Efficient CFI Enforcement for C++ Dynamic Dispatch

Dimitar Bounov, Rami Kici, Sorin Lerner

UCSD

UCSDCSE
Computer Science and Engineering

Why Attack Dynamic Dispatch?

- Valuable targets (e.g. browsers)

11M LOC
C/C++

7M LOC
C/C++

?M LOC
C/C++

~30M LOC
C/C++

Why Attack Dynamic Dispatch?

- Valuable targets (e.g. browsers)
- Prevalence of Dynamic Dispatch
  - 91.8 % of Indirect Calls in Chrome [Tice ’14]

Why Attack Dynamic Dispatch?

- Valuable targets (e.g. browsers)
- Prevalence of Dynamic Dispatch
- Exploited in the wild

Prior Work and Contribution

- Prior defenses

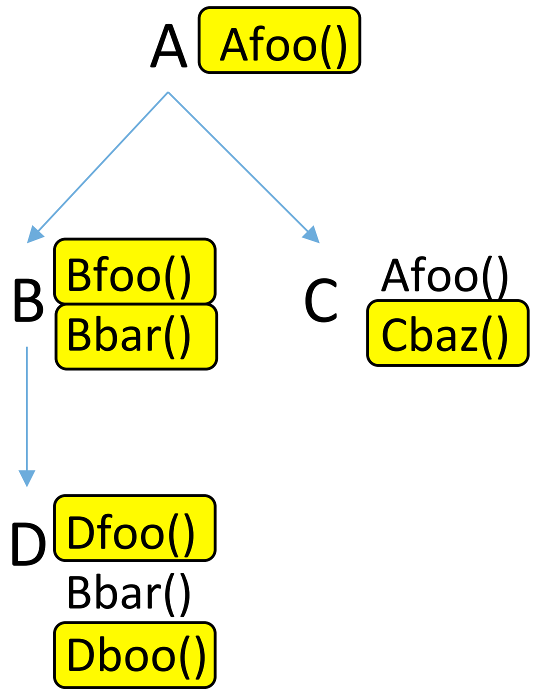
  vfGuard [Prakash’15], VTInt [Zhao’15], SafeDispatch [Jang’14], VTV [Tice’14] ...

- Our contribution: novel VTable layouts

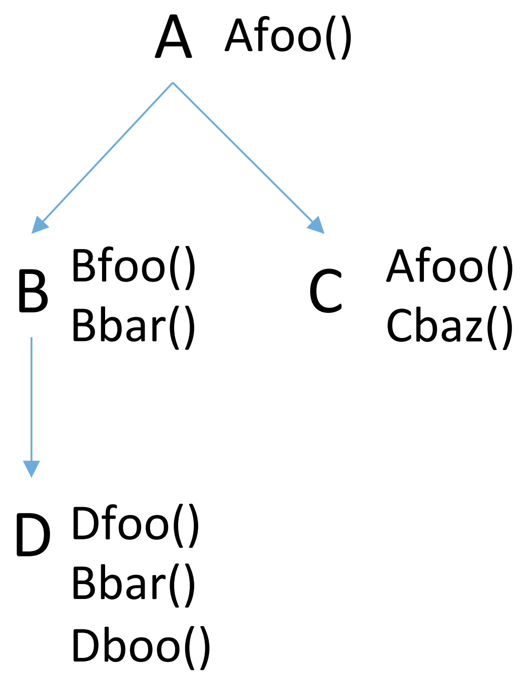
- Lower overhead & no profiling

Example

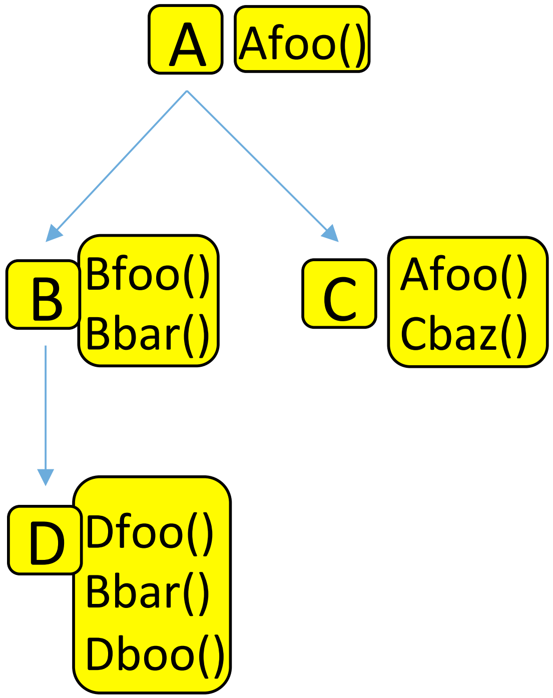
```
class A {
 virtual void foo();
}

class B : public A {
 virtual void foo();
 virtual void 
bar(); }

class C : public A {
 virtual void baz();
}

class D : public B {
 virtual void foo();
 virtual void boo();
}
```

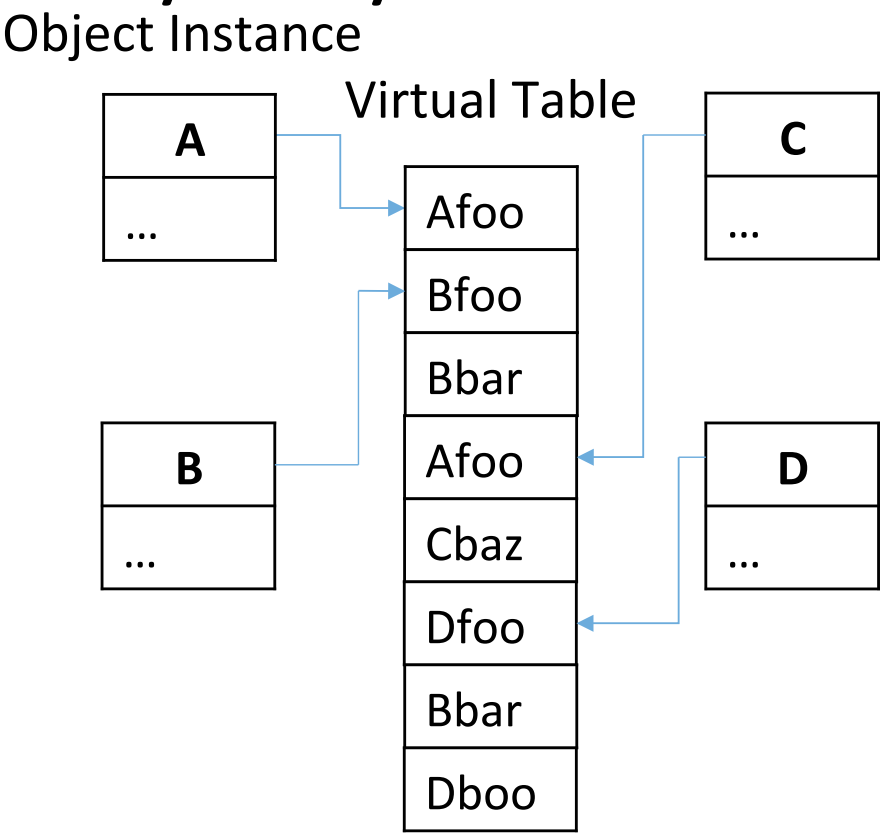


C++ Memory Layout

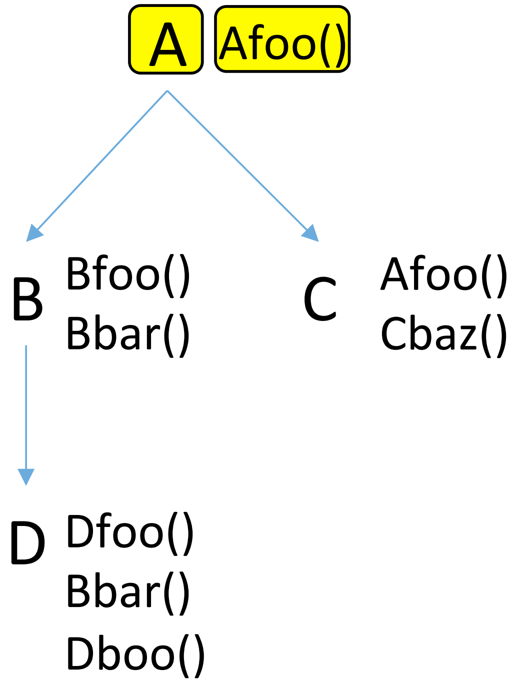


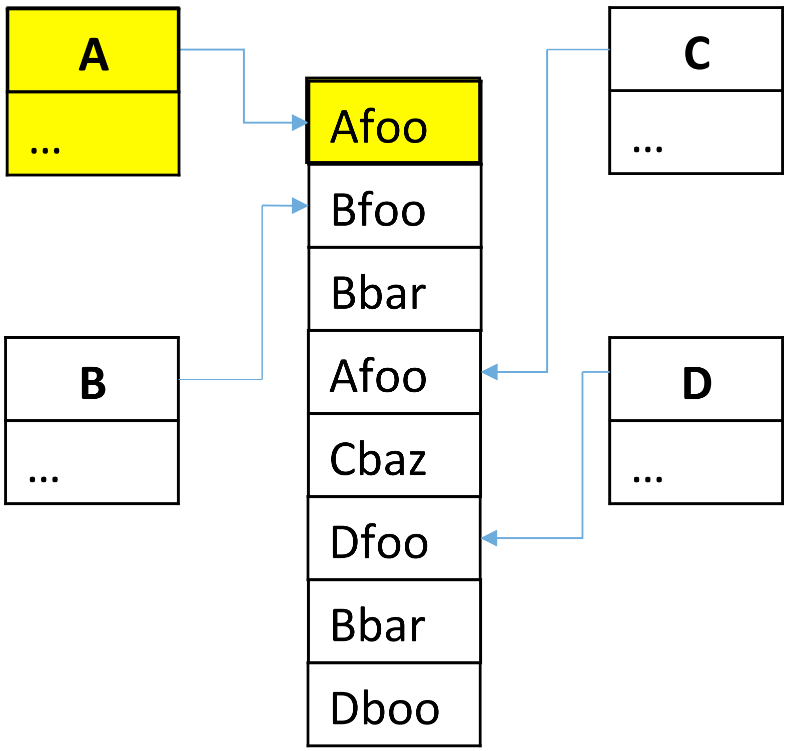
C++ Memory Layout


A
Afoo()
B
Bfoo()
Bbar()
C
Afoo()
Cbaz()
D
Dfoo()
Bbar()
Dboo()


Object Instance
Virtual Table
A
...
B
...
C
...
D
...
Afoo
Bfoo
Bbar
Afoo
Cbaz
Dfoo
Bbar
Dboo

Dynamic Dispatch



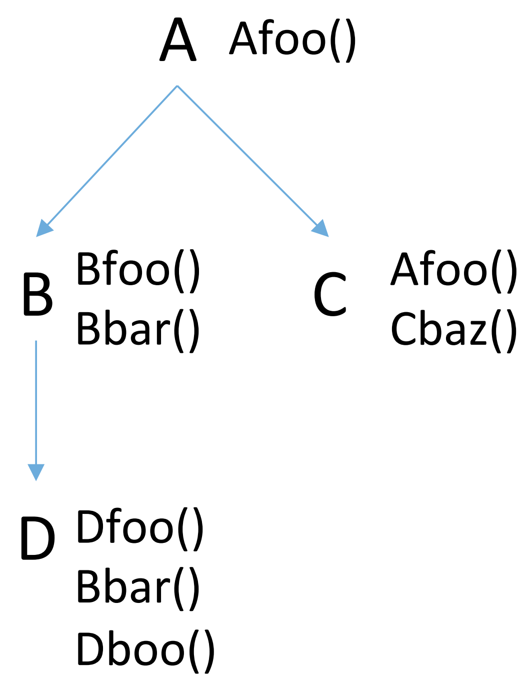


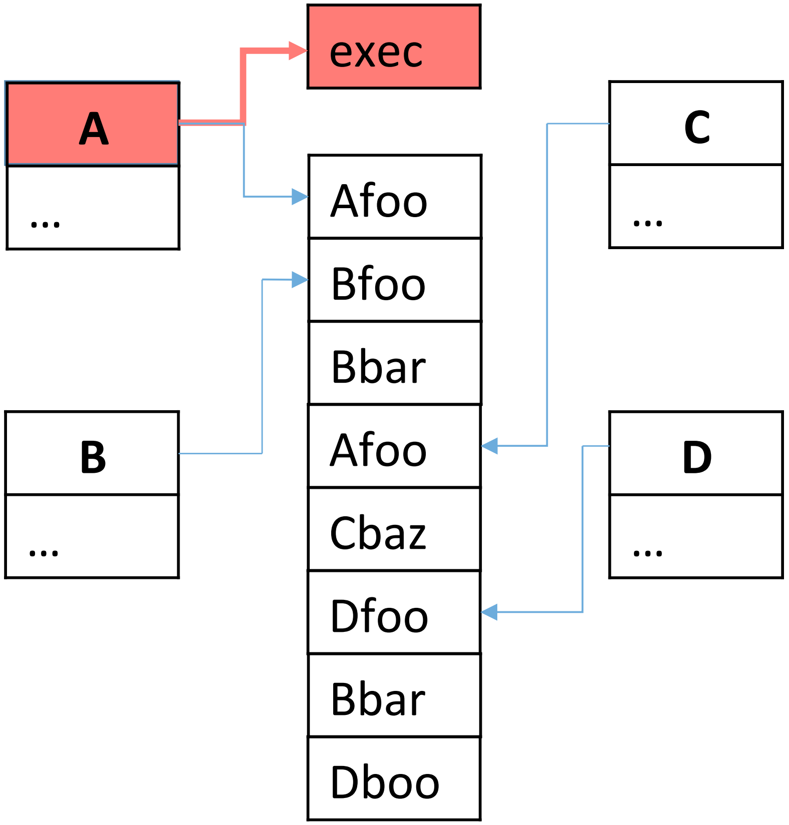
```
A* a = (A*) …
a->foo()
```

```
vptr = (*a)
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Method Index

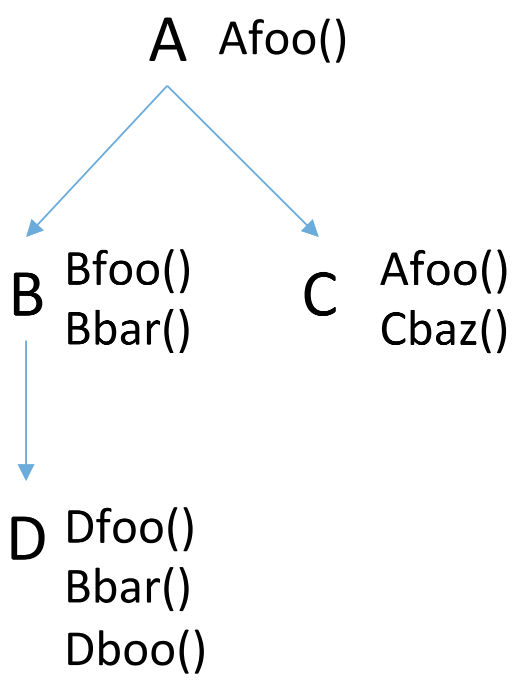
Exploiting Dynamic Dispatch

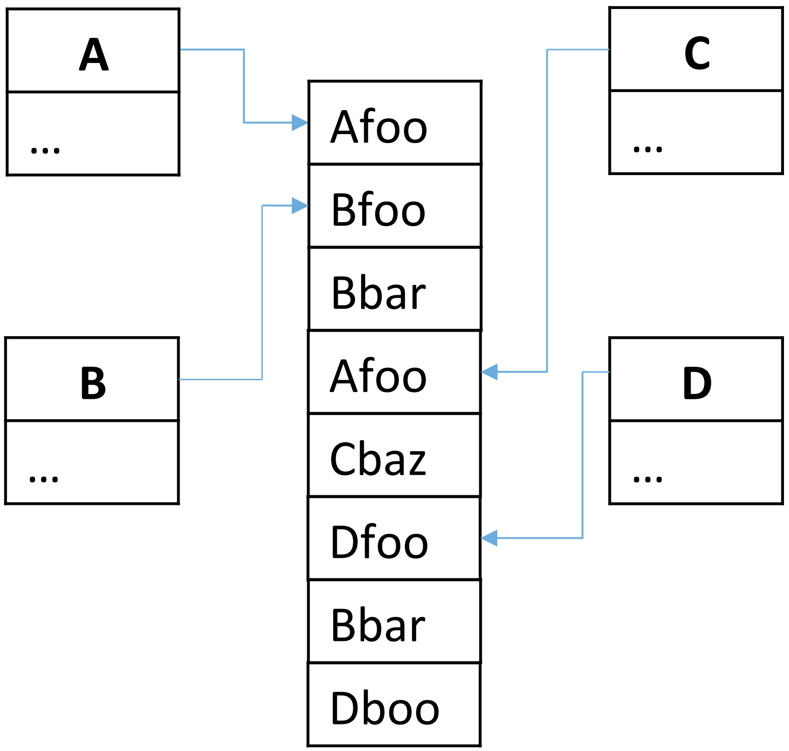




```
vptr = (*a)
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

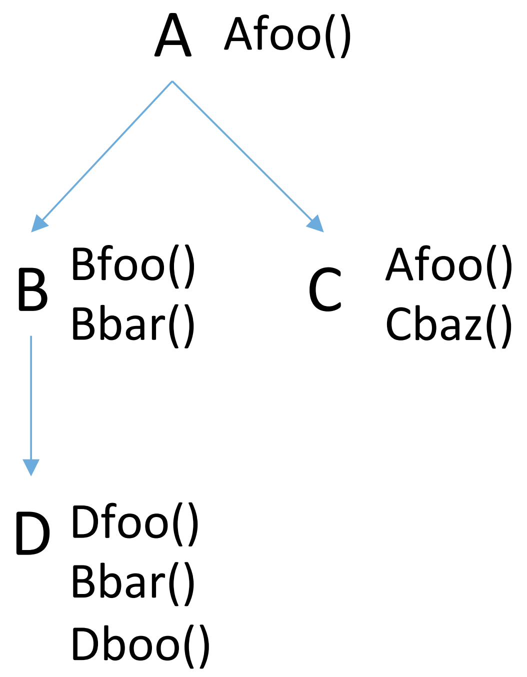
Protecting Dynamic Dispatch

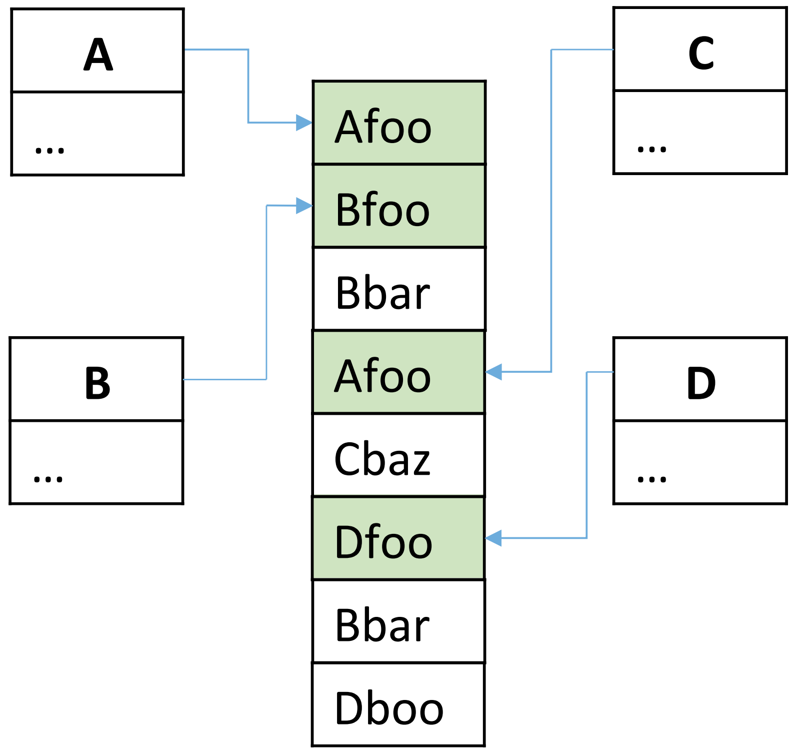




```
vptr = (*a)
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

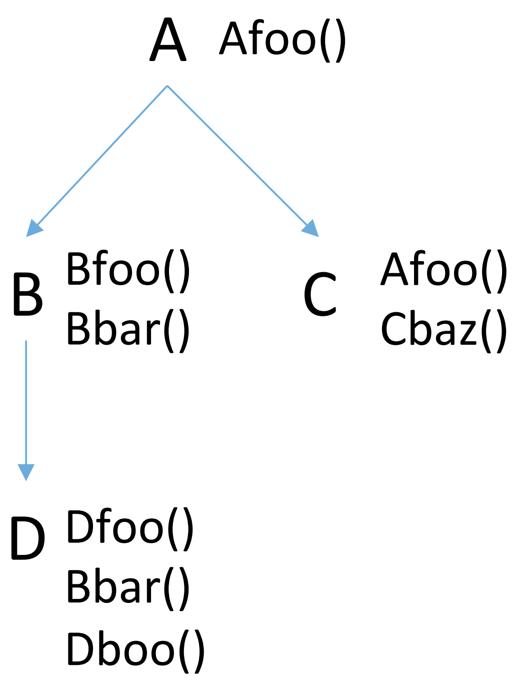
Protecting Dynamic Dispatch

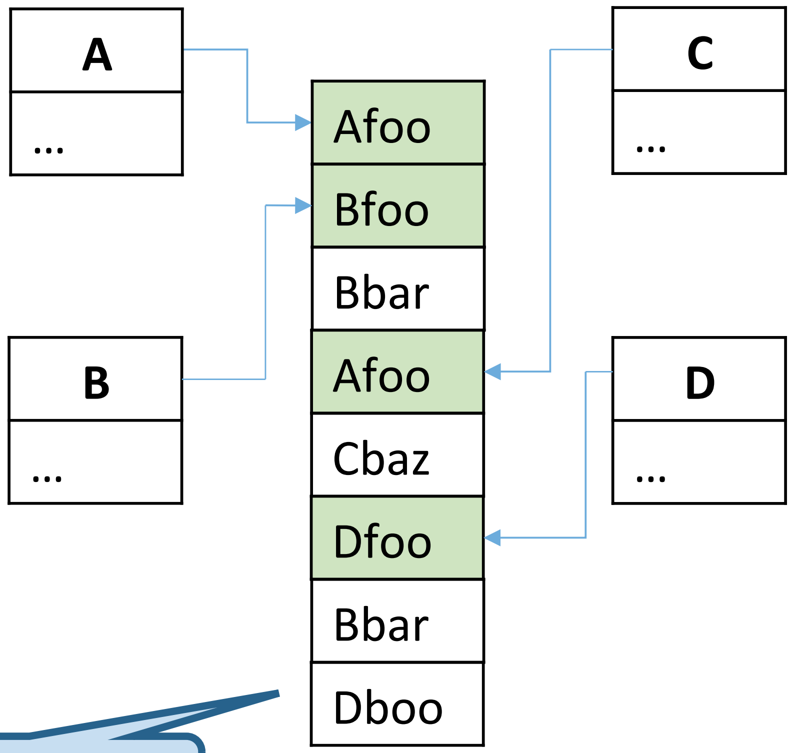




```
vptr = (*a)
assert (vptr ∊ { A, B, C, D })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Protecting Dynamic Dispatch



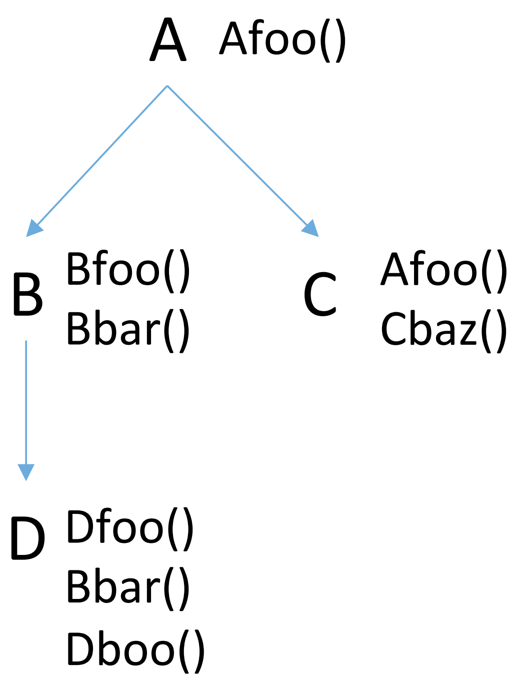


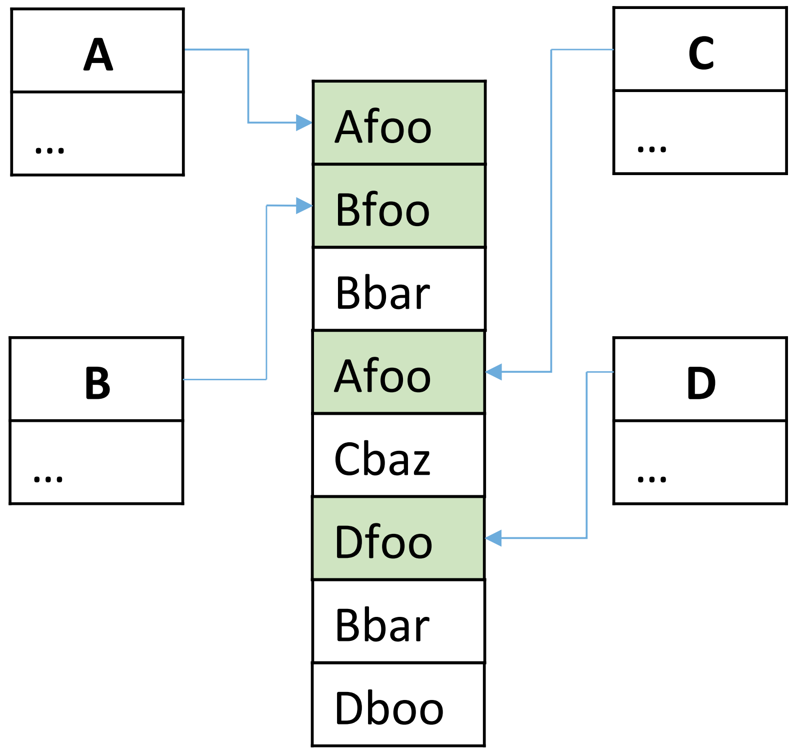
Read-Only

```
vptr = (*a)
assert (vptr ∊ { A, B, C, D })
fn_ptr = (*(vptr + 0))
(*fn_ptr)(); ✓
```

Inline Constant

Protecting Dynamic Dispatch

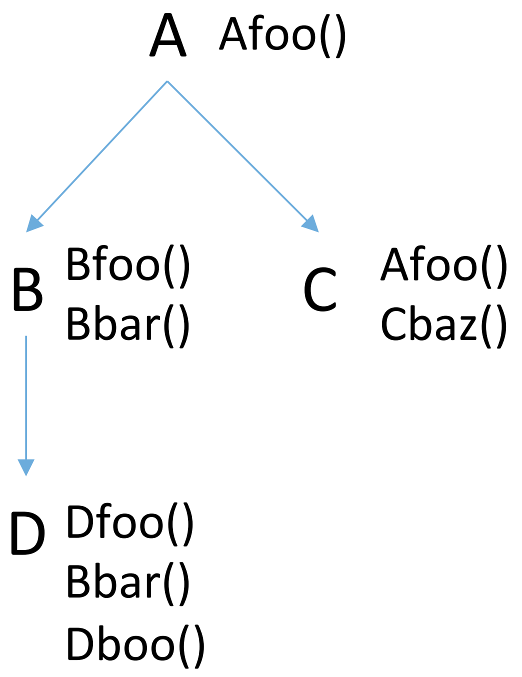


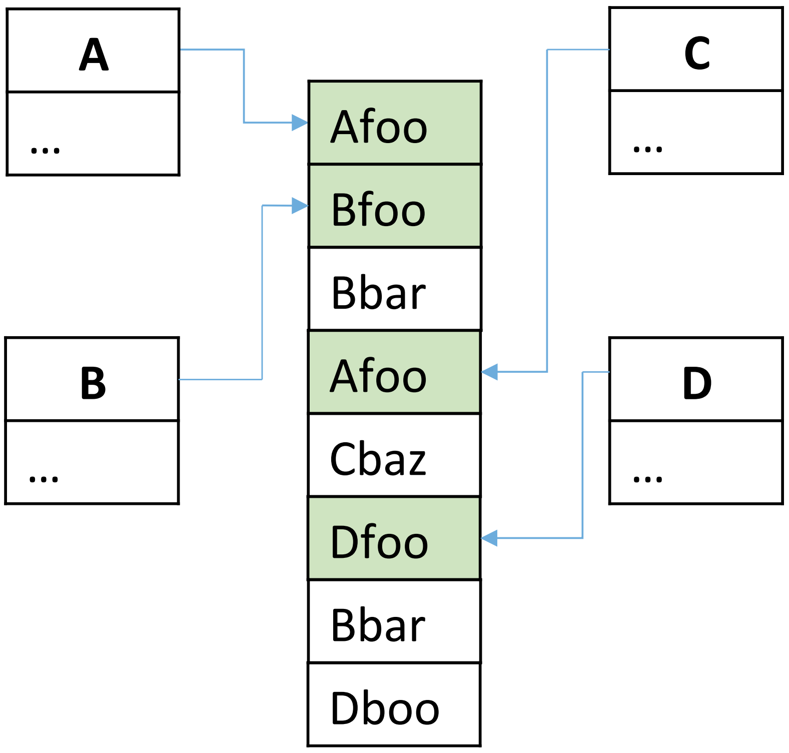


```
vptr = (*a)
assert (vptr ∊ { A, B, C, D })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

How to implement safety check efficiently?

Protecting Dynamic Dispatch

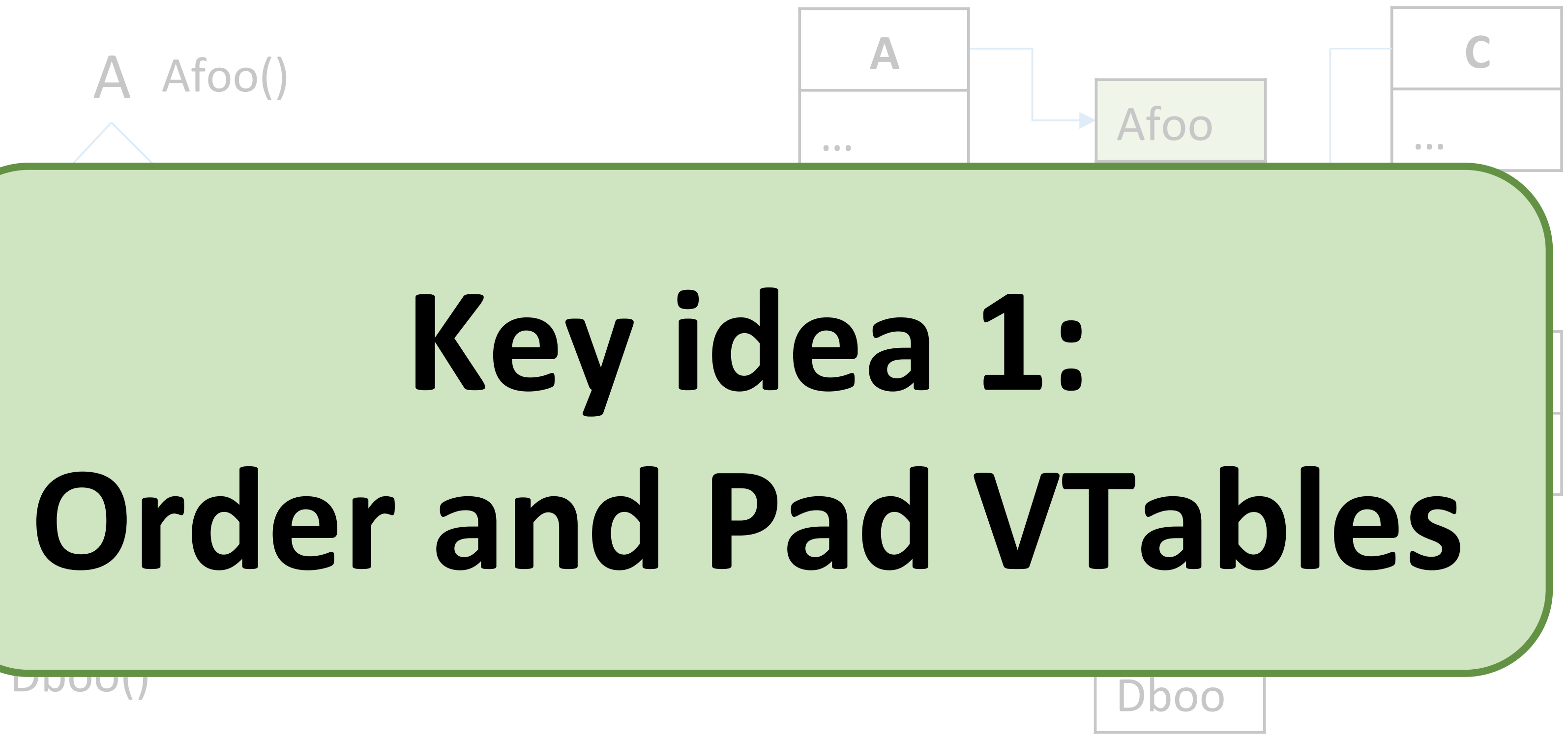




```
vptr = (*a)
assert (vptr ∊ { 0x0, 0x8, 0x18, 0x28 })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Non-regular values
Hard to test

Protecting Dynamic Dispatch

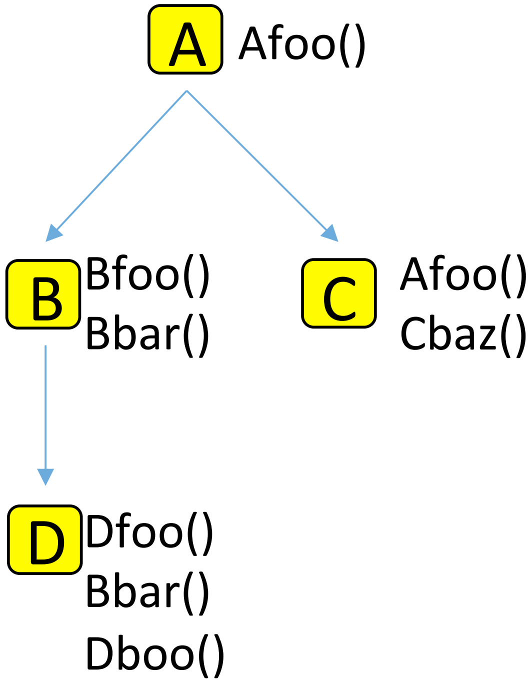


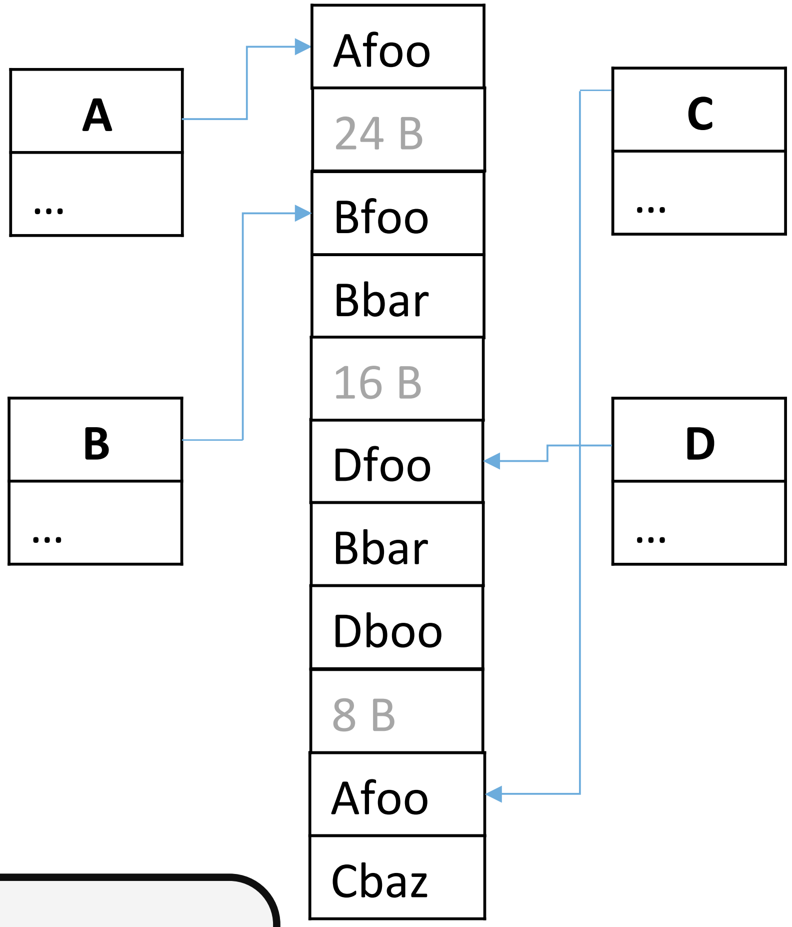
Key idea 1:
Order and Pad VTables

```
vptr = (*a)
assert (vptr ∊ { 0x0, 0x8, 0x18, 0x28 })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Non-regular values
Hard to test

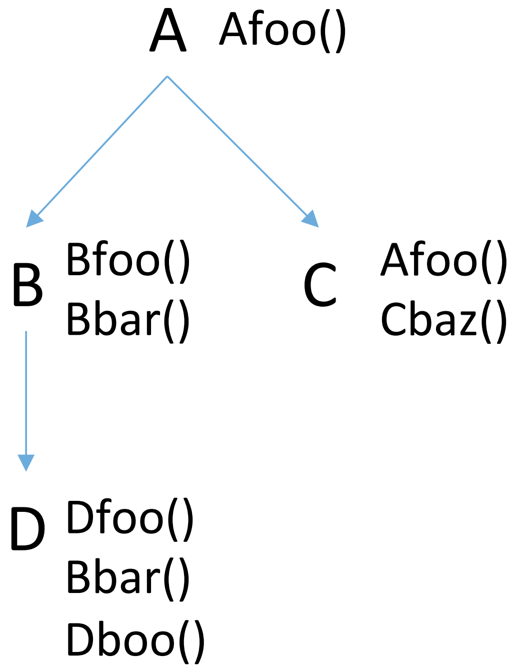
Ordered Memory Layout

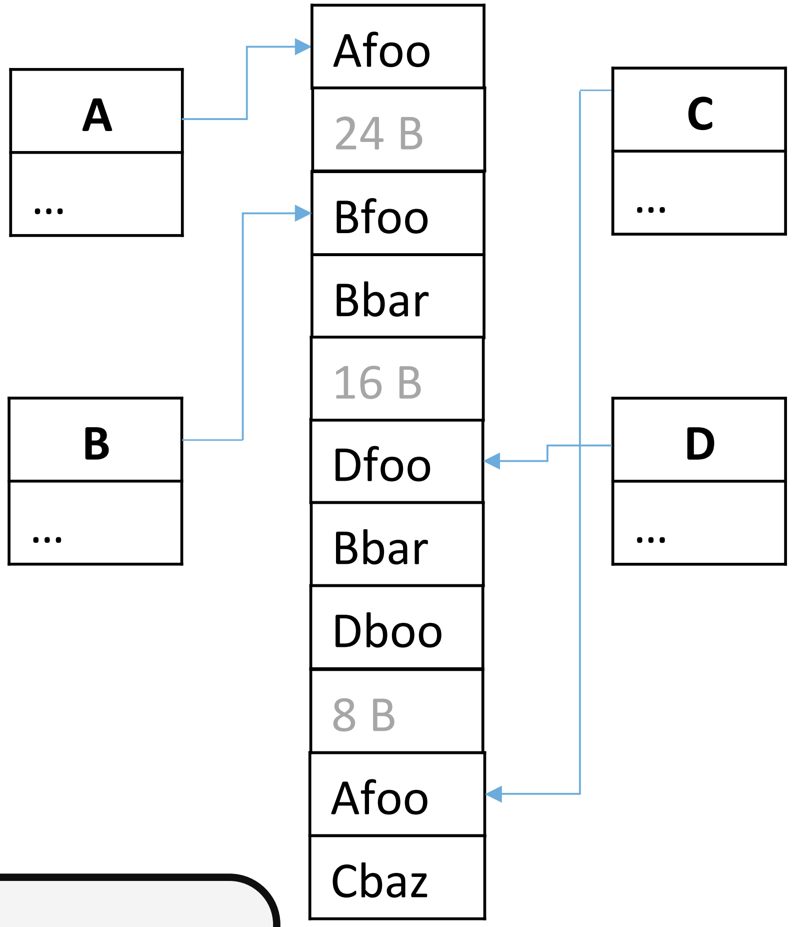




1. Traverse in pre-order: A, B, D, C
2. For each class layout vtable and pad

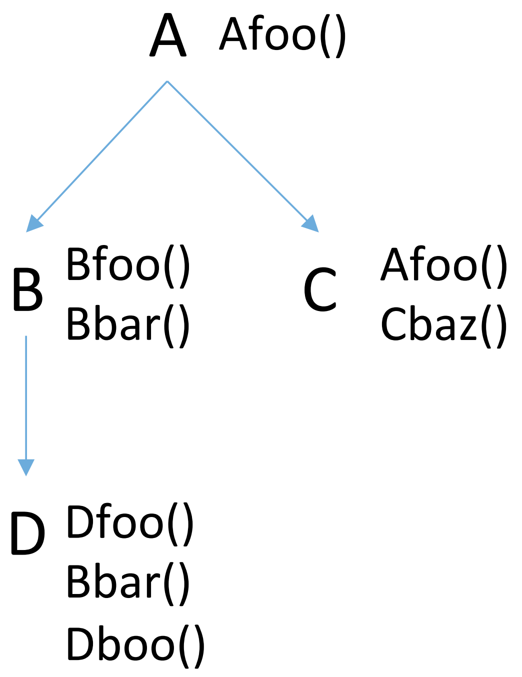
Ordered Memory Layout





1. Traverse in pre-order: A, B, D, C
2. For each class layout vtable and pad

Ordered Memory Layout

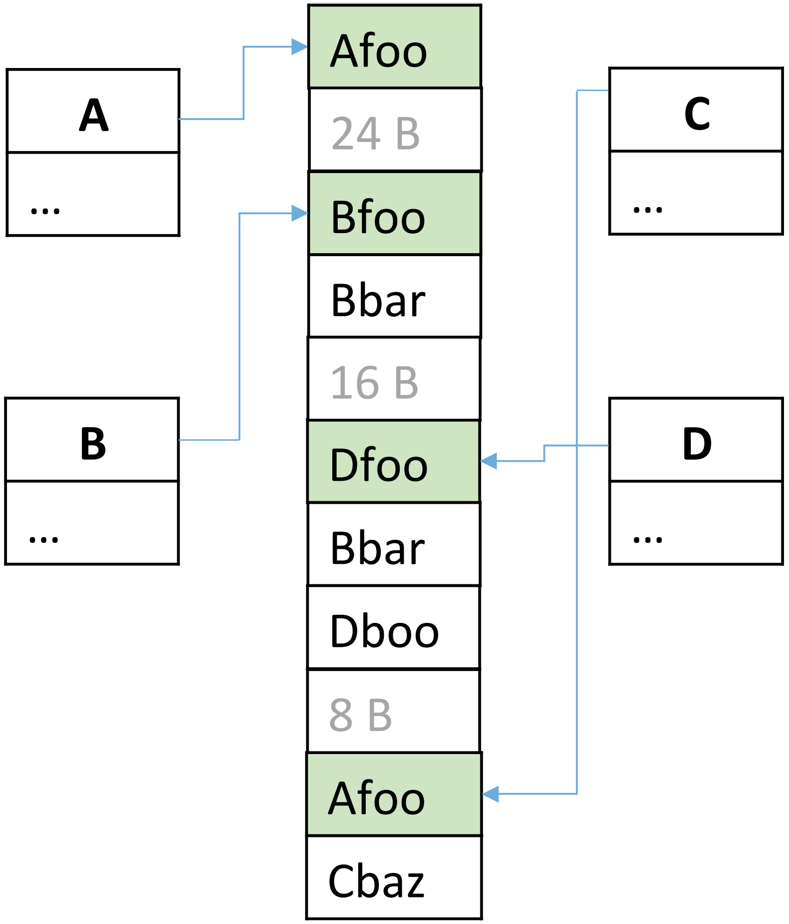


A Afoo()

B Bfoo()
Bbar()

C Afoo()
Cbaz()

D Dfoo()
Bbar()
Dboo()



A
...

B
...

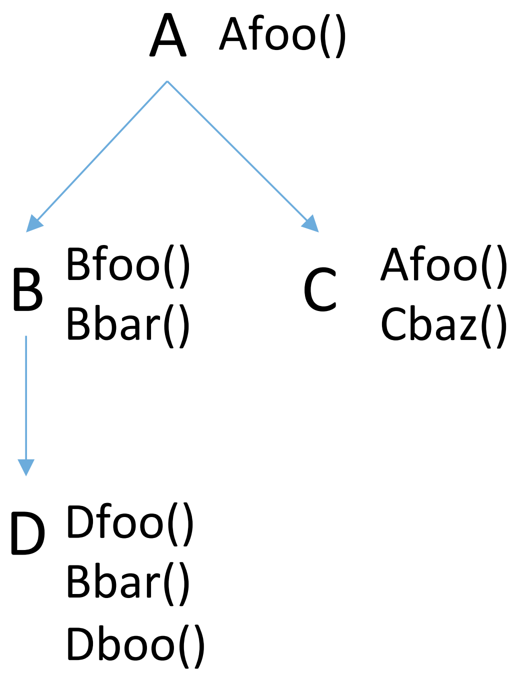
C
...

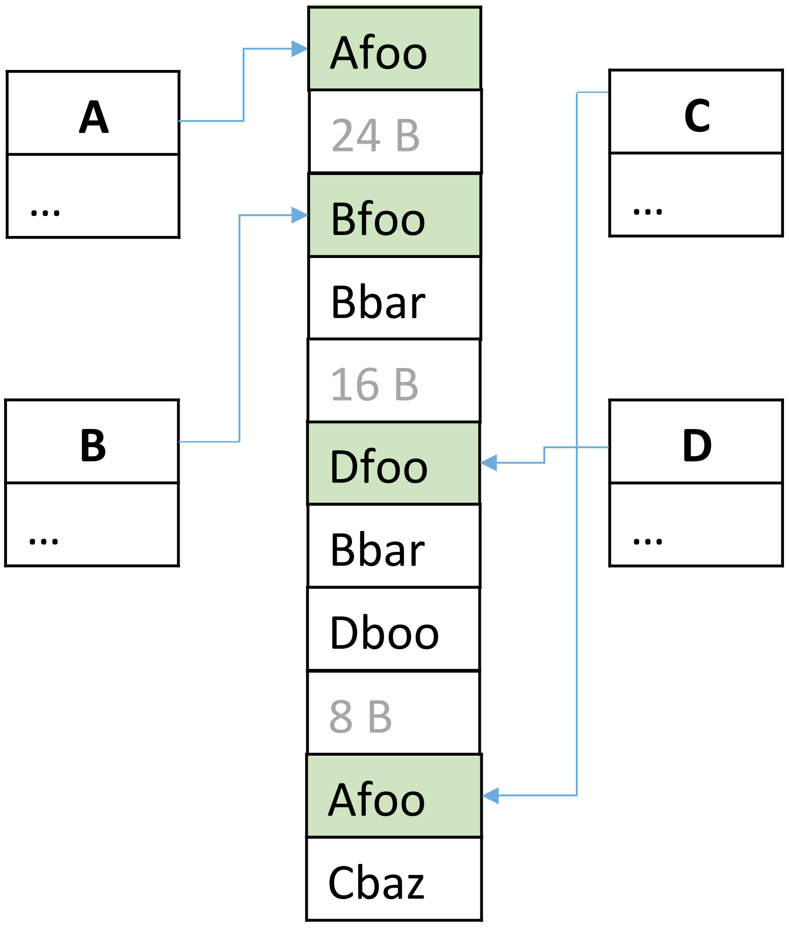
D
...

Afoo
24 B
Bfoo
Bbar
16 B
Dfoo
Bbar
Dboo
8 B
Afoo
Cbaz

```
vptr = (*a)
assert (vptr ∊ { A,B,C,D })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Ordered Memory Layout

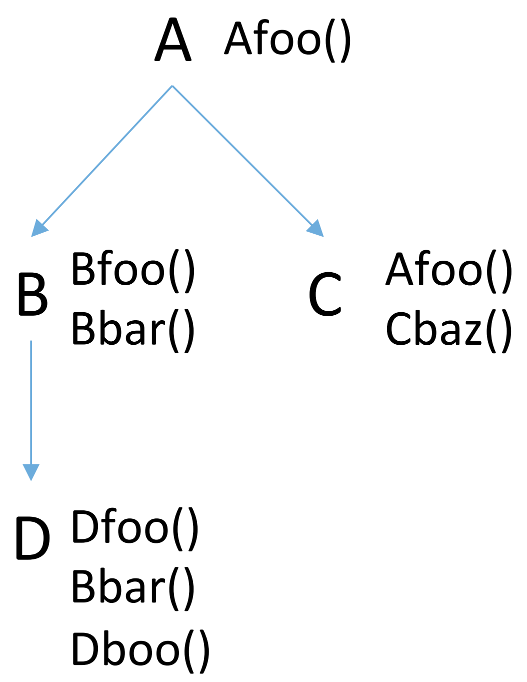


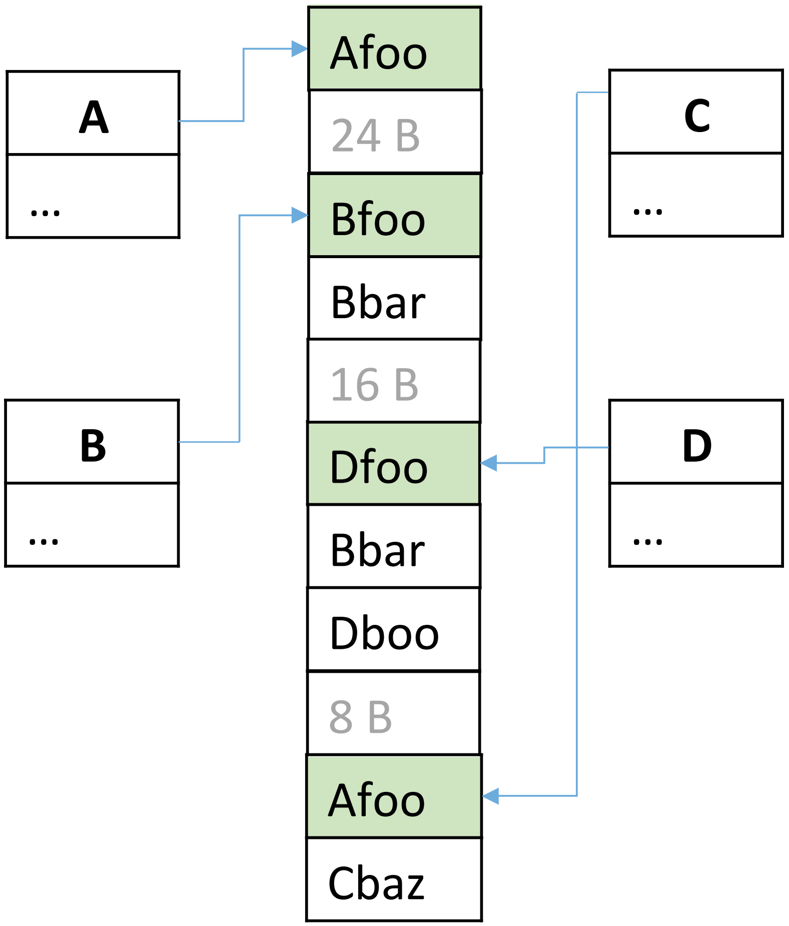


```
vptr = (*a)
assert (vptr ∊ { 0x0, 0x20, 0x40, 0x60 })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Regular
Address Points

Ordered Memory Layout

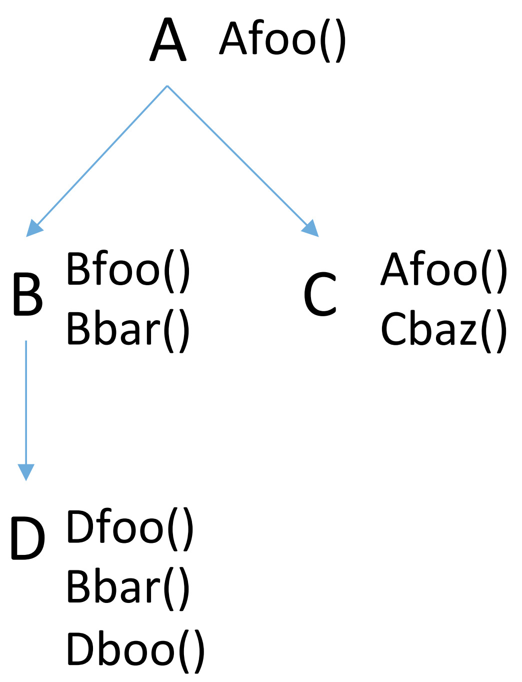


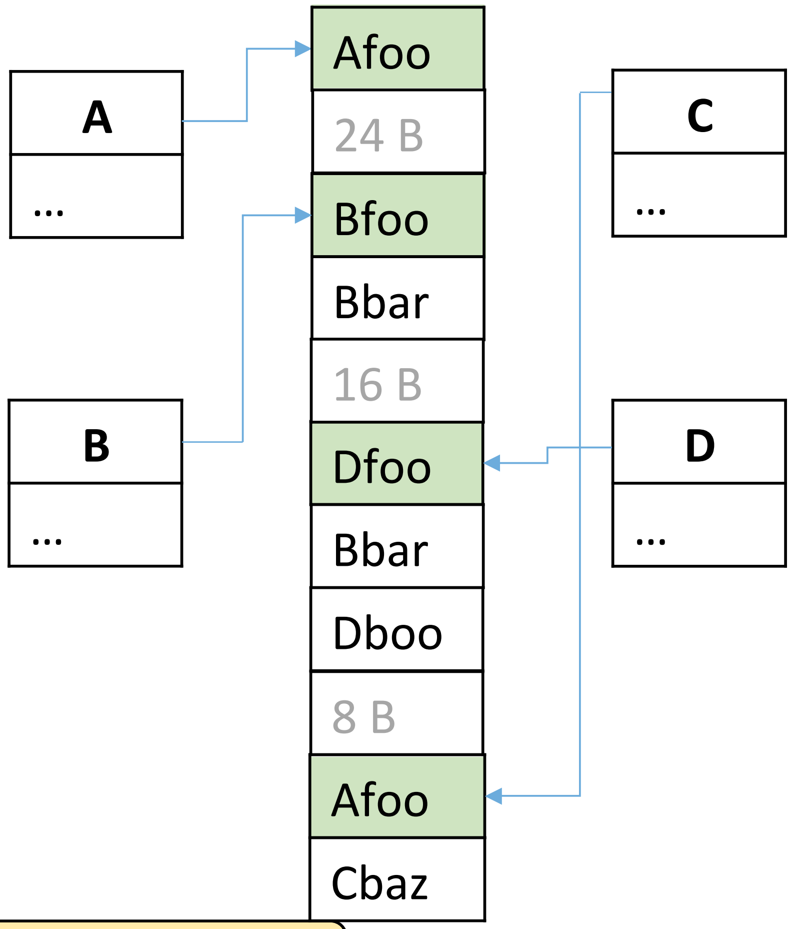


```
vptr = (*a)
assert (vptr ∊ { 0x0, 0x20, 0x40, 0x60 })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Efficient Check

Ordered Memory Layout

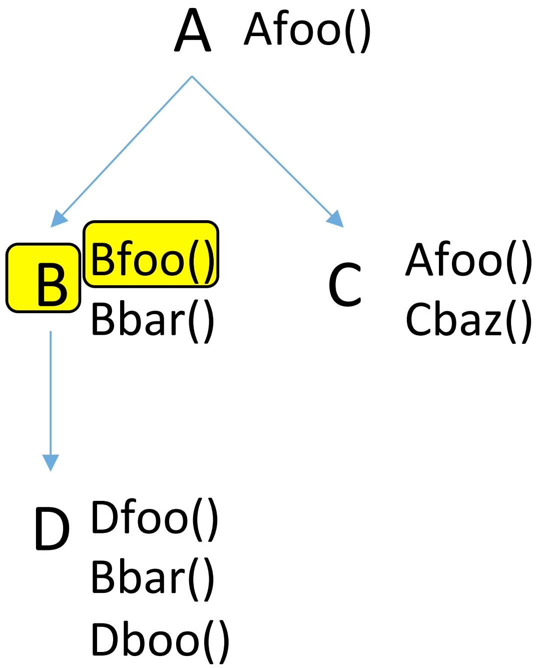


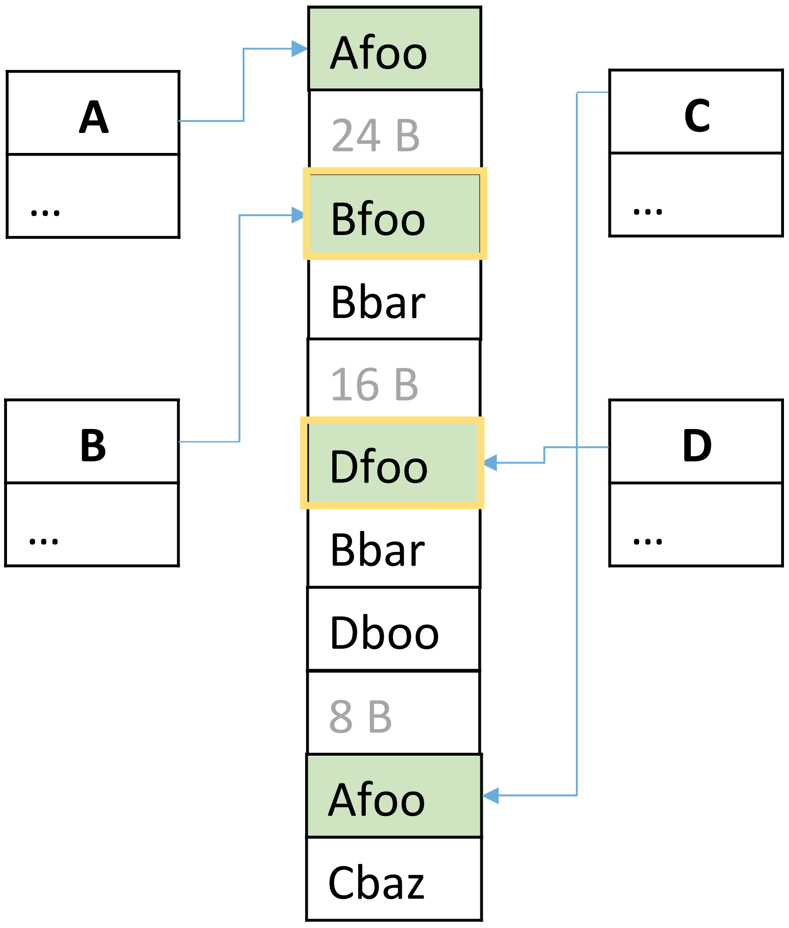


```
vptr = (*a)
assert (0x0 ≤ vptr ≤ 0x60 ∧ vptr % 0x20 = 0)
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Efficient Check

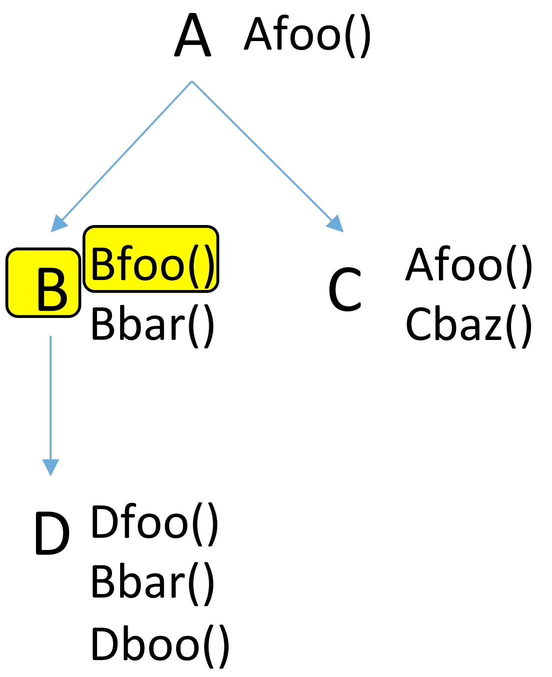
Ordered Memory Layout

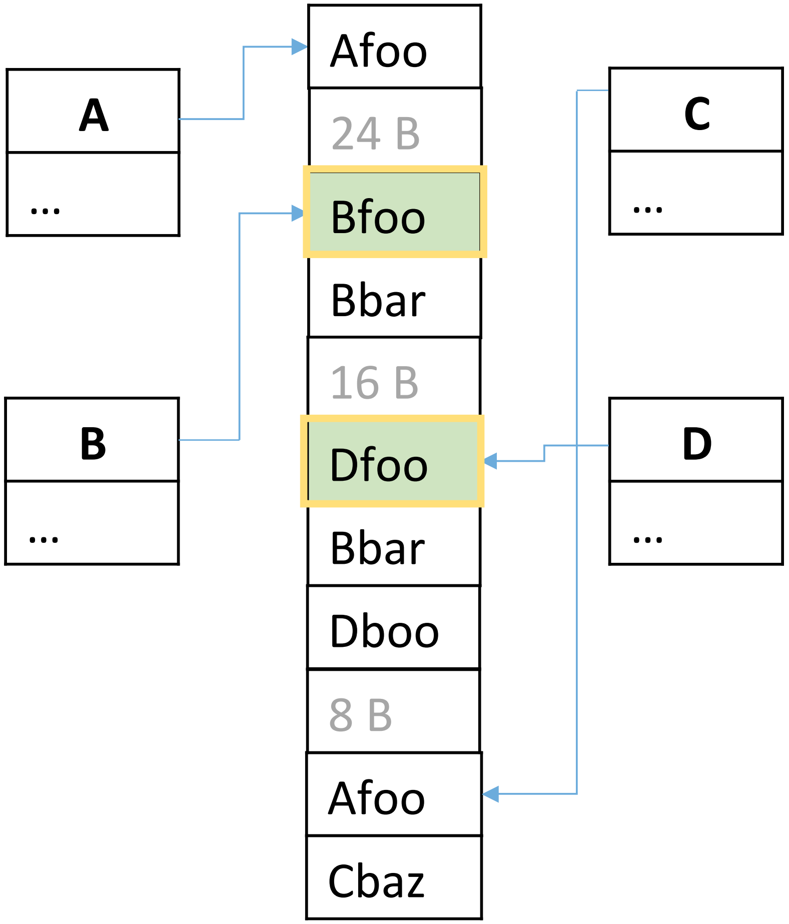




```
vptr = (*b)
assert (vptr ∊ { B, D })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

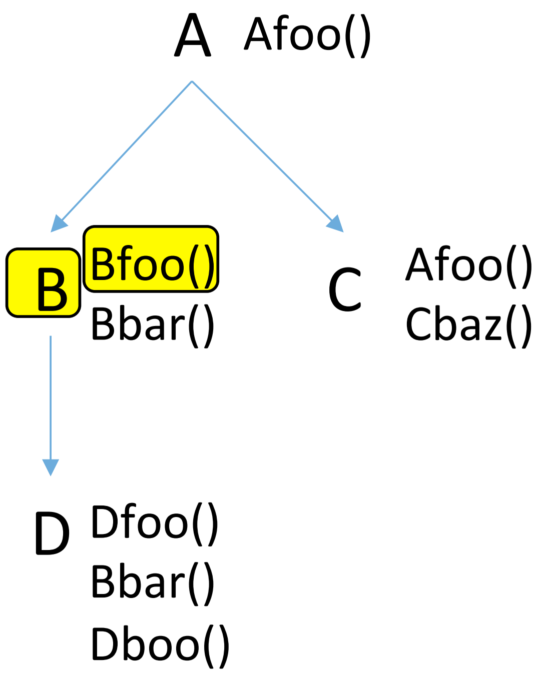
Ordered Memory Layout

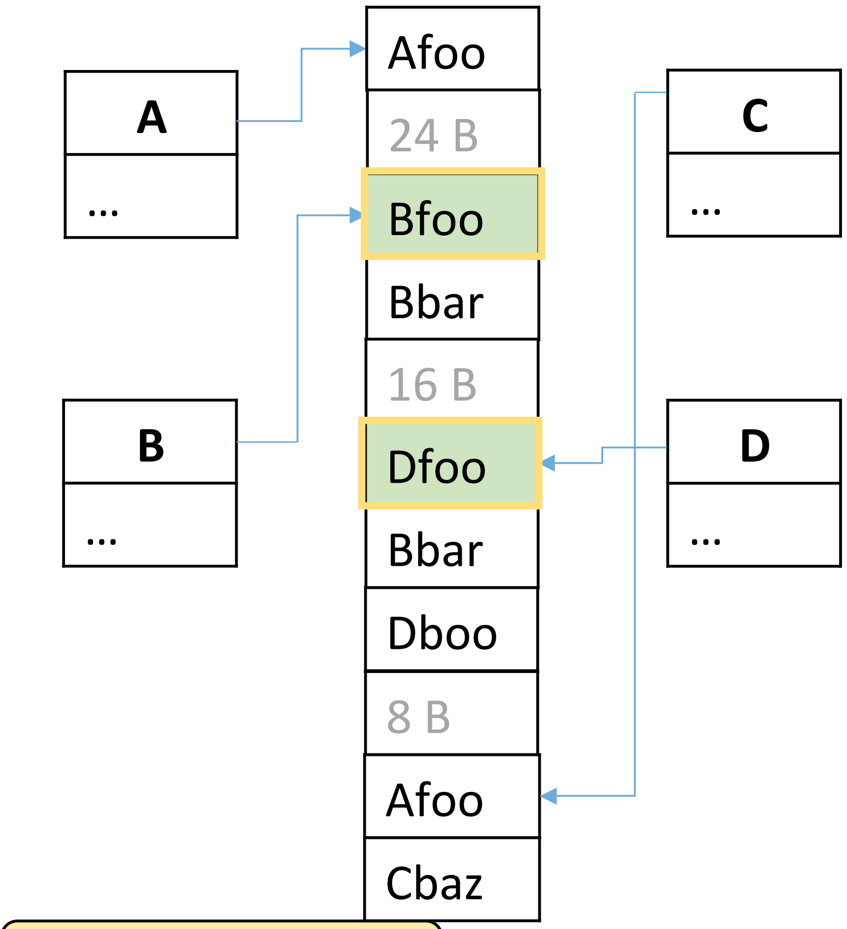




```
vptr = (*b)
assert (vptr ∊ { 0x20, 0x40 })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Ordered Memory Layout





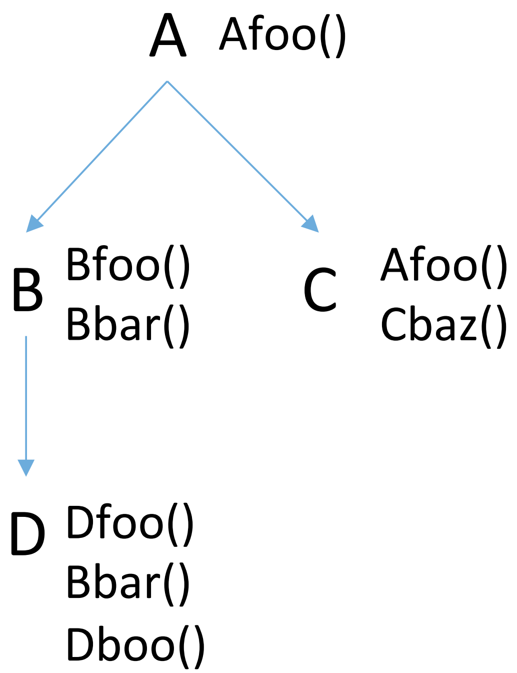
vptr = (*b)

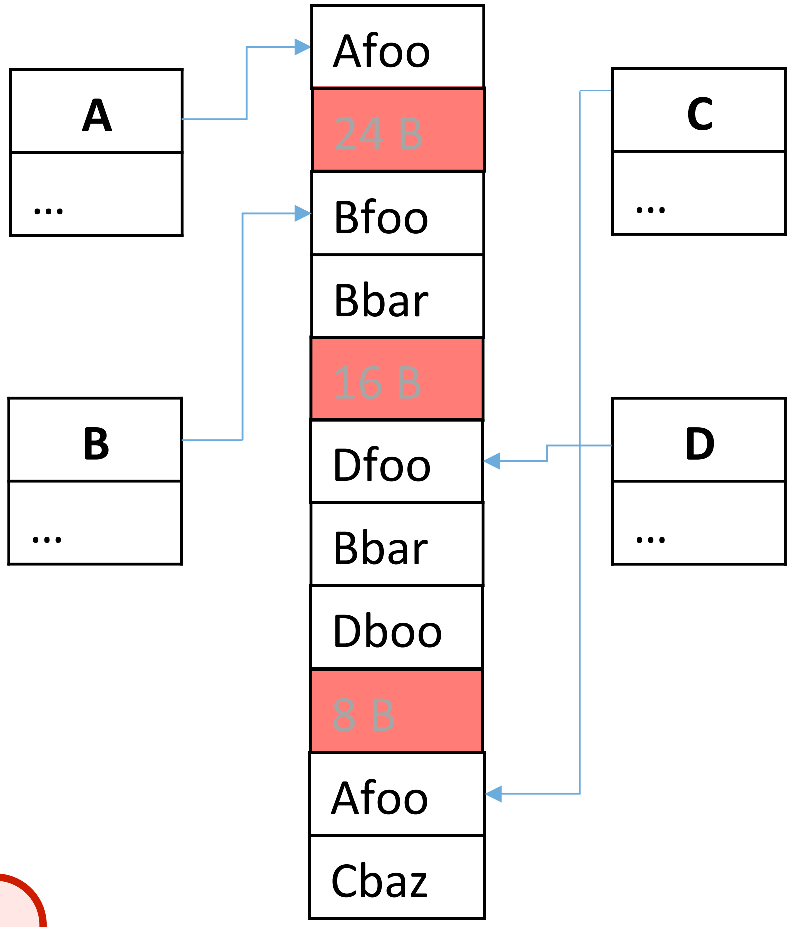
assert (0x20 ≤ vptr ≤ 0x40 ∧ vptr % 0x20 = 0)

fn_ptr = (*(vptr + 0))

(*fn_ptr)();

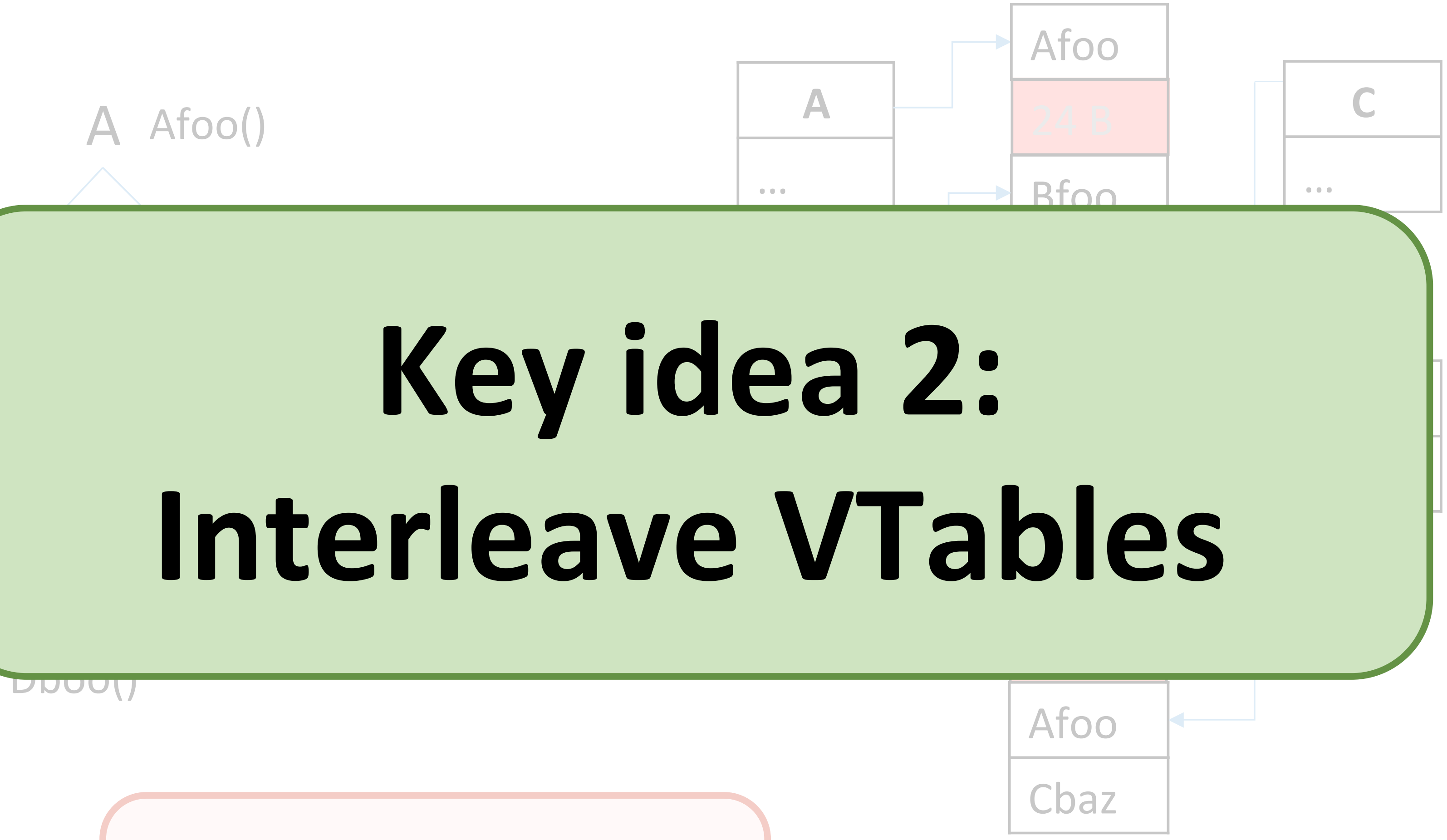
Ordered Memory Layout





Wasteful Extra Padding

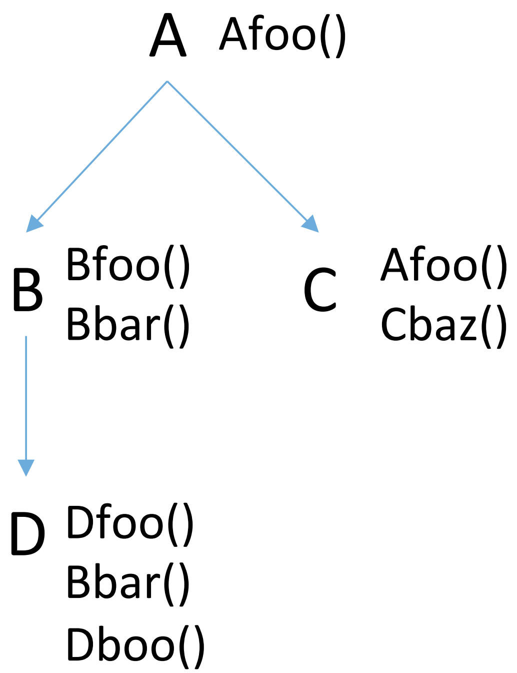
Ordered Memory Layout

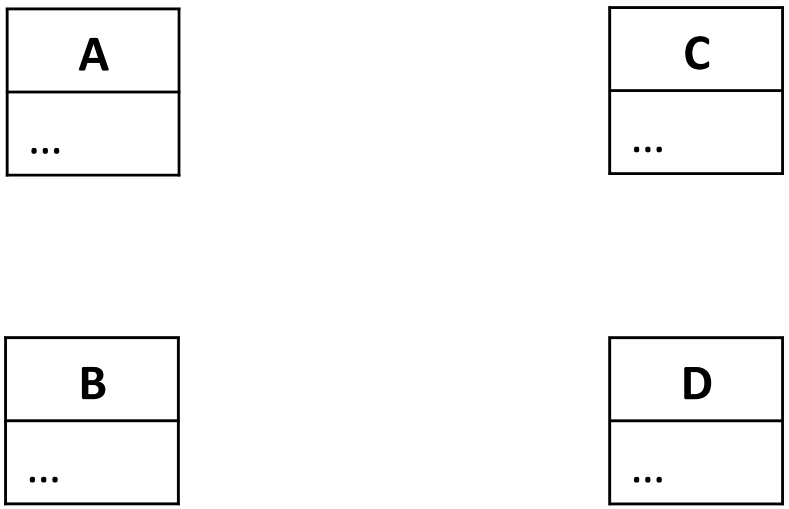


Key idea 2:
Interleave VTables

Wasteful Extra Padding

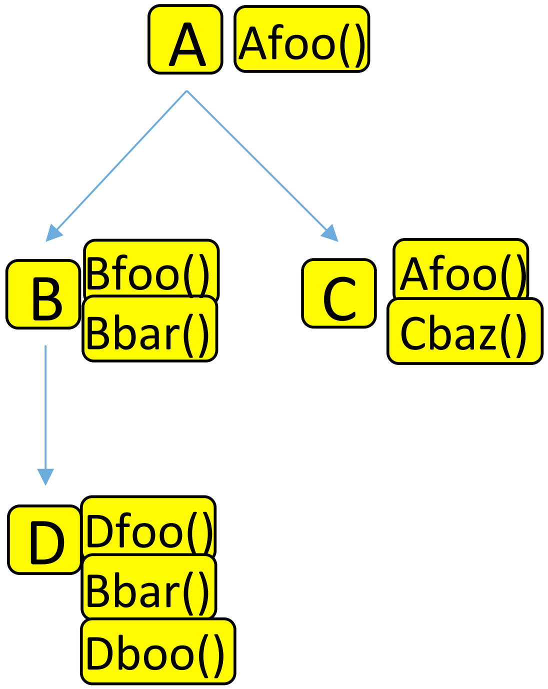
Interleaved Memory Layout

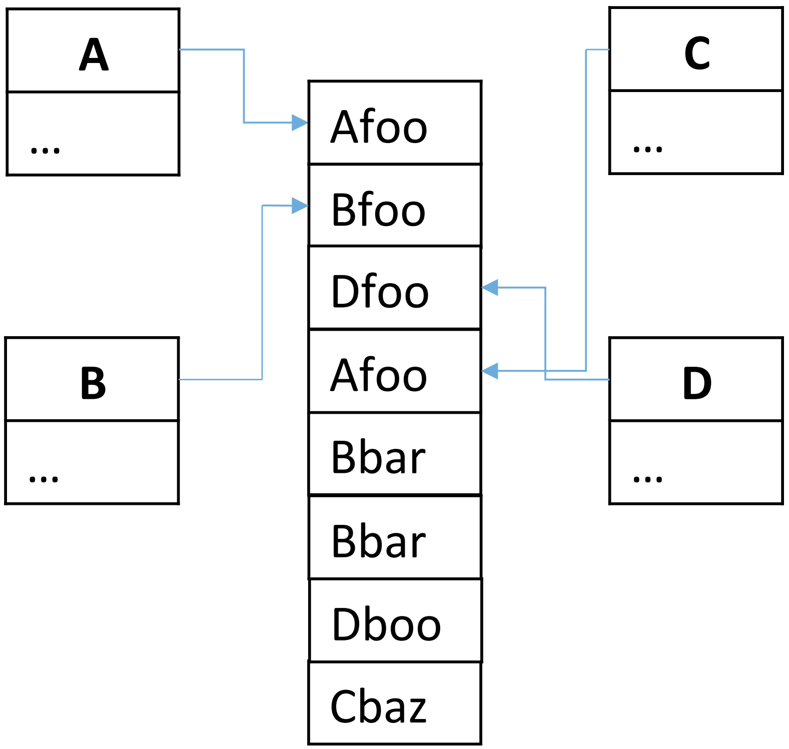




1. Traverse in pre-order: A, B, D, C
2. Layout each method

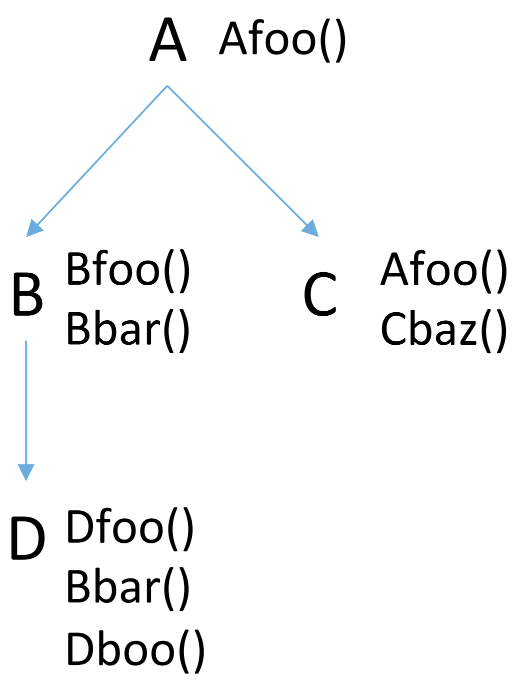
Interleaved Memory Layout

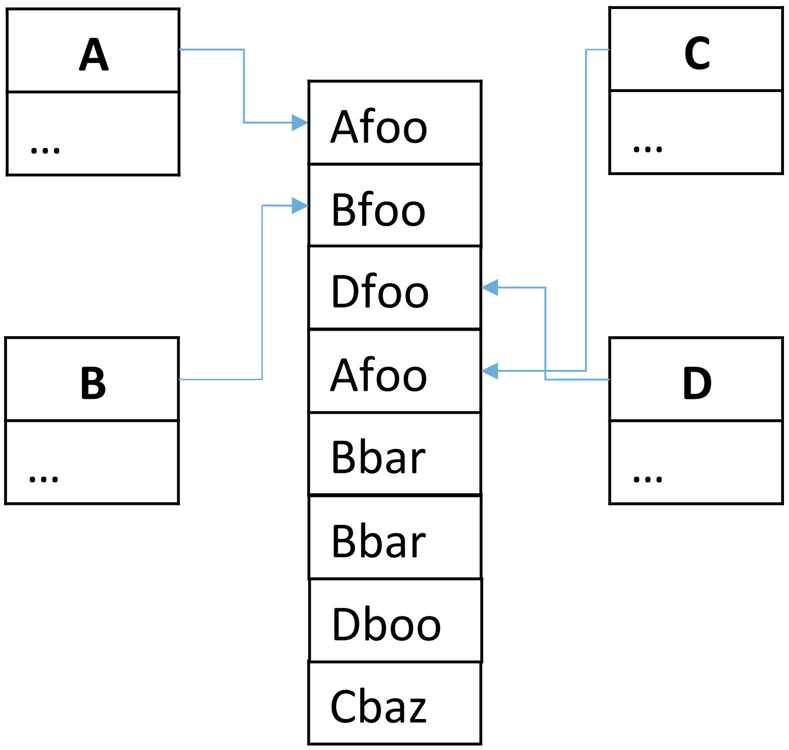




1. Traverse in pre-order: A, B, D, C
2. Layout each method

Interleaved Dynamic Dispatch

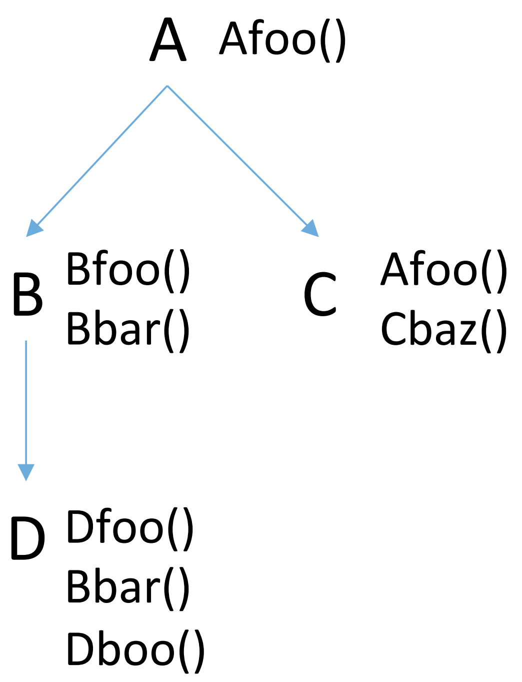


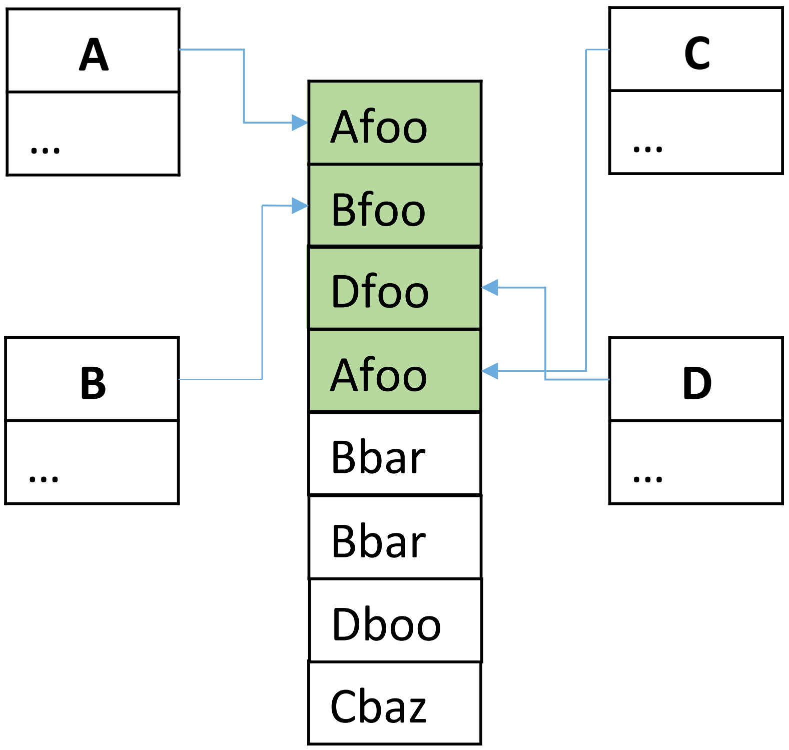


```
vptr = (*a)
assert(vptr in {A,B,C,D})
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

1. Traverse in pre-order: A, B, D, C
2. Layout each method

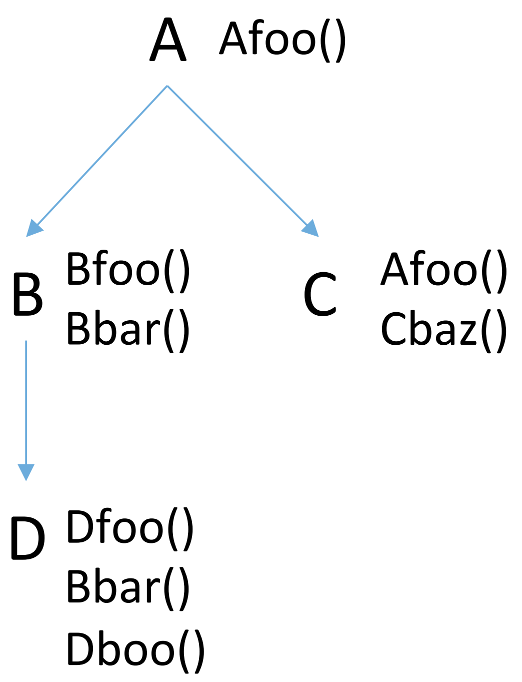
Interleaved Dynamic Dispatch

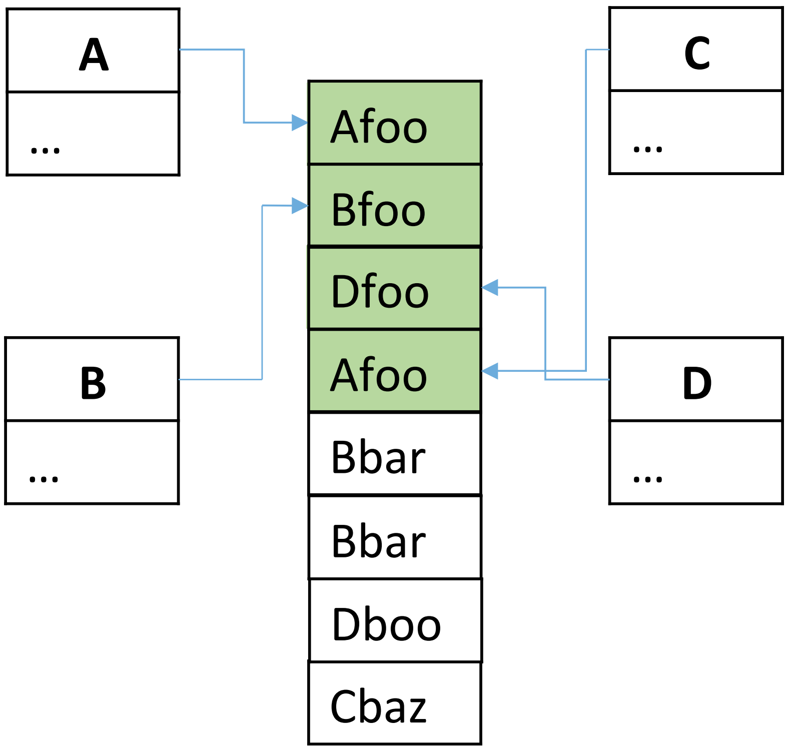




```
vptr = (*a)
assert (vptr ∊ { A,B,C,D })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Interleaved Dynamic Dispatch

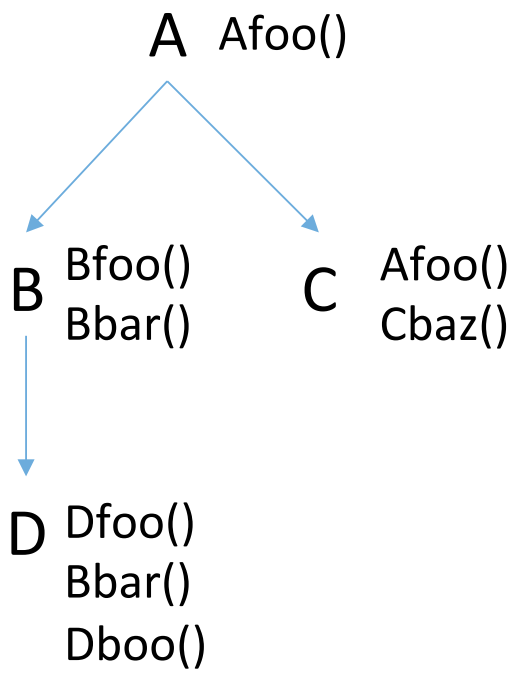


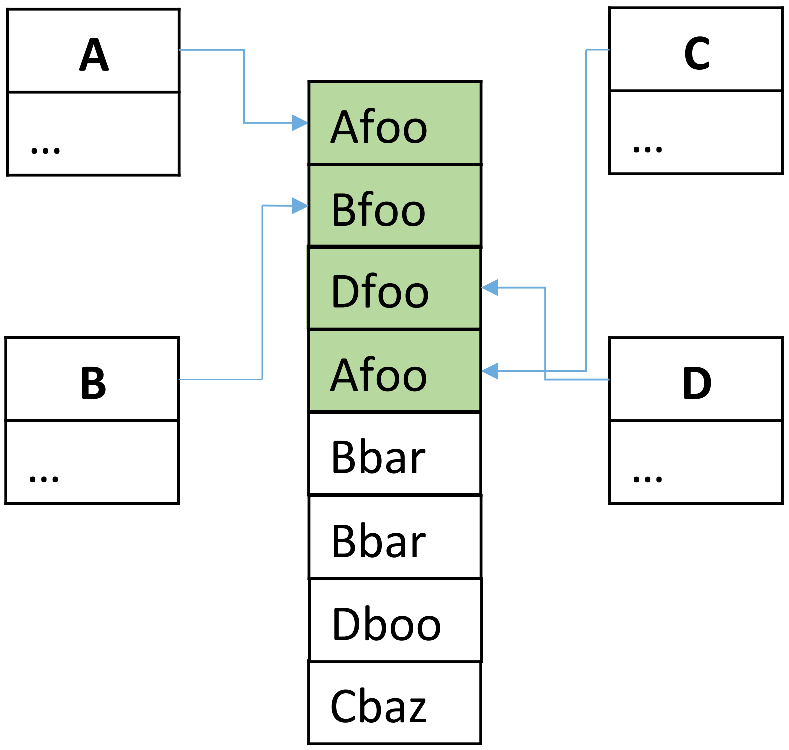


```
vptr = (*a)
assert (vptr ∊ { 0x0, 0x8, 0x10, 0x18 })
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Address Points
Consecutive Addrs.

Interleaved Dynamic Dispatch

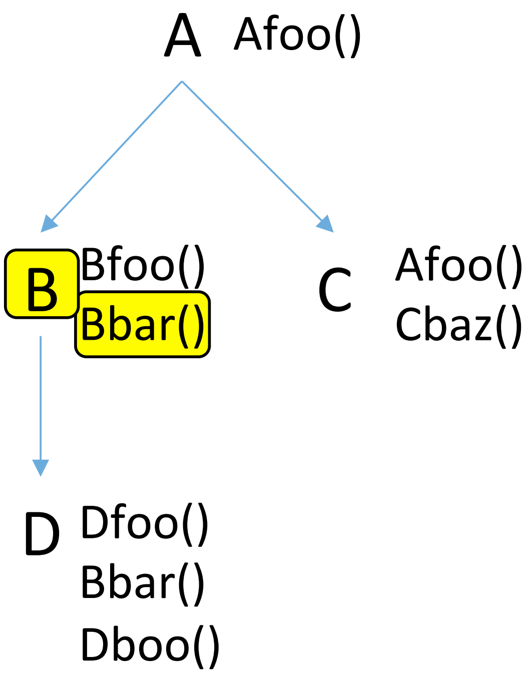


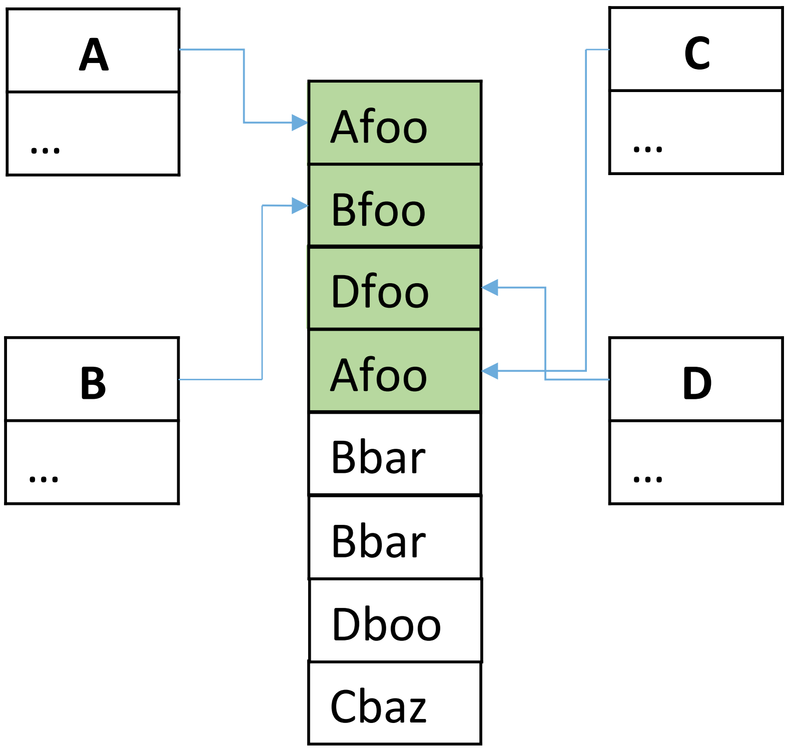


```
vptr = (*a)
assert (0x0 ≤ vptr ≤ 0x18 ∧ vptr % 0x8 = 0)
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Same Check
Different range & alignment

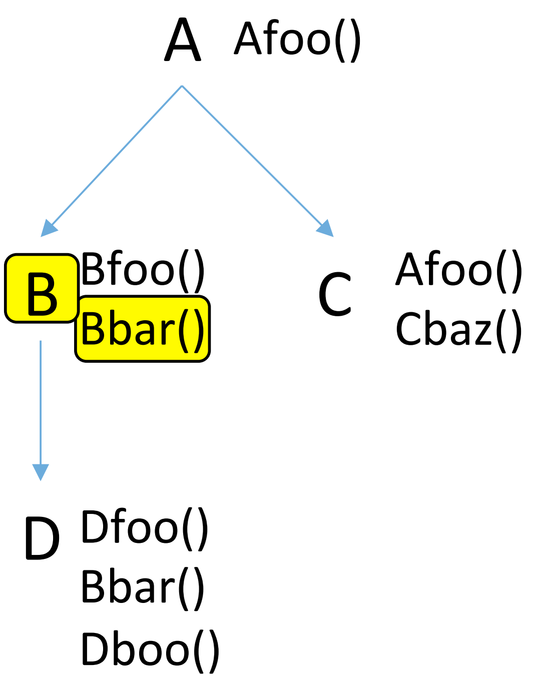
Interleaved Dynamic Dispatch

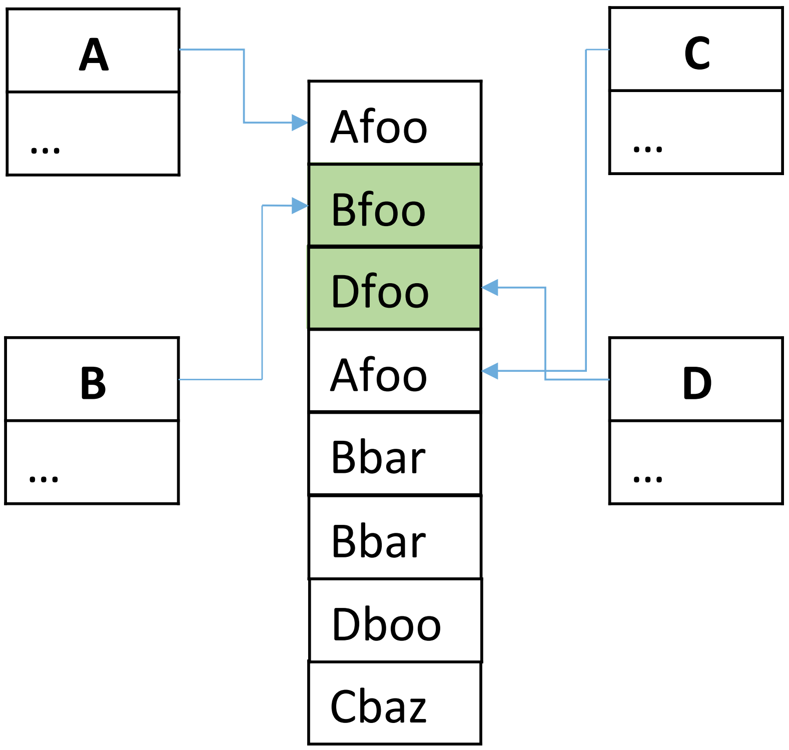




```
vptr = (*b)
assert (0x0 ≤ vptr ≤ 0x18 ∧ vptr % 0x8 = 0)
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

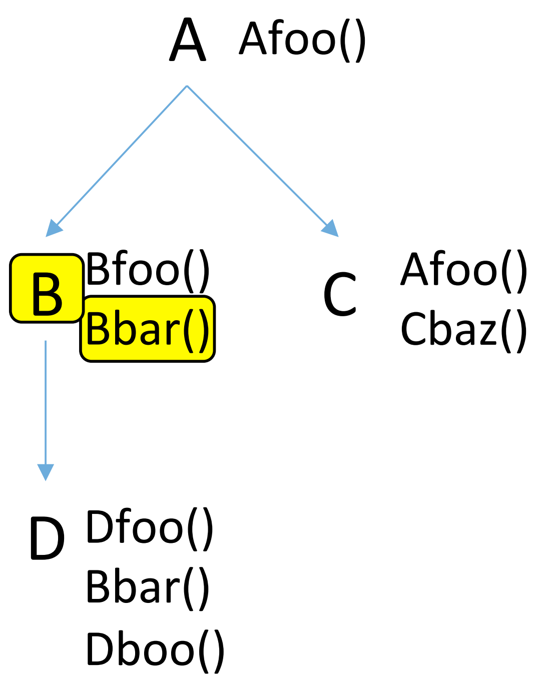
Interleaved Dynamic Dispatch

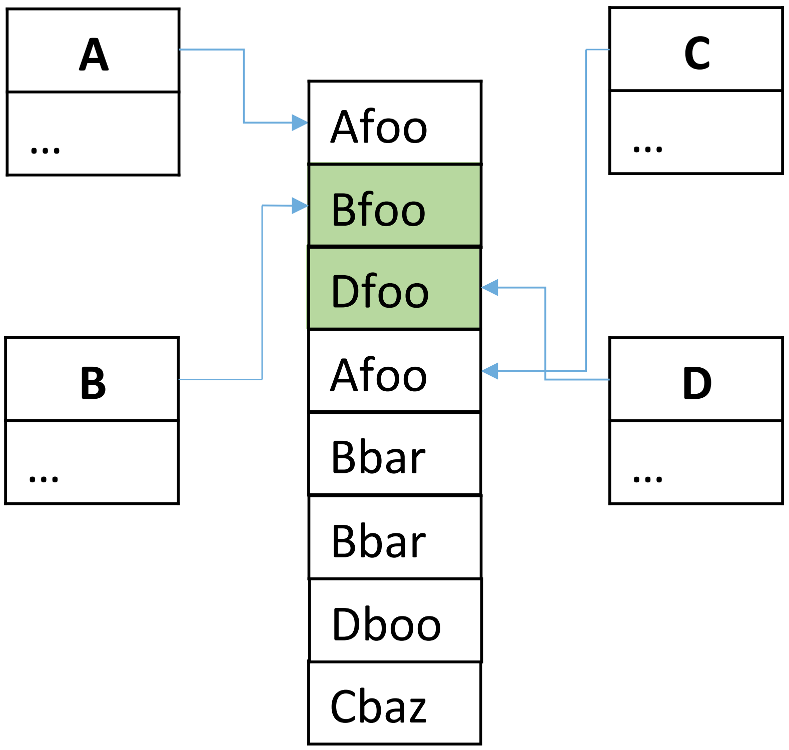




```
vptr = (*b)
assert (0x8 ≤ vptr ≤ 0x10 ∧ vptr % 0x8 = 0)
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

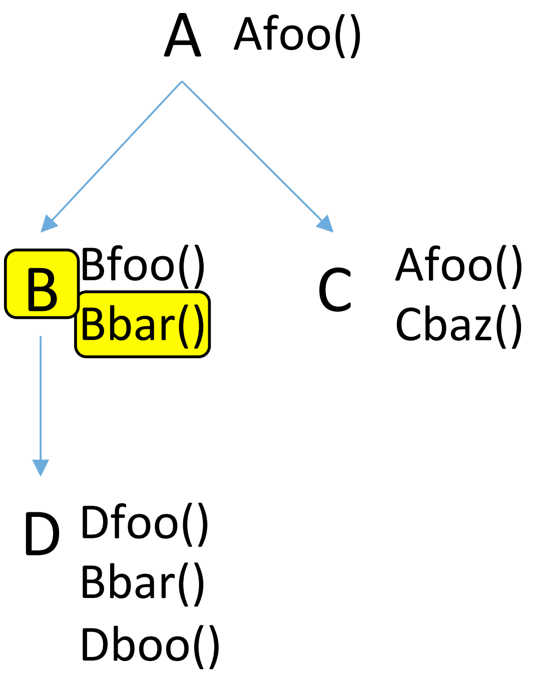
Interleaved Dynamic Dispatch

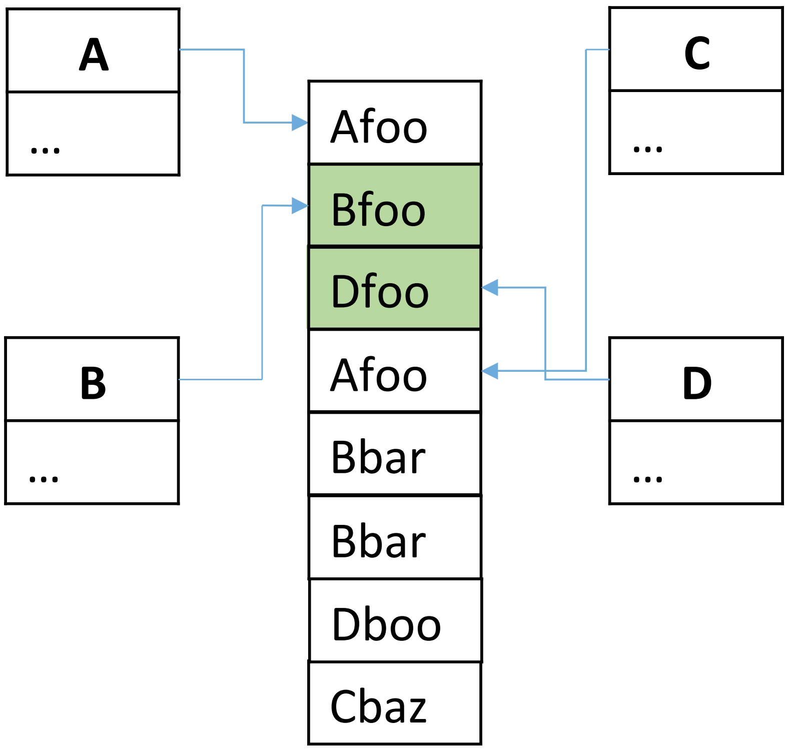




```
vptr = (*b)
assert (0x8 ≤ vptr ≤ 0x10 ∧ vptr % 0x8 = 0)
fn_ptr = (*(vptr + 0))
(*fn_ptr)();
```

Interleaved Dynamic Dispatch

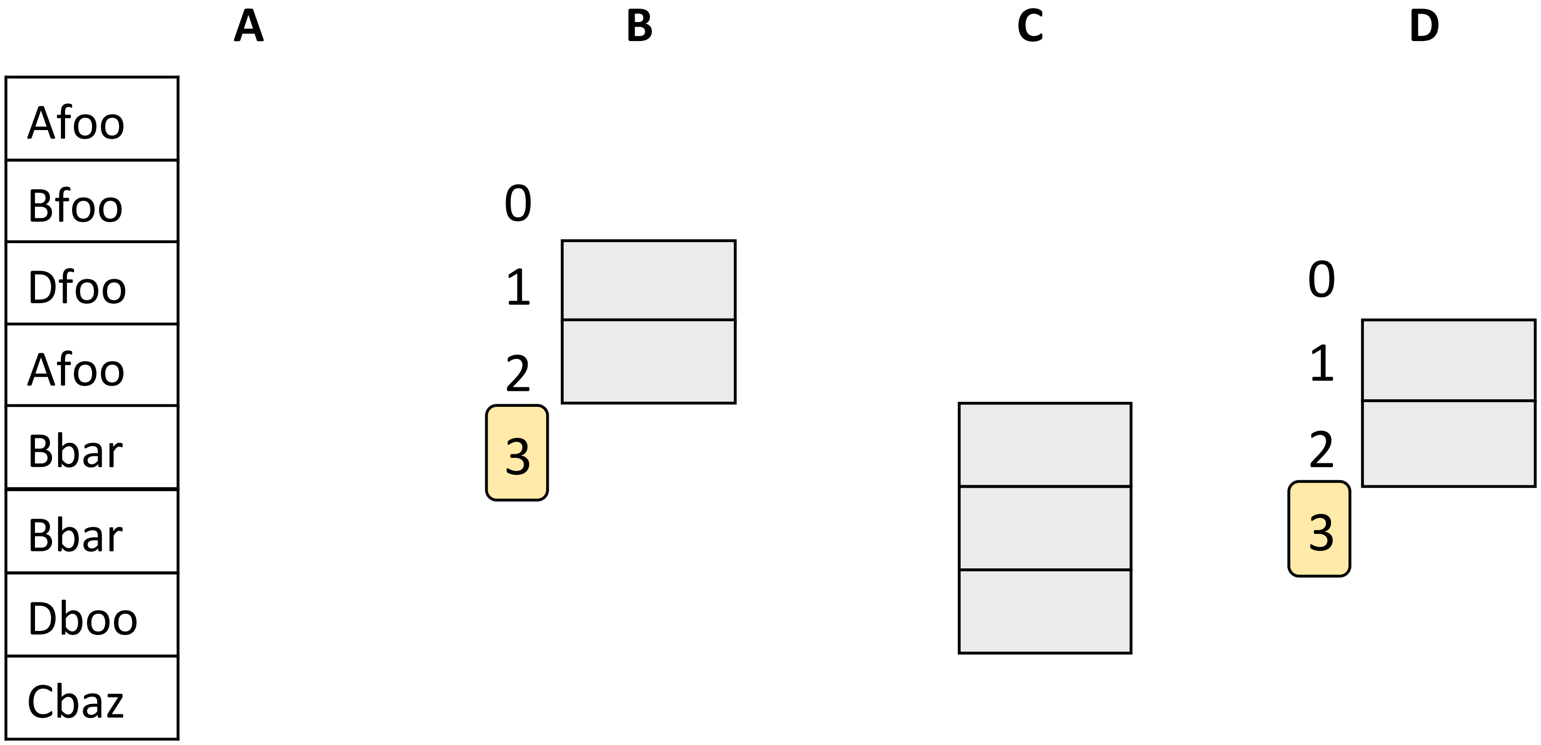




```
vptr = (*b)
assert (0x8 ≤ vptr ≤ 0x10 ∧ vptr % 0x8 = 0)
fn_ptr = (*(vptr + 0x18))
(*fn_ptr)();
```

Same index in all vtables

Where are the Vtables?



Bar method index same
VTable Dynamic Dispatch still works!

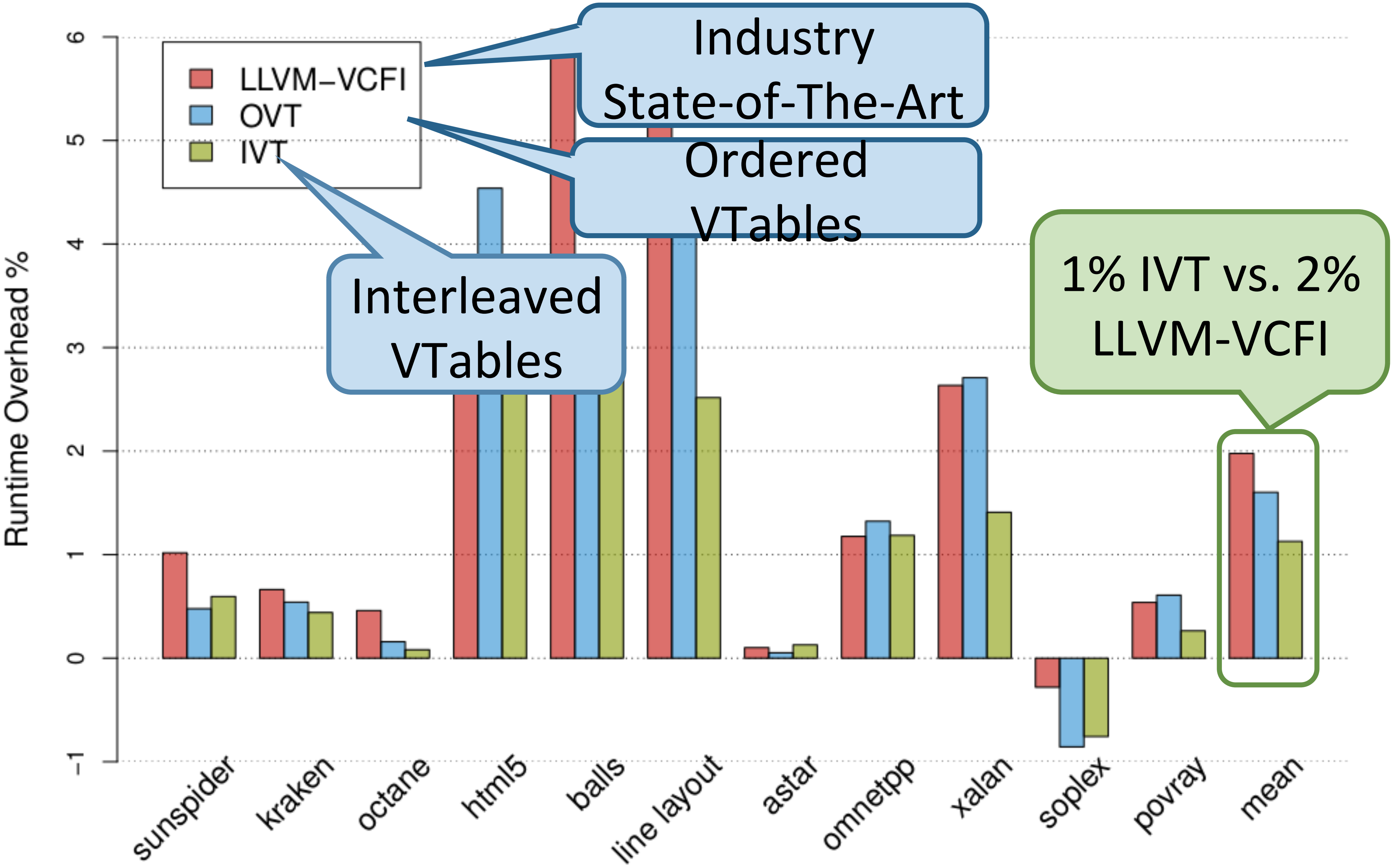
Implementation

- Implemented technique in LLVM
- Handled all cases of C++ inheritance
- Algorithms with proofs of correctness

Evaluation

- Evaluated on SPEC 2006/Chrome
- Compared to state-of-the-art industry CFI
- One (benign) CFI violation in SPEC 2006

Results: Runtime Overhead



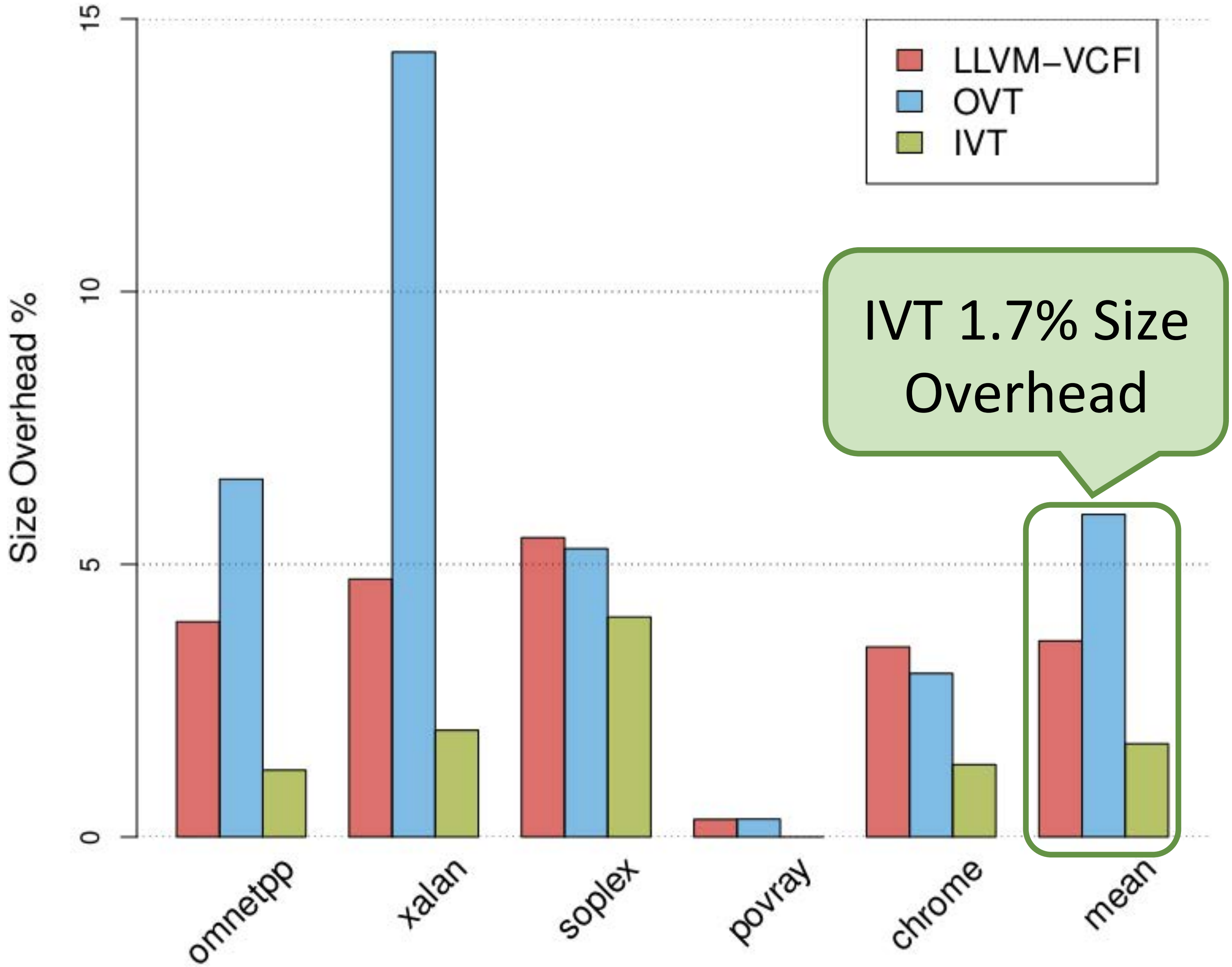
Industry State-of-The-Art

Ordered VTables

Interleaved VTables

1% IVT vs. 2% LLVM-VCFI

Results: Size Overhead



IVT 1.7% Size Overhead

Future Work

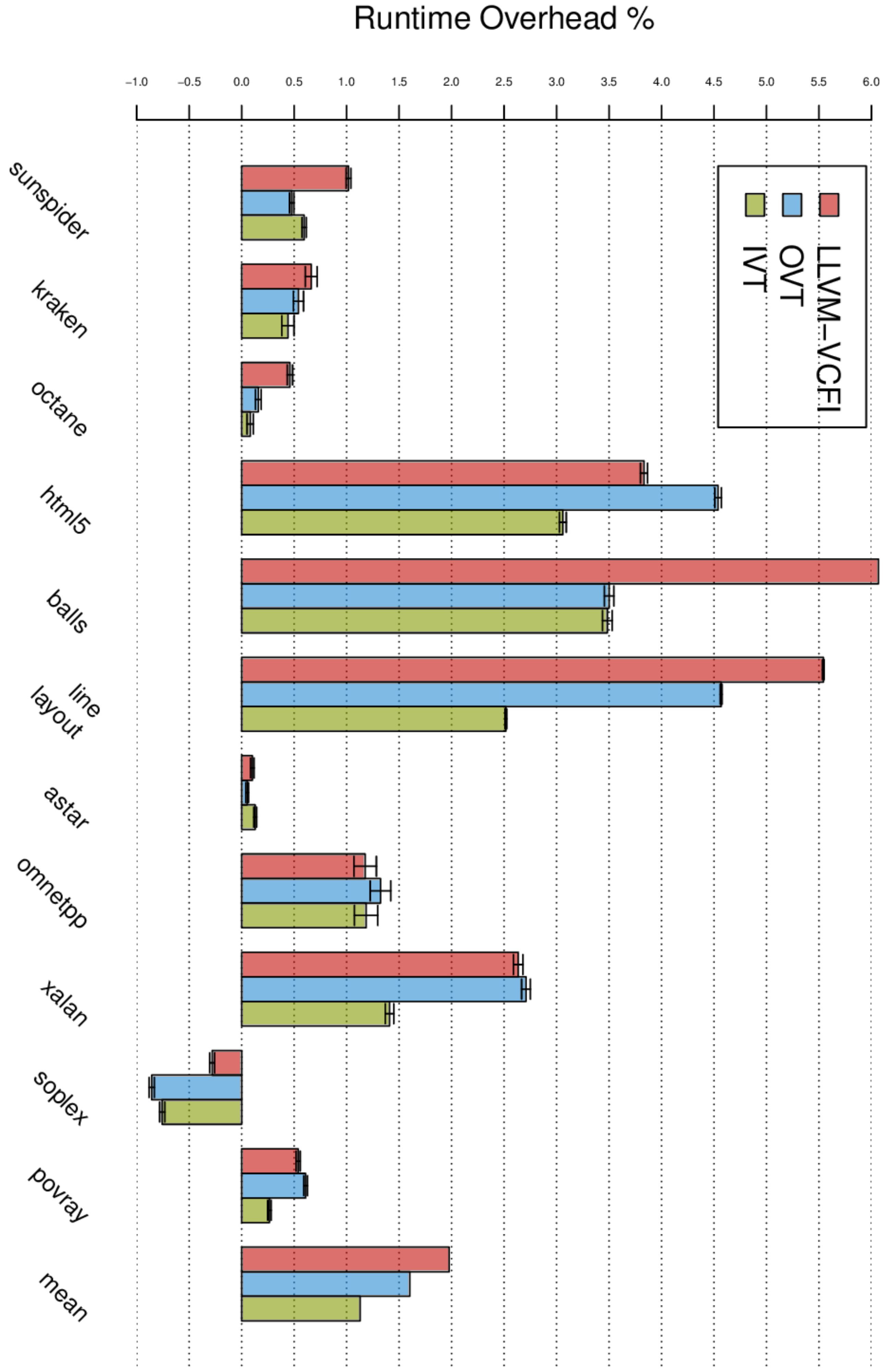
- Dynamic linking/loading
- Handle C++ member pointers
- Hardware assisted range checks

Summary

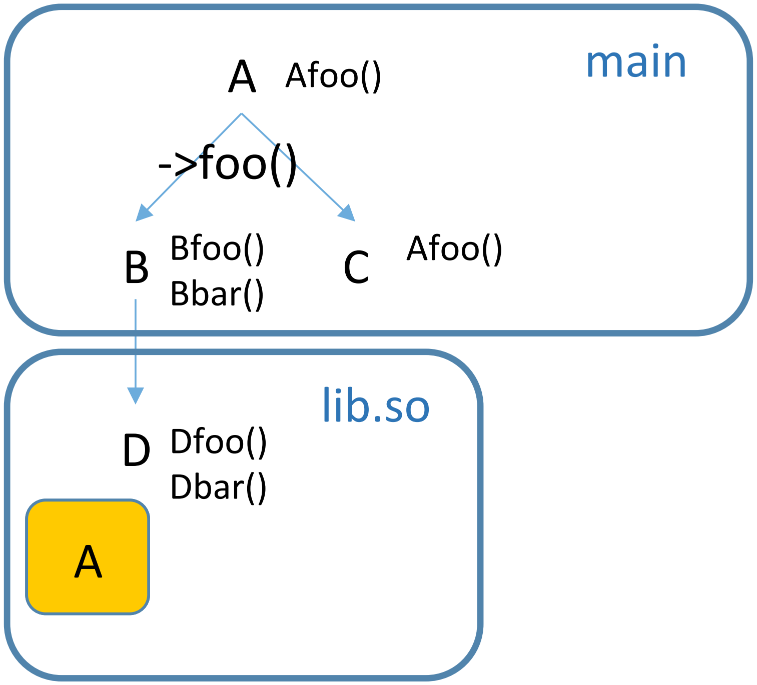
- New approach based on VTable layouts
- Efficient range checks for dynamic dispatch
- https://github.com/UCSD-PL/ivt

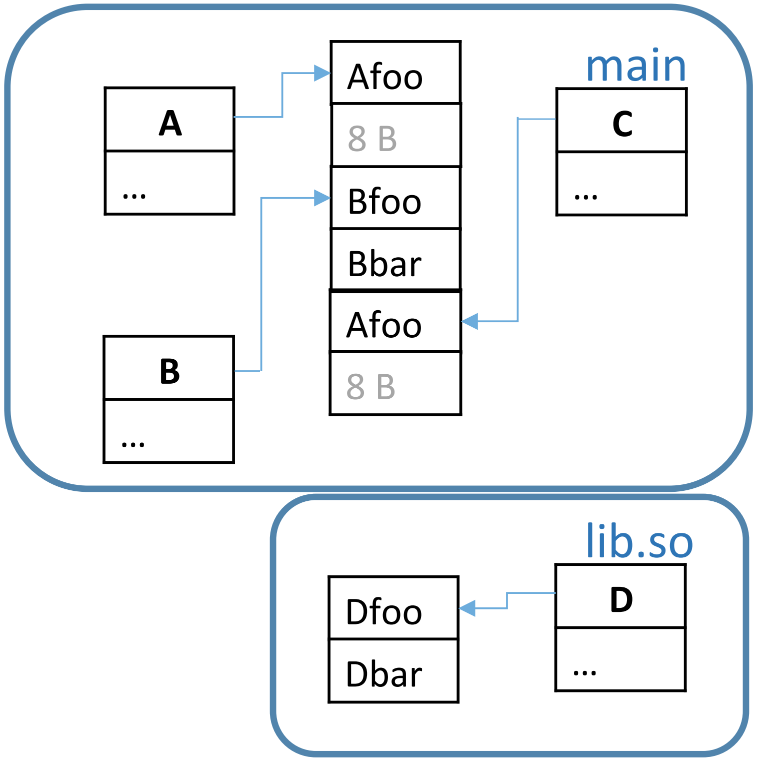
THANK YOU!

Backup


Runtime Overhead %
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5
6.0
LLVM–VCFI
OVT
IVT
sunspider
kraken
octane
html5
balls
line layout
astar
omnetpp
xalan
soplex
povray
mean

Dynamic Linking



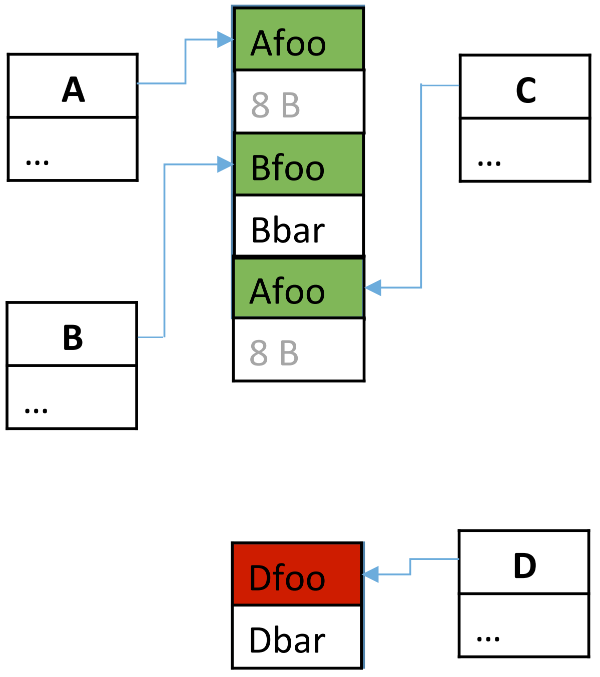


Only 0.012% of dynamic Firefox virtual calls!

Dynamic Fault Handlers

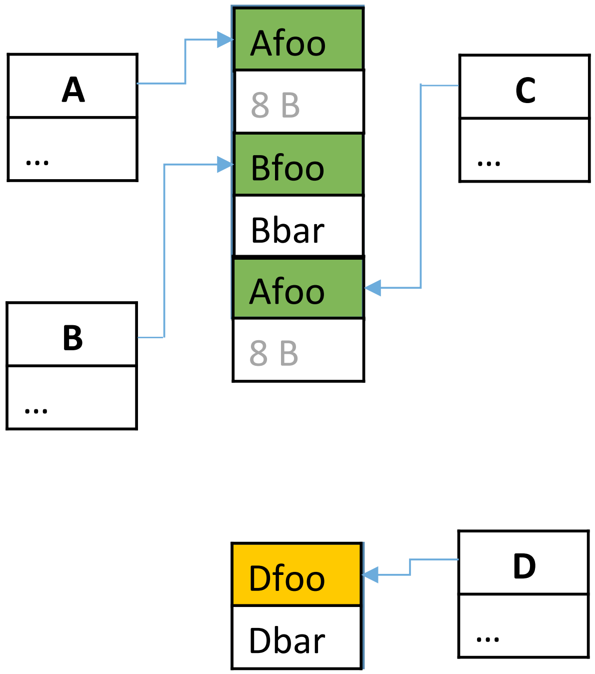
```
vptr = (*a)
if (0x00 ≤ vptr ≤ 0x20 ∧ vptr % 0x10 = 0)
   goto S;
assert(false);
```

```
S   fn_ptr = (*(vptr + 0))
:   (fn_ptr*)();
```

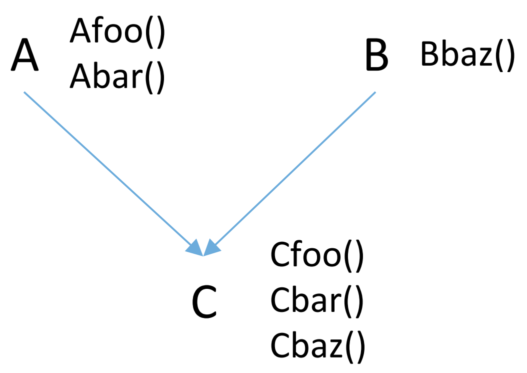


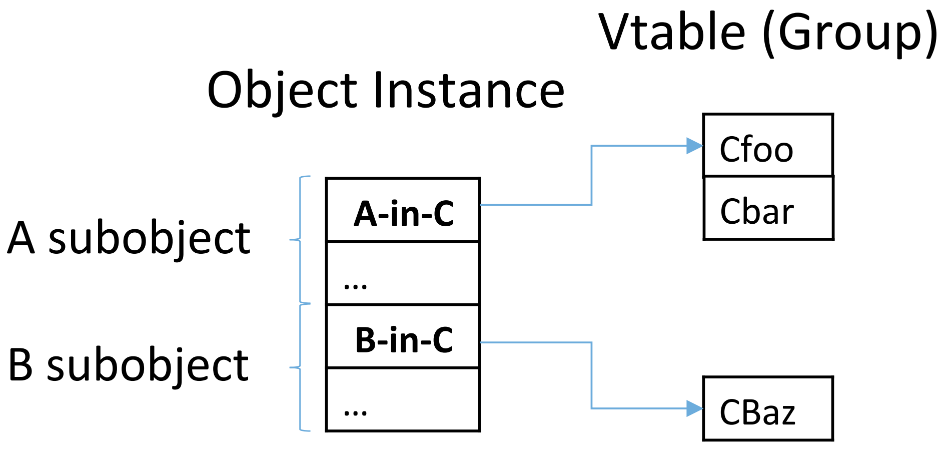
Dynamic Fault Handlers

```
    vptr = (*a)
    if (0x00 ≤ vptr ≤ 0x20 ∧ vptr % 0x10 = 0)
       goto S;
    L = link_unit (vptr) // e.g. dladddr()
    if vptr_safe(L, vptr, class A)
       goto S
    assert(false);

S   fn_ptr = (*(vptr + 0))
:   (fn_ptr*)();
```

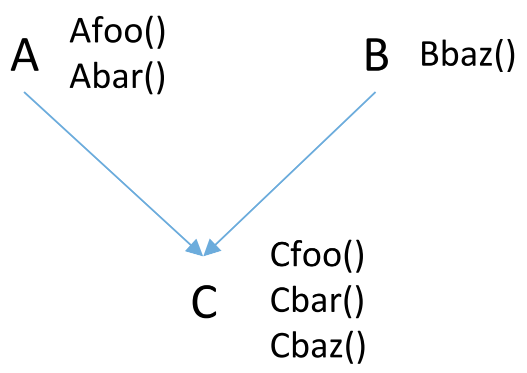


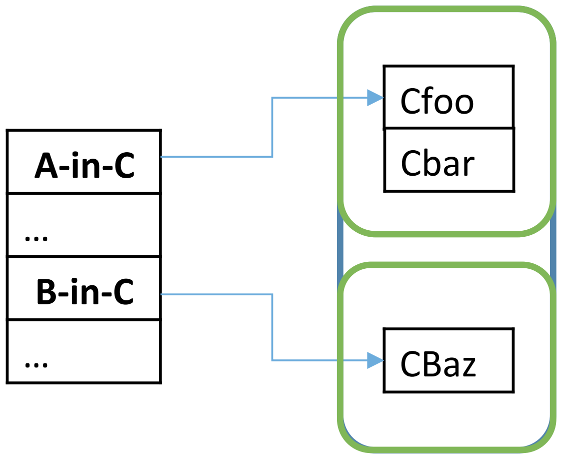
Multiple Inheritance



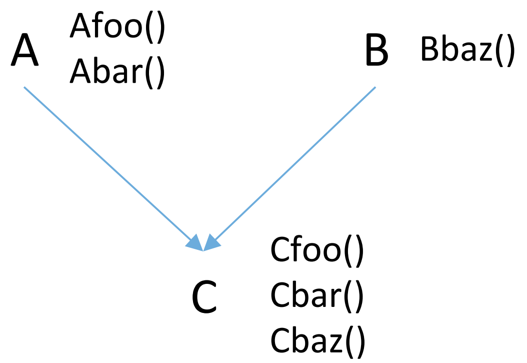


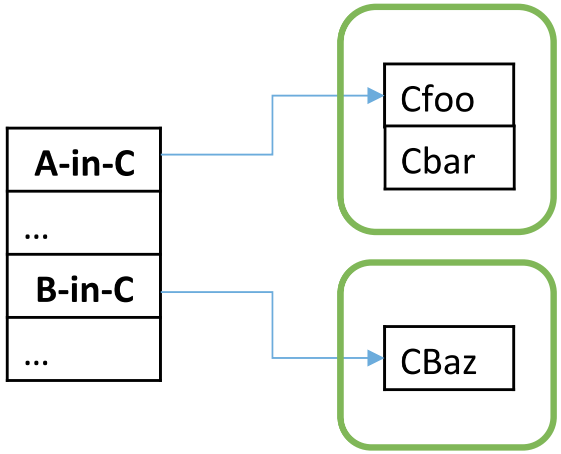
Multiple Inheritance



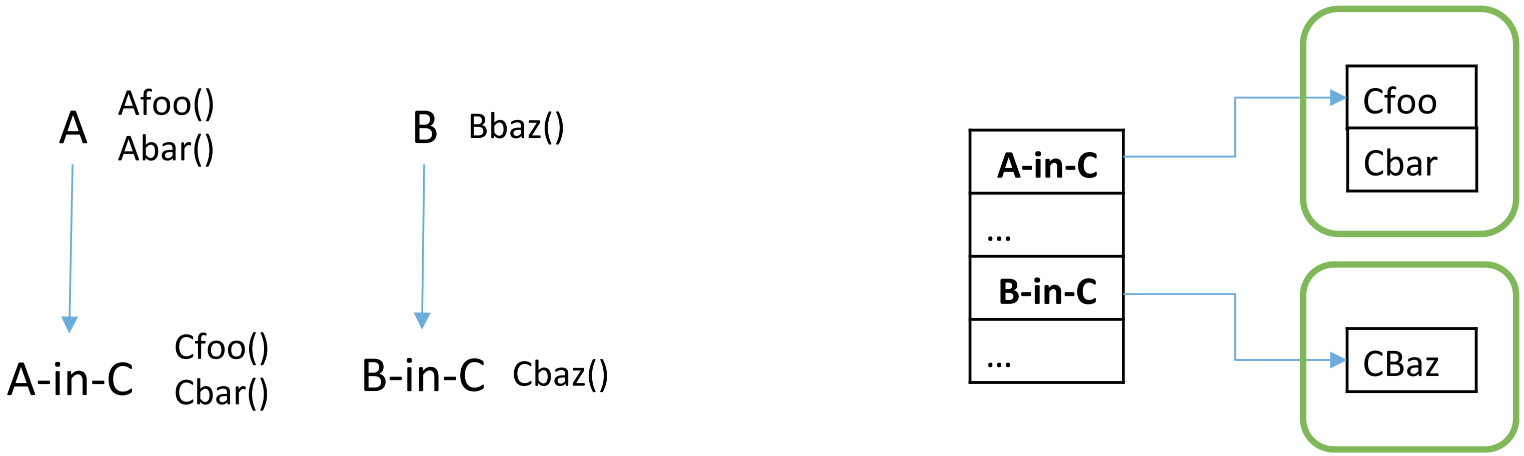


Multiple Inheritance





Multiple Inheritance



A Afoo() Abar()

A-in-C Cfoo() Cbar()

B Bbaz()

B-in-C Cbaz()

A-in-C

…

B-in-C

…

Cfoo

Cbar

CBaz