

L25: Pointer Analysis

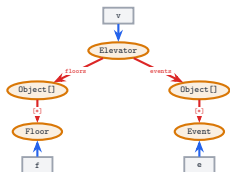
Software Analysis

IIT Guwahati · October 6, 2026

3: Three Heap Abstractions (1/3)

← more precise

more efficient →

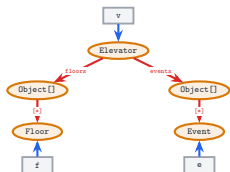


Allocation-site

5 ovals

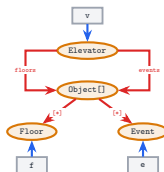
3: Three Heap Abstractions (2/3)

← more precise more efficient →



Allocation-site

5 ovals



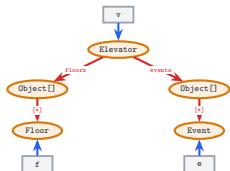
Type-based

4 ovals

3: Three Heap Abstractions (3/3)

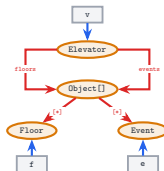
← more precise

more efficient →



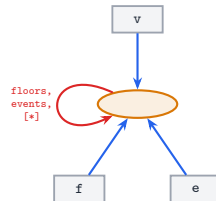
Allocation-site

5 ovals



Type-based

4 ovals



Heap-insensitive

1 oval

Naik, CIS 5470, Penn: Pointer Analysis

Answering May-Alias Queries (1/6)

May-alias?

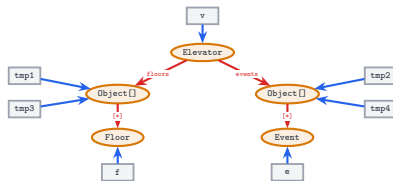
e, f

v.floors, v.events

v.floors[0], v.events[0]

v.events[0], v.events[2]

Allocation-site



Answering May-Alias Queries (2/6)

May-alias?

e, f

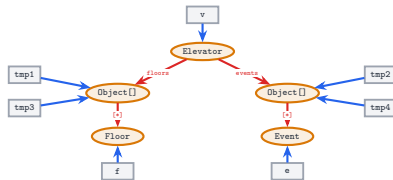
v.floors, v.events

v.floors[0], v.events[0]

v.events[0], v.events[2]

Allocation-site

NO



Answering May-Alias Queries (3/6)

May-alias?

e, f

v.floors, v.events

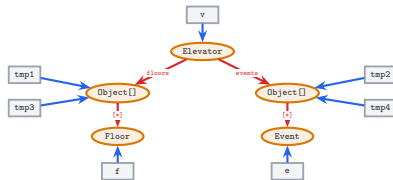
v.floors[0], v.events[0]

v.events[0], v.events[2]

Allocation-site

NO

NO



Answering May-Alias Queries (4/6)

May-alias?

e, f

v.floors, v.events

v.floors[0], v.events[0]

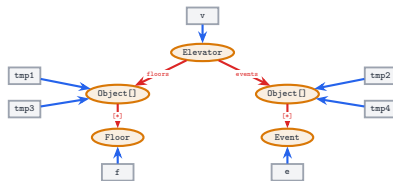
v.events[0], v.events[2]

Allocation-site

NO

NO

NO



Answering May-Alias Queries (5/6)

May-alias?

e, f

v.floors, v.events

v.floors[0], v.events[0]

v.events[0], v.events[2]

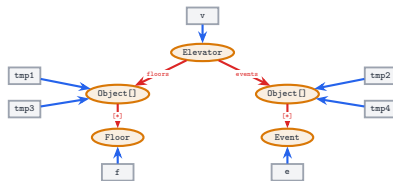
Allocation-site

NO

NO

NO

YES



Naik, CIS 5470, Penn: Pointer Analysis

Answering May-Alias Queries (6/6)

May-alias?

Allocation-site

e, f

NO

v.floors, v.events

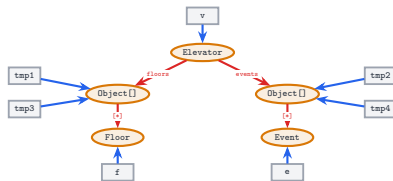
NO

v.floors[0], v.events[0]

NO

v.events[0], v.events[2]

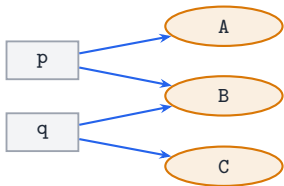
YES



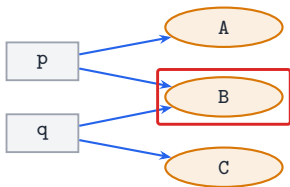
Read each expression as a **path** in the graph. Disjoint end sets means **NO**; overlapping means **YES**.

From Points-to Sets to Alias Answers (1/5)

$$pts(p) = \{A, B\}, \quad pts(q) = \{B, C\}$$



From Points-to Sets to Alias Answers (2/5)

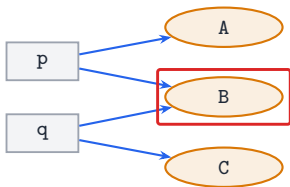


$pts(p) = \{A, B\}, \quad pts(q) = \{B, C\}$

$pts(p) \cap pts(q) = \{B\} \neq \emptyset$:

may-alias.

From Points-to Sets to Alias Answers (3/5)



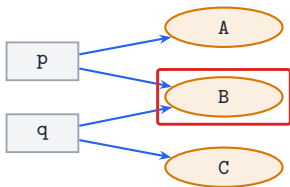
$pts(p) = \{A, B\}, \quad pts(q) = \{B, C\}$

$pts(p) \cap pts(q) = \{B\} \neq \emptyset$:

may-alias.

$\cap = \emptyset$ would be a **proof** of no-alias.

From Points-to Sets to Alias Answers (4/5)



$pts(p) = \{A, B\}, \quad pts(q) = \{B, C\}$

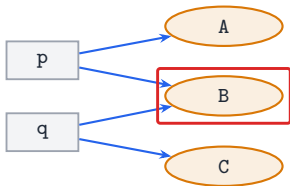
$pts(p) \cap pts(q) = \{B\} \neq \emptyset$:

may-alias.

$\cap = \emptyset$ would be a **proof** of no-alias.

$\neq \emptyset$ is only “cannot rule it out” — possibly a **false positive**.

From Points-to Sets to Alias Answers (5/5)



$pts(p) = \{A, B\}, \quad pts(q) = \{B, C\}$

$pts(p) \cap pts(q) = \{B\} \neq \emptyset$:

may-alias.

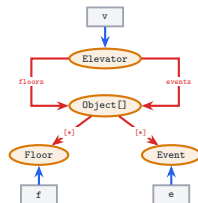
$\cap = \emptyset$ would be a **proof** of no-alias.

$\neq \emptyset$ is only “cannot rule it out” — possibly a **false positive**.

$pts(p) = \{A, B\}$ means “p may hold an address from A or B” — one address at a time, **not both at once**.

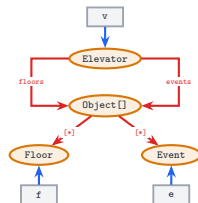
Type-Based (1/6)

May-alias?	Alloc-site	Type
<code>e, f</code>	NO	
<code>v.floors, v.events</code>	NO	
<code>v.floors[0], v.events[0]</code>	NO	
<code>v.events[0], v.events[2]</code>	YES	



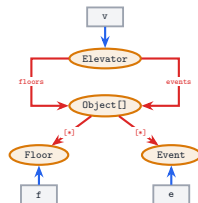
Type-Based (2/6)

May-alias?	Alloc-site	Type
<code>e, f</code>	NO	NO
<code>v.floors, v.events</code>	NO	
<code>v.floors[0], v.events[0]</code>	NO	
<code>v.events[0], v.events[2]</code>	YES	



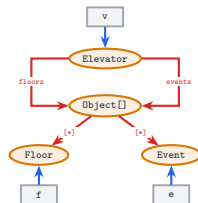
Type-Based (3/6)

May-alias?	Alloc-site	Type
<code>e, f</code>	NO	NO
<code>v.floors, v.events</code>	NO	YES
<code>v.floors[0], v.events[0]</code>	NO	
<code>v.events[0], v.events[2]</code>	YES	



Type-Based (4/6)

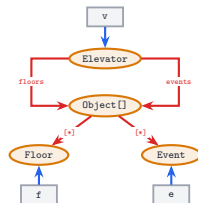
May-alias?	Alloc-site	Type
<code>e, f</code>	NO	NO
<code>v.floors, v.events</code>	NO	YES
<code>v.floors[0], v.events[0]</code>	NO	YES
<code>v.events[0], v.events[2]</code>	YES	



Type-Based (5/6)

May-alias?	Alloc-site	Type
e, f	NO	NO
v.floors, v.events	NO	YES
v.floors[0], v.events[0]	NO	YES
v.events[0], v.events[2]	YES	YES

Red marks an answer this scheme lost.

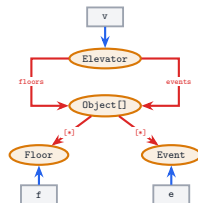


Type-Based (6/6)

May-alias?	Alloc-site	Type
e, f	NO	NO
v.floors, v.events	NO	YES
v.floors[0], v.events[0]	NO	YES
v.events[0], v.events[2]	YES	YES

Red marks an answer this scheme lost.

One merged oval, two proofs gone. **That is the price of the cheaper scheme.**



4: Modelling Arrays (1/4)

- Aggregate data structures include **arrays** and **records**. We first consider arrays.

4: Modelling Arrays (2/4)

- Aggregate data structures include **arrays** and **records**. We first consider arrays.
- A common abstraction represents all elements by a **single summary field** `[*]`.

4: Modelling Arrays (3/4)

- Aggregate data structures include **arrays** and **records**. We first consider arrays.
- A common abstraction represents all elements by a **single summary field** $[*]$.
- Individual elements are therefore not distinguished:

$$a[0] \equiv a[*] \equiv a[2].$$

4: Modelling Arrays (4/4)

- Aggregate data structures include **arrays** and **records**. We first consider arrays.
- A common abstraction represents all elements by a **single summary field** $[*]$.
- Individual elements are therefore not distinguished:

$$a[0] \equiv a[*] \equiv a[2].$$

- **Array dependence analysis** provides finer-grained reasoning about array subscripts.

4: Modelling Records (1/5)

Three choices on a record type: fields f_1 , f_2 and objects a_1 , a_2 , so four slots. **Same colour** = merged.

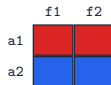
1. **Field-insensitive** merge **all** fields of **each** object
2. **Field-based** merge **each** field across **all** objects
3. **Field-sensitive** keep **each** field of **each** object apart

4: Modelling Records (2/5)

Three choices on a record type: fields `f1`, `f2` and objects `a1`, `a2`, so four slots. **Same colour** = merged.

1. Field-insensitive

merge **all** fields of **each** object



2. Field-based

merge **each** field across **all** objects

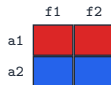
3. Field-sensitive

keep **each** field of **each** object apart

4: Modelling Records (3/5)

Three choices on a record type: fields f_1 , f_2 and objects a_1 , a_2 , so four slots. **Same colour** = merged.

1. **Field-insensitive** merge **all** fields of **each** object



2. **Field-based** merge **each** field across **all** objects

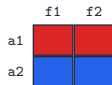


3. **Field-sensitive** keep **each** field of **each** object apart

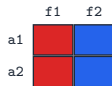
4: Modelling Records (4/5)

Three choices on a record type: fields f_1 , f_2 and objects a_1 , a_2 , so four slots. **Same colour** = merged.

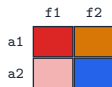
1. **Field-insensitive** merge **all** fields of **each** object



2. **Field-based** merge **each** field across **all** objects



3. **Field-sensitive** keep **each** field of **each** object apart



4: Modelling Records (5/5)

Three choices on a record type: fields $f1$, $f2$ and objects $a1$, $a2$, so four slots. **Same colour** = merged.

1. **Field-insensitive** merge **all** fields of **each** object

	f1	f2
a1	red	red
a2	blue	blue

2. **Field-based** merge **each** field across **all** objects

	f1	f2
a1	red	blue
a2	red	blue

3. **Field-sensitive** keep **each** field of **each** object apart

	f1	f2
a1	red	orange
a2	pink	blue

Ours is **field-sensitive**: floors and events never merged.

Stack-Directed Pointers (1/4)

```
void f(void) {  
    int a = 3, b = 4;  
    int *p = &a;      /* stack */  
    int *q = malloc(4); /* heap */  
    p = q;  
}
```

Stack-Directed Pointers (2/4)

```
void f(void) {  
    int a = 3, b = 4;  
    int *p = &a;      /* stack */  
    int *q = malloc(4); /* heap */  
    p = q;  
}
```

Heap-directed: from malloc or new. All of Java, and everything so far.

Stack-Directed Pointers (3/4)

```
void f(void) {  
    int a = 3, b = 4;  
    int *p = &a;      /* stack */  
    int *q = malloc(4); /* heap */  
    p = q;  
}
```

Heap-directed: from malloc or new. All of Java, and everything so far.

Stack-directed: from &, pointing at a named variable.

Stack-Directed Pointers (4/4)

```
void f(void) {  
    int a = 3, b = 4;  
    int *p = &a;      /* stack */  
    int *q = malloc(4); /* heap */  
    p = q;  
}
```

Heap-directed: from malloc or new. All of Java, and everything so far.

Stack-directed: from &, pointing at a named variable.

Statements for Stack-Based Analysis (1/6)

```
s ::= v1 = &v2  
    | v1 = v2  
    | v1 = *v2  
    | *v1 = v2
```

Statements for Stack-Based Analysis (2/6)

$s ::= v1 = \&v2$ address-of
| $v1 = v2$
| $v1 = *v2$
| $*v1 = v2$

Statements for Stack-Based Analysis (3/6)

<code>s ::= v1 = &v2</code>	address-of
<code> v1 = v2</code>	copy
<code> v1 = *v2</code>	
<code> *v1 = v2</code>	

Statements for Stack-Based Analysis (4/6)

<code>s ::= v1 = &v2</code>	address-of
<code>v1 = v2</code>	copy
<code>v1 = *v2</code>	indirect read
<code>*v1 = v2</code>	indirect write

Statements for Stack-Based Analysis (5/6)

$s ::= v1 = \&v2$	address-of
$v1 = v2$	copy
$v1 = *v2$	indirect read
$*v1 = v2$	indirect write

Four forms instead of six: with no fields, reads and writes merge into one pair.

Statements for Stack-Based Analysis (6/6)

$s ::= v1 = \&v2$	address-of
$v1 = v2$	copy
$v1 = *v2$	indirect read
$*v1 = v2$	indirect write

Four forms instead of six: with no fields, reads and writes merge into one pair.

Every node in the graph is now a **variable box**. There are no ovals.

Stack Rules: Creating and Copying (1/4)



`u = &v`

After



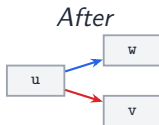
`u = v`

After

Stack Rules: Creating and Copying (2/4)



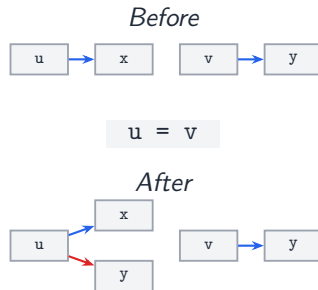
`u = &v`



`u = v`

After

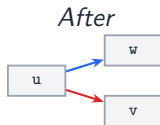
Stack Rules: Creating and Copying (3/4)



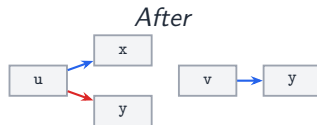
Stack Rules: Creating and Copying (4/4)



`u = &v`

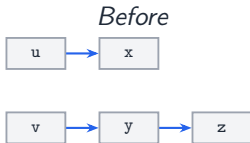


`u = v`



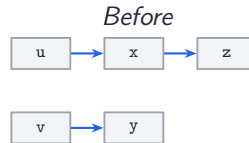
Weak updates again: `u` keeps its old edge to `w`, and to `x`.

Stack Rules: Dereferencing (1/4)



$u = *v$

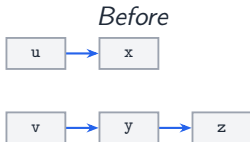
After



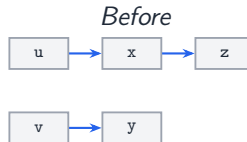
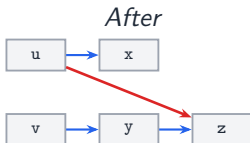
$*u = v$

After

Stack Rules: Dereferencing (2/4)



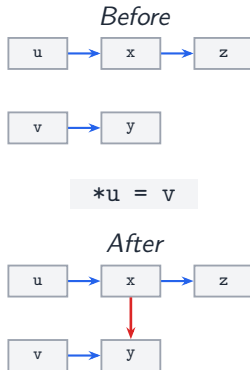
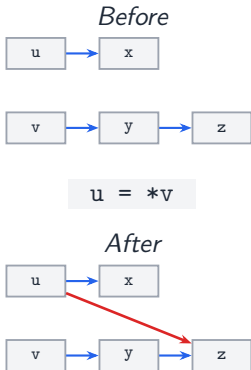
`u = *v`



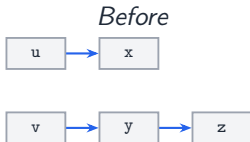
`*u = v`

After

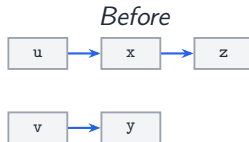
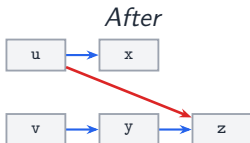
Stack Rules: Dereferencing (3/4)



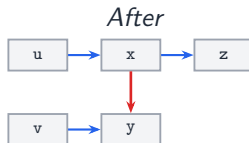
Stack Rules: Dereferencing (4/4)



`u = *v`



`*u = v`



Note where the write lands: **not** on `u`, but on everything `u` points to.

A C Program, Statement by Statement (1/7)

```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```

A C Program, Statement by Statement (2/7)

```
p = &a;
```

```
q = &b;
```

```
p = q;
```

```
r = &p;
```

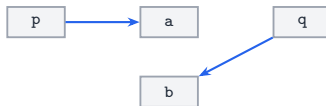
```
*r = &c;
```

```
q = *r;
```



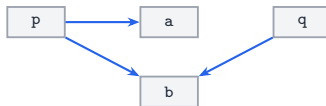
A C Program, Statement by Statement (3/7)

```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



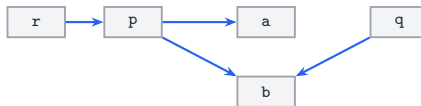
A C Program, Statement by Statement (4/7)

```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



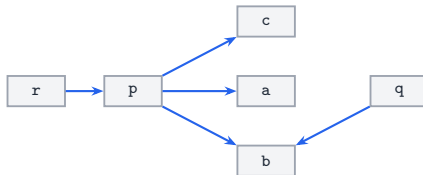
A C Program, Statement by Statement (5/7)

```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



A C Program, Statement by Statement (6/7)

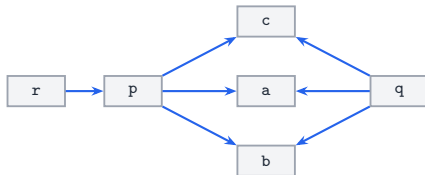
```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



`*r = &c` is two statements: `tmp = &c` then `*r = tmp`.

A C Program, Statement by Statement (7/7)

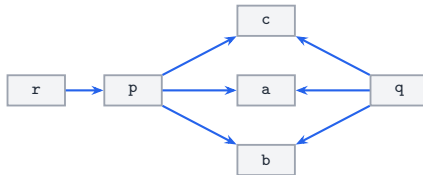
```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



`*r = &c` is two statements: `tmp = &c` then `*r = tmp`. One pass suffices here: no loops.

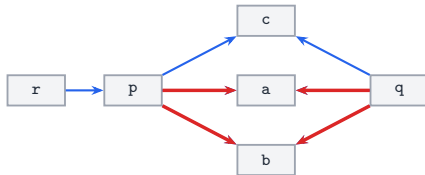
Sound, and Imprecise (1/3)

```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



Sound, and Imprecise (2/3)

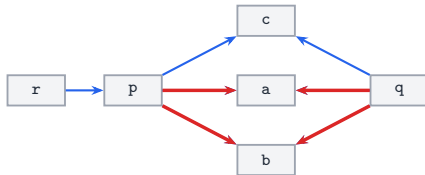
```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



At the end, p and q hold **only** &c. The four **red** edges, to a and b, are **false positives**.

Sound, and Imprecise (3/3)

```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



At the end, p and q hold **only** &c. The four **red** edges, to a and b, are **false positives**.

Flow insensitivity: p held a, then b, then c, one at a time, but no value was ever killed. (And q never held a at all.)

Flow Sensitivity (1/2)

One graph **per program point**. A single-target assignment **kills** the old value (**red**: a strong update).

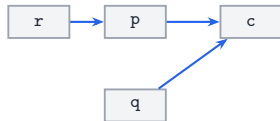
after	p	q	r
p = &a	a		
q = &b	a	b	
p = q	b	b	
r = &p	b	b	p
*r = &c	c	b	p
q = *r	c	c	p

Flow Sensitivity (2/2)

One graph **per program point**. A single-target assignment **kills** the old value (**red**: a strong update).

after	p	q	r
<code>p = &a</code>	a		
<code>q = &b</code>	a	b	
<code>p = q</code>	b	b	
<code>r = &p</code>	b	b	p
<code>*r = &c</code>	c	b	p
<code>q = *r</code>	c	c	p

At the end of the function:



$pts(p) = \{c\}, \quad pts(q) = \{c\}.$

All **four** false positives gone.

Quiz: Decompose These C Statements (1/4)

Rewrite using only `v1 = &v2`, `v1 = v2`, `v1 = *v2`, `*v1 = v2`.

```
*p = &a;
```

```
*p = *r;
```

Quiz: Decompose These C Statements (2/4)

Rewrite using only `v1 = &v2`, `v1 = v2`, `v1 = *v2`, `*v1 = v2`.

```
*p = &a;      tmp = &a  
               *p = tmp  
  
*p = *r;
```

Quiz: Decompose These C Statements (3/4)

Rewrite using only `v1 = &v2`, `v1 = v2`, `v1 = *v2`, `*v1 = v2`.

```
*p = &a;
```

```
tmp = &a
```

```
*p = tmp
```

```
*p = *r;
```

```
tmp = *r
```

```
*p = tmp
```

Quiz: Decompose These C Statements (4/4)

Rewrite using only $v1 = \&v2$, $v1 = v2$, $v1 = *v2$, $*v1 = v2$.

<code>*p = &a;</code>	<code>tmp = &a</code>
	<code>*p = tmp</code>
<code>*p = *r;</code>	<code>tmp = *r</code>
	<code>*p = tmp</code>

Both follow one rule: a statement may dereference on **one side only**. A temporary carries the value across.

Andersen: Example (1/5)

1. Read each statement as a constraint:

$p = \&a; \quad a \in pts(p)$

$q = \&b; \quad b \in pts(q)$

$r = p; \quad pts(p) \subseteq pts(r)$

$p = q; \quad pts(q) \subseteq pts(p)$

2. Solve in **passes**: sweep all four constraints in order;
repeat until a pass changes nothing (**red** grew).

after pass 1

after pass 2

$pts(p)$

$pts(q)$

$pts(r)$

Andersen: Example (2/5)

1. Read each statement as a constraint:

$p = \&a;$ $a \in pts(p)$

$q = \&b;$ $b \in pts(q)$

$r = p;$ $pts(p) \subseteq pts(r)$

$p = q;$ $pts(q) \subseteq pts(p)$

2. Solve in **passes**: sweep all four constraints in order;
repeat until a pass changes nothing (**red** grew).

after pass 1

after pass 2

$pts(p)$

{a, b}

$pts(q)$

{b}

$pts(r)$

{a}

Andersen: Example (3/5)

1. Read each statement as a constraint:

$p = \&a;$ $a \in pts(p)$

$q = \&b;$ $b \in pts(q)$

$r = p;$ $pts(p) \subseteq pts(r)$

$p = q;$ $pts(q) \subseteq pts(p)$

2. Solve in **passes**: sweep all four constraints in order;
repeat until a pass changes nothing (**red** grew).

	after pass 1	after pass 2
$pts(p)$	{a,b}	{a,b}
$pts(q)$	{b}	{b}
$pts(r)$	{a}	{a,b}

Andersen: Example (4/5)

1. Read each statement as a constraint:

$p = \&a;$ $a \in pts(p)$

$q = \&b;$ $b \in pts(q)$

$r = p;$ $pts(p) \subseteq pts(r)$

$p = q;$ $pts(q) \subseteq pts(p)$

2. Solve in **passes**: sweep all four constraints in order; repeat until a pass changes nothing (**red** grew).

	after pass 1	after pass 2
$pts(p)$	{a, b}	{a, b}
$pts(q)$	{b}	{b}
$pts(r)$	{a}	{a, b}

Pass 3 changes nothing $\Rightarrow$ **fixpoint**. r needs pass 2 because $r = p$ swept before p grew.

Andersen: Example (5/5)

1. Read each statement as a constraint:

$p = \&a;$ $a \in pts(p)$

$q = \&b;$ $b \in pts(q)$

$r = p;$ $pts(p) \subseteq pts(r)$

$p = q;$ $pts(q) \subseteq pts(p)$

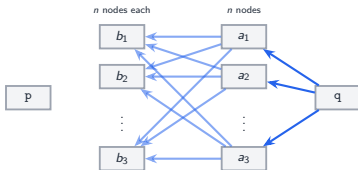
2. Solve in **passes**: sweep all four constraints in order;
repeat until a pass changes nothing (**red** grew).

	after pass 1	after pass 2
$pts(p)$	{a,b}	{a,b}
$pts(q)$	{b}	{b}
$pts(r)$	{a}	{a,b}

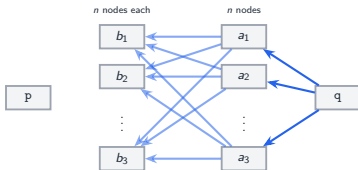
Pass 3 changes nothing $\Rightarrow$ **fixpoint**. r needs pass 2
because $r = p$ swept before p grew.

Final: $pts(p) = pts(r) = \{a, b\}$, $pts(q) = \{b\}$. How many passes
in the worst case, and how costly?

Andersen's Algorithm Is Cubic (1/4)

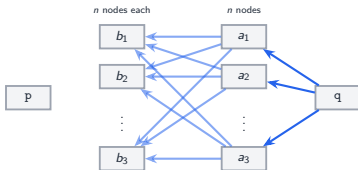


Andersen's Algorithm Is Cubic (2/4)



q reaches n nodes; each reaches n more. One $p = *q$ visits n^2 nodes.

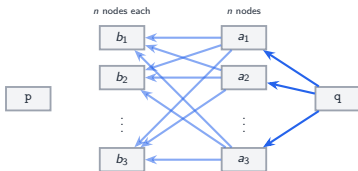
Andersen's Algorithm Is Cubic (3/4)



q reaches n nodes; each reaches n more. One $p = *q$ visits n^2 nodes.

Up to n such statements in a program of size n , so $O(n^3)$.

Andersen's Algorithm Is Cubic (4/4)



q reaches n nodes; each reaches n more. One $p \rightarrow q$ visits n^2 nodes.

Up to n such statements in a program of size n , so $O(n^3)$.

In practice nearer **quadratic** — still too slow for a codebase of millions of lines.

Steensgaard's Algorithm (1/6)

Every pointer has exactly one target node.

Steensgaard's Algorithm (2/6)

Every pointer has exactly one target node.

- If p may point to both a and b , then a and b are **merged** into one node.

Steensgaard's Algorithm (3/6)

Every pointer has exactly one target node.

- If p may point to both a and b , then a and b are **merged** into one node.
- Merging two nodes **unifies** them (union-find), and merges what they point to.

Steensgaard's Algorithm (4/6)

Every pointer has exactly one target node.

- If p may point to both a and b , then a and b are **merged** into one node.
- Merging two nodes **unifies** them (union-find), and merges what they point to.
- With one target per node, $*q$ follows a **single edge**. Union-find: $O(n\alpha(n))$, nearly linear.

α is the **inverse Ackermann function**: ≤ 4 for any realistic n , so $O(n\alpha(n))$ is linear in practice.

Steensgaard's Algorithm (5/6)

Every pointer has exactly one target node.

- If p may point to both a and b , then a and b are **merged** into one node.
- Merging two nodes **unifies** them (union-find), and merges what they point to.
- With one target per node, $*q$ follows a **single edge**. Union-find: $O(n\alpha(n))$, nearly linear.
- In LLVM's alias analysis; scales to millions of lines.

α is the **inverse Ackermann function**: ≤ 4 for any realistic n , so $O(n\alpha(n))$ is linear in practice.

Steensgaard's Algorithm (6/6)

Every pointer has exactly one target node.

- If p may point to both a and b , then a and b are **merged** into one node.
- Merging two nodes **unifies** them (union-find), and merges what they point to.
- With one target per node, $*q$ follows a **single edge**. Union-find: $O(n\alpha(n))$, nearly linear.
- In LLVM's alias analysis; scales to millions of lines.

α is the **inverse Ackermann function**: ≤ 4 for any realistic n , so $O(n\alpha(n))$ is linear in practice.

Inclusion versus Unification (1/4)

$u = v$:

Andersen: inclusion

$$pts(v) \subseteq pts(u)$$

one-way: u learns from v

Inclusion versus Unification (2/4)

$u = v$:

Andersen: inclusion

$$pts(v) \subseteq pts(u)$$

one-way: u learns from v

Steensgaard: unification

$$pts(u) = pts(v)$$

two-way: u and v **merged**

Inclusion versus Unification (3/4)

$u = v$:

Andersen: inclusion

$$pts(v) \subseteq pts(u)$$

one-way: u learns from v

Steensgaard: unification

$$pts(u) = pts(v)$$

two-way: u and v **merged**

Consider $p \rightarrow \{a\}$, $q \rightarrow \{b\}$, then $p = q$:

Andersen: $p \rightarrow \{a, b\}$, $q \rightarrow \{b\}$. Steensgaard: $p, q \rightarrow \{a, b\}$.

Inclusion versus Unification (4/4)

$u = v$:

Andersen: inclusion

$$pts(v) \subseteq pts(u)$$

one-way: u learns from v

Steensgaard: unification

$$pts(u) = pts(v)$$

two-way: u and v **merged**

Consider $p \rightarrow \{a\}$, $q \rightarrow \{b\}$, then $p = q$:

Andersen: $p \rightarrow \{a, b\}$, $q \rightarrow \{b\}$. Steensgaard: $p, q \rightarrow \{a, b\}$.

Steensgaard versus Andersen (1/3)

Andersen

```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



Steensgaard



Merged: Steensgaard now says q may point to a.

Steensgaard versus Andersen (2/3)

Andersen

```
p = &a;
```

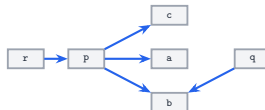
```
q = &b;
```

```
p = q;
```

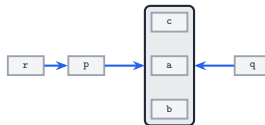
```
r = &p;
```

```
*r = &c;
```

```
q = *r;
```



Steensgaard

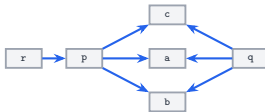


Merged again: c joins them, so q may point to c too.

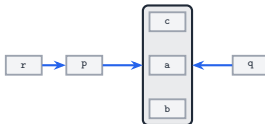
Steensgaard versus Andersen (3/3)

Andersen

```
p = &a;  
q = &b;  
p = q;  
r = &p;  
*r = &c;  
q = *r;
```



Steensgaard



Application: Control-Flow Integrity (1/5)

Most exploits **hijack control flow**: overwrite a function pointer or return address, and the attacker picks what runs next.

Application: Control-Flow Integrity (2/5)

Most exploits **hijack control flow**: overwrite a function pointer or return address, and the attacker picks what runs next.

- Define the **expected** behaviour: the program's control-flow graph.

Application: Control-Flow Integrity (3/5)

Most exploits **hijack control flow**: overwrite a function pointer or return address, and the attacker picks what runs next.

- Define the **expected** behaviour: the program's control-flow graph.
- **Instrument** every indirect transfer to check its target against that graph.

Application: Control-Flow Integrity (4/5)

Most exploits **hijack control flow**: overwrite a function pointer or return address, and the attacker picks what runs next.

- Define the **expected** behaviour: the program's control-flow graph.
- **Instrument** every indirect transfer to check its target against that graph.
- A mismatch aborts the process. The attacker may own all of memory and still cannot divert control.

Application: Control-Flow Integrity (5/5)

Most exploits **hijack control flow**: overwrite a function pointer or return address, and the attacker picks what runs next.

- Define the **expected** behaviour: the program's control-flow graph.
- **Instrument** every indirect transfer to check its target against that graph.
- A mismatch aborts the process. The attacker may own all of memory and still cannot divert control.

Microsoft **Control Flow Guard**, Clang's `-fsanitize=cfi`, ARM pointer authentication.

CFI Needs Pointer Analysis (1/5)

```
typedef int (*cmp_t)(int, int);
int lt(int x, int y);
int gt(int x, int y);
int eq(int x, int y);

void dispatch(int u, int v) {
    cmp_t a = &lt, b = &gt, c = &eq;
    cmp_t *p = &a, *q = &b;
    p = q;
    cmp_t d = *q;
    d(u, v);    /* which one? */
}
```

CFI Needs Pointer Analysis (2/5)

```
typedef int (*cmp_t)(int, int);
int lt(int x, int y);
int gt(int x, int y);
int eq(int x, int y);

void dispatch(int u, int v) {
    cmp_t a = &lt, b = &gt, c = &eq;
    cmp_t *p = &a, *q = &b;
    p = q;
    cmp_t d = *q;
    d(u, v);    /* which one? */
}
```

The CFG needs what `d` can hold: an **indirect read**.

CFI Needs Pointer Analysis (3/5)

```
typedef int (*cmp_t)(int, int);
int lt(int x, int y);
int gt(int x, int y);
int eq(int x, int y);

void dispatch(int u, int v) {
    cmp_t a = &lt, b = &gt, c = &eq;
    cmp_t *p = &a, *q = &b;
    p = q;
    cmp_t d = *q;
    d(u, v);    /* which one? */
}
```

The CFG needs what `d` can hold: an **indirect read**.

Andersen: $q \rightarrow b \rightarrow gt$, so `d` is **gt**: one edge.

CFI Needs Pointer Analysis (4/5)

```
typedef int (*cmp_t)(int, int);
int lt(int x, int y);
int gt(int x, int y);
int eq(int x, int y);

void dispatch(int u, int v) {
    cmp_t a = &ltlt, b = &gtgt, c = &eq;
    cmp_t *p = &a, *q = &b;
    p = q;
    cmp_t d = *q;
    d(u, v);    /* which one? */
}
```

The CFG needs what d can hold: an **indirect read**.

Andersen: $q \rightarrow b \rightarrow gt$, so d is **gt**: one edge.

Steensgaard: $p = q$ merged a, b, so d is lt or gt.

CFI Needs Pointer Analysis (5/5)

```
typedef int (*cmp_t)(int, int);
int lt(int x, int y);
int gt(int x, int y);
int eq(int x, int y);

void dispatch(int u, int v) {
    cmp_t a = &ltlt, b = &gtgt, c = &eq;
    cmp_t *p = &a, *q = &b;
    p = q;
    cmp_t d = *q;
    d(u, v);    /* which one? */
}
```

The CFG needs what d can hold: an **indirect read**.

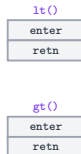
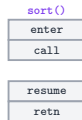
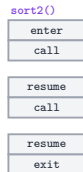
Andersen: $q \rightarrow b \rightarrow gt$, so d is **gt**: one edge.

Steensgaard: $p = q$ merged a, b, so d is lt or gt.

Still rules out eq and the rest of the binary.

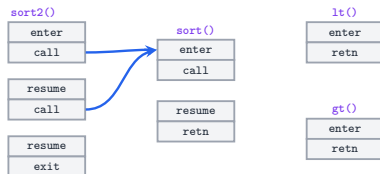
CFI Example: The Control-Flow Graph (1/6)

```
void sort(int a[], int n, cmp_t c) {  
    for (int i = 1; i < n; i++)  
        if (c(a[i], a[i-1])) { /*  
            reorder */ }  
}  
  
void sort2(int a[], int b[], int n)  
{  
    sort(a, n, lt);  
    sort(b, n, gt);  
}
```



CFI Example: The Control-Flow Graph (2/6)

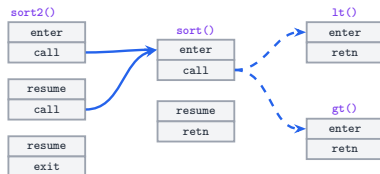
```
void sort(int a[], int n, cmp_t c) {  
    for (int i = 1; i < n; i++)  
        if (c(a[i], a[i-1])) { /*  
            reorder */ }  
}  
  
void sort2(int a[], int b[], int n)  
{  
    sort(a, n, lt);  
    sort(b, n, gt);  
}
```



— direct call

CFI Example: The Control-Flow Graph (3/6)

```
void sort(int a[], int n, cmp_t c) {  
    for (int i = 1; i < n; i++)  
        if (c(a[i], a[i-1])) { /*  
            reorder */ }  
}  
  
void sort2(int a[], int b[], int n)  
{  
    sort(a, n, lt);  
    sort(b, n, gt);  
}
```

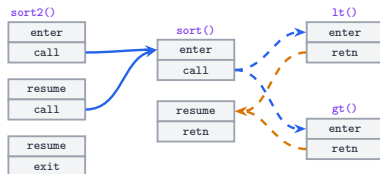


— direct call

- - - indirect call

CFI Example: The Control-Flow Graph (4/6)

```
void sort(int a[], int n, cmp_t c) {  
    for (int i = 1; i < n; i++)  
        if (c(a[i], a[i-1])) { /*  
            reorder */ }  
}  
  
void sort2(int a[], int b[], int n)  
{  
    sort(a, n, lt);  
    sort(b, n, gt);  
}
```



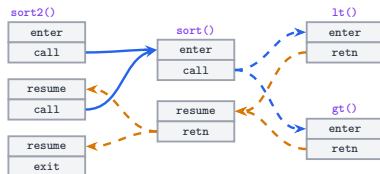
— direct call

- - - indirect call

- - - return

CFI Example: The Control-Flow Graph (5/6)

```
void sort(int a[], int n, cmp_t c) {  
    for (int i = 1; i < n; i++)  
        if (c(a[i], a[i-1])) { /*  
            reorder */ }  
}  
  
void sort2(int a[], int b[], int n)  
{  
    sort(a, n, lt);  
    sort(b, n, gt);  
}
```



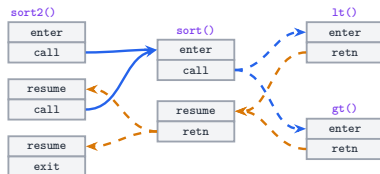
— direct call

- - - indirect call

- - - return

CFI Example: The Control-Flow Graph (6/6)

```
void sort(int a[], int n, cmp_t c) {  
    for (int i = 1; i < n; i++)  
        if (c(a[i], a[i-1])) { /*  
            reorder */ }  
}  
  
void sort2(int a[], int b[], int n)  
{  
    sort(a, n, lt);  
    sort(b, n, gt);  
}
```



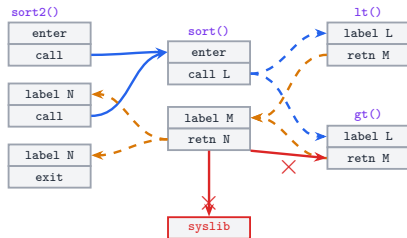
— direct call

- - - indirect call

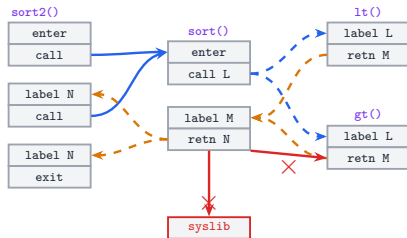
- - - return

Direct calls need no check: the target is in the instruction. **Every other edge is indirect.**

CFI Example: Instrumentation (1/3)

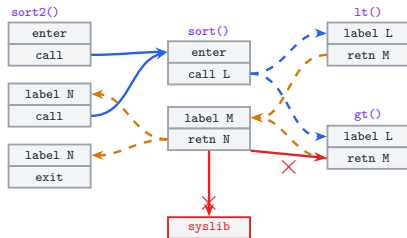


CFI Example: Instrumentation (2/3)



A **label** before every legal target; a **check** at every indirect transfer.
Mismatch aborts.

CFI Example: Instrumentation (3/3)



A **label** before every legal target; a **check** at every indirect transfer. Mismatch aborts.

So sort cannot return into gt, and no transfer can reach a **system library** gadget.