 2, j < 10
unsigned i, j;
if (array[i][j] == 7) {
  // ...
}
```

$$p \overset{\text{def}}{=} select(a[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i) + j$$

$$\overbrace{i < 2 \land j < 10 \land 400 \le p < 410}^{\text{resolution query}}$$

UNSAT



Address-Dependent Queries

$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i) + j$

$pc \stackrel{\text{def}}{=} z \neq 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 200 \leq p < 210$

query:
$pc \ \wedge \ select(a_2, p - 200) = 7$

$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i) + j$

$pc \stackrel{\text{def}}{=} z = 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$

query:
$pc \ \wedge \ select(a_2, p - 600) = 7$

Address-Dependent Queries

$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i) + j$

$pc \stackrel{\text{def}}{=} z \neq 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 200 \leq p < 210$

query:

$pc \ \wedge \ select(a_2, p - 200) = 7$

$p \stackrel{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i) + j$

$pc \stackrel{\text{def}}{=} z = 0 \ \wedge \ i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$

query:

$pc \ \wedge \ select(a_2, p - 600) = 7$

Address-Dependent Queries

$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \ 200 \leq p < 210$

query:

$pc \ \wedge \ select(a_2, p - 200) = 7$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge \ j < 10 \ \wedge \ 600 \leq p < 610$

query:

$pc \ \wedge \ select(a_2, p - 600) = 7$

Address-Dependent Queries

$p \overset{\text{def}}{=} select(a_1[0 \mapsto 200, 1 \mapsto 300, 2 \mapsto 400], i) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge\ j < 10 \ \wedge\ 200 \leq p < 210$

query:

$pc \ \wedge\ select(a_2, p - 200) = 7$

$p \overset{\text{def}}{=} select(a_1[0 \mapsto 600, 1 \mapsto 700, 2 \mapsto 800], i) + j$

$pc \overset{\text{def}}{=} i < 2 \ \wedge\ j < 10 \ \wedge\ 600 \leq p < 610$

query:

$pc \ \wedge\ select(a_2, p - 600) = 7$

- Equisatisfiable
- Query caching **fails** (No common normal form)