Past Sensitive Pointer Analysis for Symbolic Execution

David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar

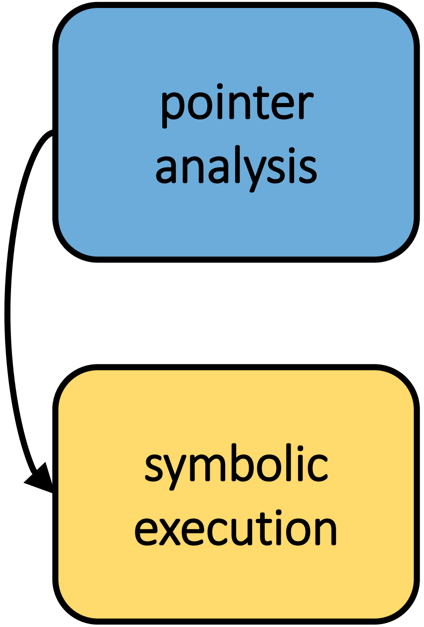
Tel-Aviv University, Israel   Imperial College, UK

ESEC/FSE 2020

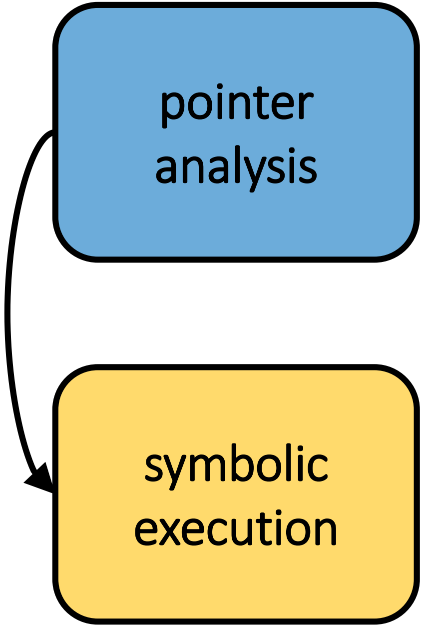
Symbolic Execution: Introduction

- Systematic program analysis technique
- Many applications:
  - Test input generation
  - Bug finding
  - ...
- Active research area
- Used in industry

Symbolic Execution & Pointer Analysis

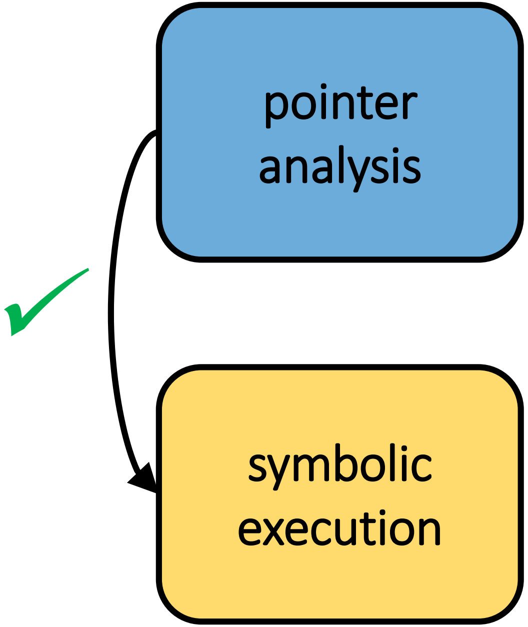


Symbolic Execution & Pointer Analysis



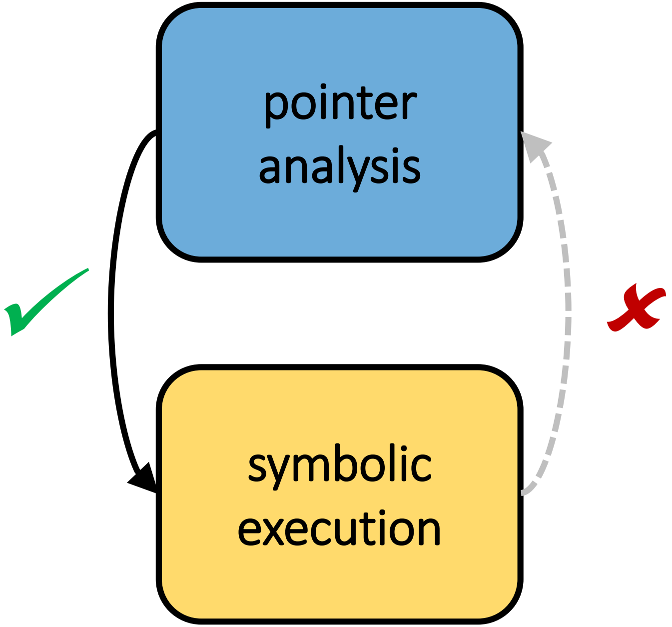
[Symbiotic] (TACAS'13)
[KATCH] (FSE'13)
[Chopper] (ICSE'18)
[Segmented memory model] (FSE'19)

Symbolic Execution & Pointer Analysis



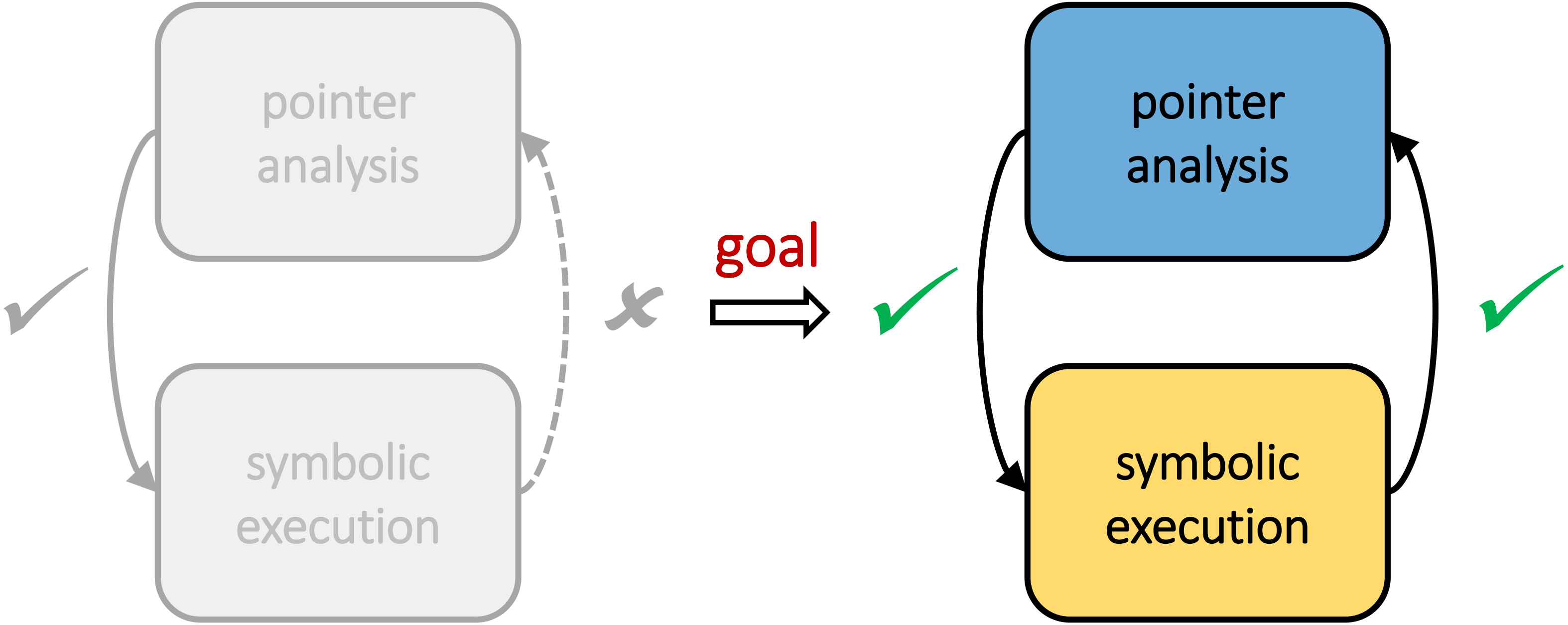
[Symbiotic] (TACAS'13)
[KATCH] (FSE'13)
[Chopper] (ICSE'18)
[Segmented memory model] (FSE'19)

Symbolic Execution & Pointer Analysis



[Symbiotic] (TACAS'13)
[KATCH] (FSE'13)
[Chopper] (ICSE'18)
[Segmented memory model] (FSE'19)

Symbolic Execution & Pointer Analysis



Example

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N];
for (int i = 0; i < N; i++)
  objs[i] = calloc(...);
...
objs[0]->p = malloc(...);
foo(objs[1]);
```

Example

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```

Example

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```

All objects allocated in the loop have **same** allocation site

Example

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```

All objects allocated in the loop have **same** allocation site

⇩

Can’t be distinguished between objs[0] and objs[1]

Example

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p) // pts: (C,0)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```

All objects allocated in the loop have **same** allocation site

⇩

Can't be distinguished between objs[0] and objs[1]

⇩

p may point to $(C, 0)$

Example

```c
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p) // pts: (C,0)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```

All objects allocated in the loop have **same** allocation site

⇩

Can't be distinguished between objs[0] and objs[1]

⇩

p may point to $(C, 0)$

⇩

False positive!

Goal

Run pointer analysis on-demand, not ahead of time:

- From a specific **symbolic state**
- On a specific function, **locally**

Past-Sensitive Pointer Analysis

- Distinguish between **past** and **future**:
  - Objects that were *already allocated*
  - Objects that might be *allocated during pointer analysis*
- **Local** pointer analysis

```
obj_t objs[N];
for (int i = 0; i < N; i++)
  objs[i] = calloc(...);
...
objs[0]->p = malloc(...);
foo(objs[1]);
```

```
void foo(obj_t *o) {
  if (o->p) // pts: (B, 1)
    o->d = 7; // pts: (B, 0)
}
```

Unique Allocation Sites

During symbolic execution:

- Allocated objects are associated with unique allocation sites

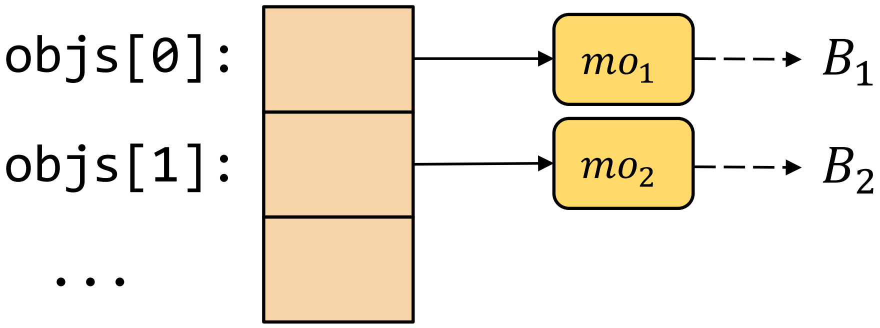
```
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
```

Unique Allocation Sites

During symbolic execution:

- Allocated objects are associated with unique allocation sites

```
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
```



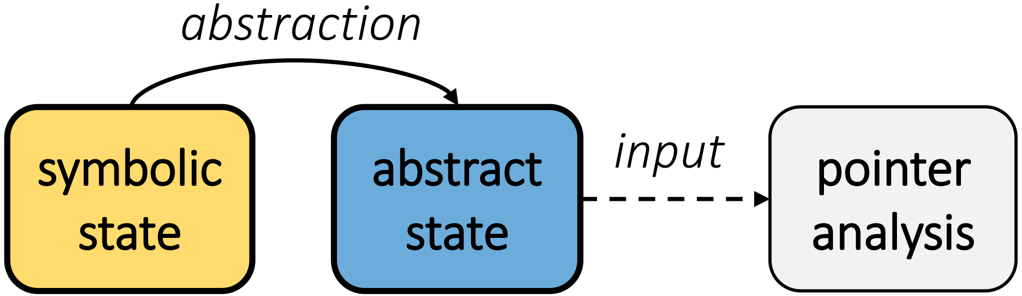
Local Pointer Analysis

When a symbolic state reaches a function call:

- Compute a **path-specific abstraction**
- Run pointer analysis from the **initial abstract state**

executed

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
...
objs[0]->p = malloc(...);
foo(objs[1]);
```



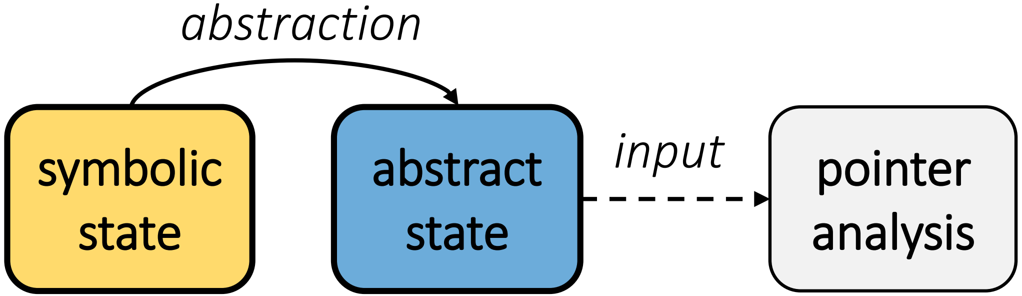
Local Pointer Analysis

Use current **symbolic state** to abstract:

- Traverse function parameters and global variables
- Translate to **points-to graph**

executed

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
...
objs[0]->p = malloc(...);
foo(objs[1]);
```



Local Pointer Analysis

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```

symbolic
state

Local Pointer Analysis

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```

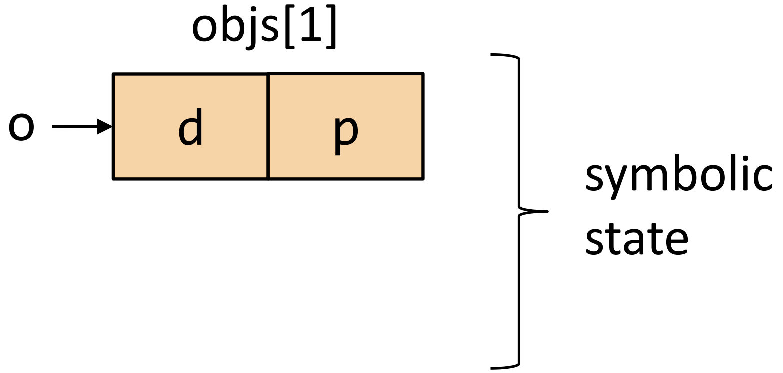
formal parameter

o

symbolic state

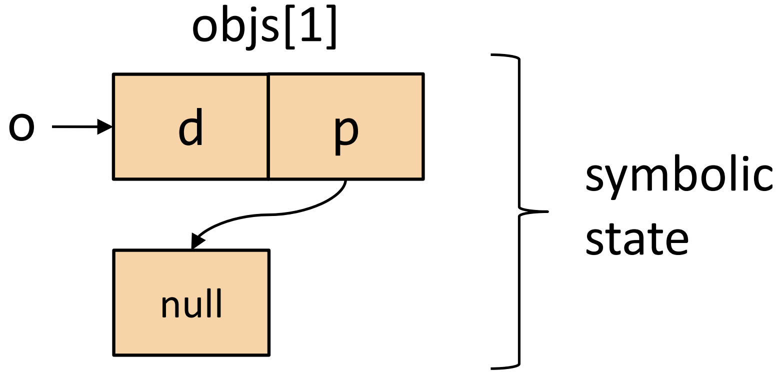
Local Pointer Analysis

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```



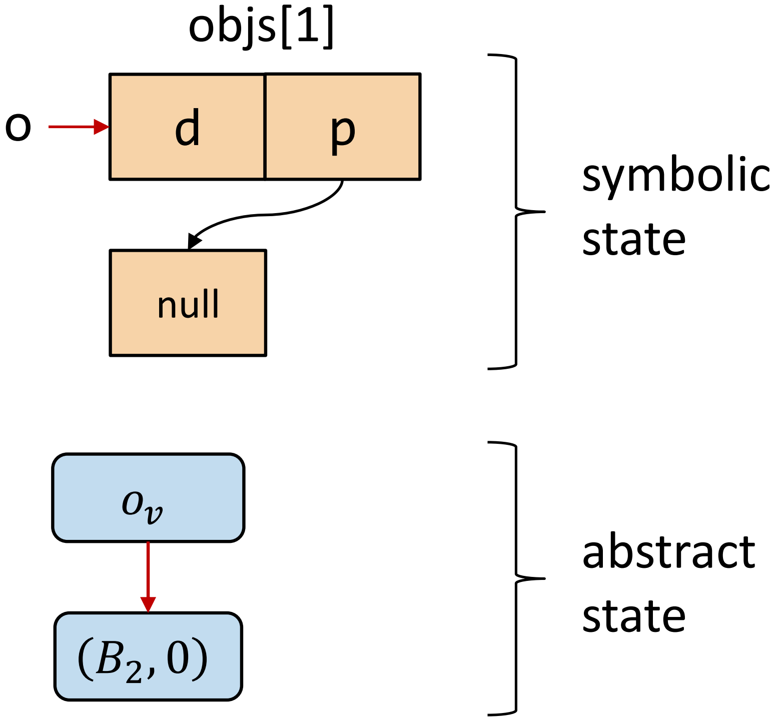
Local Pointer Analysis

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```



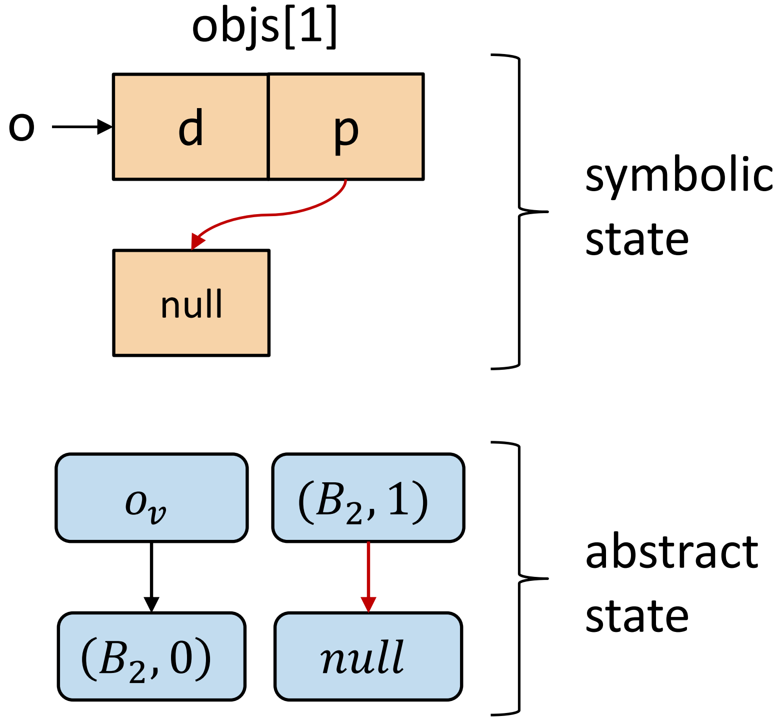
Local Pointer Analysis

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```



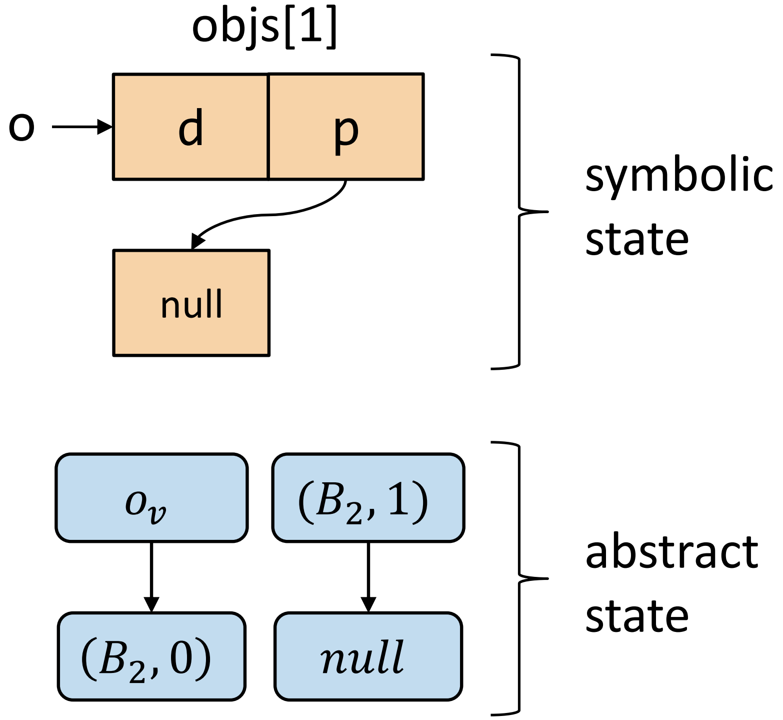
Local Pointer Analysis

```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```



Local Pointer Analysis

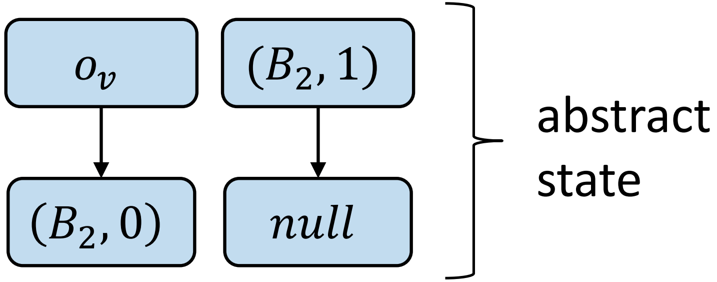
```
typedef struct { int x, *p; } obj_t;
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
obj_t objs[N]; // AS: A
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B
...
objs[0]->p = malloc(...); // AS: C
foo(objs[1]);
```



Local Pointer Analysis

Analyze **foo** from the initial abstract state:

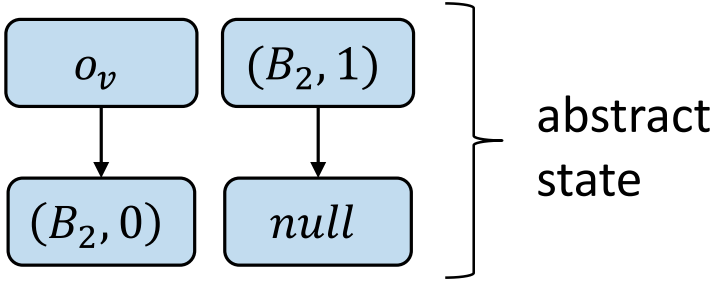
```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
```



Local Pointer Analysis

Analyze **foo** from the initial abstract state:

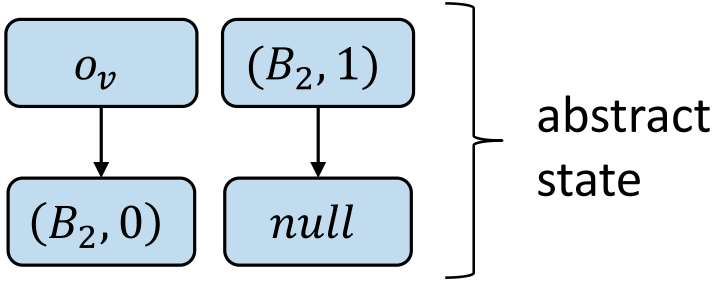
```
void foo(obj_t *o) {
  if (o->p) // pts: null
    o->d = 7;
}
```



Local Pointer Analysis

Analyze **foo** from the initial abstract state:

```
void foo(obj_t *o) {
  if (o->p) // pts: null
    o->d = 7;
}
```



No false positives!

Reusing Summaries

- Number of analyzed functions can be high
  - Running pointer analysis from scratch is **expensive**
- Empirical observation
  - Initial abstract states are **often isomorphic**

Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state

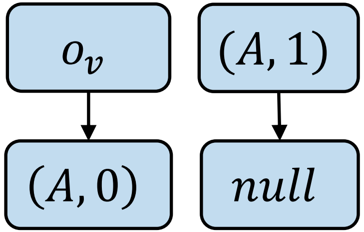
mod-set

Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state

mod-set

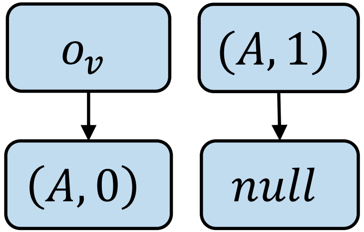


Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7; // pts: (A,0)
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state

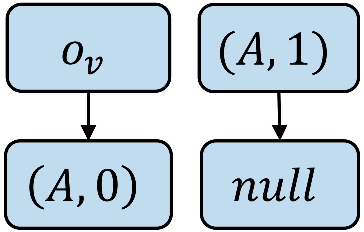
mod-set



Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state



mod-set

$\{(A, 0)\}$

Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state

$$\begin{tikzcd} o_v \arrow[d] & (A,1) \arrow[d] \\ (A,0) & null \end{tikzcd}$$

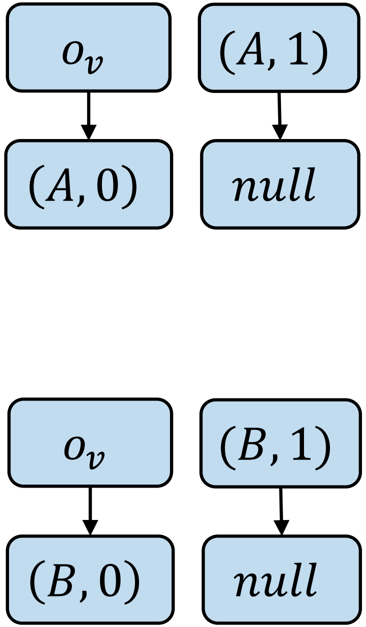
mod-set

$$\{(A,0)\}$$

Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state



mod-set

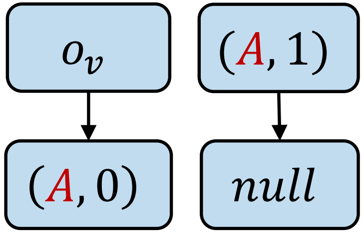
$\{(A, 0)\}$

Reusing Summaries

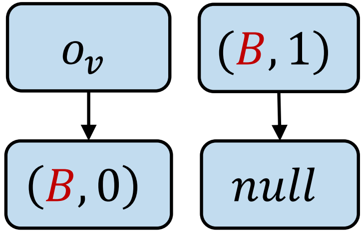
```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state

mod-set



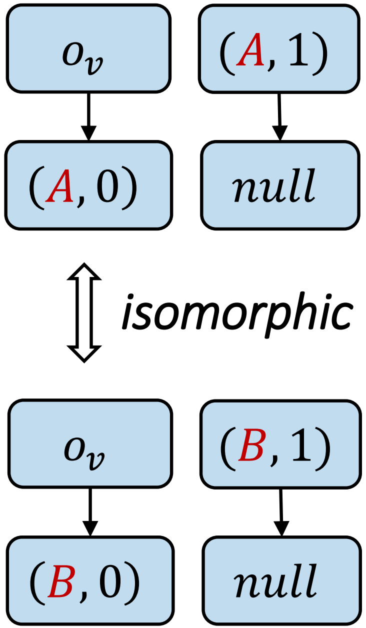
$\{(A, 0)\}$



Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state



mod-set

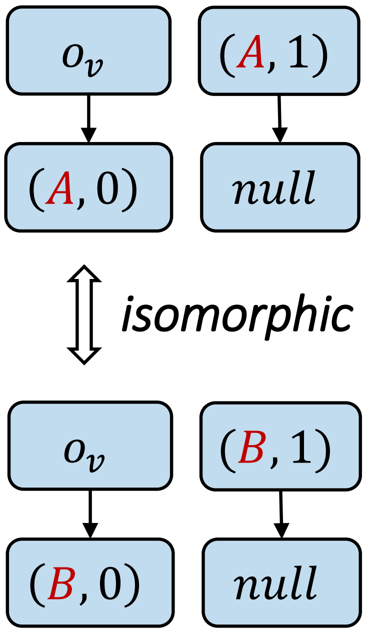
$\{(A, 0)\}$

Reusing Summaries

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
foo(o1);
...
foo(o2);
```

initial
abstract state

mod-set



$\{(A, 0)\}$

$\{(B, 0)\}$

Evaluation

Implemented using:

- KLEE *(https://github.com/klee/klee)*
- SVF *(https://github.com/SVF-tools/SVF)*

Client applications:

- Chopped symbolic execution (ICSE'18)
- Symbolic pointer resolution
- Write integrity testing (WIT)

Application: Chopped Symbolic Execution

- Skip user-specified functions
- Dynamically resolve side effects of skipped function
  - Relies on **static** mod-ref analysis

```
...
x = a + b;
foo(x);
z = x + 1;
...
```

```
void foo(x) {
  y = x * 2;
  ...
}
```

skip

Application: Chopped Symbolic Execution

Can we skip *foo* with **static** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B

foo(objs[1]);
int y = objs[0]->d;
```

Application: Chopped Symbolic Execution

Can we skip *foo* with **static** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B


foo(objs[1]);
int y = objs[0]->d;
```

- **mod-set** of foo: $\{(B, 0)\}$

Application: Chopped Symbolic Execution

Can we skip *foo* with **static** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B


foo(objs[1]);
int y = objs[0]->d;
```

- **mod-set** of foo: $\{(B, 0)\}$
- **read location** abstracted by $(B, 0)$

Application: Chopped Symbolic Execution

Can we skip *foo* with **static** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B


foo(objs[1]);
int y = objs[0]->d;
```

- **mod-set** of foo: $\{(B, 0)\}$
- **read location** abstracted by $(B, 0)$

Application: Chopped Symbolic Execution

Can we skip *foo* with **static** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B


foo(objs[1]);
int y = objs[0]->d;
```

- **mod-set** of foo: $\{(B, 0)\}$
- **read location** abstracted by $(B, 0)$
- false-dependency

Application: Chopped Symbolic Execution

Can we skip *foo* with **past-sensitive** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B

foo(objs[1]);
int y = objs[0]->d;
```

Application: Chopped Symbolic Execution

Can we skip *foo* with **past-sensitive** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B

foo(objs[1]);
int y = objs[0]->d;
```

- **mod-set** of foo: $\{(B_2, 0)\}$

Application: Chopped Symbolic Execution

Can we skip *foo* with **past-sensitive** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B


foo(objs[1]);
int y = objs[0]->d;
```

- **mod-set** of foo: $\{(B_2, 0)\}$
- **read location** abstracted by $(B_1, 0)$

Application: Chopped Symbolic Execution

Can we skip *foo* with **past-sensitive** pointer analysis?

```
void foo(obj_t *o) {
  if (o->p)
    o->d = 7;
}
...
for (int i = 0; i < N; i++)
  objs[i] = calloc(...); // AS: B


foo(objs[1]);
int y = objs[0]->d;
```

- **mod-set** of foo: $\{(B_2, 0)\}$
- **read location** abstracted by $(B_1, 0)$
- **no false-dependency**

Application: Chopped Symbolic Execution

Compare **static** and **past-sensitive** mod-ref analysis:

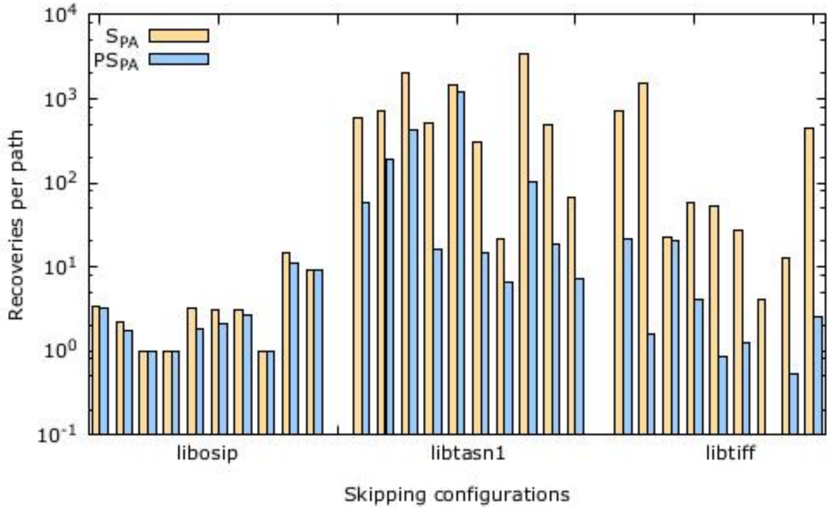
- Reducing recoveries
- Code coverage
- Failure reproduction

Application: Chopped Symbolic Execution

Reducing recoveries

- Several configurations of skipped functions
- Record number of **recoveries per path**
- Show relative reduction compared to static pointer analysis

Benchmark	Min	Max
libosip	0%	57%
libtiff	61%	99%
libtasn1	17%	99%



Application: Chopped Symbolic Execution

Code coverage

- Manually select skipped functions
- Measure coverage (lines)

Benchmark	Search	Static	PSPA
libosip	DFS	567	519
	Random	592	647
libtiff	DFS	958	1079
	Random	950	1019
Libtasn1	DFS	669	673
	Random	647	1034

Application: Chopped Symbolic Execution

Failure reproduction

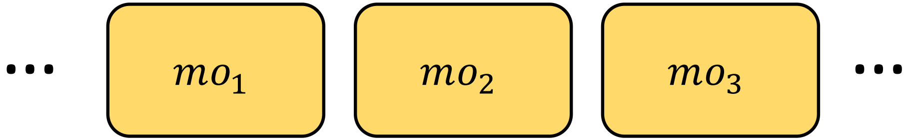
- Measure time required to find bugs *(DFS search heuristic)*

CVE	Chopping-aware heuristic			
	Without		With	
	Static	PSPA	Static	PSPA
2012-1569	04:57	01:46	00:11	00:06
2014-3467-1	04:17	02:15	00:01	00:01
2014-3467-2	T.O.	T.O.	04:23	00:37
2014-3467-3	T.O.	T.O.	00:02	00:02
2015-2806	T.O.	10:14	T.O.	10:25
2015-3622	T.O.	07:25	07:16	06:33

Time: ***mm:ss***

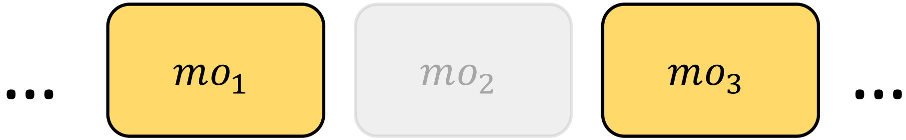
Application: Symbolic Pointer Resolution

- Symbolic pointers may point to **multiple** objects
- Resolve by **scanning** the memory
  - Construct a SMT query for each scanned object
  - SMT queries are expensive



Application: Symbolic Pointer Resolution

- Can improve using **points-to information**
- Compute the points-to set of the symbolic pointer
- If an object is not contained:
  - Skip it, and avoid solver queries…



Application: Symbolic Pointer Resolution

- Computing with static pointer analysis is **trivial**
- How to compute with past-sensitive pointer analysis?

```
void foo(char *key) {
  h = hash(key);
  // symbolic pointer
  p = o->table[h];
  if (p->x > 7)
  ...
}
```

← symbolic pointer

Application: Symbolic Pointer Resolution

- Computing with static pointer analysis is **trivial**
- How to compute with past-sensitive pointer analysis?
  - Run the analysis from the **calling function**

```
void foo(char *key) {
  h = hash(key);
  // symbolic pointer
  p = o->table[h];
  if (p->x > 7)
  ...
}
```

← analyze from here

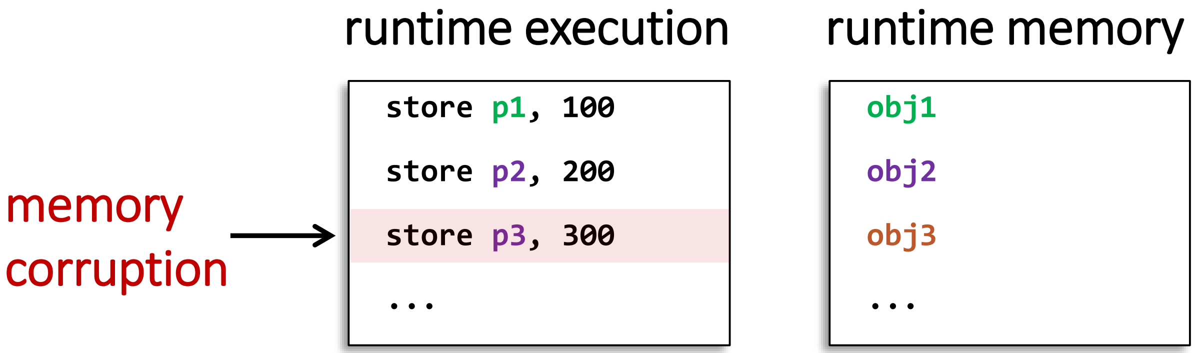
Application: Symbolic Pointer Resolution

Benchmark	Mode	Time (minutes)	Queries
m4	Baseline	49	1902
	Static	47	1836
	PSPA	34	960
make	Baseline	65	21832
	Static	60	18872
	PSPA	41	6222
sqlite	Baseline	43	7726
	Static	51	7726
	PSPA	33	1166

Application: WIT

Write integrity testing (WIT)

- Detect memory corruptions in runtime (in production)
- Each **pointer** and **object** are assigned a **color**
- Mismatch means memory corruption



Application: WIT

Color assignment relies on **static pointer analysis**

- Compute points-to sets for each store operand (pointer)
- Merge intersecting points-to sets until all are disjoint
- Each points-to set corresponds to a unique color

Application: WIT

- Need a more precise color assignment
- Compute colors only **after the initialization code** completes
  - Use past-sensitive pointer analysis

```
int main() {
  // initialization code
  ...
  run();
  ...
}
```

← analyze from here

Application: WIT

Evaluation:

- Compute the number of colors
- Record the number of color transitions (between allocations)
    - During one hour

Benchmark	Paths	Colors		Transitions	
		Static	PSPA	Static	PSPA
libosip	12,084,552	70	277	108,532,593	302,069,717
libtasn1	90,289	157	645	8,848,322	39,456,279
libtiff	300	1047	1101	1,938	1,938

Future Work

- Integrate symbolic execution with other static analyses
  - Constant propagation, numerical analysis, etc.
- Apply to other pointer analyses
  - Flow-sensitive, context sensitive, etc.
- More client applications

Summary

- **Tighter integration** between symbolic execution and pointer analysis
- Evaluated with several client applications:
    - Chopped symbolic execution
    - Symbolic pointer resolution
    - WIT

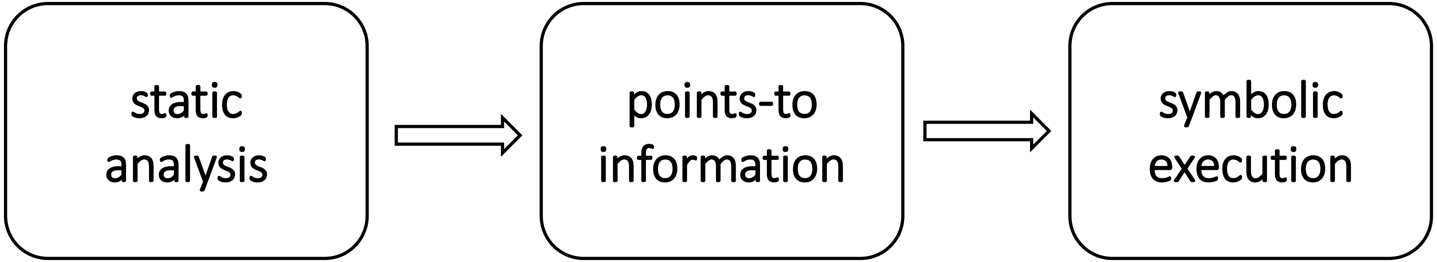
Available on github: https://github.com/davidtr1037/klee-pspa
Project page: https://srg.doc.ic.ac.uk/projects/pspa/

Backup

Symbolic Execution & Pointer Analysis

Combined in an offline manner:

- First run pointer analysis
- Then use the results during the symbolic execution



Enhances symbolic execution ✓

Whole-program analysis is imprecise ✗

Not using dynamic information ✗

Static Pointer Analysis

- Statically computes an **over-approximation** of **points-to** information
- Typically a **whole-program** analysis

Abstract Domain

In ***static*** pointer analysis we have:

- Static allocation sites (AS)
- Top level pointers (V)
- $O = AS \times (N \cup \{*\})$
    - Objects (field sensitive and field insensitive)

Abstract domain:

- $D = 2^V \times 2^O \times 2^{(V \cup O) \times O}$

Past-Sensitive Abstract Domain

We extend the original domain:

- Static (AS) and **unique** (AS_{unique}) allocation sites
  - $AS_{unique} = AS \times N$ (*distinct copies* of the static allocation sites)
- Top level pointers (V)
- $O = AS \times (N \cup \{*\})$

Abstract domain:

- $D = 2^V \times 2^O \times 2^{(V \cup O) \times O}$

Application: Symbolic Pointer Resolution

From where do we get the symbolic state snapshot?

- Take snapshots at every function call entry
  - Might be expensive on some benchmarks
- Learn the relevant location on the fly
  - Symbolic pointers usually appear in the same locations