black hat BRIEFINGS

AUGUST 6-7, 2025
MANDALAY BAY / LAS VEGAS

Coroutine Frame-Oriented Programming

Breaking Control Flow Integrity by Abusing Modern C++

Marcos Bajo *h3xduck*

Christian Rossow

The Old Ages

1972 **2000** **2010** **2020**

Buffer overflows 1st mentioned

ESD-TR-73-51, Vol. II

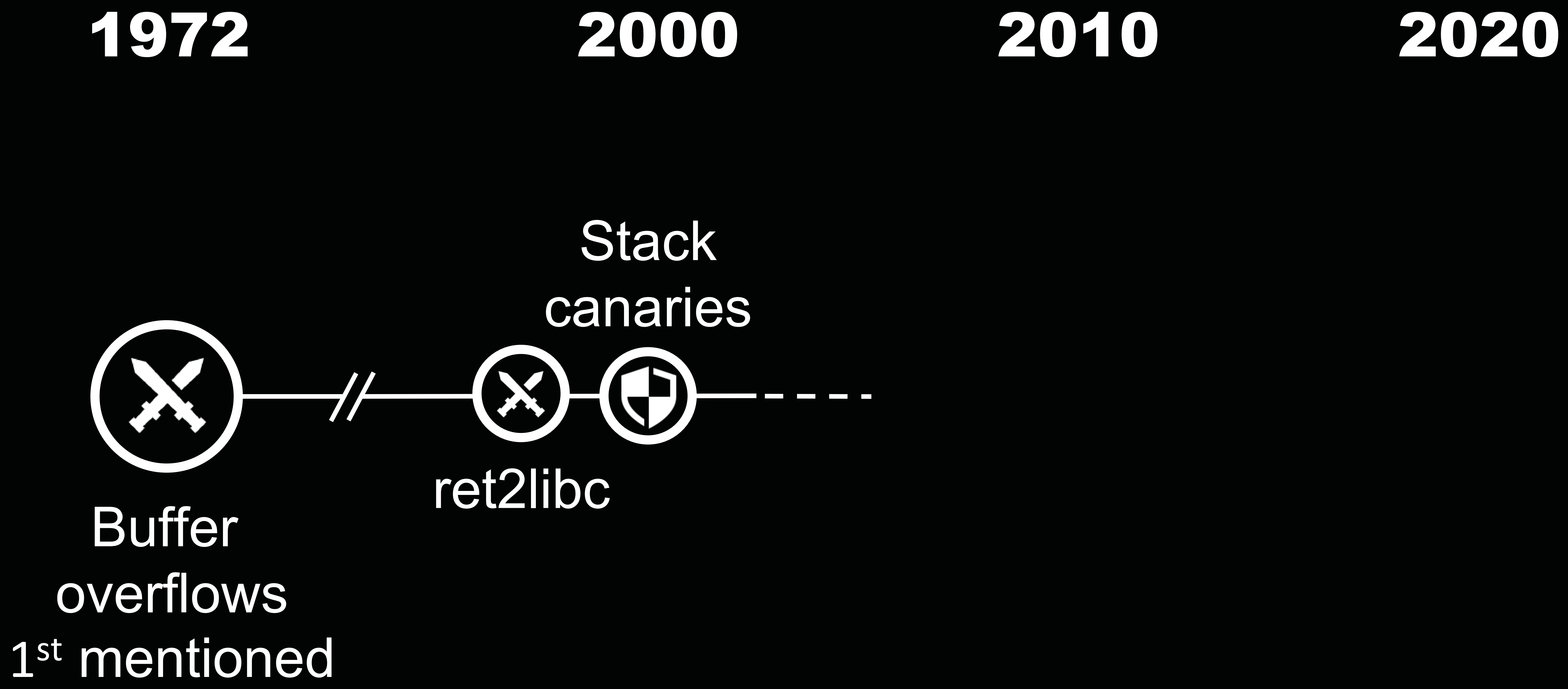
COMPUTER SECURITY TECHNOLOGY PLANNING STUDY

James P. Anderson

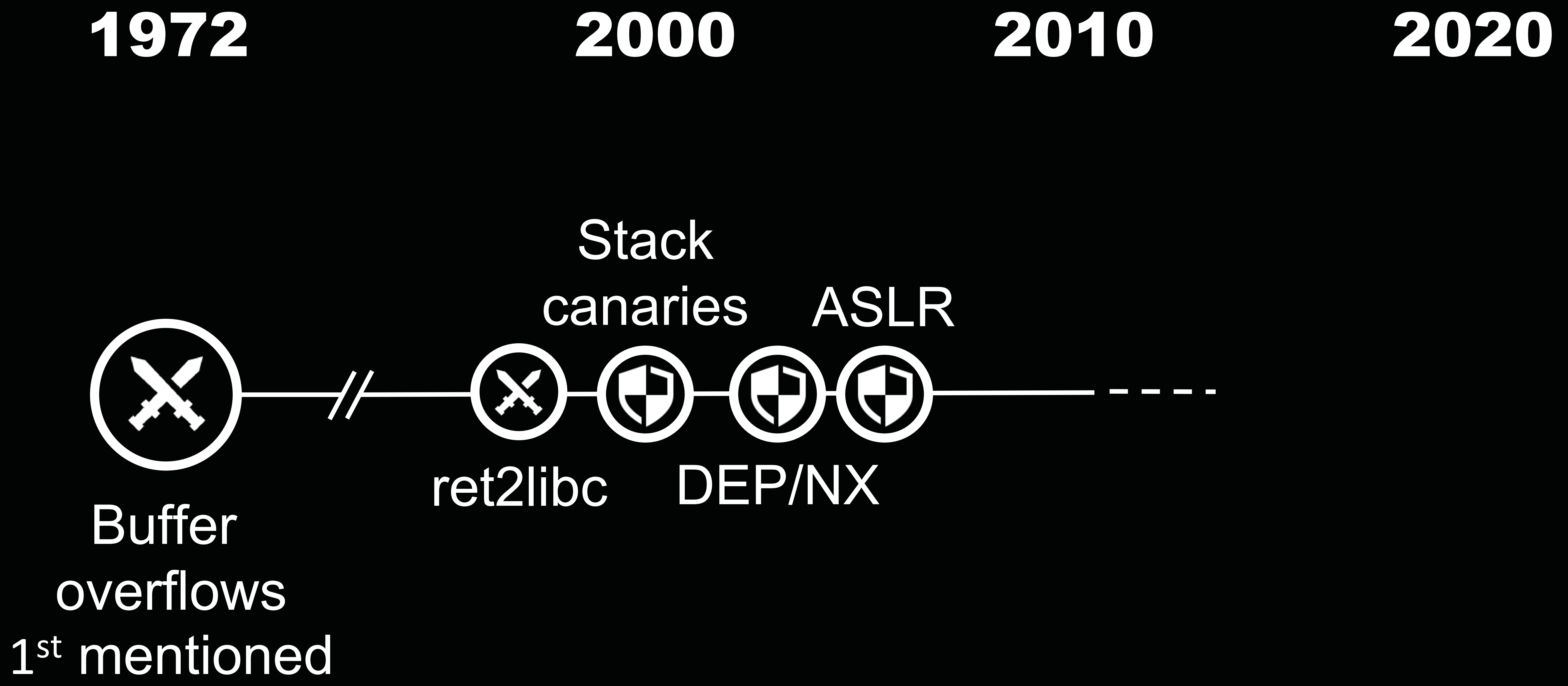
October 1972

ELECTRONIC SYSTEMS DIVISION

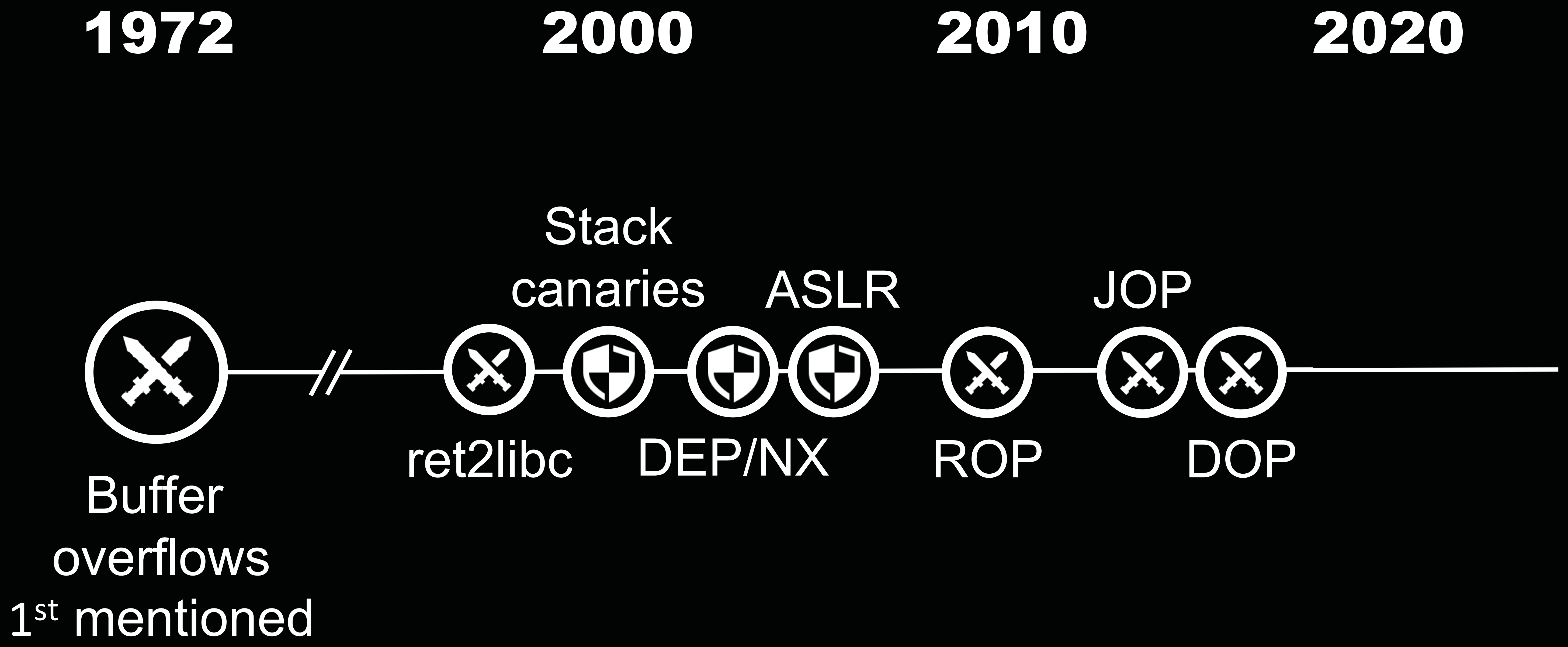
The Old Ages


1972
2000
2010
2020
Stack canaries
ret2libc
Buffer overflows 1st mentioned

The Old Ages

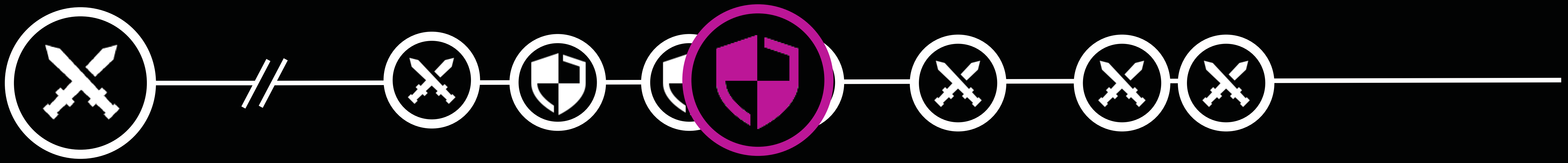

1972
2000
2010
2020
Stack canaries
ASLR
ret2libc
DEP/NX
Buffer overflows 1st mentioned

The Old Ages


1972
2000
2010
2020
Stack canaries
ASLR
JOP
ret2libc
DEP/NX
ROP
DOP
Buffer overflows 1st mentioned

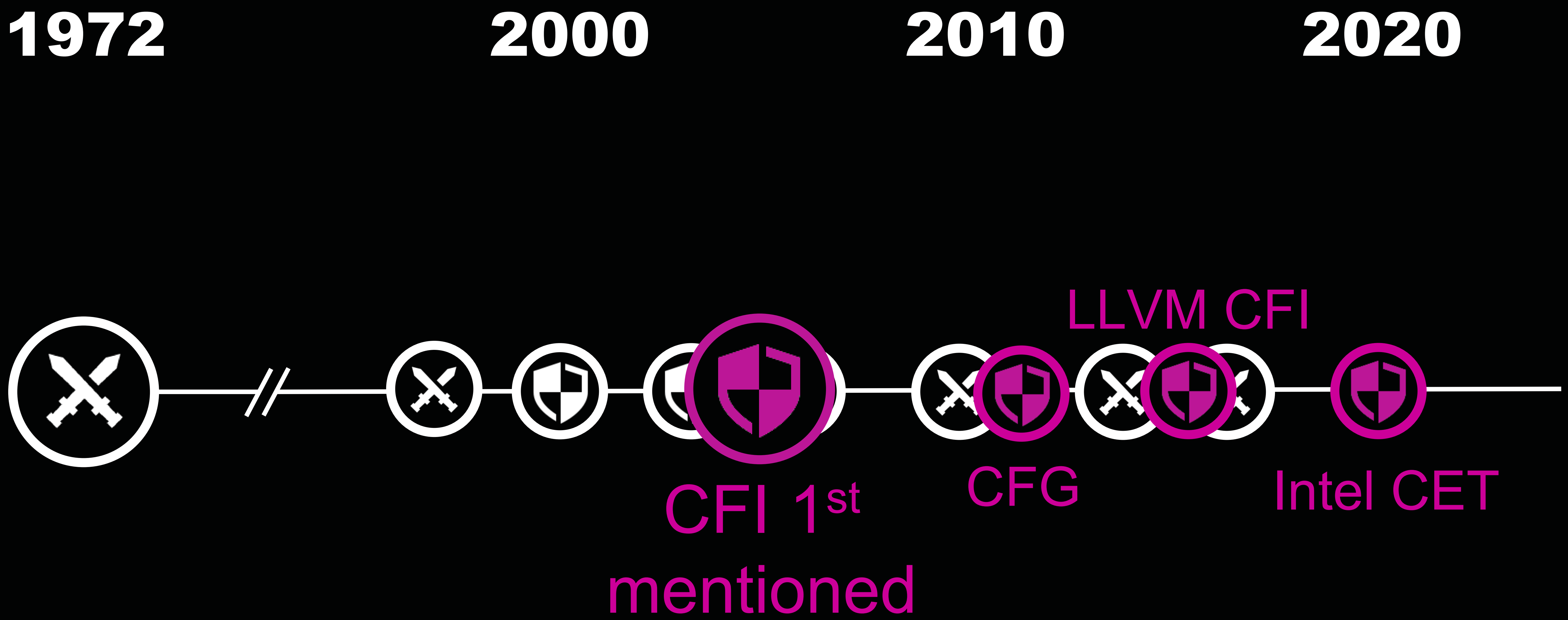
The Modern Ages

1972 2000 2010 2020

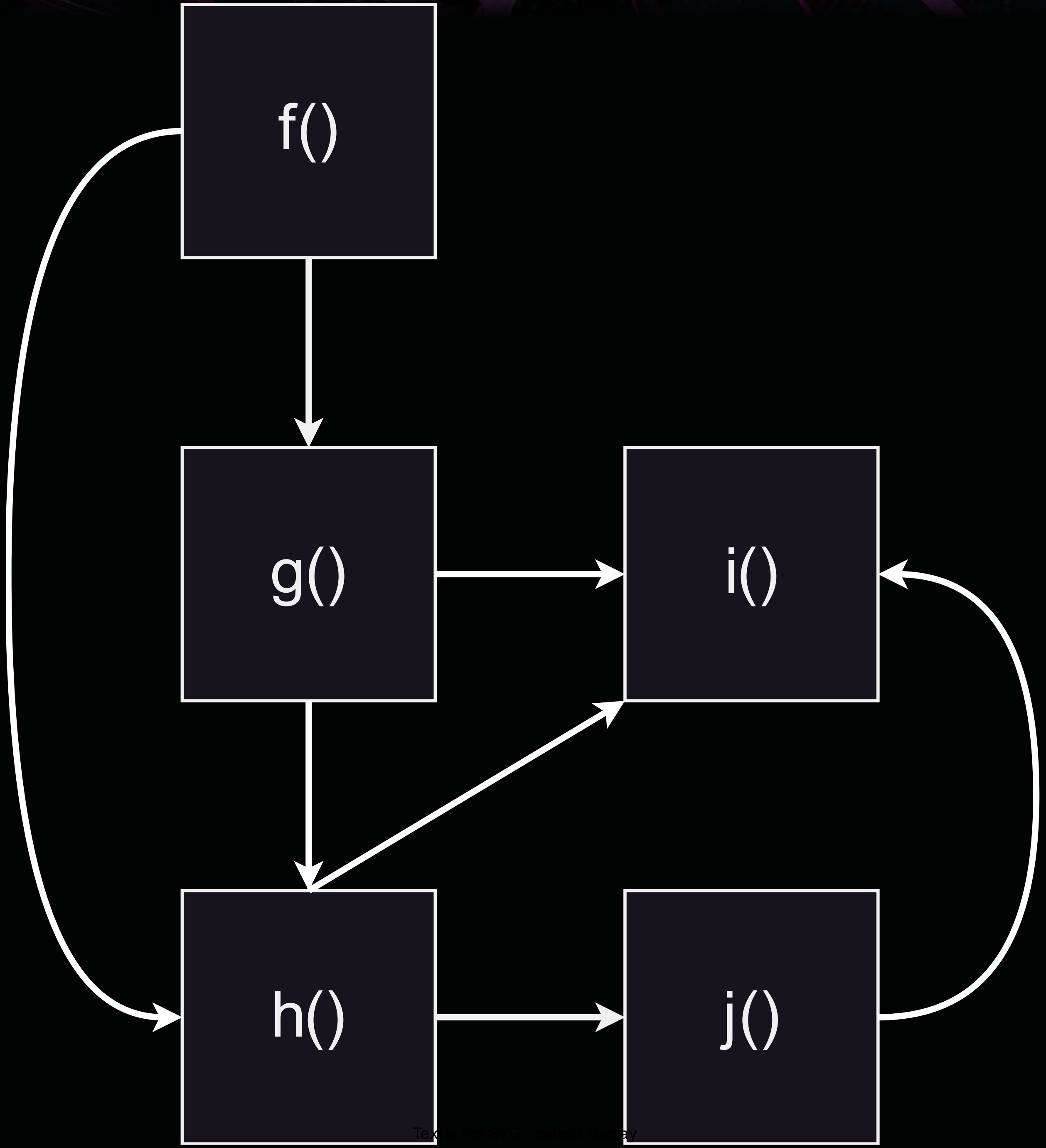


CFI 1st mentioned

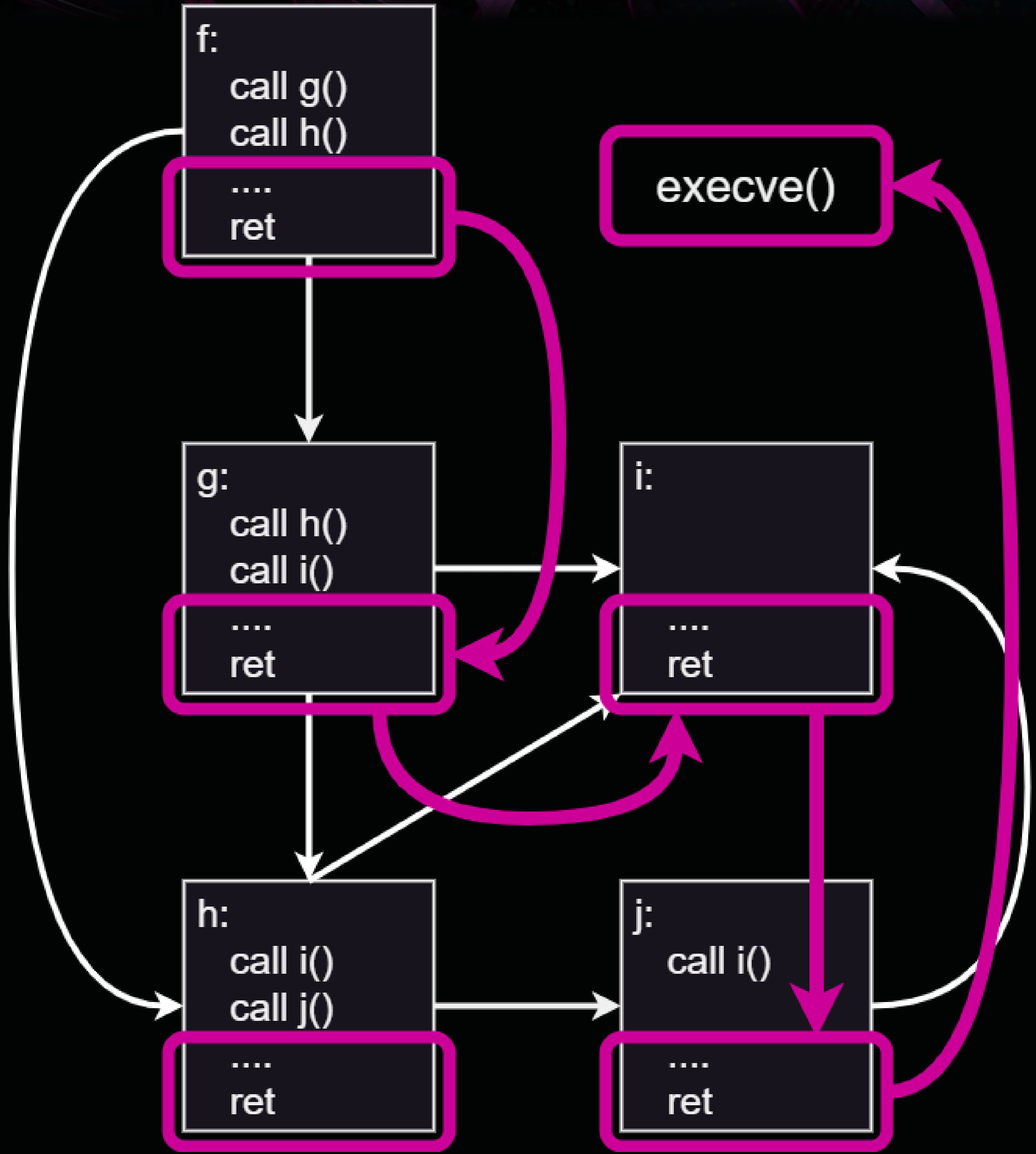
The Modern Ages



Code Reuse



Code Reuse

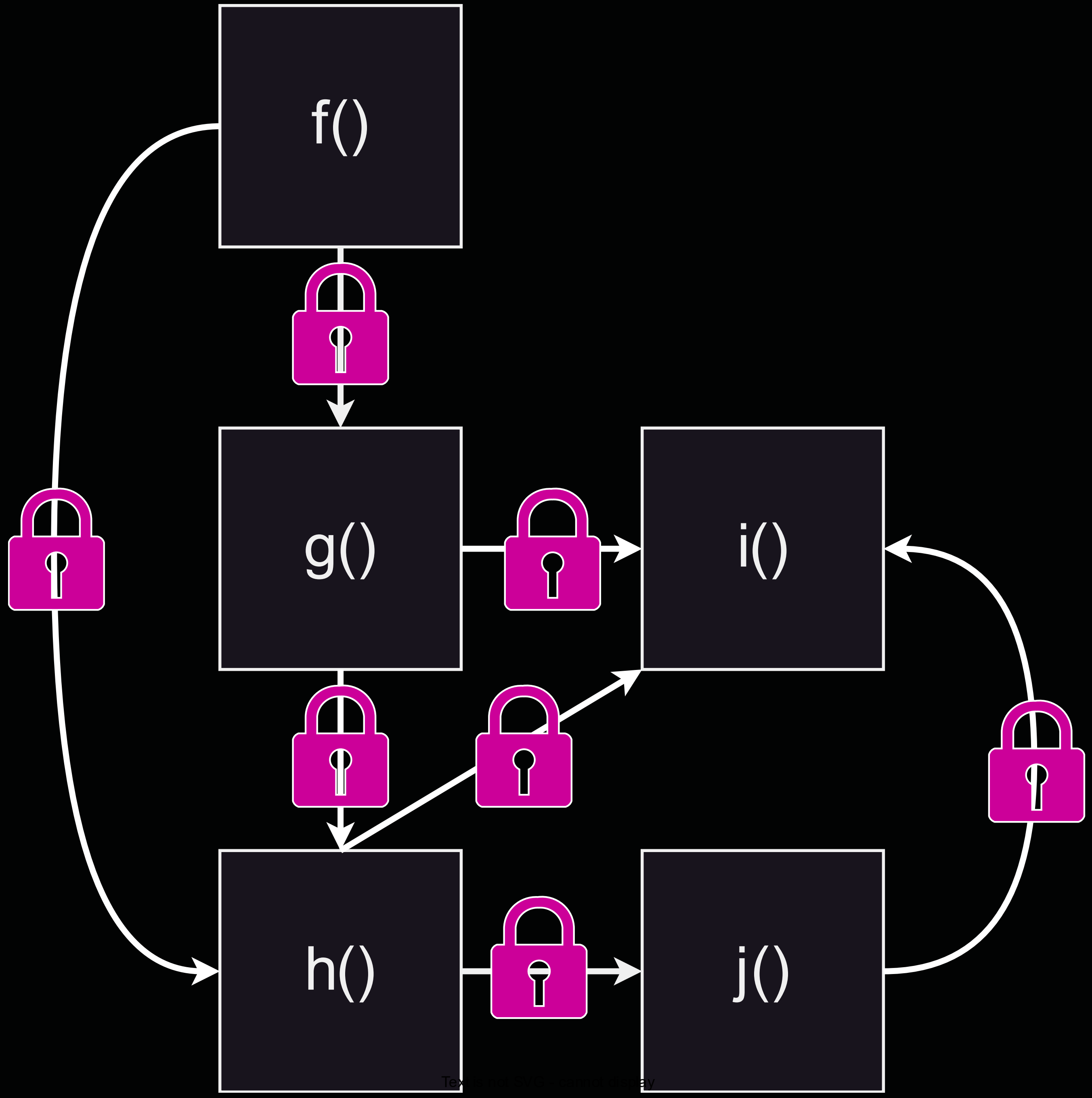


f:
call g()
call h()
....
ret
execve()
g:
call h()
call i()
....
ret
i:
....
ret
h:
call i()
call j()
....
ret
j:
call i()
....
ret

black hat BRIEFINGS

Control Flow Integrity

- "Classic" defenses
  - ASLR, DEP, canaries…
  - Make exploits harder
- Control Flow Integrity
  - Construct Control Flow Graph (CFG)
  - Instrumentation to enforce CFG
  - Code-reuse techniques stopped
    - Sorry, yes, ROP is dead



Who We Are

Marcos Bajo

aka ***h3xduck***

@h3xduck

h3xduck@gmail.com

- PhD Student at CISPA (Germany)
- https://github.com/h3xduck
- Three things I love:
  - Malware
  - Exploits
  - Ducks

CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

Who We Are

Marcos Bajo

aka ***h3xduck***

@h3xduck

h3xduck@gmail.com

- PhD Student at CISPA (Germany)
- https://github.com/h3xduck
- Three things I love:
  - Malware
  - Exploits
  - Ducks

Christian Rossow

@chrossow

rossow@cispa.de

- Faculty at CISPA
- CS Professor at Saarbrücken & Dortmund
- Leader of the *Systems Security Group*

CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

(We do very cool things, reach out!)

What We Will Learn

1. Userspace CFI defenses

How does CFI look like in an everyday system?

What We Will Learn

1. Userspace CFI defenses

How does CFI look like in an everyday system?

2. Bypassing CFI

How can we exploit programs protected by CFI schemes?

What We Will Learn

1. Userspace CFI defenses

How does CFI look like in an everyday system?

2. Bypassing CFI

How can we exploit programs protected by CFI schemes?

3. C++20 Coroutines

Internals and security of C++ coroutines.

What We Will Learn

1. Userspace CFI defenses

How does CFI look like in an everyday system?

2. Bypassing CFI

How can we exploit programs protected by CFI schemes?

3. C++20 Coroutines

Internals and security of C++ coroutines.

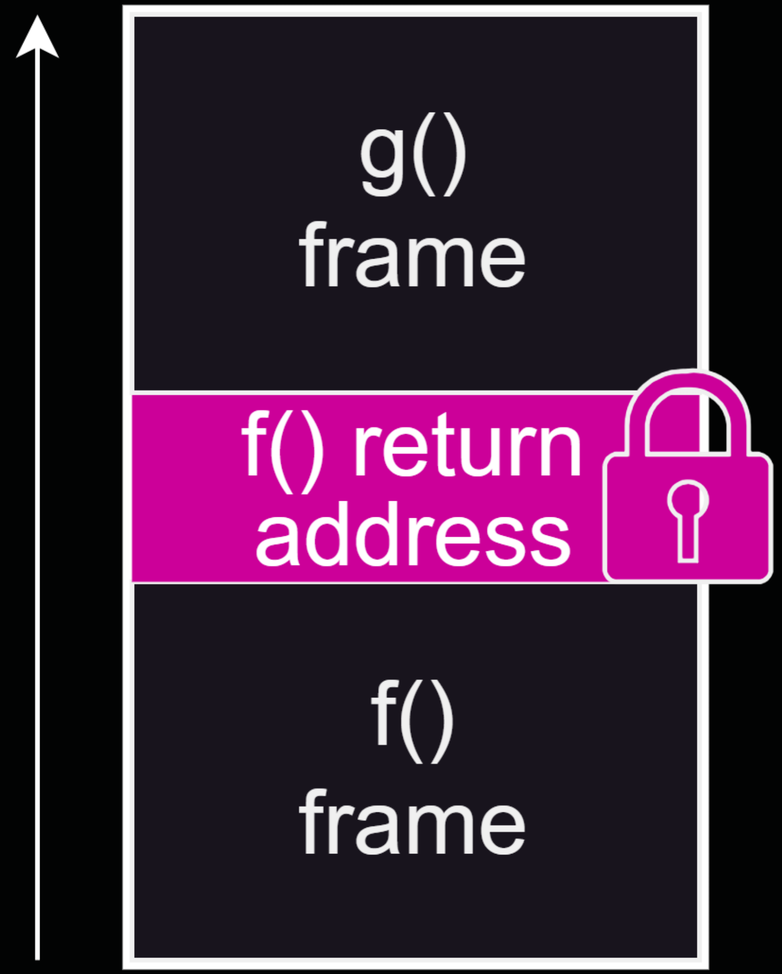
4. Coroutine Frame-Oriented Programming

Using coroutines to bypass CFI.

1 Userspace CFI Defenses

CFI Types

Backward-edge

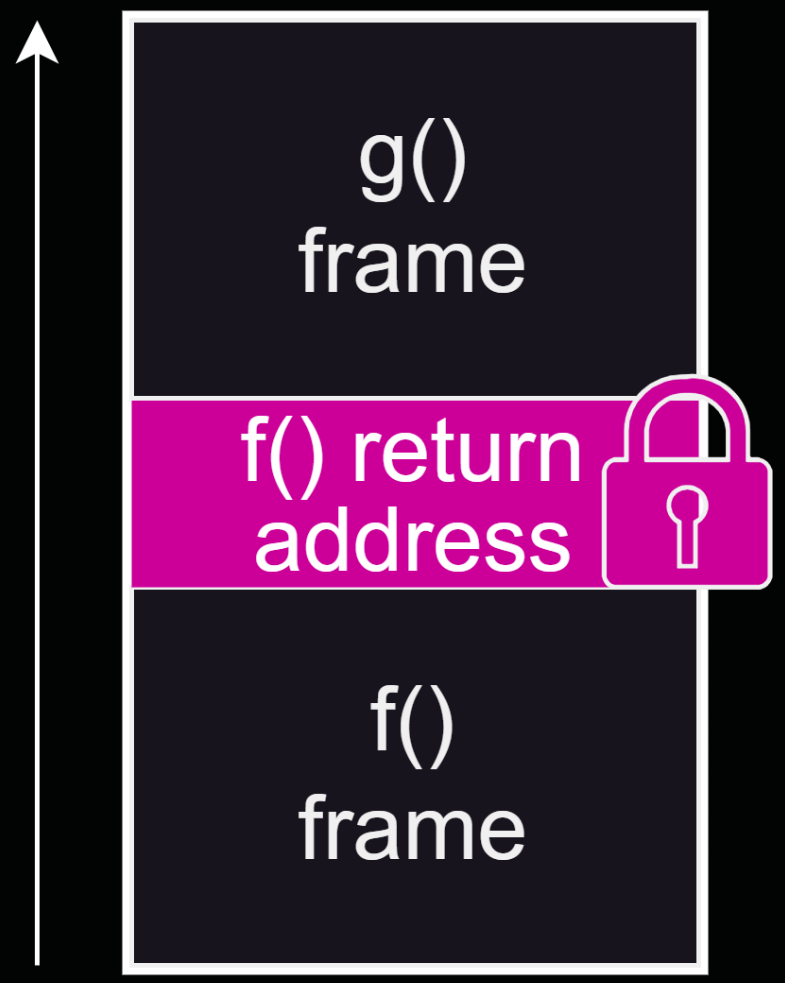



```
void g()
{
    …
}

void f()
{
    g();
}
```

CFI Types

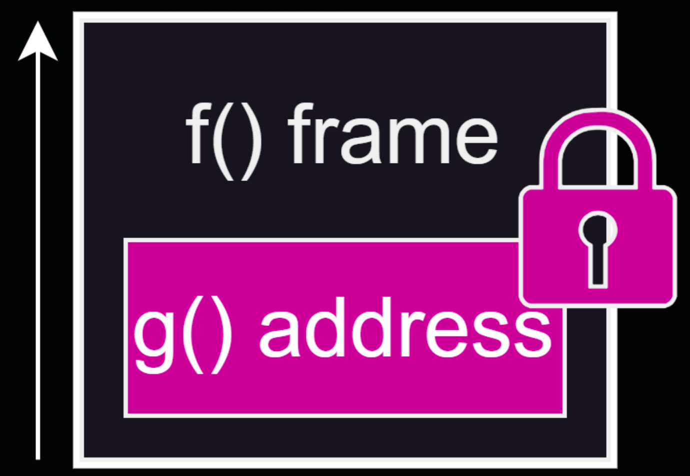
Backward-edge

```
void g()
{
      …
}

void f()
{
      g();
}
```



Forward-edge

```
void f()
{
    void *ptr = &g;
    ptr();
}
```



CFI Types

Coarse-grained CFI | **Fine-grained CFI**

- Security → + Security

+ Performance ← - Performance

Intel CET

- Coarse-grained CFI
- Hardware-assisted
- Two protections in one:
  - Backward-edge: Shadow Stack
  - Forward-edge: Indirect Branch Tracking (IBT)

Intel CET (Shadow Stack)

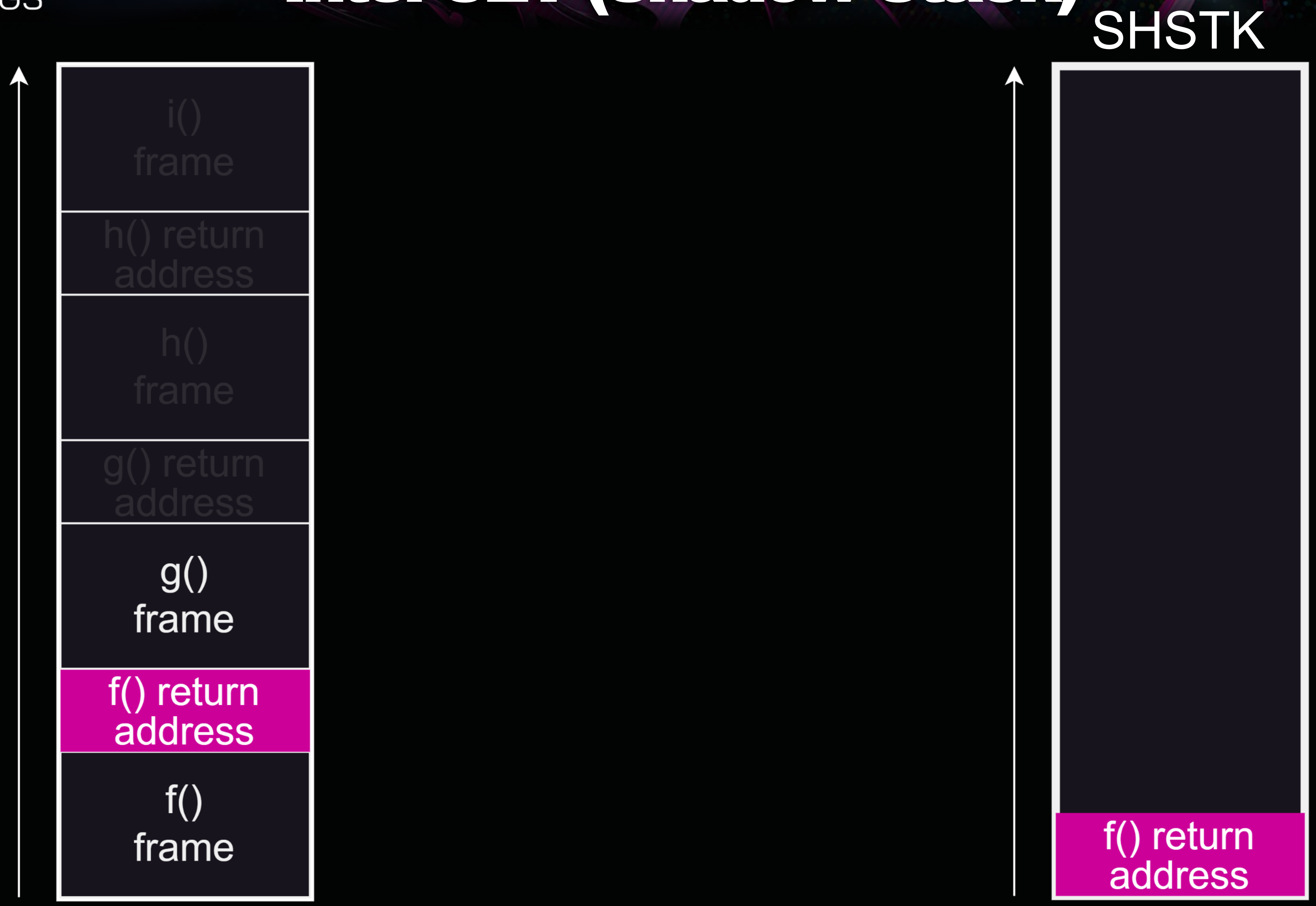
```
void f()
{
    g();
    ret;
}
```

```
void g()
{
    h();
    ret;
}
```

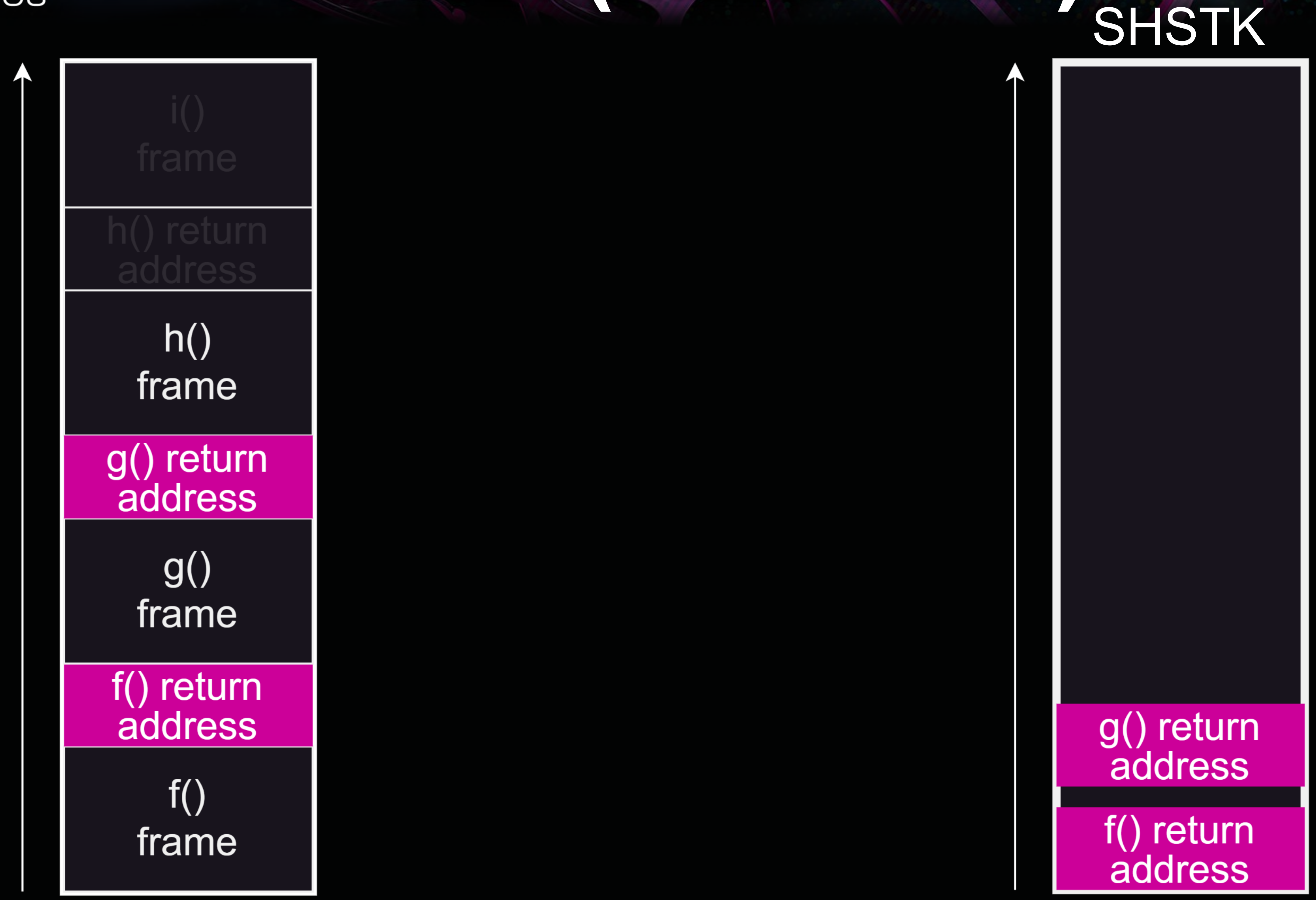
```
void h()
{
    i();
    ret;
}
```

```
void i()
{
    ret;
}
```

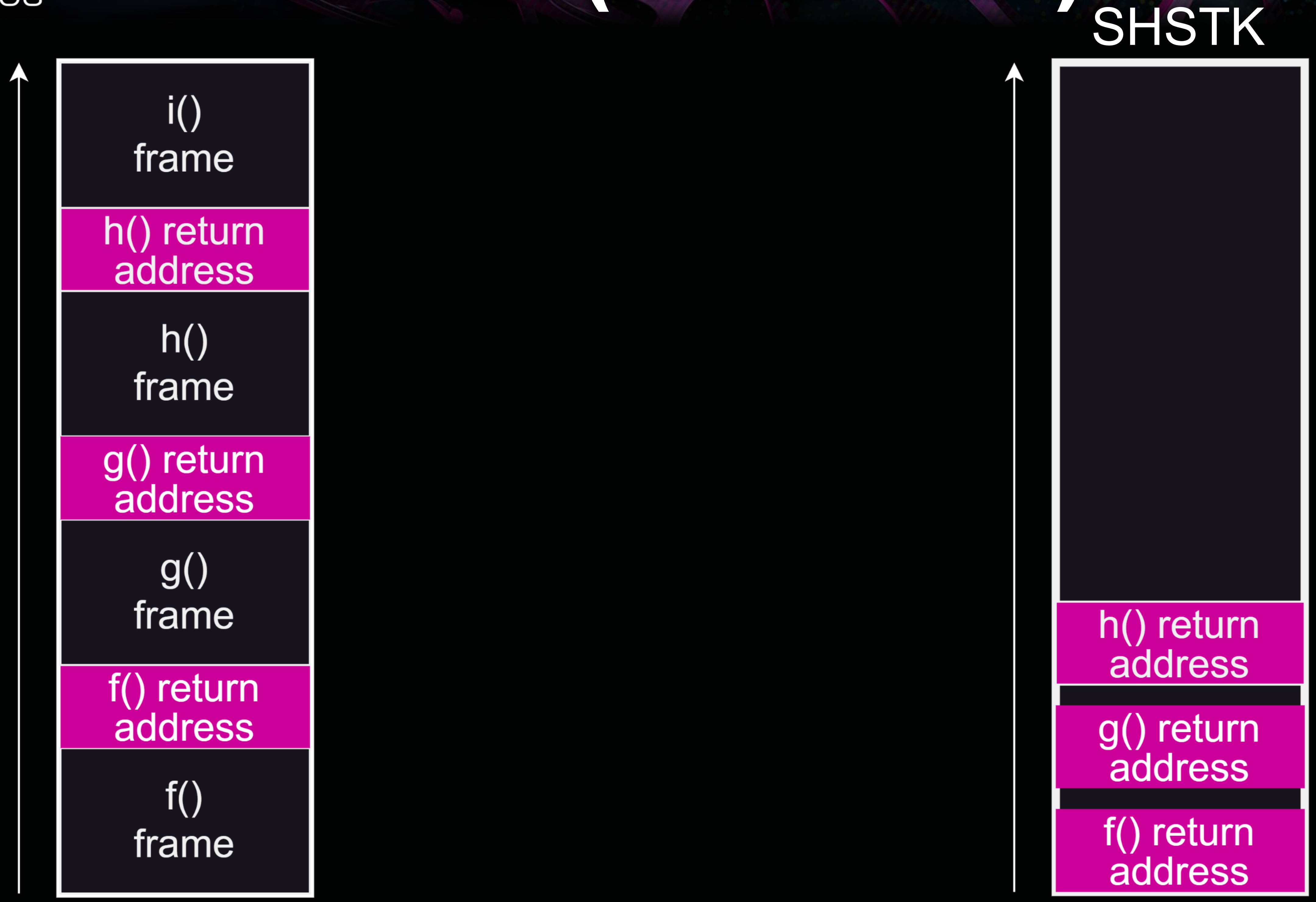
Intel CET (Shadow Stack)



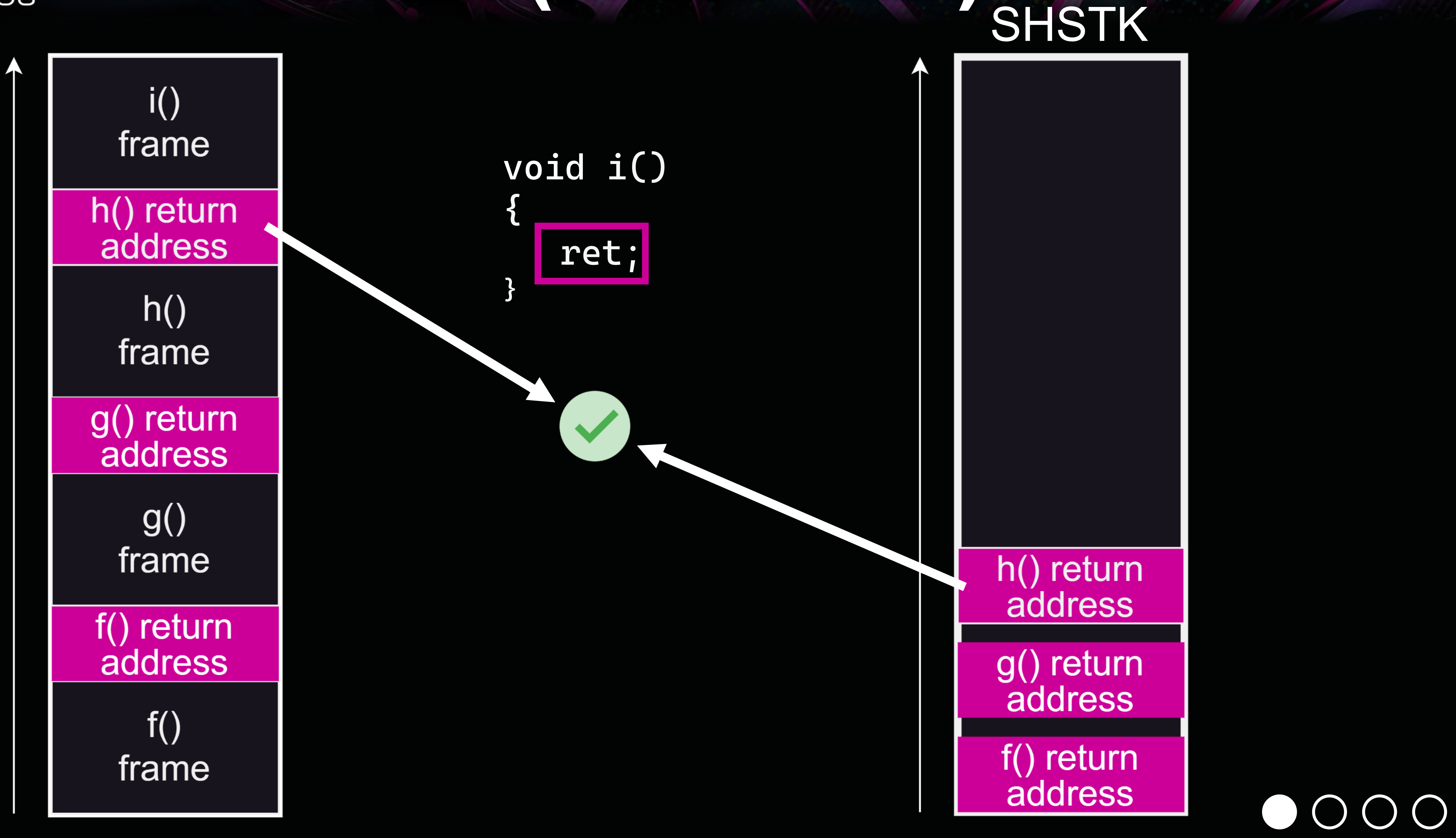
Intel CET (Shadow Stack)



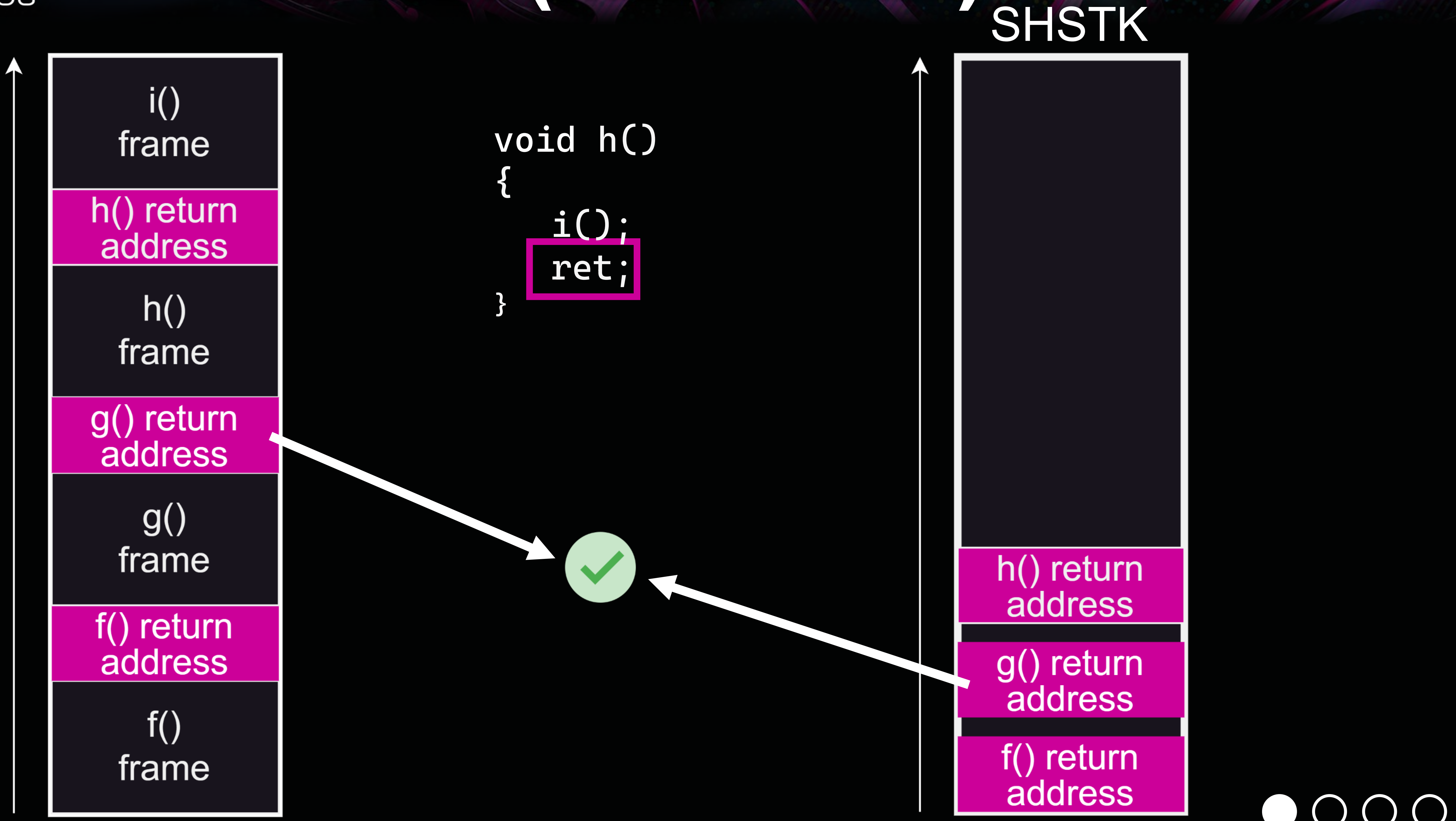
Intel CET (Shadow Stack)



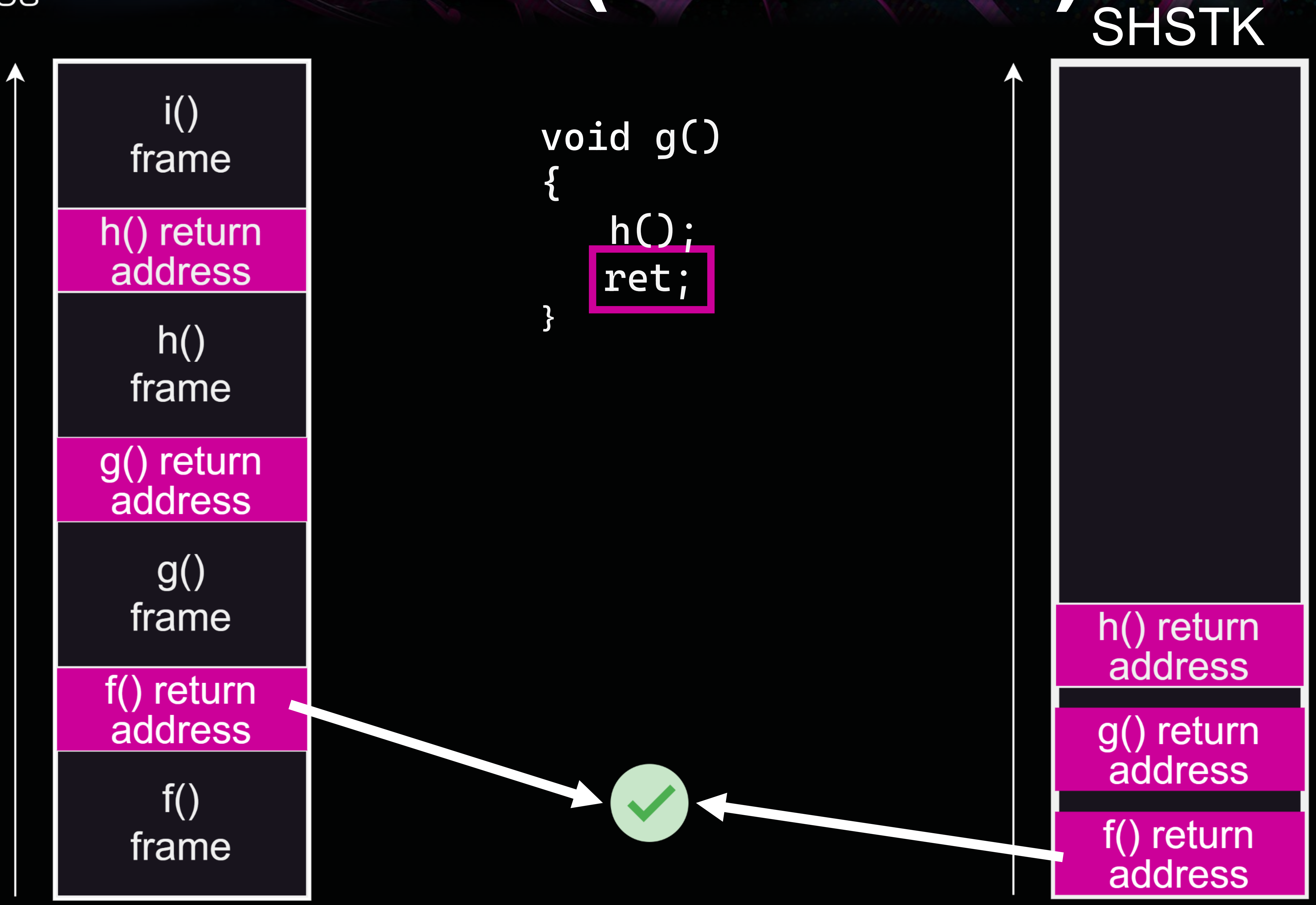
Intel CET (Shadow Stack)



Intel CET (Shadow Stack)

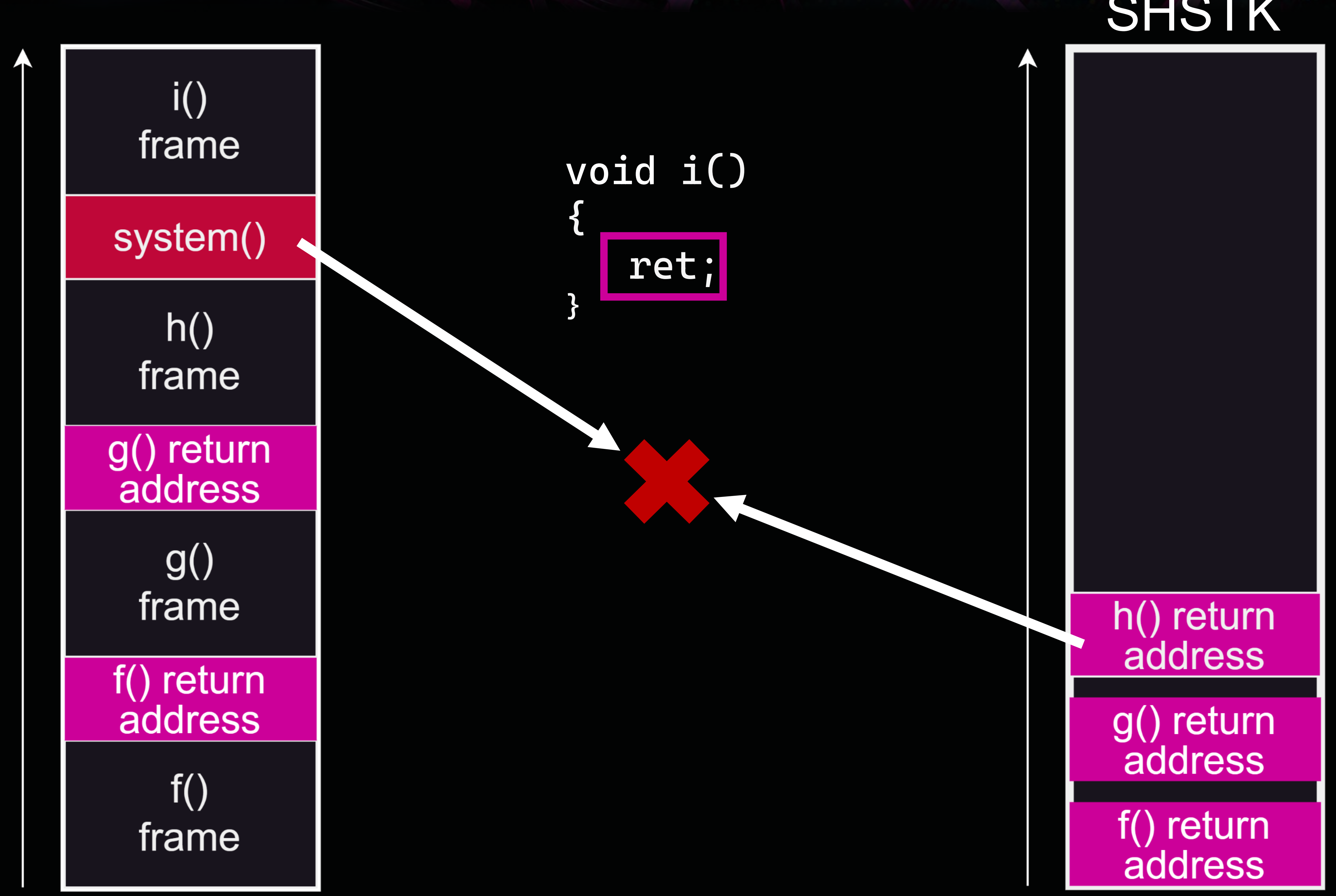


Intel CET (Shadow Stack)




```
void g()
{
    h();
    ret;
}
```

Intel CET (Shadow Stack)



Intel CET (Shadow Stack)

- For an application to be SHSTK enabled:
  - CPU Support: Intel 11th gen (Tiger Lake)
  - Kernel Support: Linux 6.6, Windows 10 19H1
  - Compiler support:
    - GCC 8.1
    - LLVM 11
    - MSVC 16.7
  - Application must be compiled with –fcf-protection=full (Linux) or /CETCOMPAT (Windows)

https://h3xduck.github.io/cfi/2025/06/26/enabling-intel-cet.html

Intel CET (IBT)

```
main:
 push rbp
 mov rbp, rsp
 mov rsi, [rdi+0x8]
 mov rax, [rbx]
 call [rax]
 leave
 ret
```

Intel CET (IBT)

```
main:
 push rbp
 mov rbp, rsp
 mov rsi, [rdi+0x8]
 mov rax, [rbx]
 call [rax]
 leave
 ret
```

```
f1:
 add rdi, 0x8
 mov rax, 0x10
 lea rdx, [rbp]
 add rdi, rax
 mov rax, rdx
 ret
```

Intel CET (IBT)

```
main:
 push rbp
 mov rbp, rsp
 mov rsi, [rdi+0x8]
 mov rax, [rbx]
 call [rax]
 leave
 ret
```

```
f1:
 endbr64
 add rdi, 0x8
 mov rax, 0x10
 lea rdx, [rbp]
 add rdi, rax
 mov rax, rdx
 ret
```

Intel CET (IBT)

- Limited availability
  - Windows: Not implemented
  - Linux: enforcement only in the kernel since 5.18
- Coarse-grained CFI
  - We still can use gadgets starting with endbr64

CFG: Control Flow Guard

- Instrumentation for every indirect call/jmp
- Windows substitute for IBT
- Coarse-grained

call qword ptr [rdi]

⬇

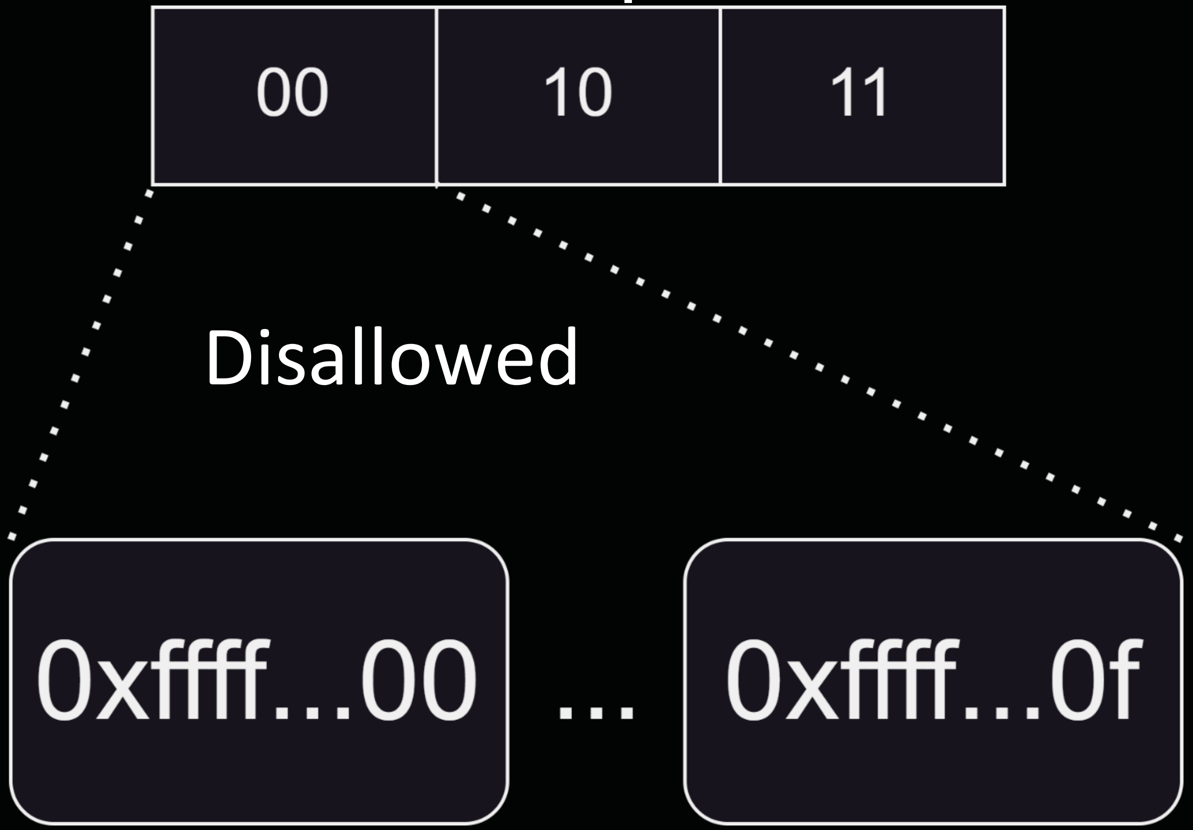
call qword ptr [binary!__guard_dispatch_icall_fptr]

CFG: Control Flow Guard

```
00007ffa`13a451c0 48ba00001a23f57d0000 mov rdx,7DF5231A0000h
00007ffa`13a451ca 488bc1          mov     rax,rcx
00007ffa`13a451cd 48c1e809        shr     rax,9
00007ffa`13a451d1 488b14c2        mov     rdx,qword ptr [rdx+rax*8]
00007ffa`13a451d5 488bc1          mov     rax,rcx
00007ffa`13a451d8 48c1e803        shr     rax,3
00007ffa`13a451dc f6c10f          test    cl,0Fh
00007ffa`13a451df 7507            jne     00007ffa`13a451e8
00007ffa`13a451e1 480fa3c2        bt      rdx,rax
00007ffa`13a451e5 730c            jae     00007ffa`13a451f3
00007ffa`13a451e7 c3              ret
00007ffa`13a451e8 480fbaf000      btr     rax,0
00007ffa`13a451ed 480fa3c2        bt      rdx,rax
00007ffa`13a451f1 730b            jae     00007ffa`13a451fe
00007ffa`13a451f3 4883c801        or      rax,1
00007ffa`13a451f7 480fa3c2        bt      rdx,rax
00007ffa`13a451fb 7301            jae     00007ffa`13a451fe
00007ffa`13a451fd c3              ret
00007ffa`13a451fe 488bc1          mov     rax,rcx
00007ffa`13a45201 4533d2          xor     r10d,r10d
00007ffa`13a45204 eb3a            jmp     00007ffa`13a45240
```

CFG: Control Flow Guard

- *__guard_dispatch_icall_fptr* ensures that the call target is valid
  - If yes, make the call
  - If not, abort the process
- Uses a 2-bit map




```
d0000 mov rdx,7DF5231A0000h
 mov     rax,rcx
 shr     rax,9
 mov     rdx,qword ptr [rdx+rax*8]
 mov     rax,rcx
 shr     rax,3
 test    cl,0Fh
 jne     00007ffa`13a451e8
 bt      rdx,rax
 jae     00007ffa`13a451f3
 ret
 btr     rax,0
 bt      rdx,rax
 jae     00007ffa`13a451fe
 or      rax,1
 bt      rdx,rax
 jae     00007ffa`13a451fe
 ret
 mov     rax,rcx
 xor     r10d,r10d
 jmp     00007ffa`13a45240
```

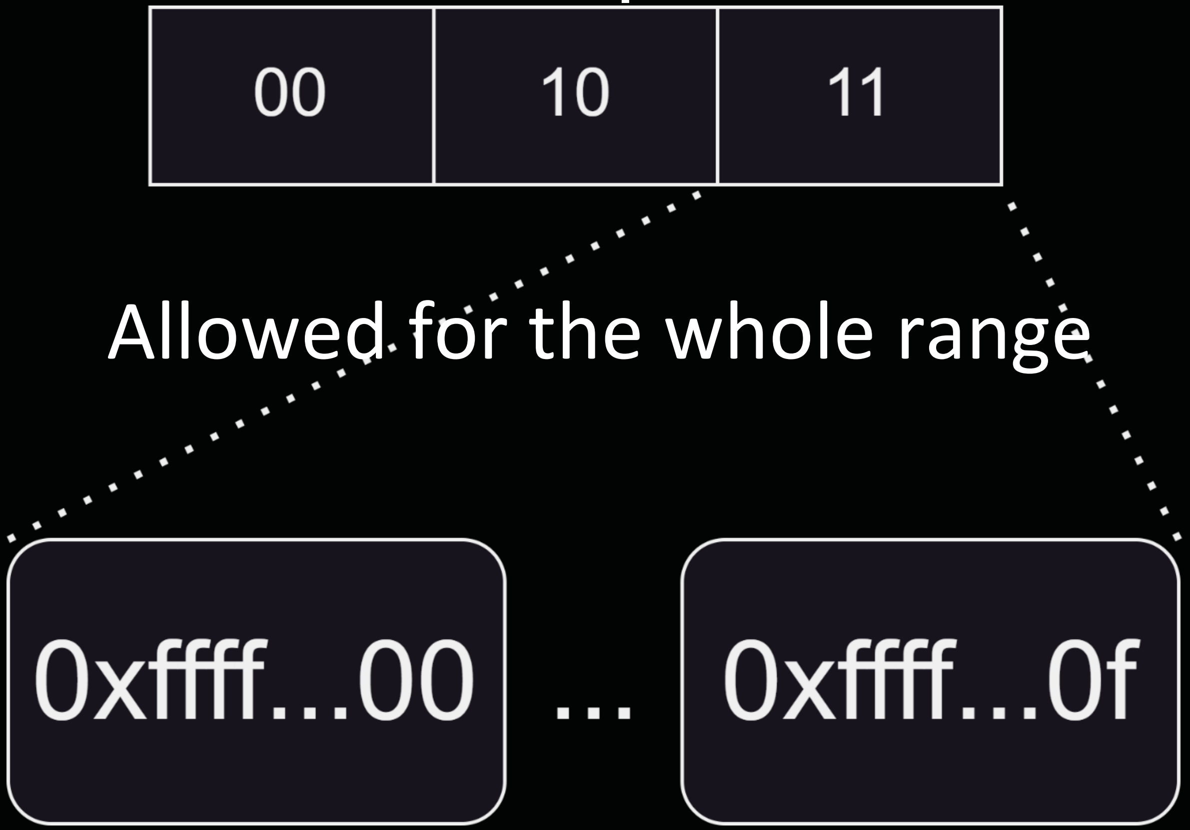
CFG: Control Flow Guard

- *__guard_dispatch_icall_fptr* ensures that the call target is valid
  - If yes, make the call
  - If not, abort the process
- Uses a 2-bit map

```
'd0000 mov rdx,7DF5231A0000h
 mov     rax,rcx
 shr     rax,9
 mov     rdx,qword ptr [rdx+rax*8]
 mov     rax,rcx
 shr     rax,3
 test    cl,0Fh
 jne     00007ffa`13a451e8
 bt      rdx,rax
 jae     00007ffa`13a451f3
 ret
 btr     rax,0
 bt      rdx,rax
 jae     00007ffa`13a451fe
 or      rax,1
 bt      rdx,rax
 jae     00007ffa`13a451fe
 ret
 mov     rax,rcx
 xor     r10d,r10d
 jmp     00007ffa`13a45240
```

CFG: Control Flow Guard

- *__guard_dispatch_icall_fptr* ensures that the call target is valid
  - If yes, make the call
  - If not, abort the process
- Uses a 2-bit map




```
d0000 mov rdx,7DF5231A0000h
 mov     rax,rcx
 shr     rax,9
 mov     rdx,qword ptr [rdx+rax*8]
 mov     rax,rcx
 shr     rax,3
 test    cl,0Fh
 jne     00007ffa`13a451e8
 bt      rdx,rax
 jae     00007ffa`13a451f3
 ret
 btr     rax,0
 bt      rdx,rax
 jae     00007ffa`13a451fe
 or      rax,1
 bt      rdx,rax
 jae     00007ffa`13a451fe
 ret
 mov     rax,rcx
 xor     r10d,r10d
 jmp     00007ffa`13a45240
```

CFG: Control Flow Guard

- *__guard_dispatch_icall_fptr* ensures that the call target is valid
  - If yes, make the call
  - If not, abort the process

```
00007ff5`0b62b420  00000000  00000010  00000000  00000000  00100000  00000000
```

```
KERNEL32!WinExec:
 mov      rax,rsp
 mov      qword ptr [rax+10h],rbx
 mov      qword ptr [rax+18h],rsi
 mov      qword ptr [rax+20h],rdi
 push     rbp
 lea      rbp,[rax-38h]
```

LLVM CFI (cfi-icall)

- Fine(r)-grade CFI: label based
- Flag *–fsanitize=cfi-icall* in Clang/LLVM
- Each function is assigned a dynamic type

```
ind_func();
```

```
int puts(const char* s)
int close(int fd)
int kill(pid_t pid, int sig)
int system(const char* c)
```

LLVM CFI (cfi-icall)

- Fine(r)-grade CFI: label based
- Flag *–fsanitize=cfi-icall* in Clang/LLVM
- Each function is assigned a dynamic type

```
ind_func = &close;
ind_func();
```

```
int puts(const char* s)
int close(int fd)
int kill(pid_t pid, int sig)
int system(const char* c)
```

LLVM CFI (cfi-icall)

- Fine(r)-grade CFI: label based
- Flag *–fsanitize=cfi-icall* in Clang/LLVM
- Each function is assigned a dynamic type

```
if()
  ind_func = &close;
else
  ind_func = &kill;
ind_func();
```

```
int puts(const char* s)
int close(int fd)
int kill(pid_t pid, int sig)
int system(const char* c)
```

LLVM CFI (cfi-icall)

```
0x1e080 <f1>:        jmp    0x1d310 <f1>
0x1e085 <f1+5>:      int3
0x1e086 <f1+6>:      int3
0x1e087 <f1+7>:      int3
0x1e088 <f3>:        jmp    0x1d330 <f3>
0x1e08d <f3+5>:      int3
0x1e08e <f3+6>:      int3
0x1e08f <f3+7>:      int3
0x1e090 <main>:      jmp    0x1d340 <main>
0x1e095 <main+5>:    int3
0x1e096 <main+6>:    int3
0x1e097 <main+7>:    int3
```

black hat BRIEFINGS

LLVM CFI (cfi-icall)

```
0x000000000001d3a8 <+104>:   lea    rax,[rip+0xcd1]        # 0x1e080 <f1>
0x000000000001d3af <+111>:   mov    rcx,rbx
0x000000000001d3b2 <+114>:   sub    rcx,rax
0x000000000001d3b5 <+117>:   rol    rcx,0x3d
0x000000000001d3b9 <+121>:   cmp    rcx,0x2
0x000000000001d3bd <+125>:   jae    0x1d3cb <main+139>
0x000000000001d3bf <+127>:   xor    edi,edi
0x000000000001d3c1 <+129>:   call   rbx
0x000000000001d3c3 <+131>:   xor    eax,eax
0x000000000001d3c5 <+133>:   add    rsp,0x10
0x000000000001d3c9 <+137>:   pop    rbx
0x000000000001d3ca <+138>:   ret
0x000000000001d3cb <+139>:   movabs rdi,0x4c550309df1cf4c1
0x000000000001d3d5 <+149>:   mov    rsi,rbx
0x000000000001d3d8 <+152>:   call   0x1cd60 <__cfi_slowpath>
```

LLVM CFI (cfi-icall)

```
0x000000000001d3a8 <+104>:   lea    rax,[rip+0xcd1]        # 0x1e080 <f1>
0x000000000001d3af <+111>:   mov    rcx,rbx
0x000000000001d3b2 <+114>:   sub    rcx,rax
0x000000000001d3b5 <+117>:   rol    rcx,0x3d
0x000000000001d3b9 <+121>:   cmp    rcx,0x2
0x000000000001d3bd <+125>:   jae    0x1d3cb <main+139>
0x000000000001d3bf <+127>:   xor    edi,edi
0x000000000001d3c1 <+129>:   call   rbx
0x000000000001d3c3 <+131>:   xor    eax,eax
0x000000000001d3c5 <+133>:   add    rsp,0x10
0x000000000001d3c9 <+137>:   pop    rbx
0x000000000001d3ca <+138>:   ret
0x000000000001d3cb <+139>:   movabs rdi,0x4c550309df1cf4c1
0x000000000001d3d5 <+149>:   mov    rsi,rbx
0x000000000001d3d8 <+152>:   call   0x1cd60 <__cfi_slowpath>
```

LLVM CFI (cfi-icall)

```
0x000000000001d3a8 <+104>:   lea     rax,[rip+0xcd1]        # 0x1e080 <f1>
0x000000000001d3af <+111>:   mov     rcx,rbx
0x000000000001d3b2 <+114>:   sub     rcx,rax
0x000000000001d3b5 <+117>:   rol     rcx,0x3d
0x000000000001d3b9 <+121>:   cmp     rcx,0x2
0x000000000001d3bd <+125>:   jae     0x1d3cb <main+139>
0x000000000001d3bf <+127>:   xor     edi,edi
0x000000000001d3c1 <+129>:   call    rbx
0x000000000001d3c3 <+131>:   xor     eax,eax
0x000000000001d3c5 <+133>:   add     rsp,0x10
0x000000000001d3c9 <+137>:   pop     rbx
0x000000000001d3ca <+138>:   ret
0x000000000001d3cb <+139>:   movabs  rdi,0x4c550309df1cf4c1
0x000000000001d3d5 <+149>:   mov     rsi,rbx
0x000000000001d3d8 <+152>:   call    0x1cd60 <__cfi_slowpath>
```

LLVM CFI (cfi-icall)

- Fine(r)-grade CFI: label based
- Flag *–fsanitize=cfi-icall* in Clang/LLVM
- Each function is assigned a dynamic type

```
ind_func = &puts;
ind_func();
```

```
int puts(const char* s)
int close(int fd)
int kill(pid_t pid, int sig)
int system(const char* c)
```

2 Bypassing CFI Defenses

Approach

- We used to…
  - Return to arbitrary gadgets: ROP
  - Jump to arbitrary gadgets: JOP

Approach

- We used to…
  - Return to arbitrary gadgets: ~~ROP~~ Backward-edge CFI
  - Jump to arbitrary gadgets: ~~JOP~~ Forward-edge CFI
- How is a new exploitation technique built?

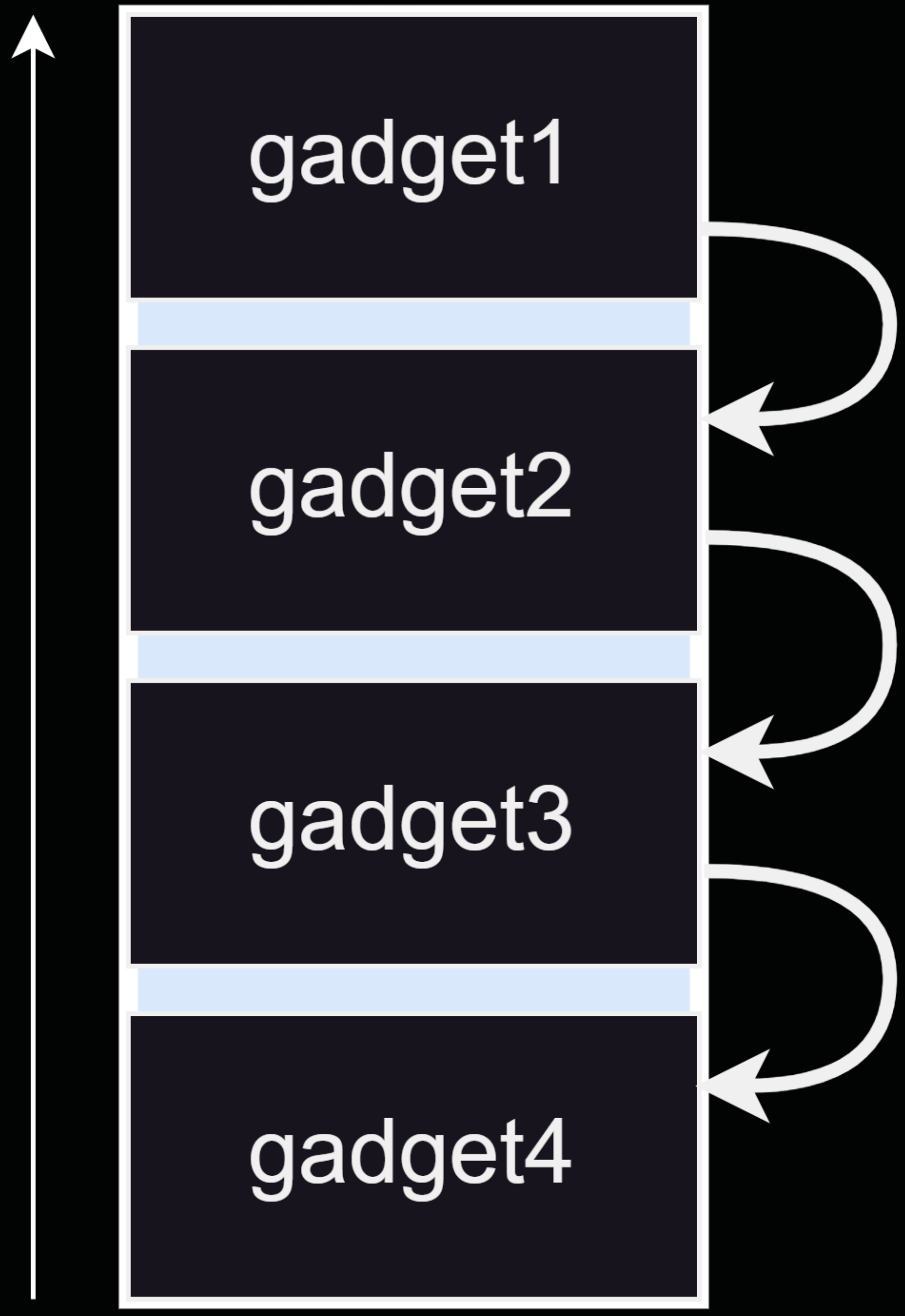
Approach

Gadgets

```
…
inc rax
ret

…
pop rdi
ret

…
pop rsi
ret

…
mov rdx, rcx
ret
```

Approach

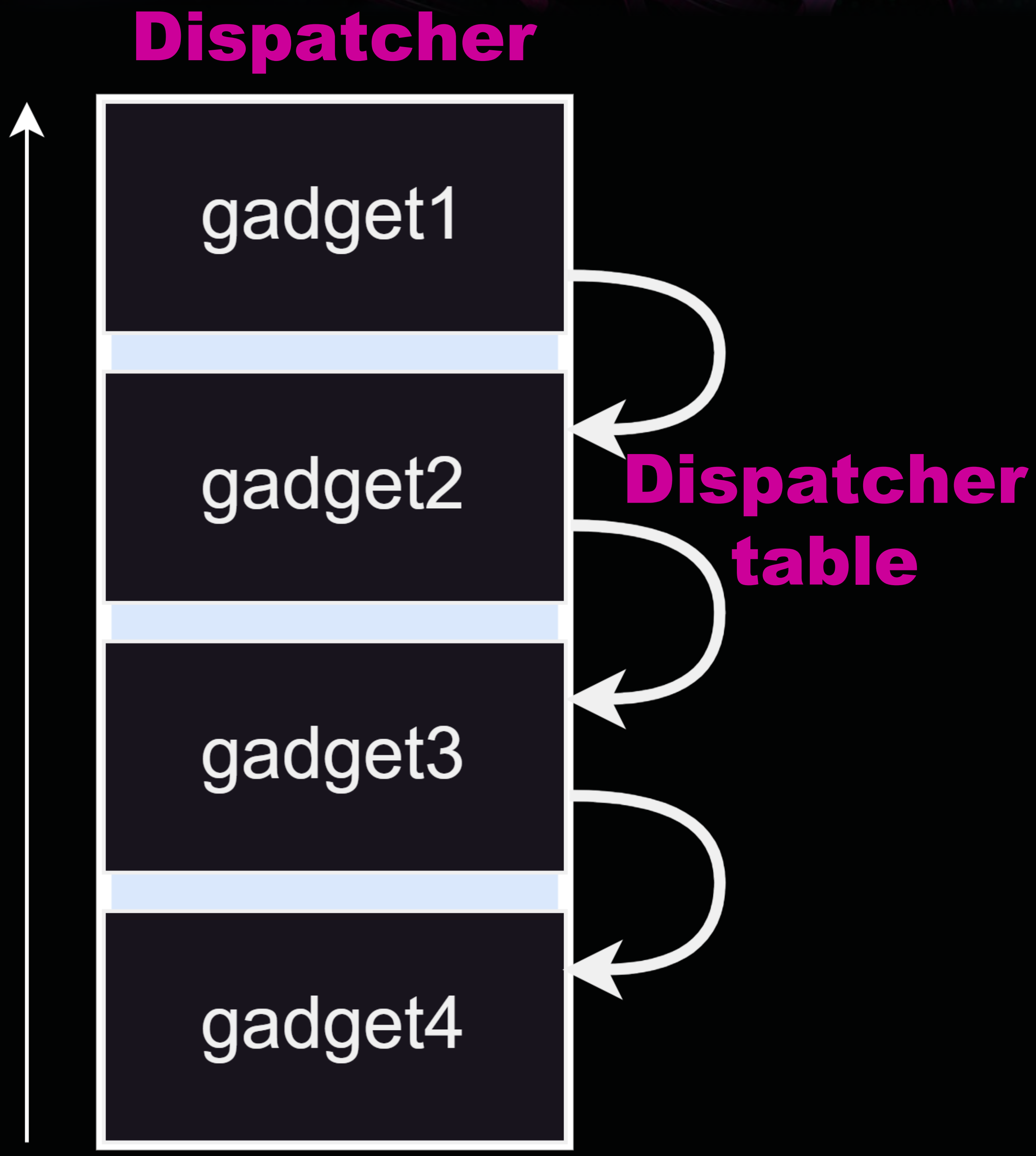
Dispatcher



Gadgets

```
…
inc rax
ret

…
pop rdi
ret

…
pop rsi
ret

…
mov rdx, rcx
ret
```

Approach



Gadgets

```
…
inc rax
ret

…
pop rdi
ret

…
pop rsi
ret

…
mov rdx, rcx
ret
```

Approach

- Every exploitation technique must have some form of…
  - Dispatcher: a loop that iterates over gadgets
  - Dispatcher table: memory containing the gadgets to call
  - Gadgets: code to be executed e.g., set registers, etc

COOP

- Counterfeit Object-Oriented Programming (COOP)
  - Leverage virtual pointers (VPs) in C++
  - Dispatcher: a loop that calls VPs
  - Dispatcher table: overwritten VPs
  - Gagdets: (complete) virtual functions

```
class Student {
 virtual void study();
}
class Course {
   Student **students;
   virtual func(){
     for(;;){
        students[i]->study();
     }
   }
}
```

How to bypass CFI

- Bypassing coarse-grained CFI (CET and CFG) requires
  - Not tampering with return addresses
  - Tampering with code pointers in writable memory, but only pointing them to the beginning of functions
- Bypassing finer-grade CFI (LLVM CFI) requires
  - Finding some useful collision (rare)
  - Otherwise, every pointer is instrumented

How to bypass CFI

- Bypassing coarse-grained CFI (CET and CFG) requires
  - Not tampering with return addresses
  - Tampering with code pointers in writable memory, but only pointing them to the beginning of functions
- Bypassing finer-grade CFI (LLVM CFI) requires
  - Finding some useful collision (rare)
  - Otherwise, every pointer is instrumented

Is this really true though?

3 C++20 Coroutines

What is a Coroutine

- TL;DR: A coroutine is a function that can suspend and resume

```
void foo()
{
   bar();

   bar();
}
```

```
void bar()
{

    …

}
```

What is a Coroutine

- TL;DR: A coroutine is a function that can suspend and resume

Function

```
void foo()
{
   bar();

   bar();
}
```

```
void bar()
{

   …

}
```

What is a Coroutine

- TL;DR: A coroutine is a function that can suspend and resume

Function

```
void foo()
{
   bar();

   bar();
}
```

```
void bar()
{

   …

}
```

What is a Coroutine

- TL;DR: A coroutine is a function that can suspend and resume

```
void foo()
{
   coro();

   coro();
}
```

Coroutine

```
void coro()
{

  <suspend>;

}
```

Suspension point (SP)

black hat BRIEFINGS

What is a Coroutine

- TL;DR: A coroutine is a function that can **suspend** and **resume**

```
void foo()
{
   coro();

   coro();
}
```

Coroutine

```
void coro()
{

   <suspend>;

}
```

The Coroutine (task) Object

- Every coroutine returns a *task* object, that describes its state

```
void foo()
{
   task t = coro();
}
```

```
task coro()
{
   ...
   <suspend>;
   ...
}
```

Coroutine Handle

- The coroutine handle refers to an instance of a coroutine

```
void foo()
{
   task t1 = coro();
   coroutine_handle<> h1 = t1.handle;
}
```

```
task coro()
{
   ...
   <suspend>;
   ...
}
```

Coroutine Handle

- The coroutine handle refers to an instance of a coroutine

```
void foo()
{
   coroutine_handle<> h1 = coro().handle;
   coroutine_handle<> h2 = coro().handle;
}
```

```
task coro()
{
   ...
   <suspend>;
   ...
}
```

Coroutine Handle

- The coroutine handle allows *resuming* & *destroying* a coroutine

```
void foo()
{
   coroutine_handle<> h1 = coro().handle;
   coroutine_handle<> h2 = coro().handle;

   h1.resume();
}
```

h1

```
task coro()
{
   ...
   <suspend>;
   ...
}
```

Coroutine Handle

- The coroutine handle allows *resuming* & *destroying* a coroutine

```
void foo()
{
   coroutine_handle<> h1 = coro().handle;
   coroutine_handle<> h2 = coro().handle;

   h1.resume();
   h2.resume();
}
```

h1&h2 →

```
task coro()
{
  ...
  <suspend>;
  ...
}
```

Coroutine Handle

- The coroutine handle allows *resuming* & *destroying* a coroutine

```
void foo()
{
   coroutine_handle<> h1 = coro().handle;
   coroutine_handle<> h2 = coro().handle;

   h1.destroy();
}
```

```
task coro()
{
   ...
   <suspend>;
   ...
}
```

What is a Coroutine

- The compiler treats a function as a coroutine whenever one of the three coroutine keywords appear:

```
void coro()
{

   <suspend>;

}
```

co_await

co_yield

co_return

What is a Coroutine

- *co_yield* suspends and returns a value

```
void main()
{
   handle coro = fib().handle;

   coro.resume();
   coro.resume();
   coro.resume();
}
```

returns 1

```
task fib()
{
   int a=0, b=1;
   for(;;){
     co_yield a+b;
     int temp = b;
     b = a+b;
     a = temp;
   }
}
```

What is a Coroutine

- *co_yield* suspends and returns a value

```
void main()
{
   handle coro = fib().handle;

   coro.resume();
   coro.resume();
   coro.resume();
}
```

returns 2

```
task fib()
{
   int a=0, b=1;
   for(;;){
      co_yield a+b;
      int temp = b;
      b = a+b;
      a = temp;
   }
}
```

What is a Coroutine

- *co_yield* suspends and returns a value

```
void main()
{
   handle coro = fib().handle;

   coro.resume();
   coro.resume();
   coro.resume();
}
```

```
task fib()
{
   int a=0, b=1;
   for(;;){
      co_yield a+b;
      int temp = b;
      b = a+b;
      a = temp;
   }
}
```

returns 3

What is a Coroutine

- *co_return* suspends and returns a value

```
void main()
{
   handle coro = fib().handle;

   coro.resume();
   coro.resume();
   coro.resume();
}
```

returns for the final time

```
task fib()
{
   int a=0, b=1;
   for(int i=0; i<10; i++){
      co_yield a+b;
      int temp = b;
      b = a+b;
      a = temp;
   }
   co_return a+b;
}
```

Returning a value

- Coroutines return values by storing them in the promise object

```
void main()
{
   handle coro = coro().handle;

   coro.resume();

   int res = coro.promise().value;
}
```

```
task coro()
{
   co_return 42;
}
```

(Basic) Coroutine Lifetime

Creation stub

```
void foo()
{
   coroutine_handle<> h = coro().handle;
   h.resume();
   h.destroy();
}
```

(Basic) Coroutine Lifetime

Creation stub

Resume stub

```
void foo()
{
   coroutine_handle<> h = coro().handle;
   h.resume();
   h.destroy();
}
```

(Basic) Coroutine Lifetime

Creation stub

Resume stub

Destroy stub

```
void foo()
{
   coroutine_handle<> h = coro().handle;
   h.resume();
   h.destroy();
}
```

(Basic) Coroutine Lifetime

```
task coro()
{
   co_return 42;
}
```

```
task
{

}
```

```
void foo()
{
   handle h = coro().h;
   h.resume();
   h.destroy();
}
```

(Basic) Coroutine Lifetime

```
task coro()
{
   co_return 42;
}
```

```
task
{
      handle h;
      struct promise_type{};
}
```

```
void foo()
{
   handle h = coro().h;
   h.resume();
   h.destroy();
}
```

(Basic) Coroutine Lifetime

```
task coro()
{
   co_return 42;
}


void foo()
{
   handle h = coro().h;
   h.resume();
   h.destroy();
}
```

```
task
{
      handle h;
      struct promise_type
      {
         int return_value;
         suspend_always initial_suspend();
         suspend_always final_suspend();
      };
}
```

(Basic) Coroutine Lifetime

```
task coro()
{
   co_return 42;
}


void foo()
{
   handle h = coro().h;
   h.resume();
   h.destroy();
}
```

```
task
{
    handle h;
    struct promise_type
    {
       int return_value;
       suspend_always initial_suspend();
       suspend_always final_suspend();
    };
}
```

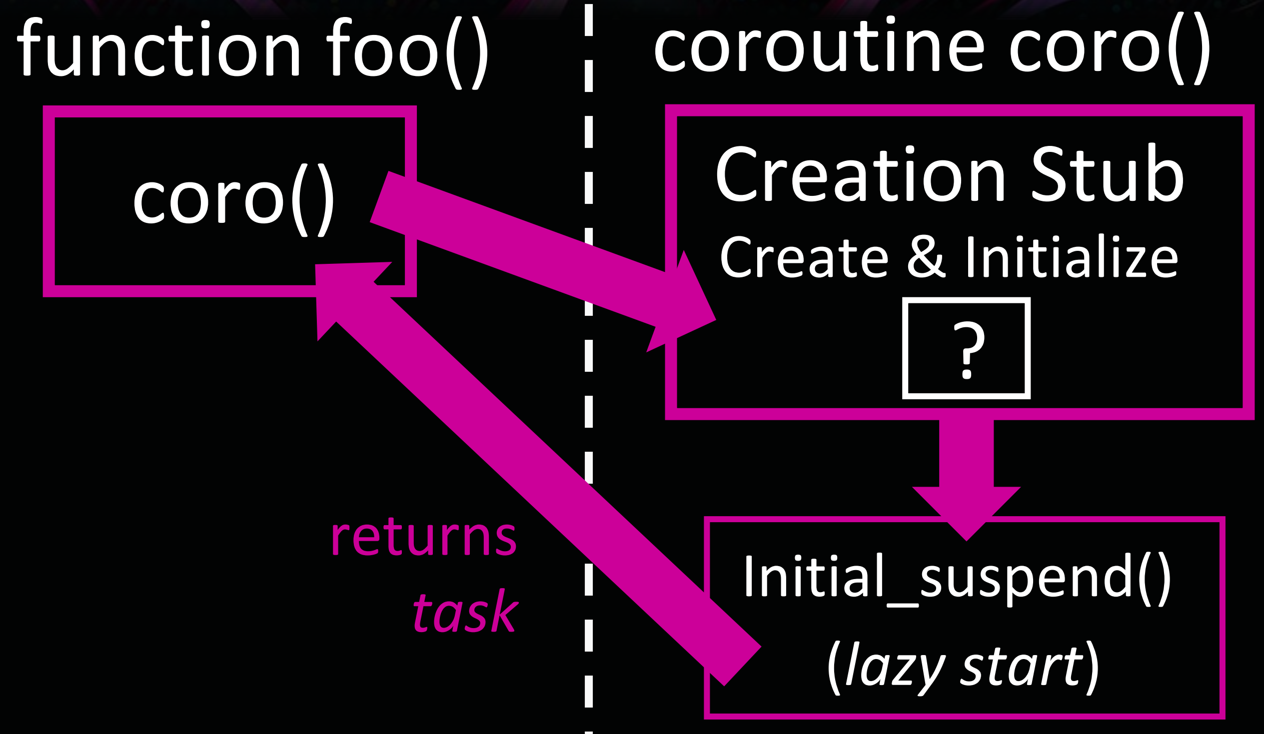
Creation stub

Resume stub

Destroy stub

(Basic) Coroutine Lifetime

```
task
{
      handle h;
      struct promise_type
      {
         int return_value;
         suspend_always initial_suspend();
         suspend_always final_suspend();
      };
}

void foo()                  task coro()
{                           {
   handle h = coro().h;        co_return 42;
   h.resume();              }
   h.destroy();
}
```

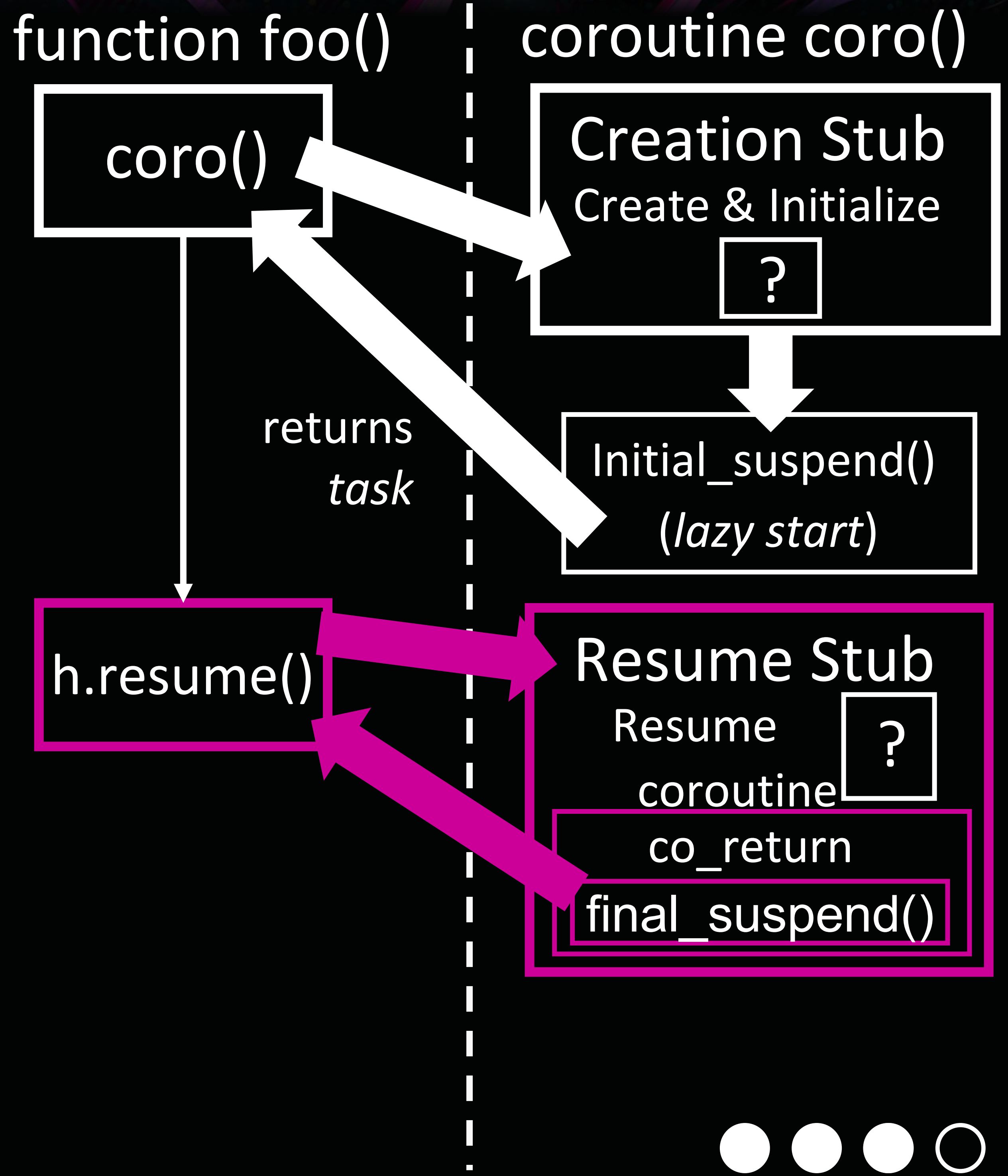


(Basic) Coroutine Lifetime

```
task
{
        handle h;
        struct promise_type
        {
            int return_value;
            suspend_always initial_suspend();
            suspend_always final_suspend();
        };
}
```

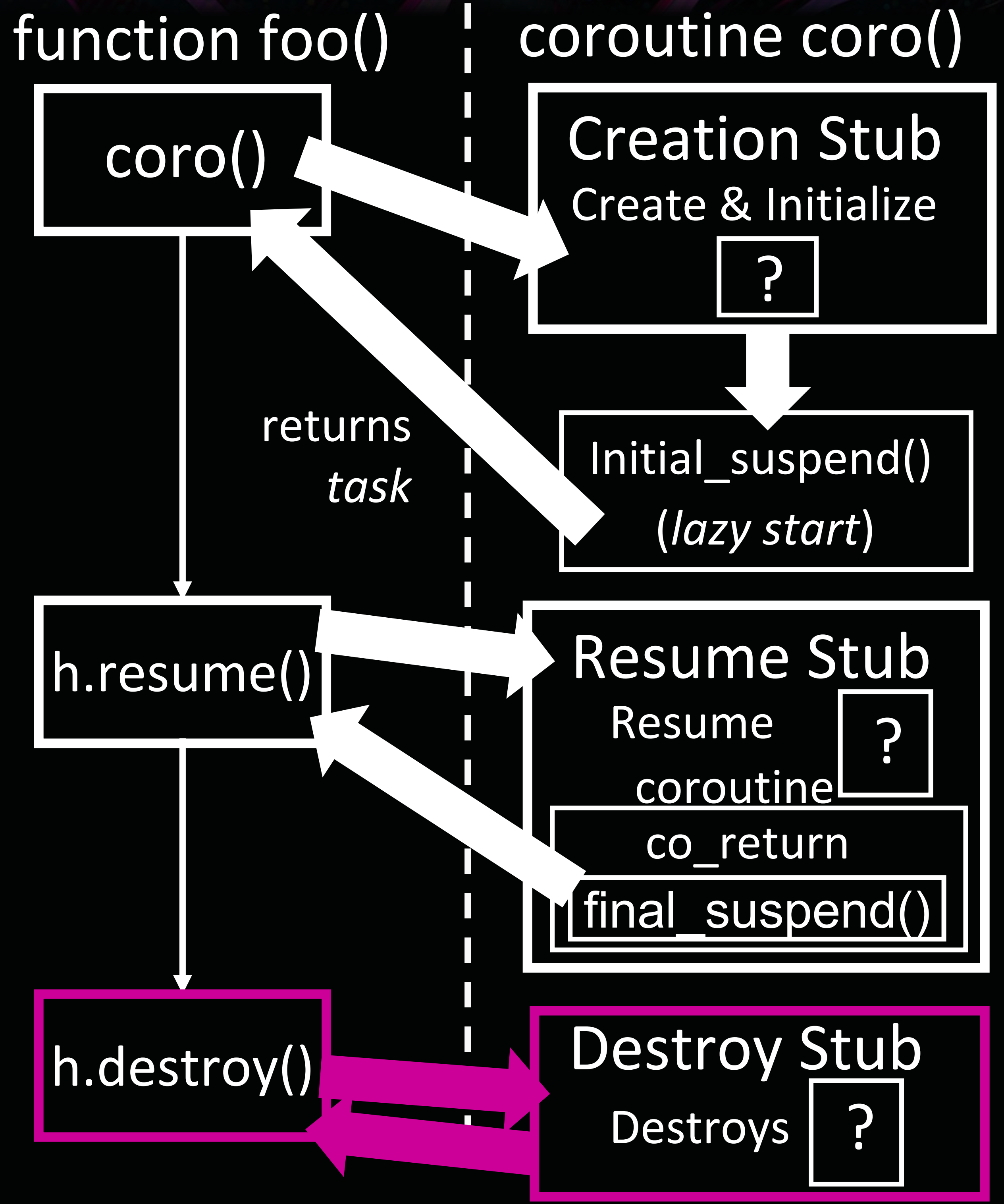
```
void foo()
{
    handle h = coro().h;
    h.resume();
    h.destroy();
}
```

```
task coro()
{
    co_return 42;
}
```



(Basic) Coroutine Lifetime

```
task
{
        handle h;
        struct promise_type
        {
            int return_value;
            suspend_always initial_suspend();
            suspend_always final_suspend();
        };
}

void foo()                          task coro()
{                                   {
    handle h = coro().h;                co_return 42;
    h.resume();                     }
    h.destroy();
}
```



The Coroutine Frame

- Coroutines in C++ are stackless
  - Can only be suspended from the coroutine itself (you cannot call another function and suspend from there)
  - Other stackless coroutines: C#, JS, Python, Rust, Swift

The Coroutine Frame

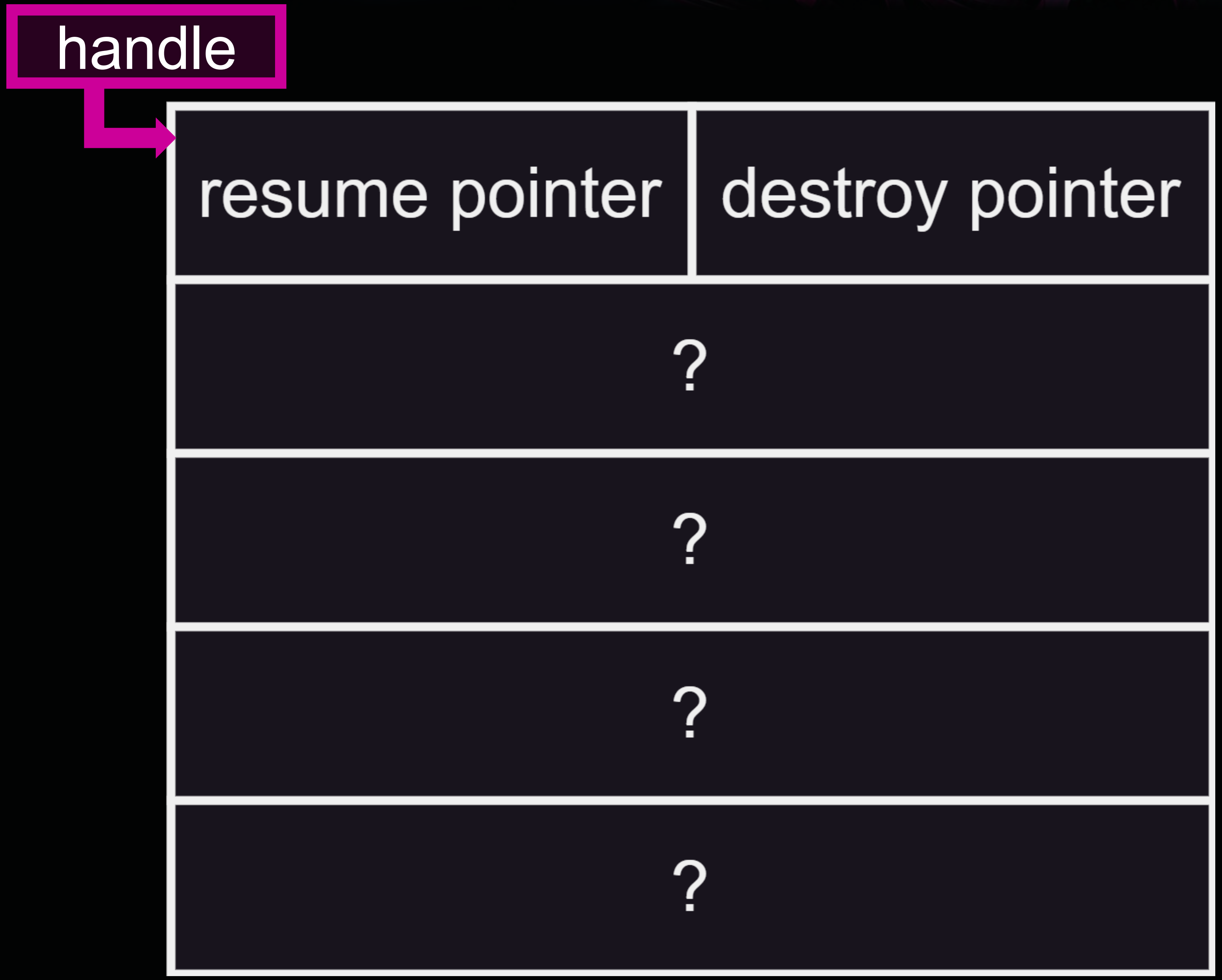
- Coroutines in C++ are stackless
  - Can only be suspended from the coroutine itself (you cannot call another function and suspend from there)
  - Other stackless coroutines: C#, JS, Python, Rust, Swift

- The coroutine is stored in a heap-allocated coroutine frame

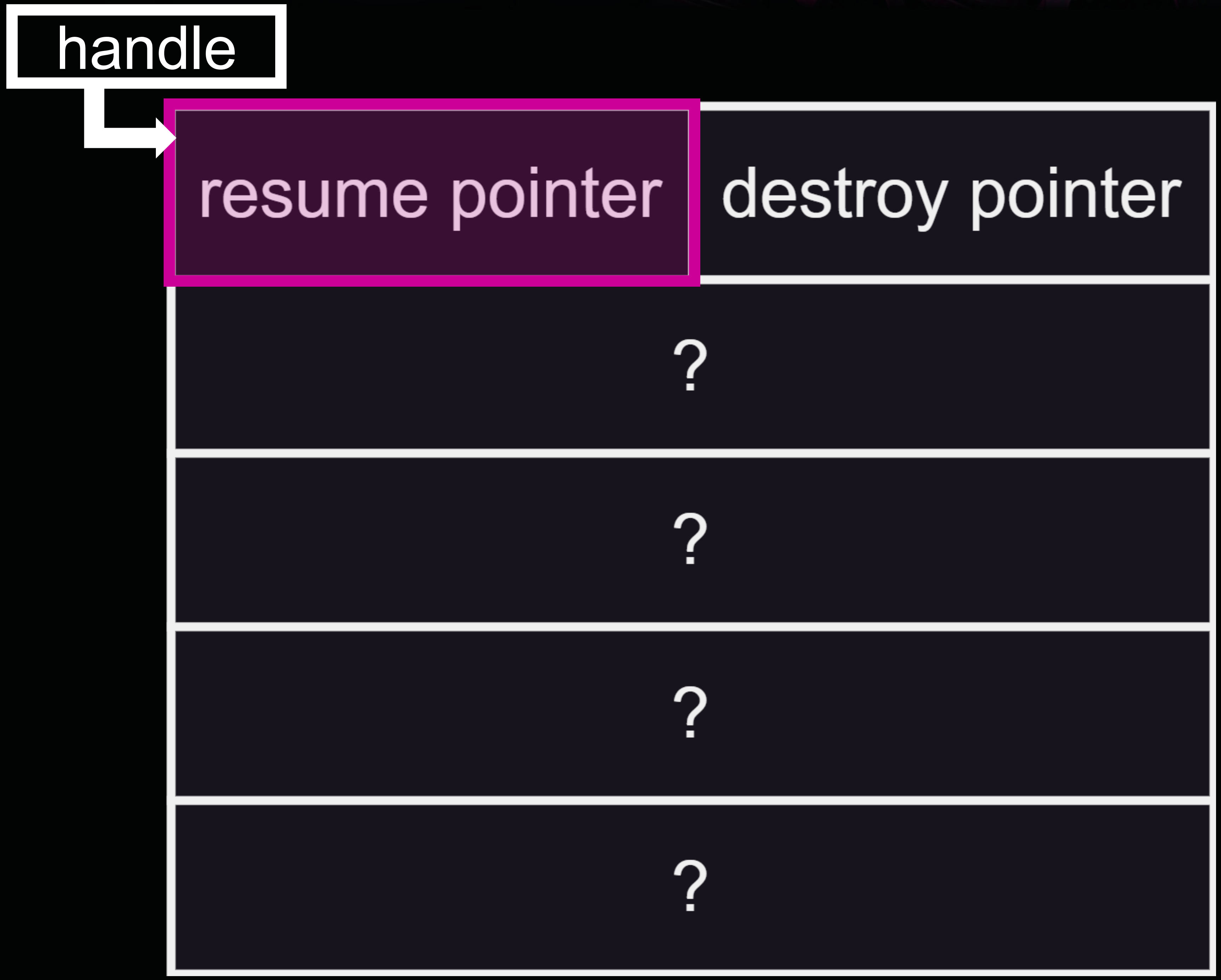
```
void foo()
{
   coroutine_handle<> h1 = coro().handle;
   coroutine_handle<> h2 = coro().handle;
}
```

2 allocated frames

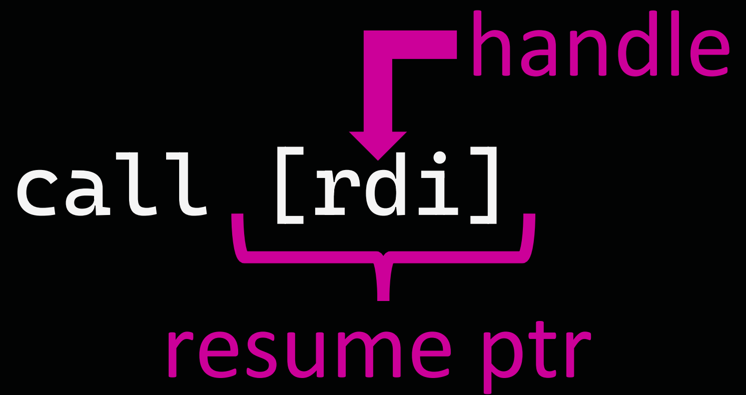
The Coroutine Frame



The Coroutine Frame

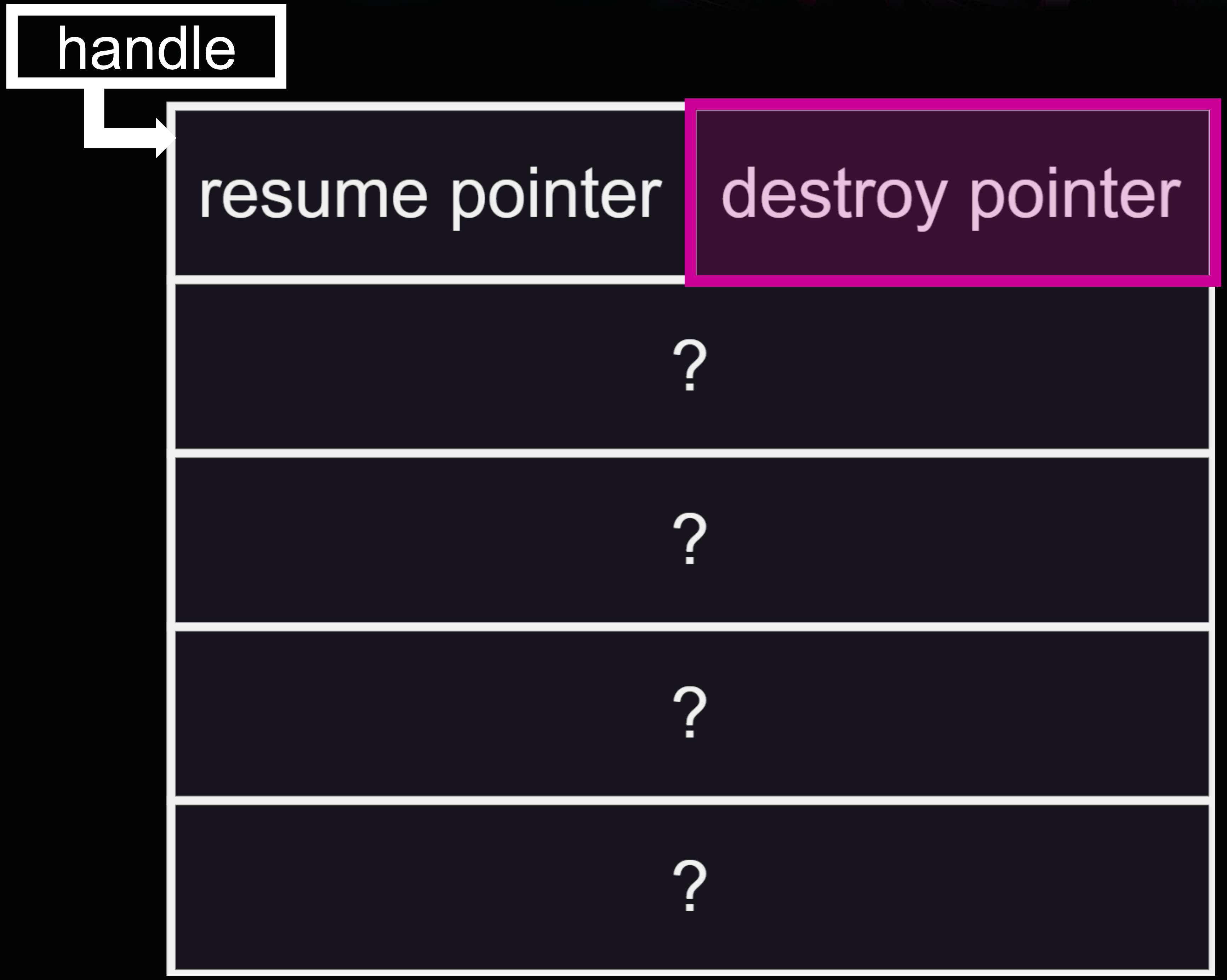



```
handle.resume()
```



- Points to the resume stub

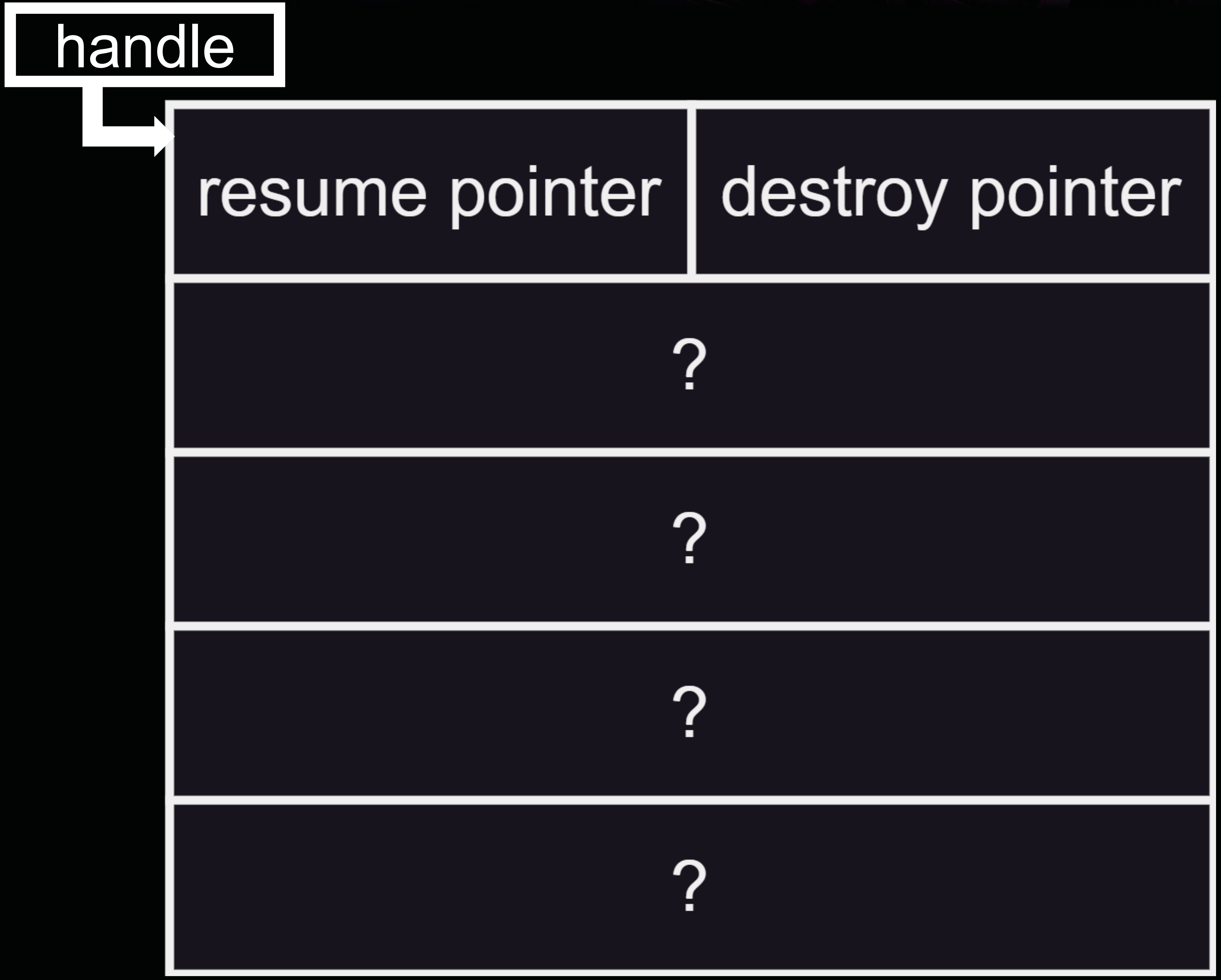
The Coroutine Frame



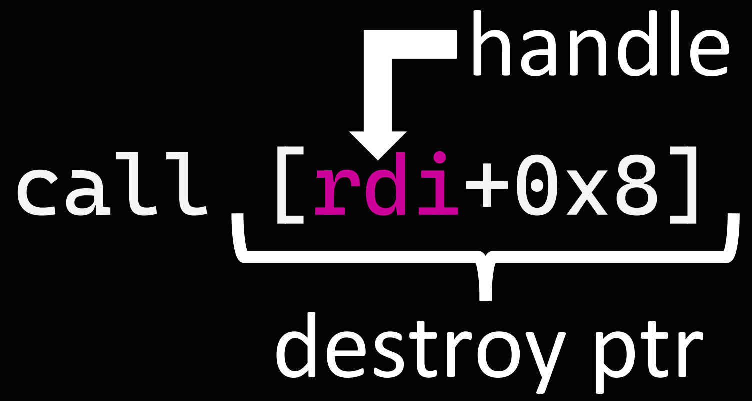
`handle.destroy()`

- Points to the destroy stub

The Coroutine Frame




```
handle.destroy()
```




```
void resume() const {
  coro_resume(pointer_to_frame);
}
void destroy() const {
  coro_destroy(pointer_to_frame);
}
```

The Coroutine Frame

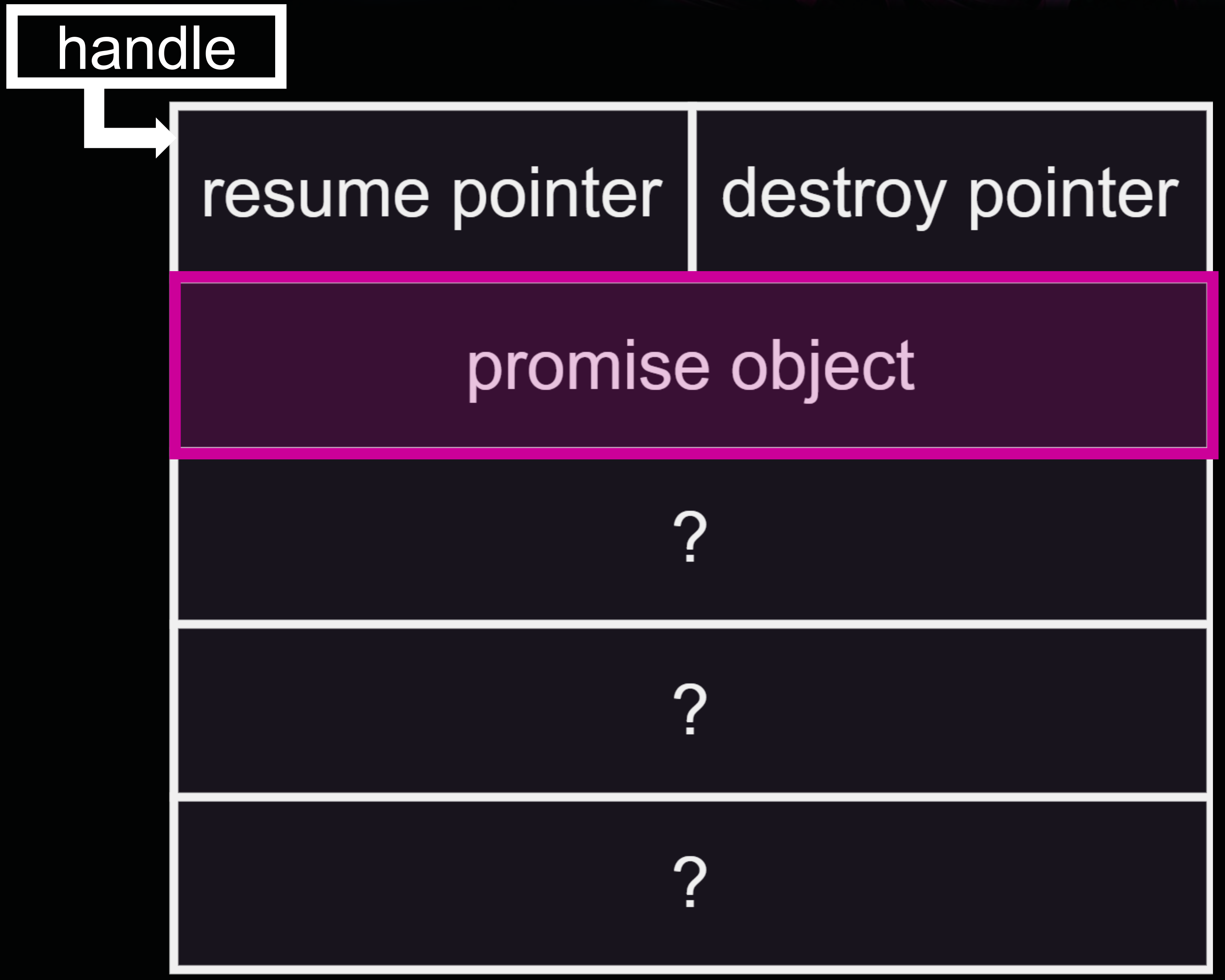
resume pointer	destroy pointer
?	
?	
?	
?	

resume pointer	destroy pointer
?	
?	
?	
?	

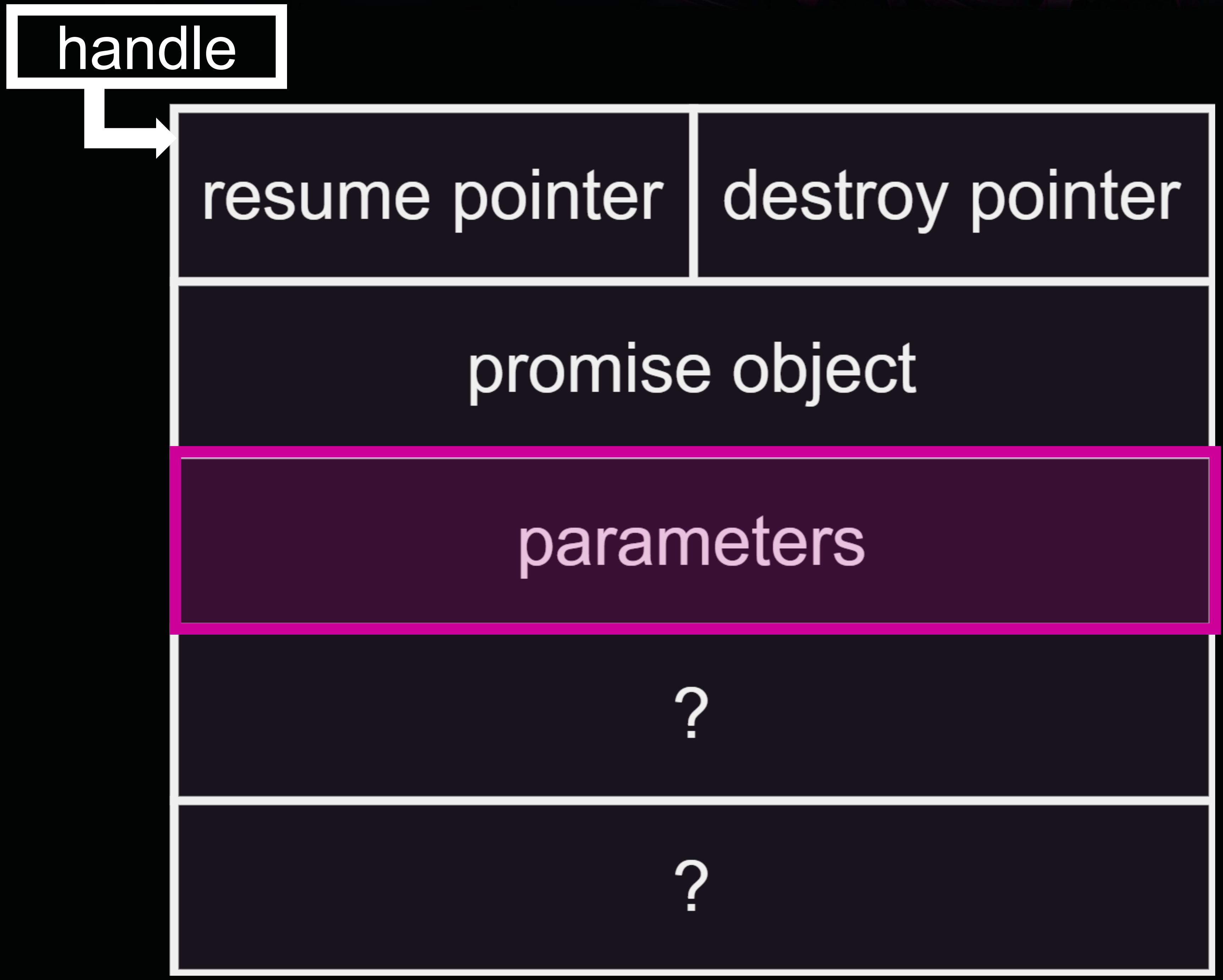
resume pointer	destroy pointer
?	
?	
?	
?	

resume pointer	destroy pointer
?	
?	
?	
?	

The Coroutine Frame

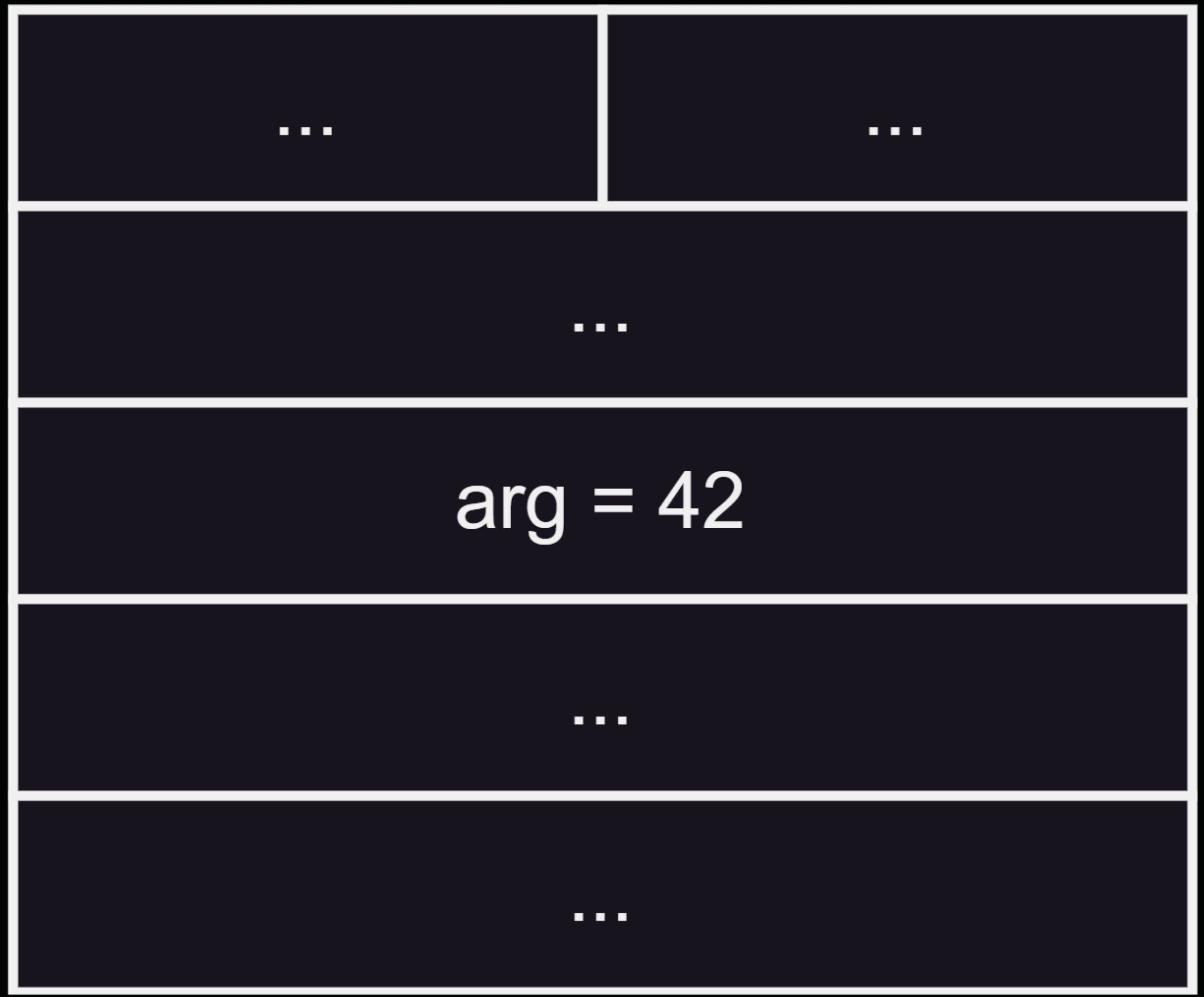


The Coroutine Frame



The Coroutine Frame

```
void main()
{
   coro(42);
}

task coro(int arg)
{
   co_return;
}
```



The Coroutine Frame

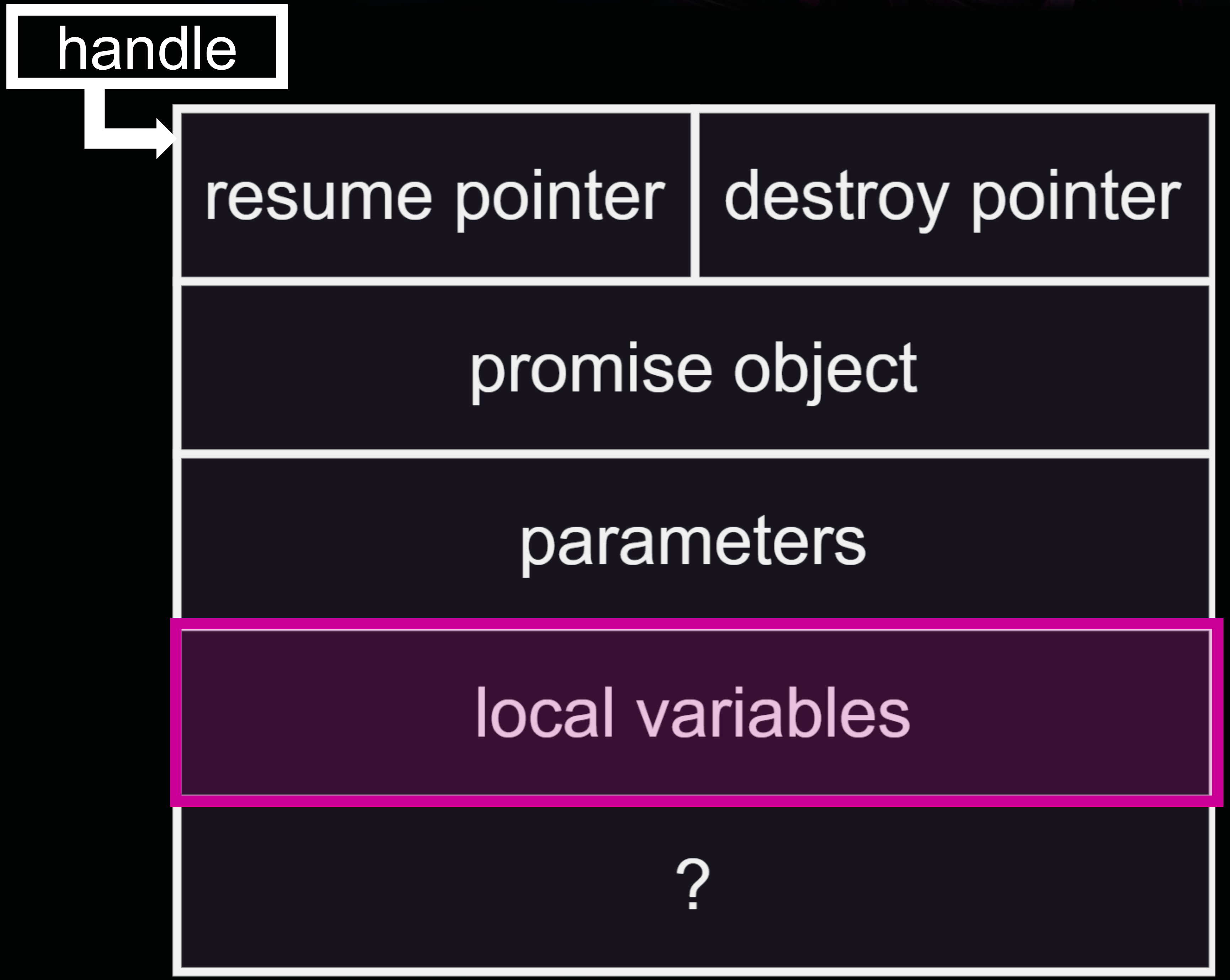
```
void main()
{
  string s = “hello”;
  coro(s);
}
task coro(string arg)
{
  co_return;
}
```

The Coroutine Frame

```
void main()
{
   char* buf;
   coro(buf);
}

task coro(char* arg)
{
   co_return;
}
```

...	...
...	
arg = <ptr to buf>	
...	
...	

The Coroutine Frame



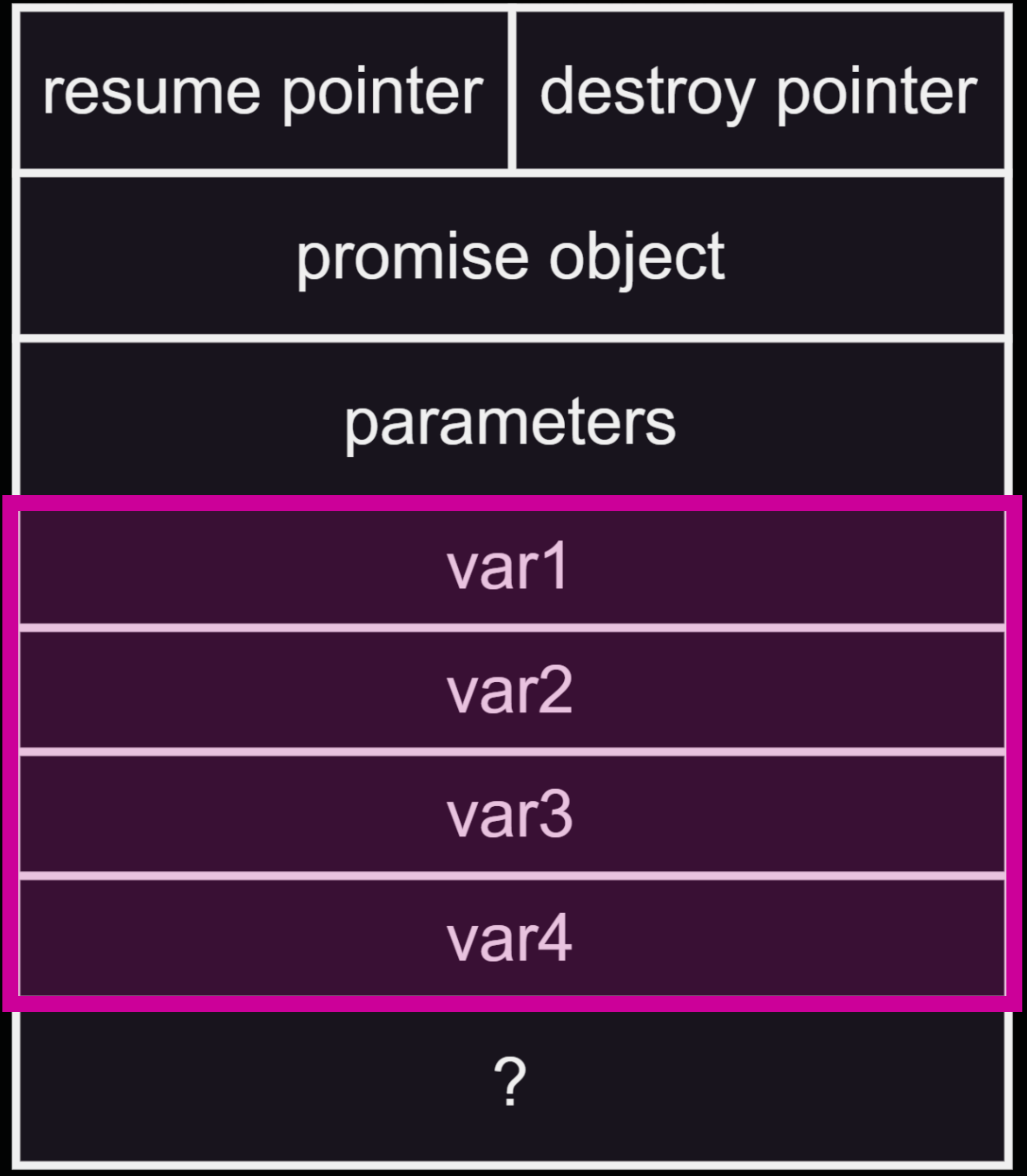
The Coroutine Frame

Stack


```
task coro()
{
   int var1,var2,var3,var4;
}
```

The Coroutine Frame

Heap

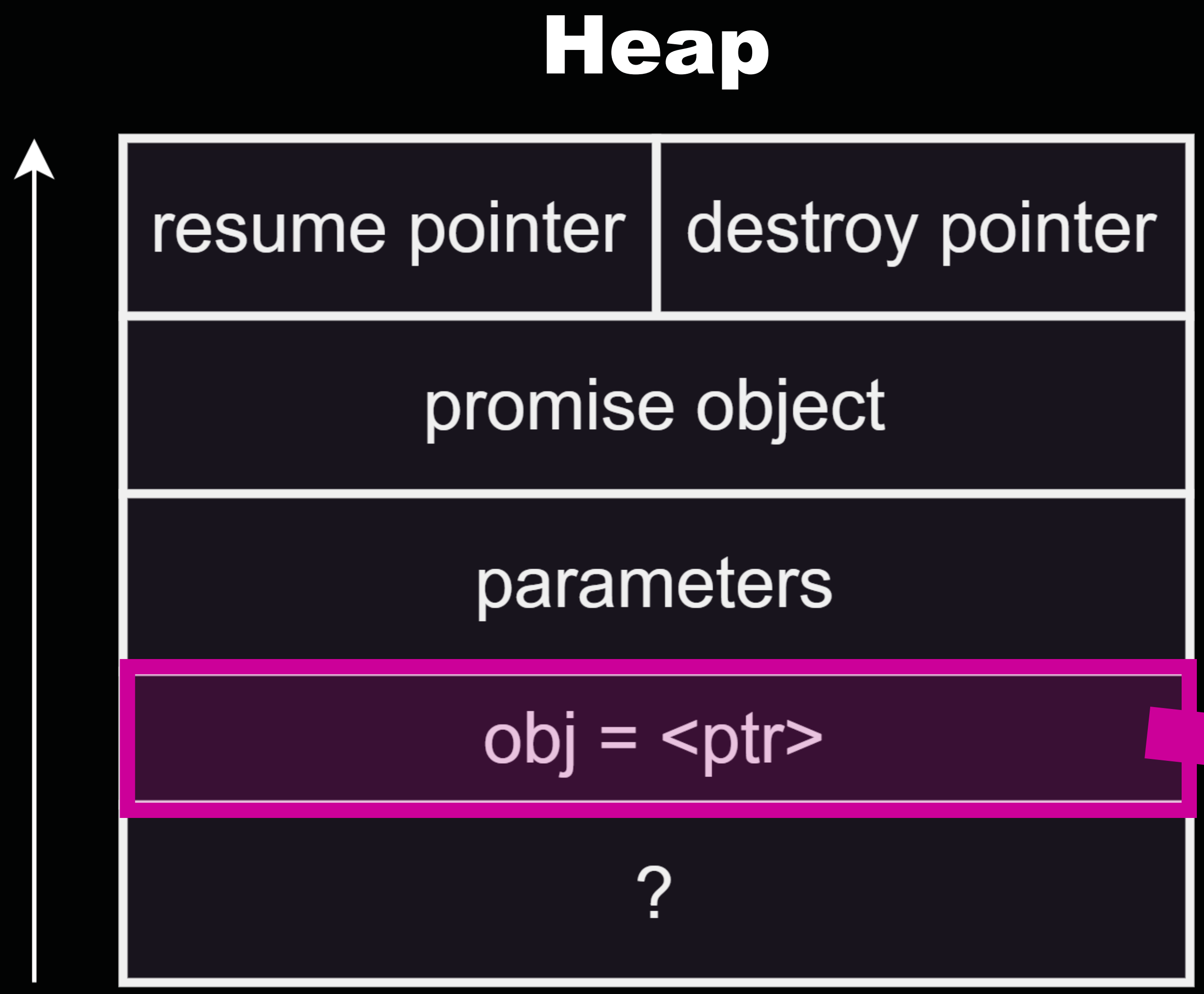



```
task coro()
{
   int var1,var2,var3,var4;
}
```

- Stack-based vars → heap-based vars
- Heap-based vars → heap-based vars

Stack

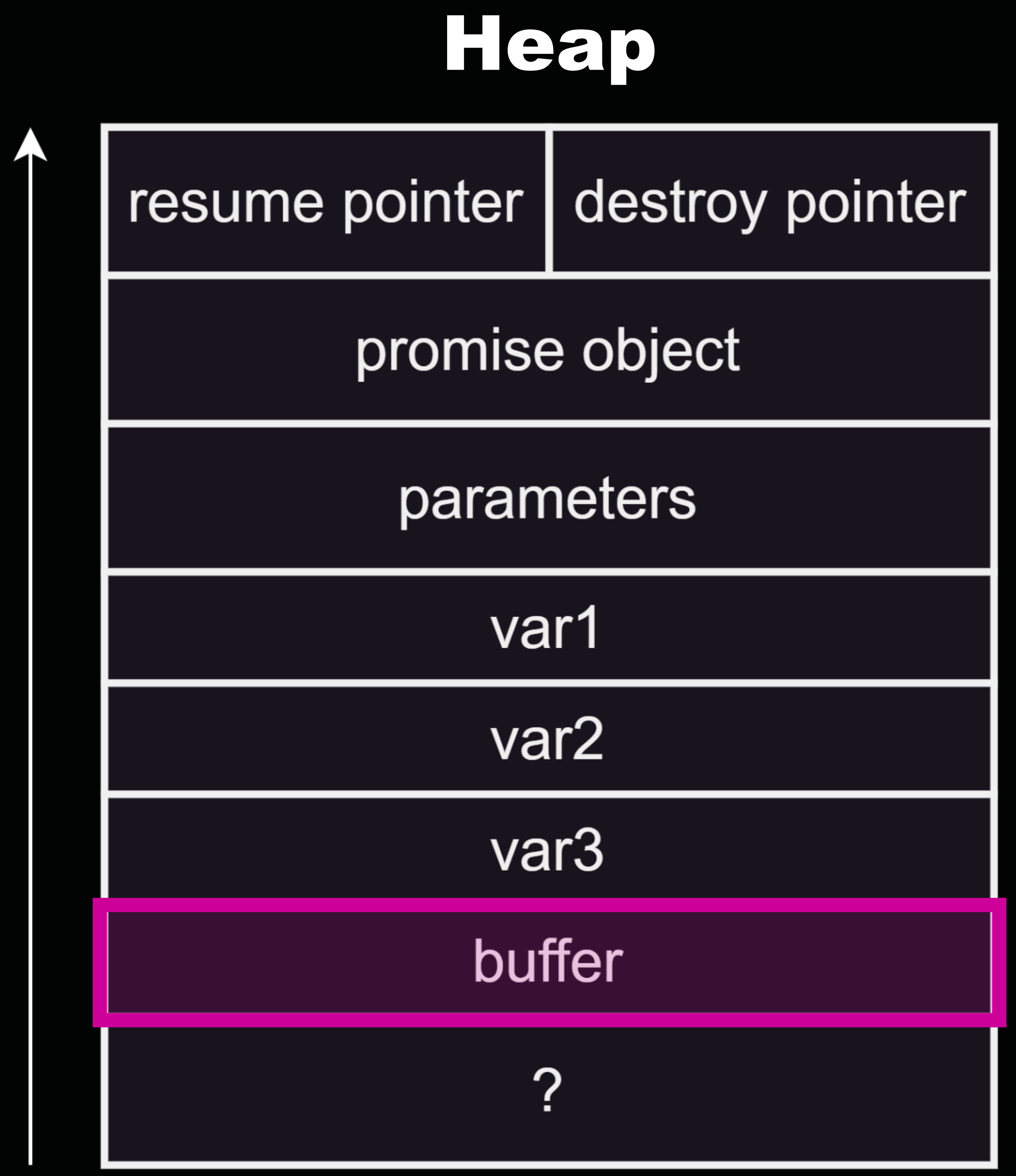
The Coroutine Frame




```
task coro()
{
   Object *obj = new Object();
}
```

- Stack-based vars → heap-based vars
- Heap-based vars → heap-based vars

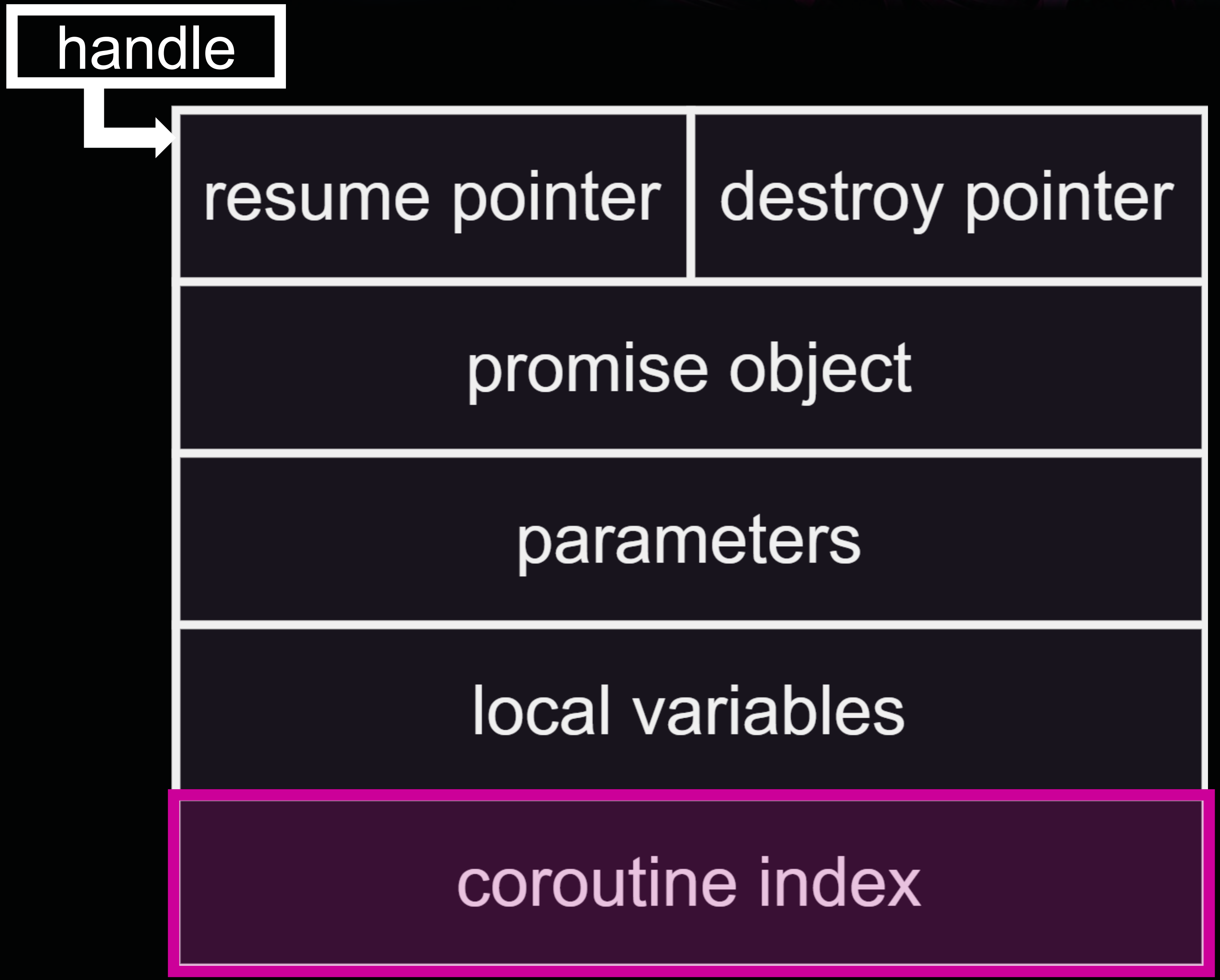
The Coroutine Frame




```
task coro()
{
   char buffer[];
   int var1,var2,var3;
}
```

- Compiler *sometimes* reorders "stack-based" buffers to safer positions (we will see when)

The Coroutine Frame




```
CI = 0  task coro()
        {
CI = 1  →   co_yield “one”;
CI = 2  →   co_yield “two”;
CI = 3  →   co_return “three”;
        }
```

The Stubs in Depth

Creation stub

Resume stub

Destroy stub

```
coroframe creation_stub()
{
   coroframe = new()
   coroIndex = 0;

   resumePtr = &resume_stub
   destroyPtr = &destroy_stub

   return coroframe;
}
```

```
call    0x5555555551d0 <_Znwm@plt>
mov     QWORD PTR [rbp-0x20],rax
mov     rax,QWORD PTR [rbp-0x20]
mov     BYTE PTR [rax+0x32],0x1
mov     rax,QWORD PTR [rbp-0x20]
lea     rdx,[rip+0xf5]        # 0x555555555ab0 <foo(_Z3fooPv.Frame *)>
mov     QWORD PTR [rax],rdx
mov     rax,QWORD PTR [rbp-0x20]
lea     rdx,[rip+0x490]        # 0x555555555e59 <foo(_Z3fooPv.Frame *)>
mov     QWORD PTR [rax+0x8],rdx
```

The Stubs in Depth

Creation stub

Resume stub

Destroy stub

```
void resume_stub(coroFrame)
{
   switch(coroFrame.coroIndex)
   {
      case 0:
         //first suspension point
      case 1:
         //second SP
      default:
         //err
   }
}
```

```
je     0x555555555d70 <foo(_Z3fooPv.Frame *)+704>
cmp    eax,0x6
jg     0x555555555b96 <foo(_Z3fooPv.Frame *)+230>
cmp    eax,0x4
je     0x555555555cab <foo(_Z3fooPv.Frame *)+507>
cmp    eax,0x4
jg     0x555555555b96 <foo(_Z3fooPv.Frame *)+230>
test   eax,eax
je     0x555555555b51 <foo(_Z3fooPv.Frame *)+161>
cmp    eax,0x2
je     0x555555555bcf <foo(_Z3fooPv.Frame *)+287>
jmp    0x555555555b96 <foo(_Z3fooPv.Frame *)+230>
```

The Stubs in Depth

Creation stub

Resume stub

Destroy stub

```
void resume_stub(coroFrame)
{
   switch(coroFrame.coroIndex)
   {
      case 0:
         cout << “Hello”;
      case 1:
         cout << “Bye”;
      default:
         //err
   }
}
```

```
task coro()
{
   cout << “Hello”;
   <SP>
   cout << “Bye”;
}
```

The Stubs in Depth

Creation stub

Resume stub

Destroy stub

```
void resume_stub(coroFrame)
{
   switch(coroFrame.coroIndex)
   {
      case 0:
         initial_suspend();
         cout << “Hello”;
      case 1:
         cout << “Bye”;
         final_suspend();
      default:
         //err
   }
}
```

```
task coro()
{
   cout << “Hello”;
   <SP>
   cout << “Bye”;
}
```

The Stubs in Depth

black hat BRIEFINGS

Creation stub

Resume stub

Destroy stub

```
void destroy_stub(coroFrame)
{
   delete coroFrame;
}
```

What is a Coroutine

- The compiler treats a function as a coroutine whenever one of the three coroutine keywords appear:

```
void coro()
{

   <suspend>;

}
```

- **co_await**
- **co_yield**
- **co_return**

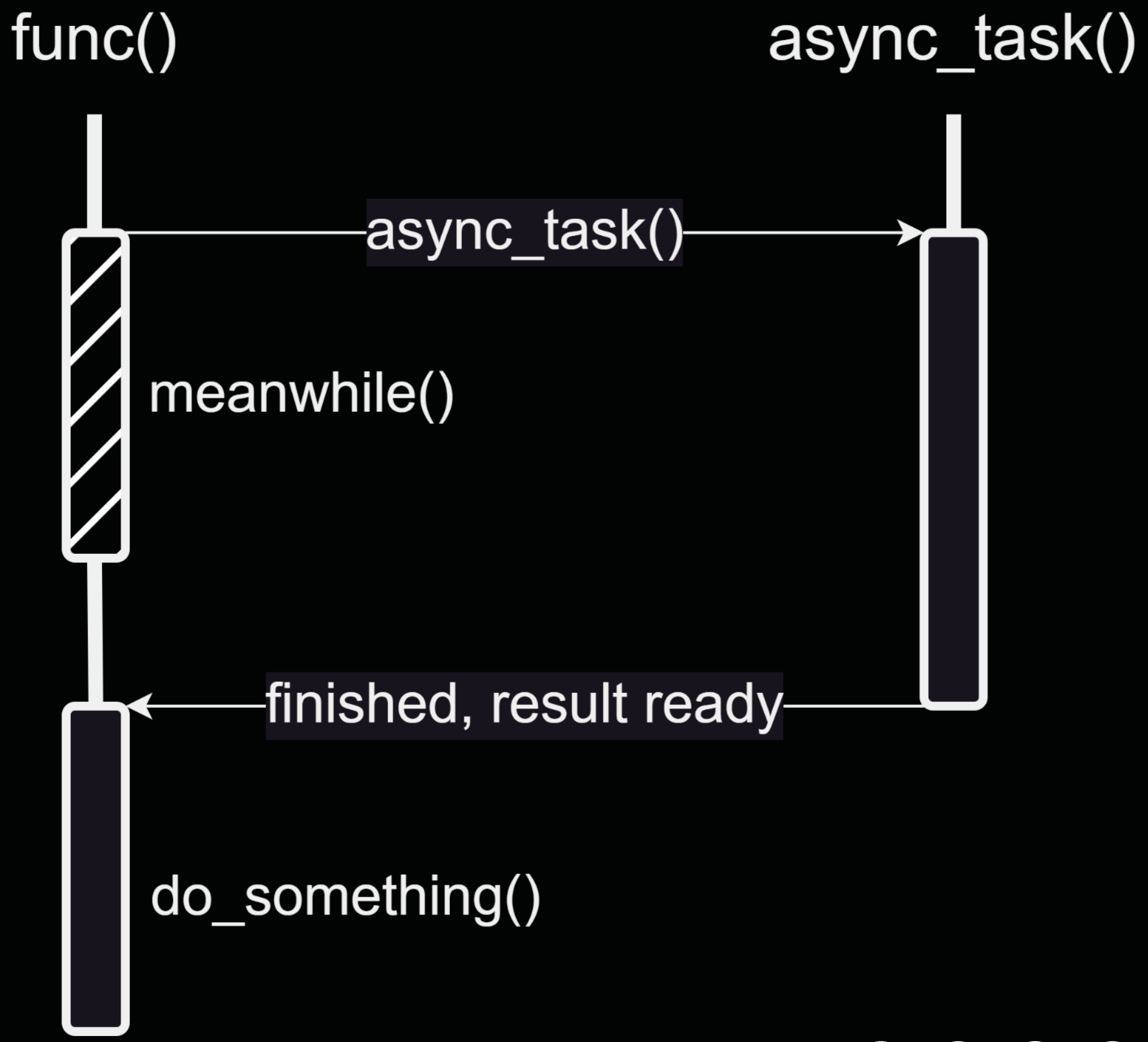
Coroutine Awaiting

- *co_await* evaluates an awaitable

- Use cases:
  - Asynchronous jobs
  - Awaitable coroutines
  - Cooperative multitasking

Coroutine Awaiting

```
void func()
{
   //Execute async
   result = async_task();

   //Do something else meanwhile
   meanwhile();


   ...
   //When result is ready, do sth
   something(result);
}
```



Coroutine Awaiting

- Without coroutines, callbacks would typically be used

```
void func()
{
   //Read the length, then read the buffer
   async_read(len, when_done={
      async_read(buf, when_done={
         process_buffer(buf());
      }
   }
}
```

Coroutine Awaiting

- Without coroutines, callbacks would typically be used

```
void func()
{
  //Read the length, then read the buffer
  async_read(len, when_done={
    async_read(buf, when_done={
      process_buffer(buf());
      async_other(…, when_done={
        process_buffer(buf());
      }
      …
    }
  }
}
```

Coroutine Awaiting

- With coroutines, code looks synchronous but is actually not
- In simple terms, *co_await* suspends the coroutine and does something else

```
task coro()
{
   len = co_await async_read();
   buf = co_await async_task();
   ...
}
```

Coroutine Awaiting

- *co_await* evaluates an awaitable

```
task coroutine()
{
   co_await Awaitable{};
}
```

co_await

Coroutine Awaiting

- *co_await* evaluates an awaitable

```
task coroutine()
{
   co_await Awaitable{};
}
```

```
struct Awaitable
{
   Awaiter operator co_await()
   {
      return{};
   }
}
```

co_await ➡ **Awaitable**

Coroutine Awaiting

- The *awaiter* controls what happens at the co_await point
  - Maybe it suspends, or it executes something else…

```
struct Awaitable
{
   Awaiter operator co_await()
   {
      return{};
   }
}
```

```
struct Awaiter()
{
   bool await_ready();

   void await_suspend(…);

   void await_resume();
}
```

co_await ➡ Awaitable ➡ Awaiter

Coroutine Awaiting

- The *awaiter* controls what happens at the co_await point
  - Maybe it suspends, or it executes something else…

```
task coroutine()
{
   co_await Awaiter{};
}
```

```
struct Awaiter()
{
   bool await_ready();

   void await_suspend(…);

   void await_resume();
}
```

co_await ➡ Awaiter

Coroutine Awaiting

```
struct Awaiter()
{
    bool await_ready();

    void await_suspend(…);

    void await_resume();
}
```

Do we need to suspend the coroutine?

Coroutine Awaiting

```
struct Awaiter()
{
    bool await_ready();

    void await_suspend(…);

    void await_resume();
}
```

The coroutine is now suspended.

Do you want to do something with the suspended coroutine?

co_await ➡ Awaiter

Coroutine Awaiting

```
void await_suspend(coroutine_handle suspended_coro)
{





   //coroutine suspended, return to the caller of the coroutine
}
```

co_await ➡ Awaiter

Coroutine Awaiting

```
void await_suspend(coroutine_handle suspended_coro)
{

•  Execute async code
•  Start new threads
•  Resume the coroutine: suspended_coro.resume()


   //coroutine suspended, return to the caller of the coroutine
}
```

Coroutine Awaiting

```
struct Awaiter()
{
    bool await_ready();

    void await_suspend(…);

    void await_resume();
}
```

Anything to do before resuming the coroutine?

co_await ➡ Awaiter

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

   ...
}
```

```
struct Awaiter
{
   bool await_ready(){}
   void await_suspend(h)
   {
      std::thread((){
         <time expensive work>
         handle.resume();
      }).detach();
   }
   void await_resume(){};
}
```

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

   ...
}
```

```
struct Awaiter
{
   bool await_ready(){}
   void await_suspend(h)
   {
      std::thread((){
         <time expensive work>
         handle.resume();
      }).detach();
   }
   void await_resume(){};
}
```

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

   ...
}
```

```
struct Awaiter
{
  bool await_ready(){}
  void await_suspend(h)
  {
    std::thread((){
      <time expensive work>
      handle.resume();
    }).detach();
  }
  void await_resume(){};
}
```

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

   ...
}
```

```
struct Awaiter
{
   bool await_ready(){}
   void await_suspend(h)
   {
      std::thread((){
         <time expensive work>
         handle.resume();
      }).detach();
   }
   void await_resume(){};
}
```

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

   <something else>
}
```

```
struct Awaiter
{
   bool await_ready(){}
   void await_suspend(h)
   {
      std::thread((){
         <time expensive work>
         handle.resume();
      }).detach();
   }
   void await_resume(){};
}
```

THREAD 1

t1

t2

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

   <something else>
}
```

t2

```
struct Awaiter
{
   bool await_ready(){}
   void await_suspend(h)
   {
      std::thread((){
         <time expensive work>
         handle.resume();
      }).detach();
   }
   void await_resume(){};
}
```

THREAD 1

t1

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

   <something else>
}
```

t2

```
struct Awaiter
{
   bool await_ready(){}
   void await_suspend(h)
   {
      std::thread([](){
         <time expensive work>
         handle.resume();
      }).detach();
   }
   void await_resume(){};
}
```

t1

black hat BRIEFINGS

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

   <something else>
}
```

```
struct Awaiter
{
   bool await_ready(){}
   void await_suspend(h)
   {
      std::thread(()){
         <time expensive work>
         handle.resume();
      }).detach();
   }
   void await_resume(){};
}
```

co_await Example

```
task coro()
{
   //
   co_await Awaiter{};
   //
}

void func()
{
   handler h = coro();
   h.resume();

t1 <join>
}
```

```
struct Awaiter
{
   bool await_ready(){}
   void await_suspend(h)
   {
      std::thread((){
         <time expensive work>
         handle.resume();
      }).detach();
   }
   void await_resume(){};
}
```

Co_awaiting Coroutines

```
task
{
    handle h;
    struct promise_type
    {
      int return_value;
      suspend_always initial_suspend();
      suspend_always final_suspend();
    };
}
```

Co_awaiting Coroutines

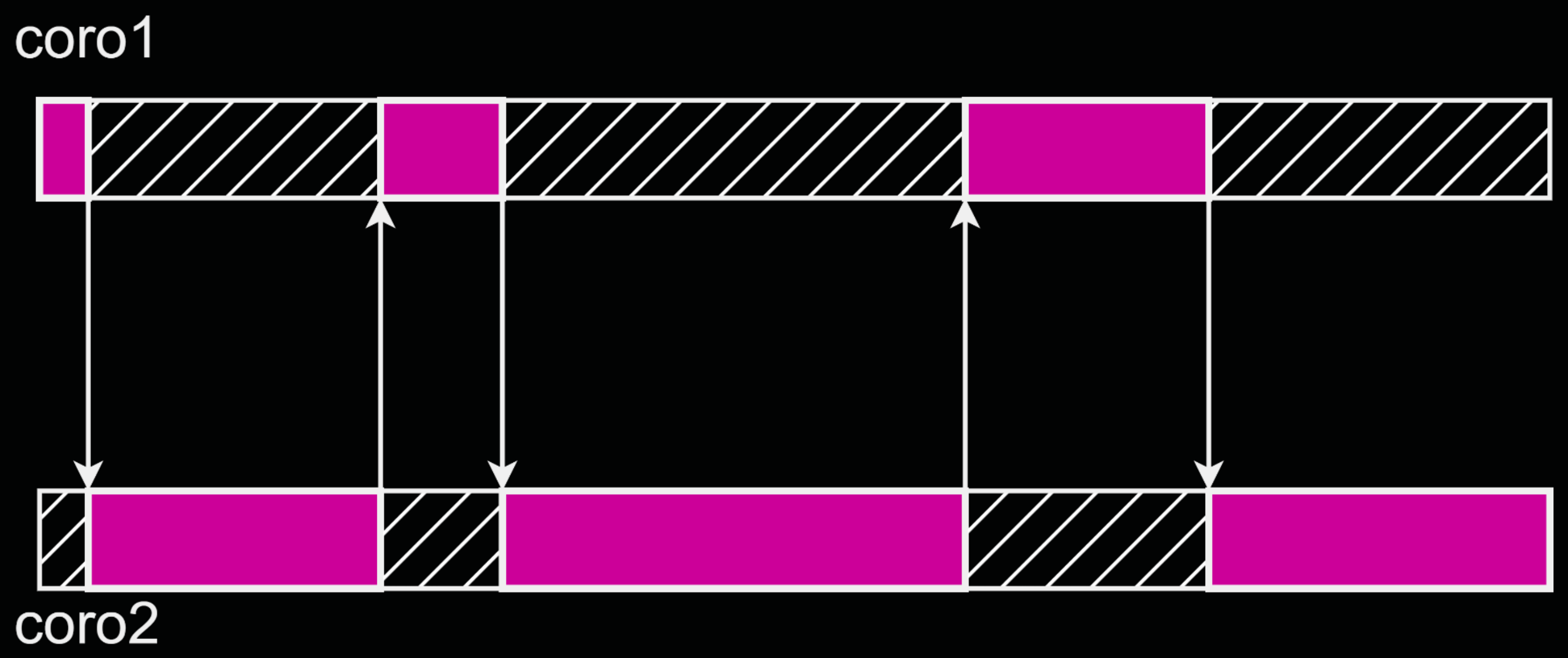
```
task
{
    handle h;
    struct promise_type
    {
        int return_value;
        suspend_always initial_suspend();
        suspend_always final_suspend();
    };
    struct awaiter
    {
        bool await_ready();
        void await_suspend();
        void await_resume();
    }
}
```

Co_awaiting Coroutines

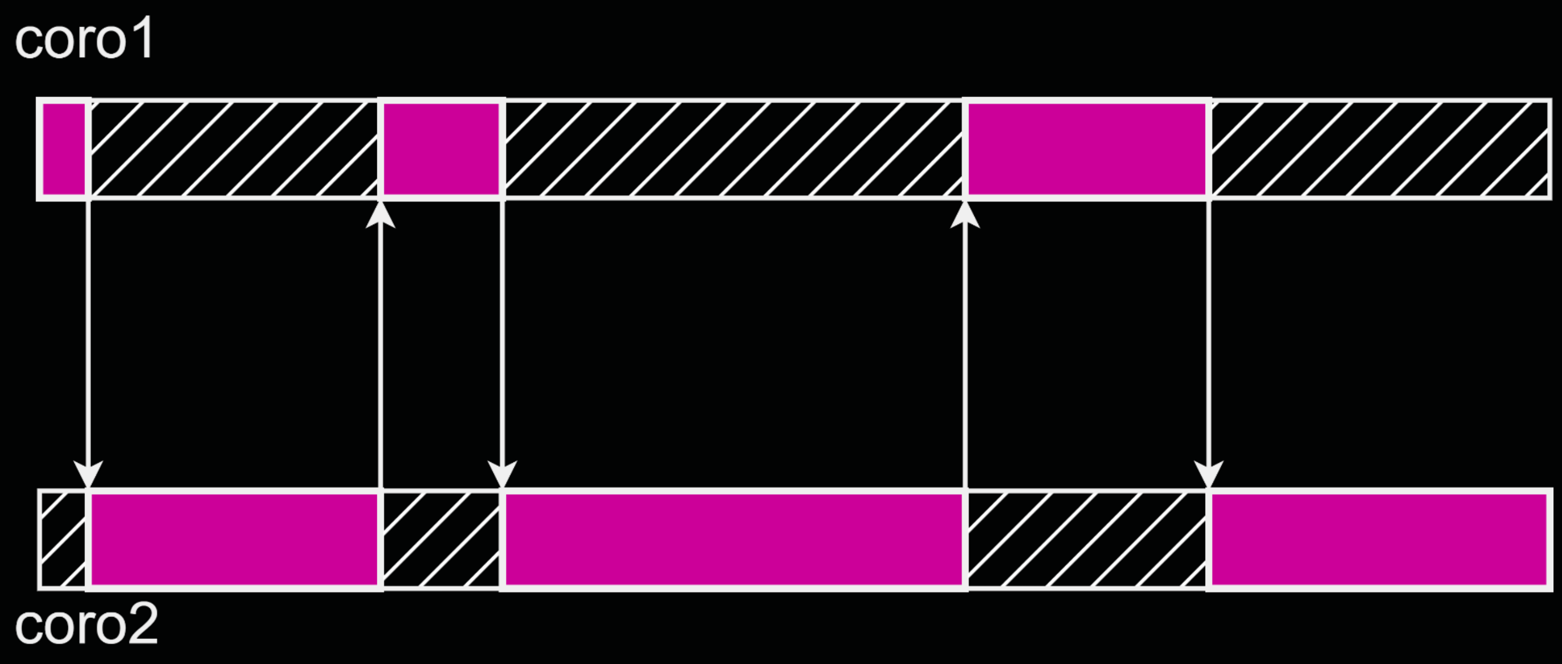
```
task coro2()
{
   co_return;
}

task coro1()
{
   co_await coro2();
}
```

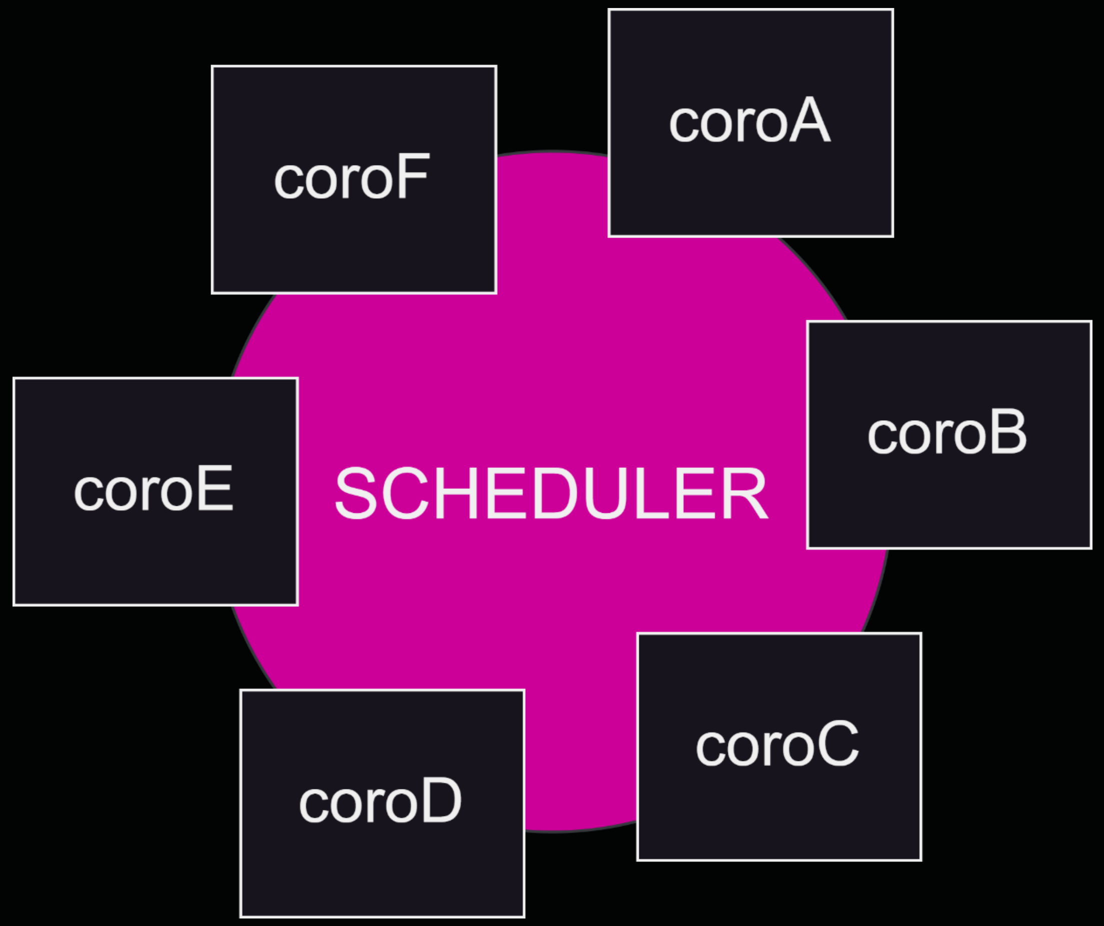
```
task
{
    handle h;
    struct promise_type
    {
       int return_value;
       suspend_always initial_suspend();
       suspend_always final_suspend();
    };
    struct awaiter
    {
       bool await_ready();
       void await_suspend();
       void await_resume();
    }
}
```

Use Cases



Use Cases





4 CFOP

Threat model

- CFI threat model:
  - ASLR bypassed
  - Infinite number of arbitrary memory writes

Threat model

- CFI threat model:
  - ASLR bypassed
  - Infinite number of arbitrary memory writes

- Our threat model:
  - ASLR bypassed
  - Attacker can leverage a single memory corruption vulnerability
  - At lease one coroutine in the code
  - CFI is in place

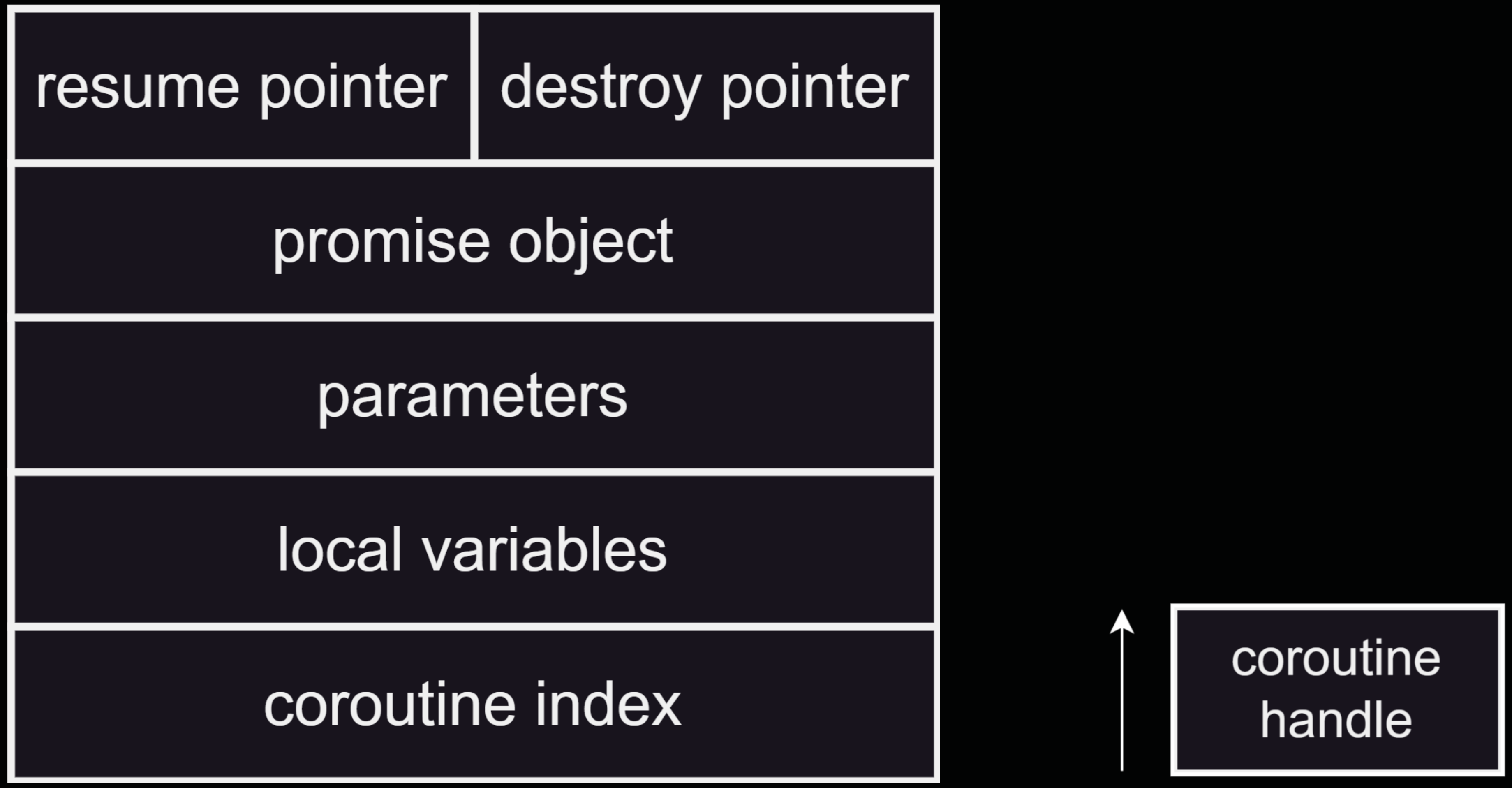
black hat BRIEFINGS

Threat model

- Intel CET
- LLVM CFI (icall)
- LLVM KCFI
- Safe Dispatch
- Control Flow Guard
- MCFI/ piCFI
- ReCFI
- Path Armor
- VfGuard
- CFIXX
- PittyPat
- VTrust
- Typro

Observations

- The coroutine handles and frames are writable



Attack primitives

FRAME MANIPULATION

- Modifying existing frames

<table>
<tr><td>resume pointer</td><td>destroy pointer</td></tr>
<tr><td colspan="2">promise object</td></tr>
<tr><td colspan="2">parameters</td></tr>
<tr><td colspan="2">local variables</td></tr>
<tr><td colspan="2">coroutine index</td></tr>
</table>

Attack primitives

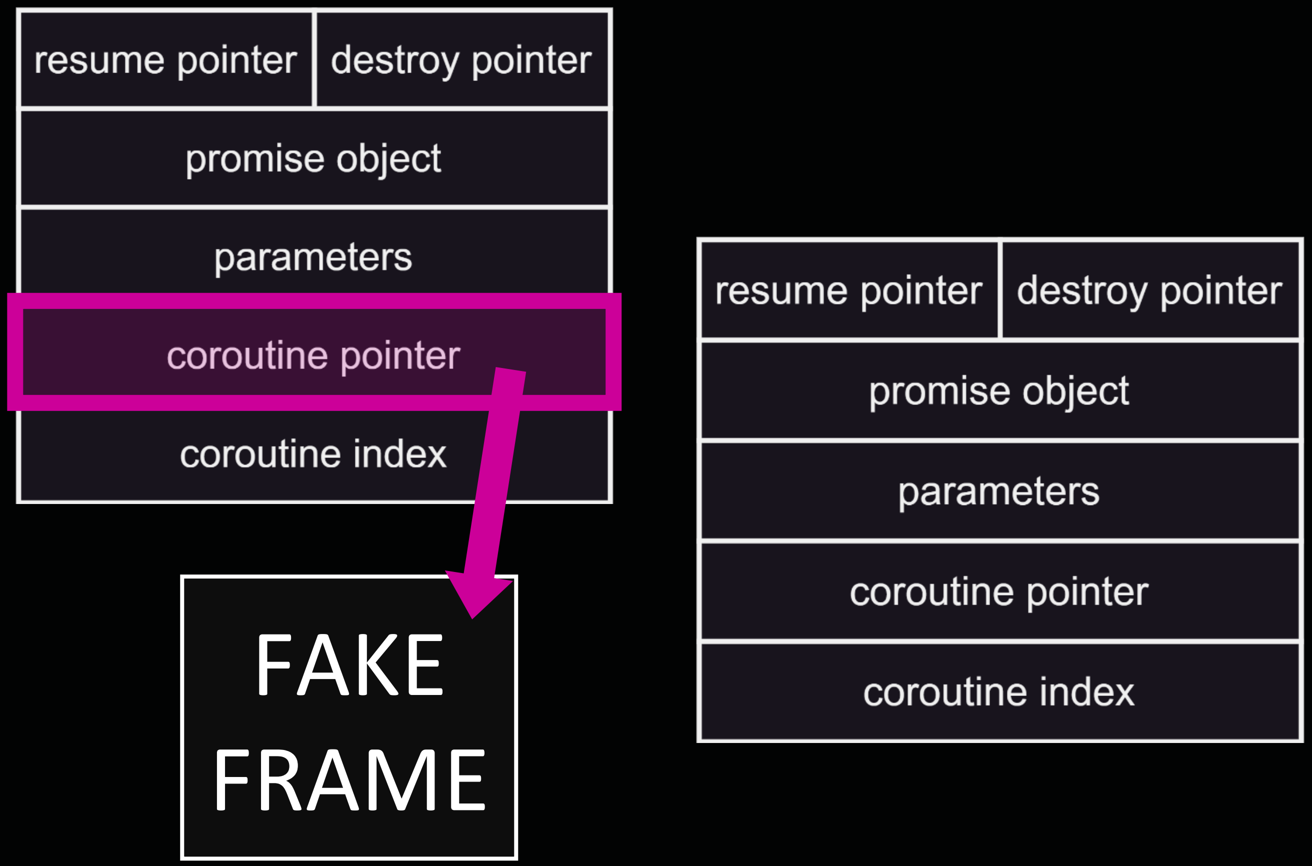
FRAME MANIPULATION

- Modifying existing frames

resume pointer	destroy pointer
promise object	
parameters	
local variables	
coroutine index	

FRAME **INJECTION**

- Inserting **new** frames



DOA: Data Only Attack

- Modifying the runtime data of a program can lead to arbitrary code execution
- Data-Only Attacks (DOA) use frame manipulation

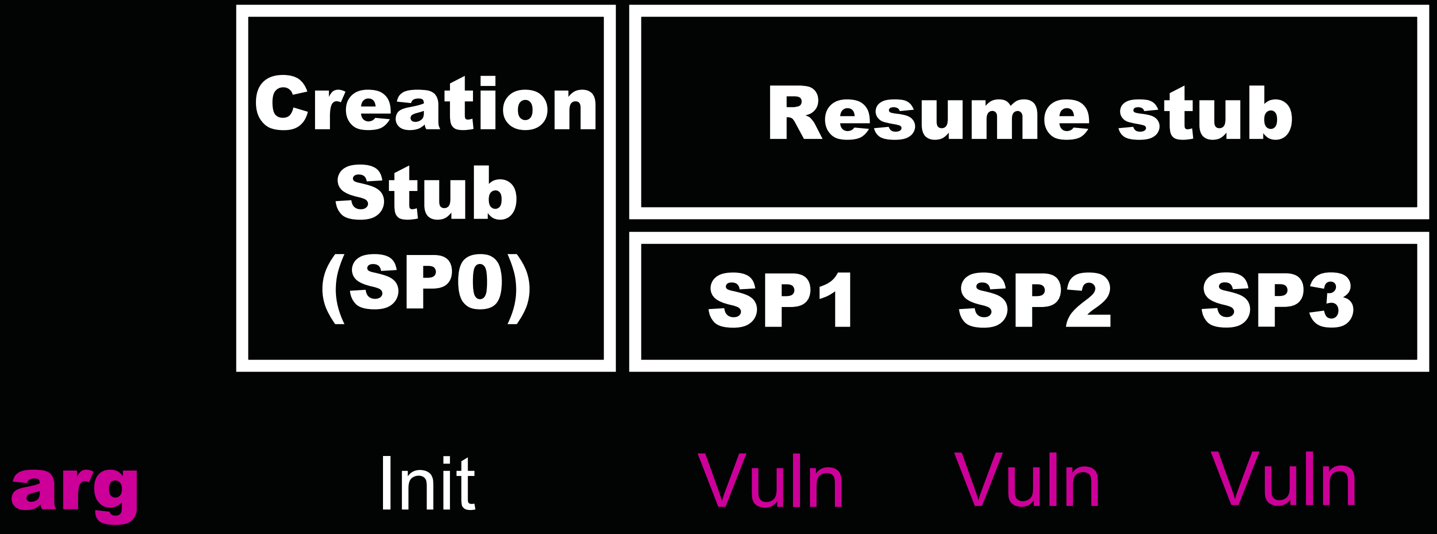
DOA: Data Only Attack

```
task coro(char* arg)
{
   co_await some_task;
   co_await some_task;
   system(arg);
}
```

DOA: Data Only Attack


```
task coro(char* arg)
{
★ //SP1//
  co_await some_task;
★ //SP2//
  co_await some_task;
★ //SP3//
  system(arg);
}
```

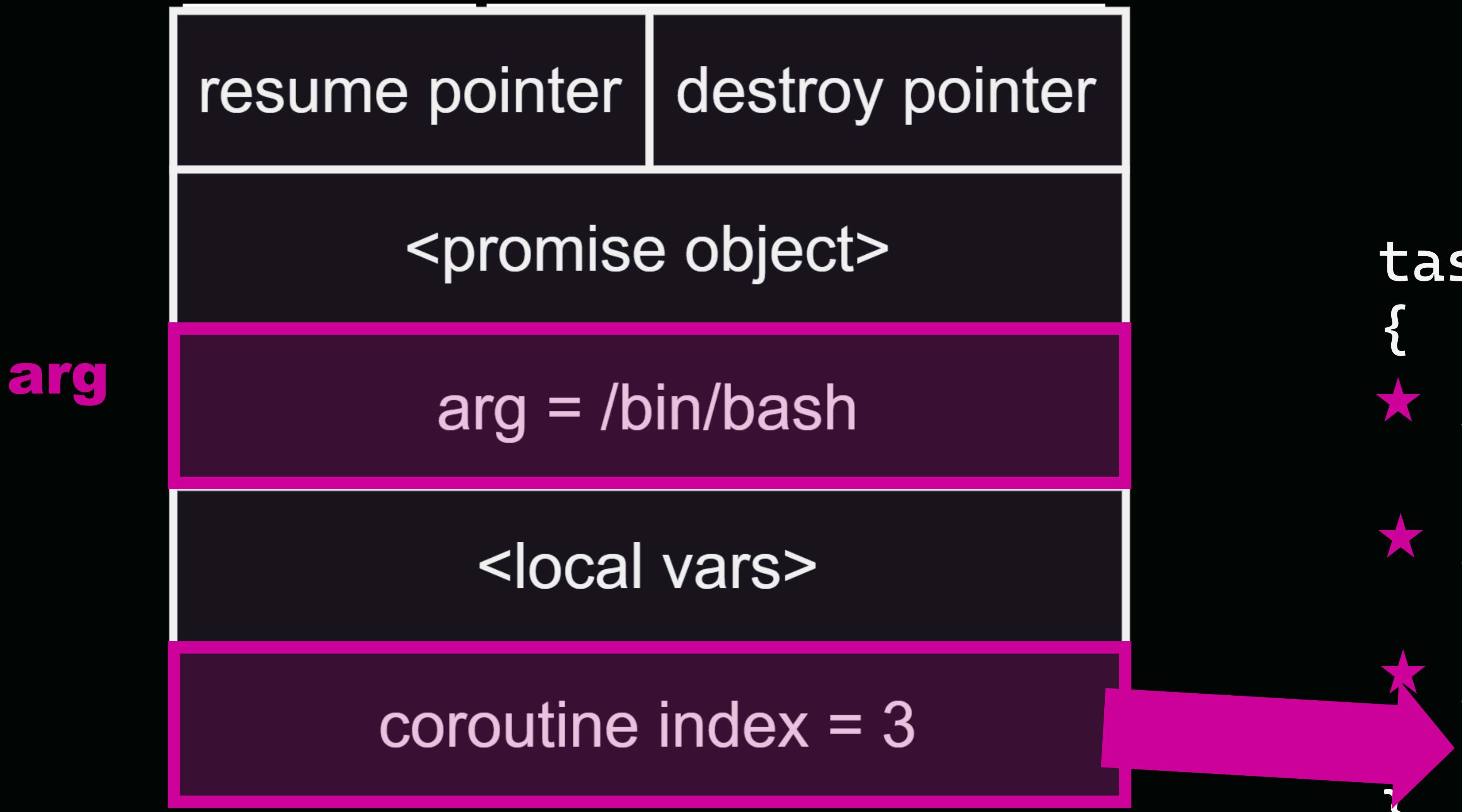
DOA: Data Only Attack




```
task coro(char* arg)
{
★ //SP1//
   co_await some_task;
★ //SP2//
   co_await some_task;
★ //SP3//
   system(arg);
}
```

1. Arguments are copied in the frame during the creation stub

DOA: Data Only Attack




```
task coro(char* arg)
{
★ //SP1//
  co_await some_task;
★ //SP2//
  co_await some_task;
★ //SP3//
  system(arg);
}
```

1. Arguments are copied in the frame during the creation stub

DOA: Data Only Attack

	Creation Stub (SP0)	Resume stub		
		SP1	SP2	SP3
arg	Init	Vuln	Vuln	Vuln

```
task coro(char* arg)
{
★ //SP1//
  char arr[10];
  co_await some_task;
★ //SP2//
  co_await some_task;
★ //SP3//
  getline(arr);
}
```

DOA: Data Only Attack

	Creation Stub (SP0)	Resume stub		
		SP1	SP2	SP3
arg	Init	Vuln	Vuln	Vuln
arr	—	—	—	Init

```
task coro(char* arg)
{
★ //SP1//
  char arr[10];
  co_await some_task;
★ //SP2//
  co_await some_task;
★ //SP3//
  getline(arr);
}
```

2. Local variables are copied to the frame on the same SP where they are first **initialized**.

DOA: Data Only Attack

	Creation Stub (SP0)	Resume stub		
		SP1	SP2	SP3
arg	Init	Vuln	Vuln	Vuln
arr	—	—	—	Init
arr2	—	—	Init	Vuln

```
task coro(char* arg)
{
★ //SP1//
  co_await some_task;
★ //SP2//
  char arr2 = “hello”;
  co_await some_task;
★ //SP3//
  puts(arr2);
}
```

2. Local variables are copied to the frame on the same SP where they are first initialized.

black hat BRIEFINGS

DOA: Data Only Attack

	Creation Stub (SP0)	Resume stub		
		SP1	SP2	SP3
arg	Init	Vuln	Vuln	Vuln
arr	—	—	—	Init
arr2	—	—	Init	Vuln

```
task coro(char* arg)
{
★ //SP1//
  co_await some_task;
★ //SP2//
  void *ptr;
  for(int ii=0; ii<3; ii++)
  {
     co_await some_task;
★    //SP3//
     write(ptr, 100);
  }
}
```

DOA: Data Only Attack

	Creation Stub (SP0)	Resume stub		
		SP1	SP2	SP3
arg	Init	Vuln	Vuln	Vuln
arr	—	—	—	Init
arr2	—	—	Init	Vuln
ii	—	—	Init	Vuln
ptr	—	—	—	Vuln

TIP: Local variables used inside a loop are always hijackable at some SP.

```
task coro(char* arg)
{
★ //SP1//
  co_await some_task;
★ //SP2//
  void *ptr;
  for(int ii=0; ii<3; ii++)
  {
    co_await some_task;
★   //SP3//
    write(ptr, 100);
  }
}
```

DOA: Data Only Attack

	Creation Stub (SP0)	Resume stub		
		SP1	SP2	SP3
arg	Init	Vuln	Vuln	Vuln
arr	—	—	—	Init
arr2	—	—	Init	Vuln
ii	—	—	Init	Vuln
ptr	—	—	—	Vuln

```
task coro(char* arg)
{
★ //SP1//
  co_await some_task;
★ //SP2//
  co_await some_task;
★ //SP3//
  int value = 0;
  value++;
}
```

DOA: Data Only Attack

	Creation Stub (SP0)	Resume stub		
		SP1	SP2	SP3
arg	Init	Vuln	Vuln	Vuln
arr	—	—	—	Init
arr2	—	—	Init	Vuln
ii	—	—	Init	Vuln
ptr	—	—	—	Vuln
value	(stack based local variable)			

```
task coro(char* arg)
{
★ //SP1//
    co_await some_task;
★ //SP2//
    co_await some_task;
★ //SP3//
    int value = 0;
    value++;
}
```

Advanced DOAs

```
task coro(char* arg)
{
★ //SP1//
   vector<int> vec;
   vec.push_back(1);
   co_await some_task;
★ //SP2//
   co_await some_task;
★ //SP3//
}
```

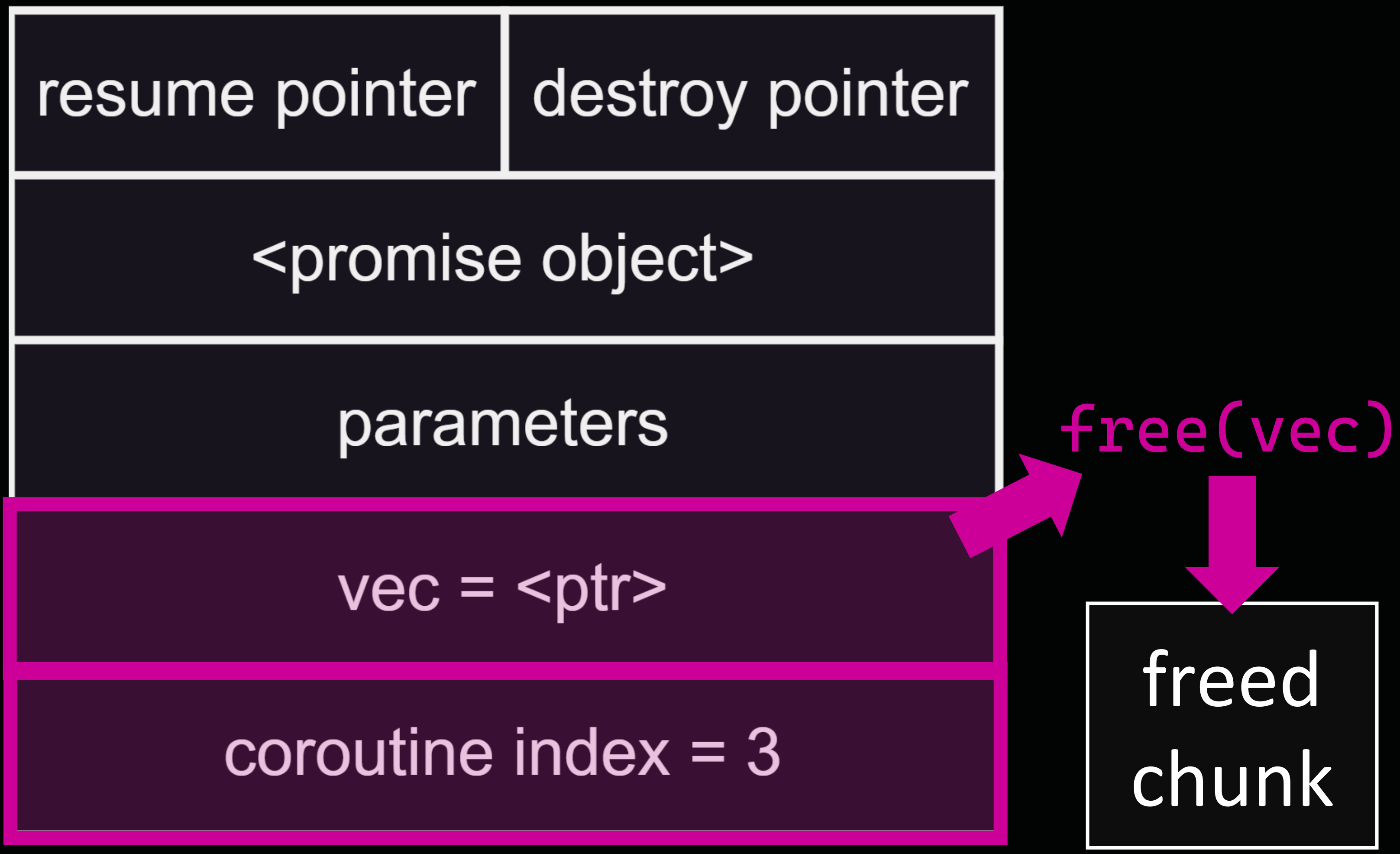
black hat BRIEFINGS

Advanced DOAs

```
task coro(char* arg)
{
★ //SP1//
  initial_suspend();
  vector<int> vec;
  vec.push_back(1);
  co_await some_task;
★ //SP2//
  co_await some_task;
★ //SP3//
  <FREE ALL VARIABLES>
  final_suspend();
}
```

Advanced DOAs

- Arbitrarily free() chunks
- Need to prepare chunk metadata



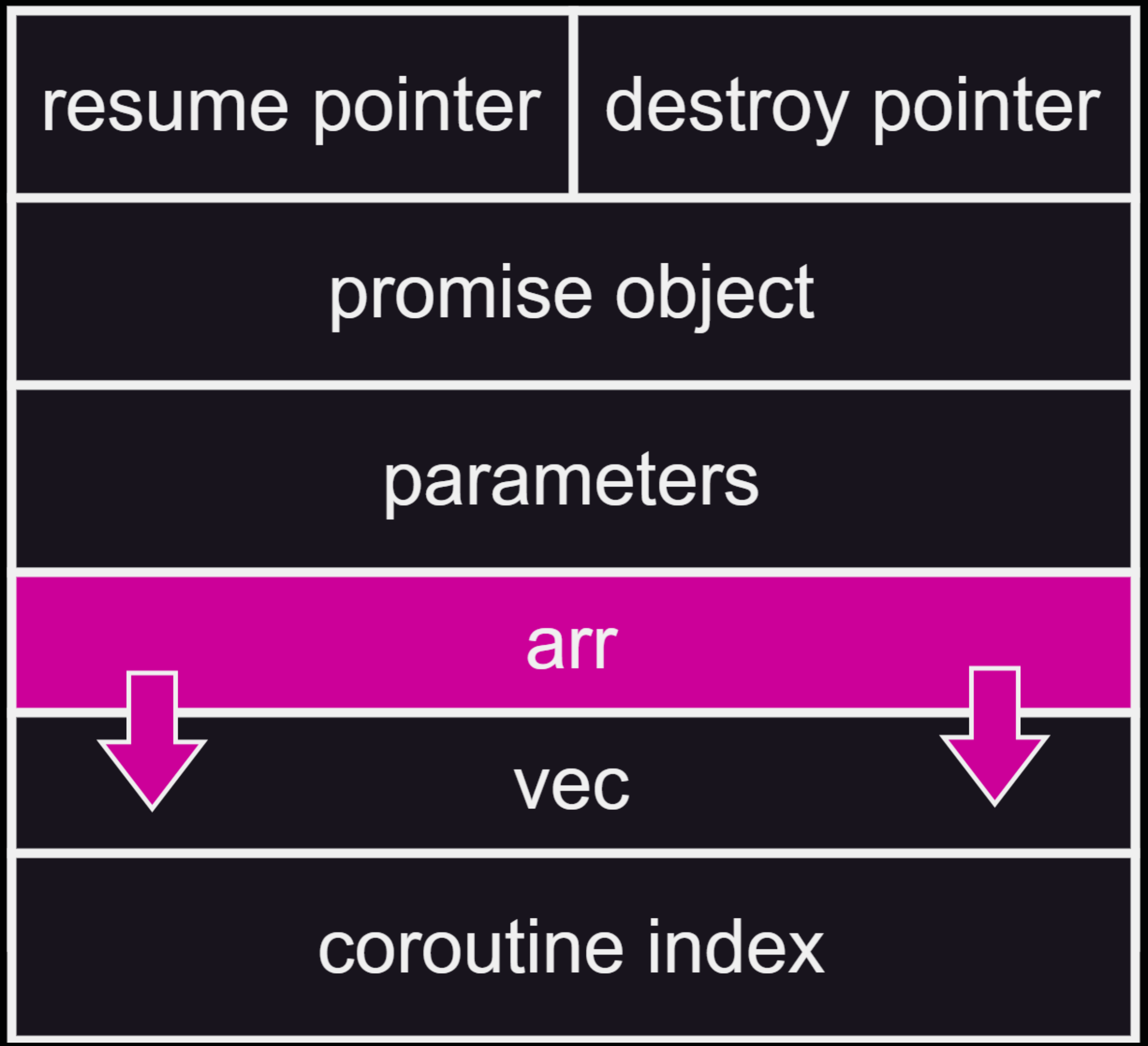

```
task coro(char* arg)
{
★  //SP1//
   initial_suspend();
   vector<int> vec;
   vec.push_back(1);
   co_await some_task;
★  //SP2//
   co_await some_task;
★  //SP3//
   <FREE ALL VARIABLES>
   final_suspend();
}
```

Advanced DOAs

g++-14
clang++-19 -O0

```
task coro(char* arg)
{
   //SP1//
   char arr[10];
   vector<int> vec;
   co_await some_task;
   //SP2//
}
```

No reordering



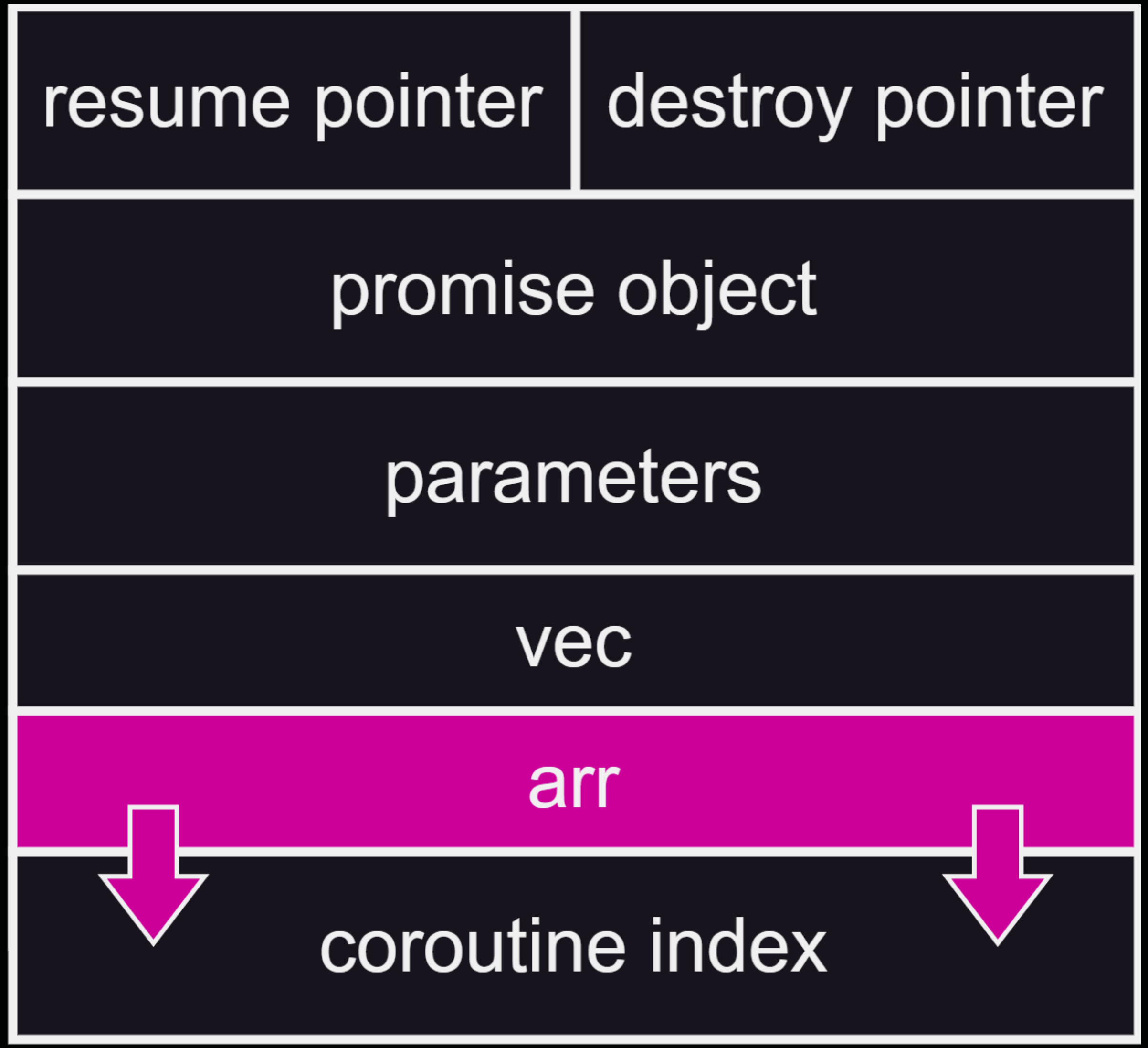
- Some compilers do variable reordering at –O3, but they do it funny

Advanced DOAs

g++-14
clang++-19 *-O0*

```
task coro(char* arg)
{
   //SP1//
   vector<int> vec;
   char arr[10];
   co_await some_task;
   //SP2//
}
```

No reordering

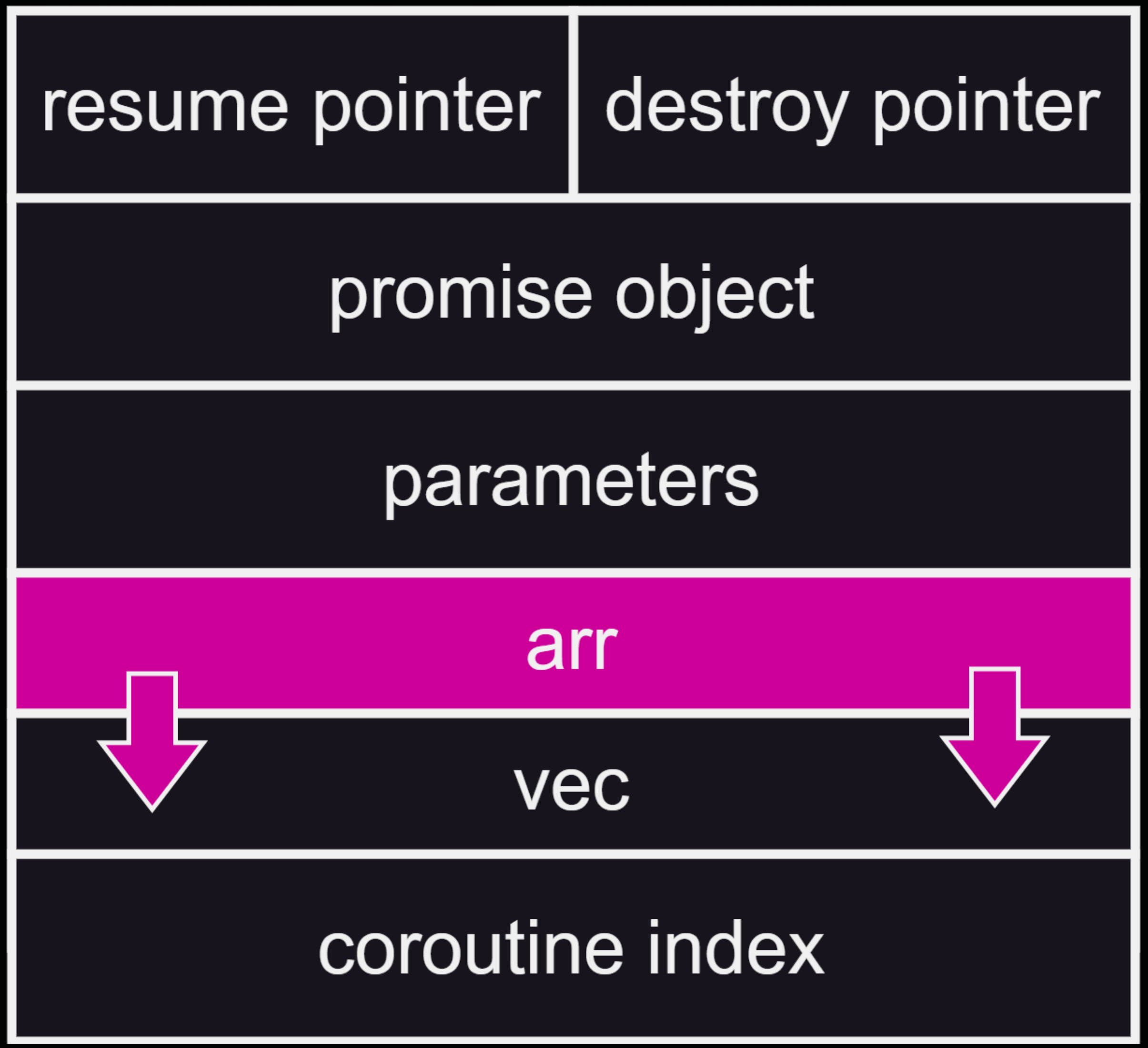


Advanced DOAs

g++-14 -O3

```
task coro(char* arg)
{
   //SP1//
   char arr[10];
   vector<int> vec;
   co_await some_task;
   //SP2//
}
```

No reordering

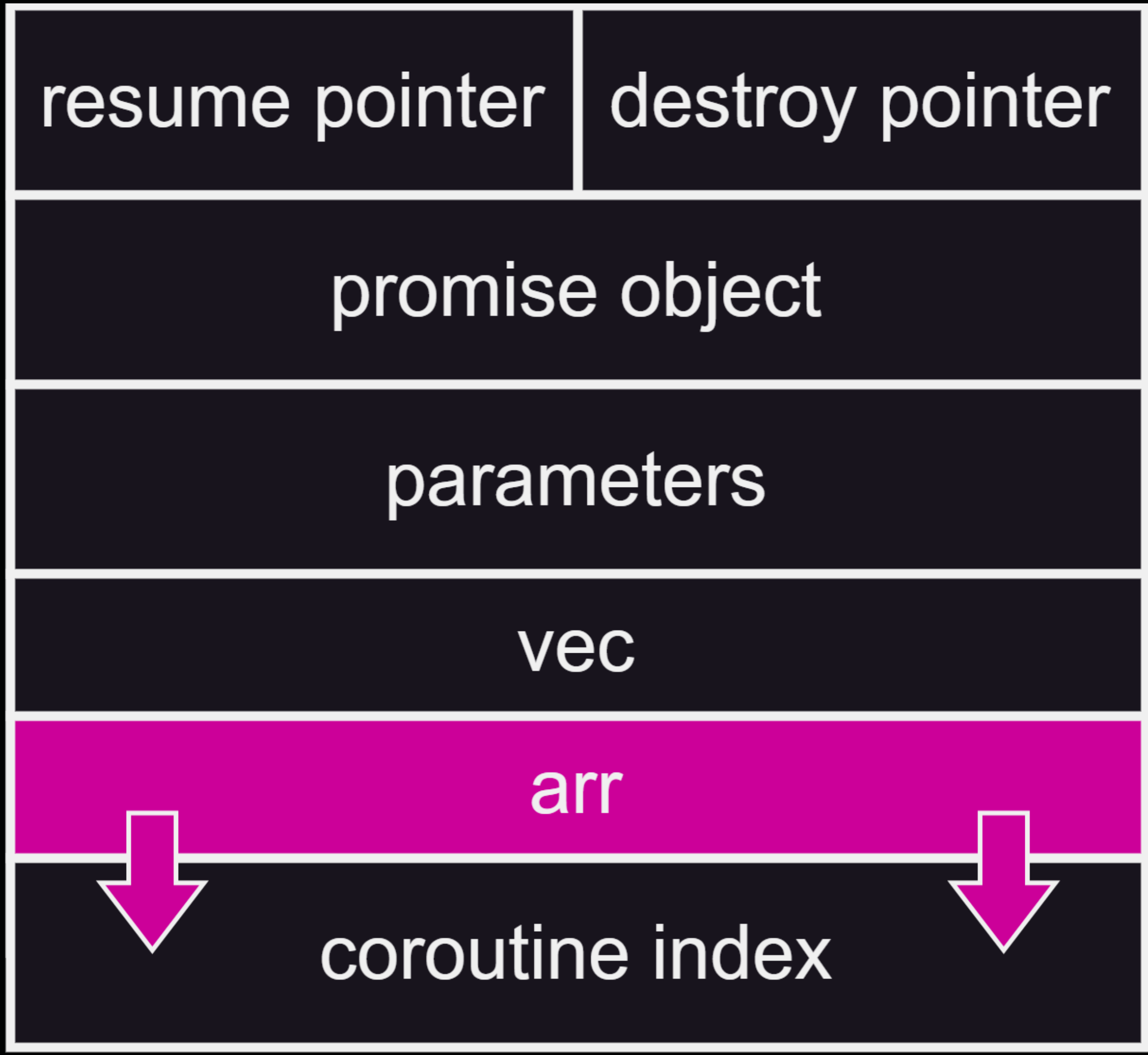


Advanced DOAs

g++-14 -O3

```
task coro(char* arg)
{
   //SP1//
   vector<int> vec;
   char arr[10];
   co_await some_task;
   //SP2//
}
```

No reordering

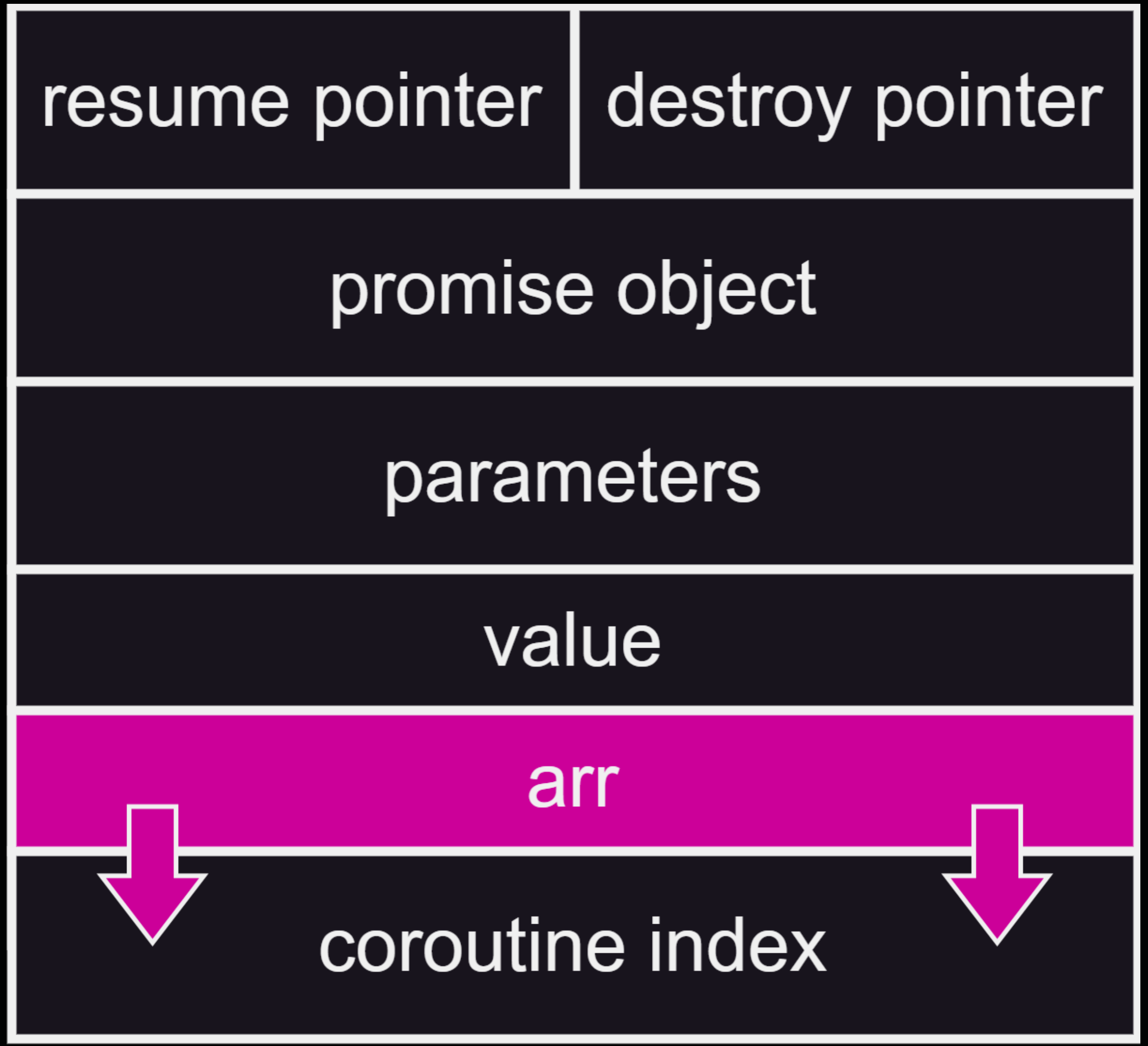


Advanced DOAs

clang++-19 -O3

```
task coro(char* arg)
{
   //SP1//
   char arr[10];
   int value;
   co_await some_task;
   //SP2//
}
```

Safe reordering



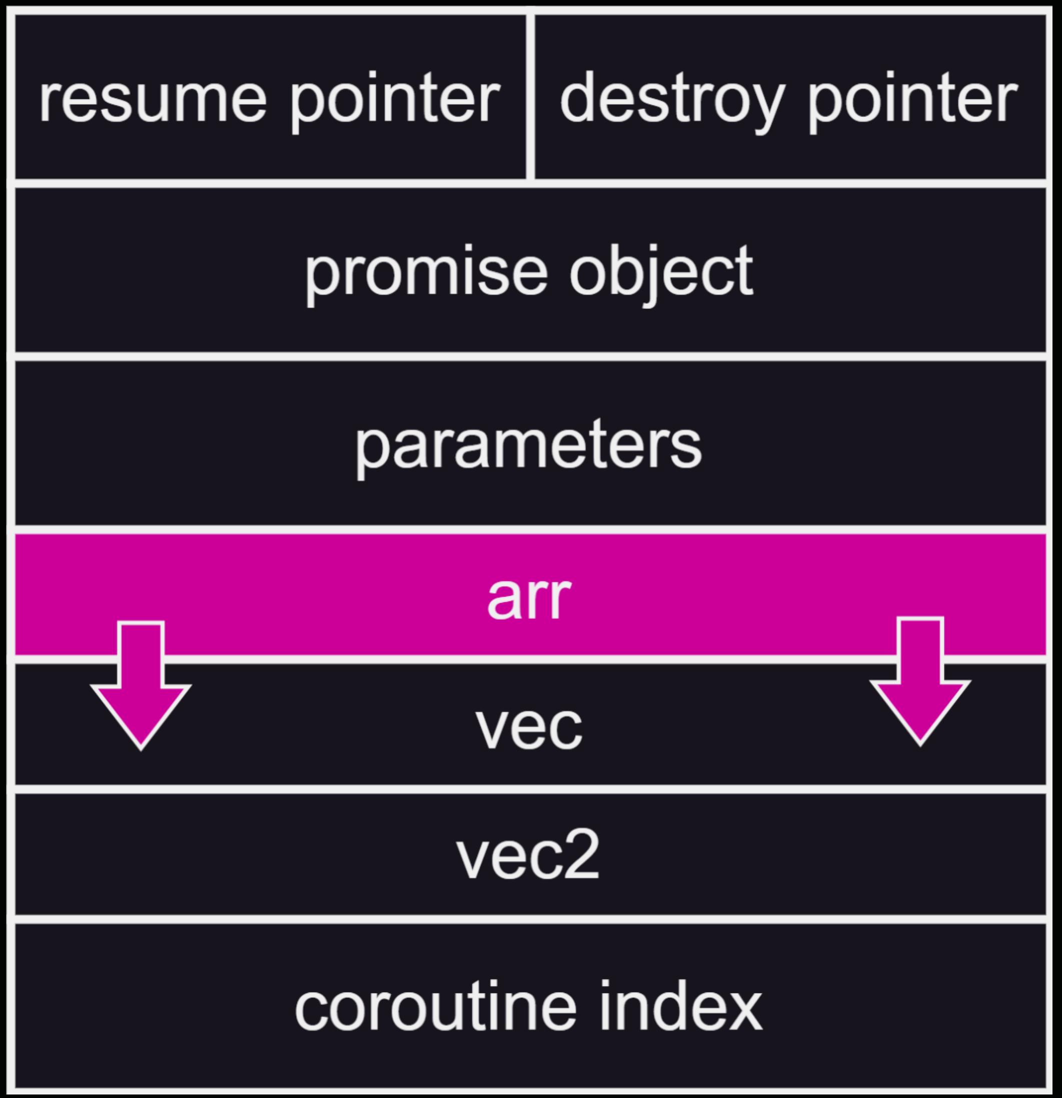
Advanced DOAs

clang++-19 -O3

```
task coro(char* arg)
{
   //SP1//
   vector<int> vec;
   vector<int> vec2;
   char arr[10];
   co_await some_task;
   //SP2//
}
```

- The reordering rules for clang are a bit messed up

Ok, that was weird



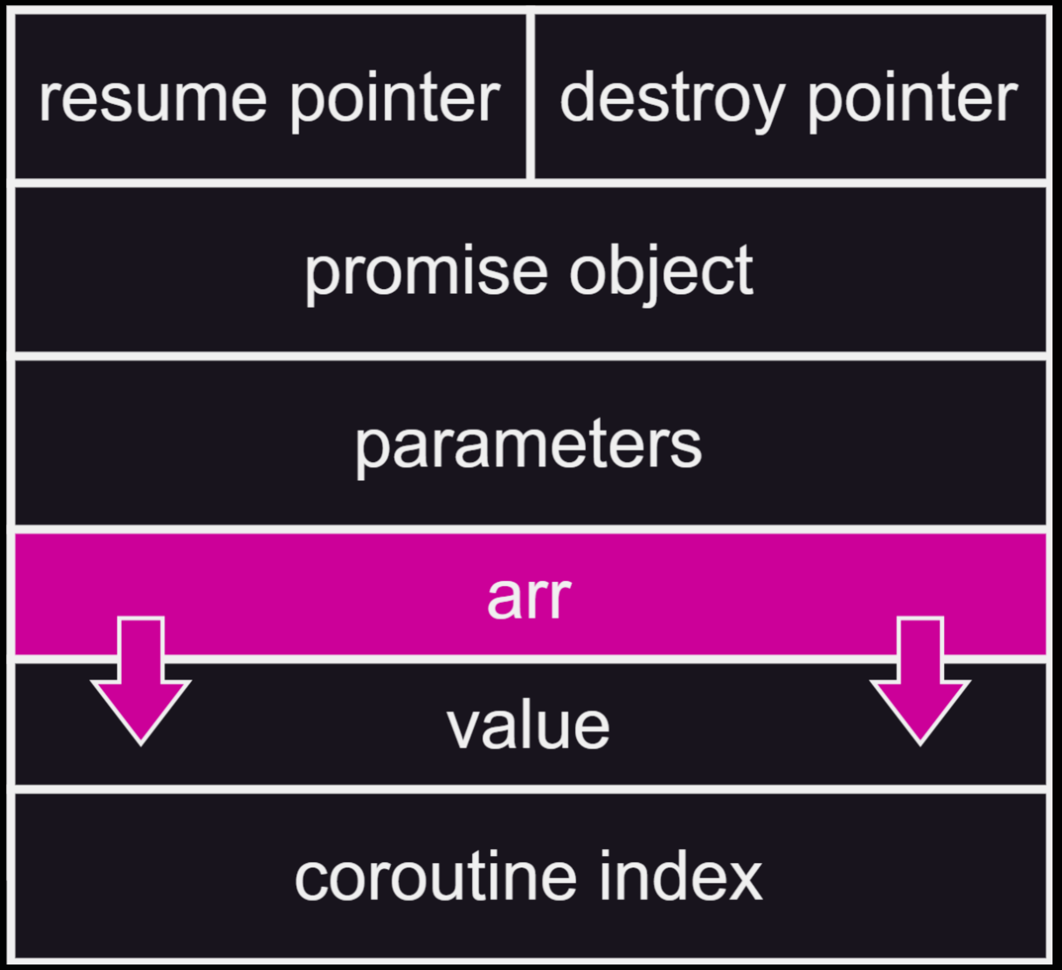
Advanced DOAs

clang++-19 *-O3*

```
task coro(char* arg)
{
   //SP1//
   int value;
   char arr[10];
   co_await some_task;
   //SP2//
}
```

- The reordering rules for clang are a bit messed up

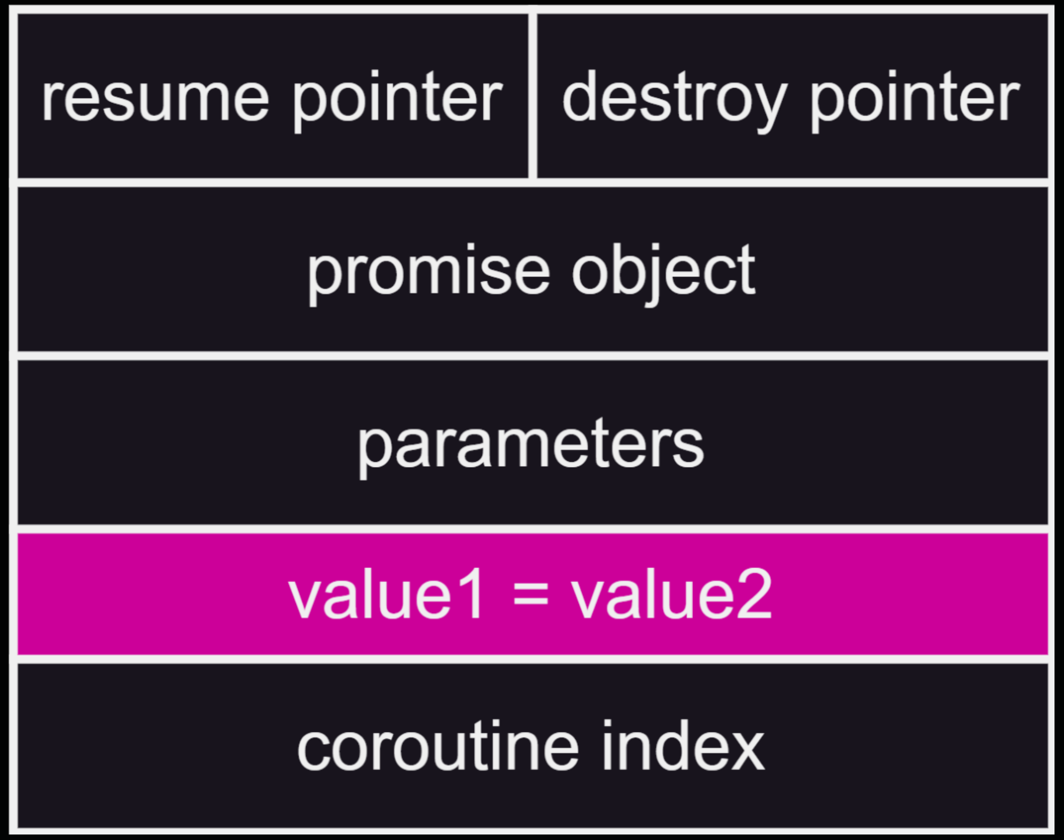
Ok, that was weird



Advanced DOAs

- The compiler saves space in the frame by reusing addresses for SP-exclusive variables.
- Variables can be ‘reused’ wrongly in other SPs.

```
task coro(char* arg)
{
   //SP1//
   int val1;
   co_await some_task;
   //SP2//
   int val2;
}
```

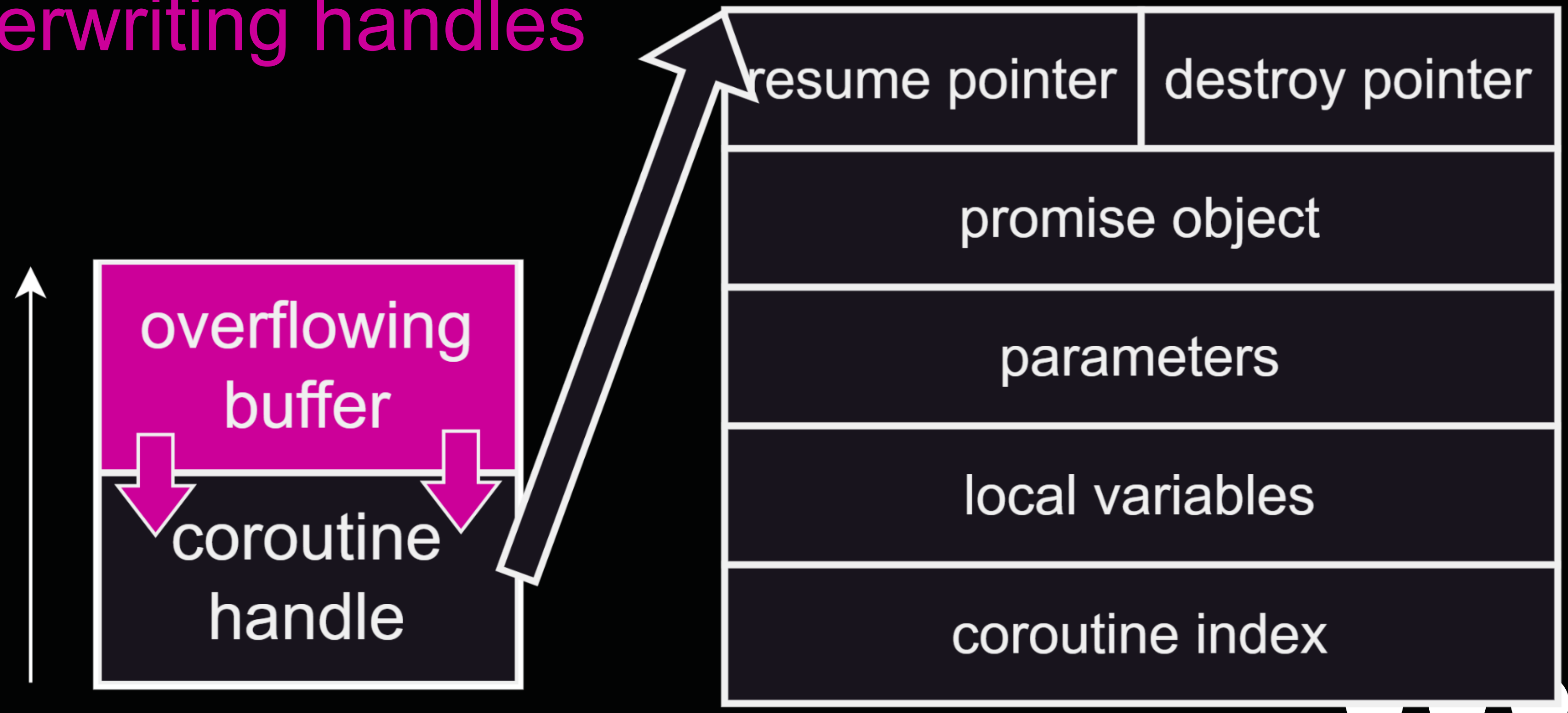


Revisiting the Threat model

- Launching a DOA (and other CFOP attacks) requires either frame manipulation or frame injection
- Options:
  1. Arbitrary memory write

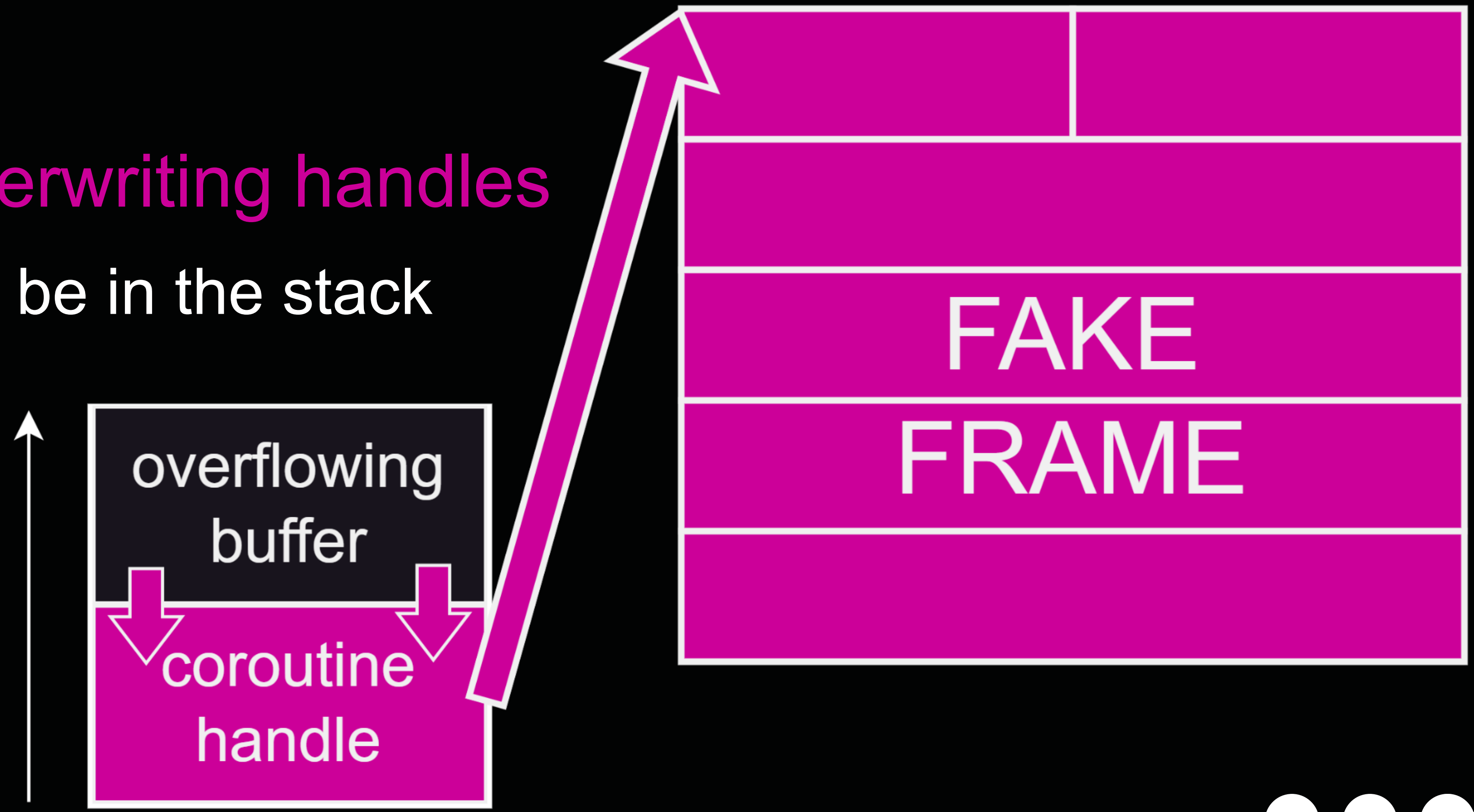
Revisiting the Threat model

- Launching a DOA (and other CFOP attacks) requires either frame manipulation or frame injection
- Options:
  1. Arbitrary memory write
  2. Stack-based overflow overwriting handles



Revisiting the Threat model

- Launching a DOA (and other CFOP attacks) requires either frame manipulation or frame injection
- Options:
  1. Arbitrary memory write
  2. Stack-based overflow overwriting handles
     - The new frame could even be in the stack



Revisiting the Threat model

- Launching a DOA (and other CFOP attacks) requires either frame manipulation or frame injection
- Options:

1. Arbitrary memory write
2. Stack-based overflow overwriting handles
3. Stack-based overflow inside the coroutine
    - Leverage no reordering… and no stack canaries!! :)
    - Parameters can always overflow almost the whole frame
    - At a minimum, you can always overwrite the coroutine index
    - In ptmalloc, you can overwrite frames further down the heap

Revisiting the Threat model

- Launching a DOA (and other CFOP attacks) requires either frame manipulation or frame injection
- Options:

1. Arbitrary memory write
2. Stack-based overflow overwriting handles
3. Stack-based overflow inside the coroutine
4. Heap-based overflow overwriting subsequent frames

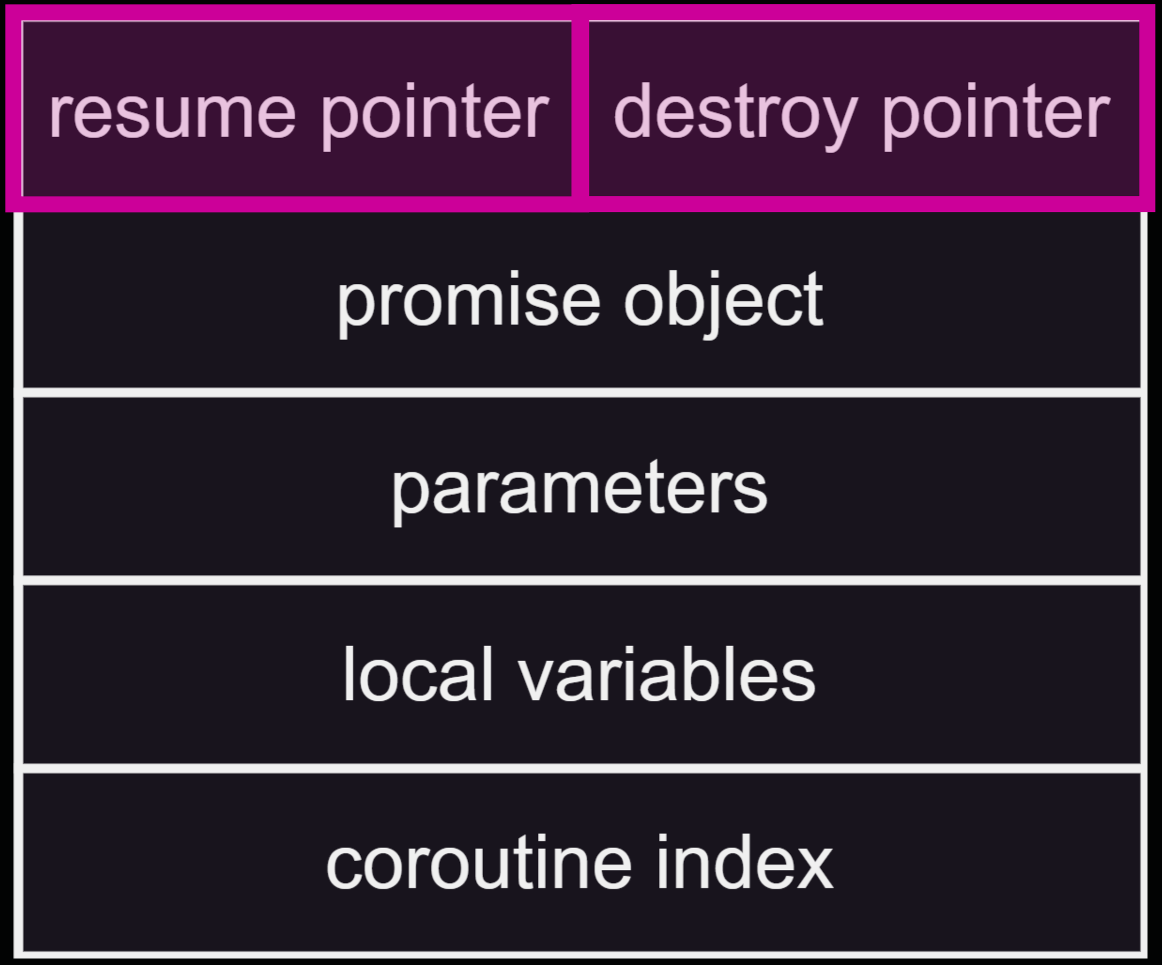
Revisiting the Threat model

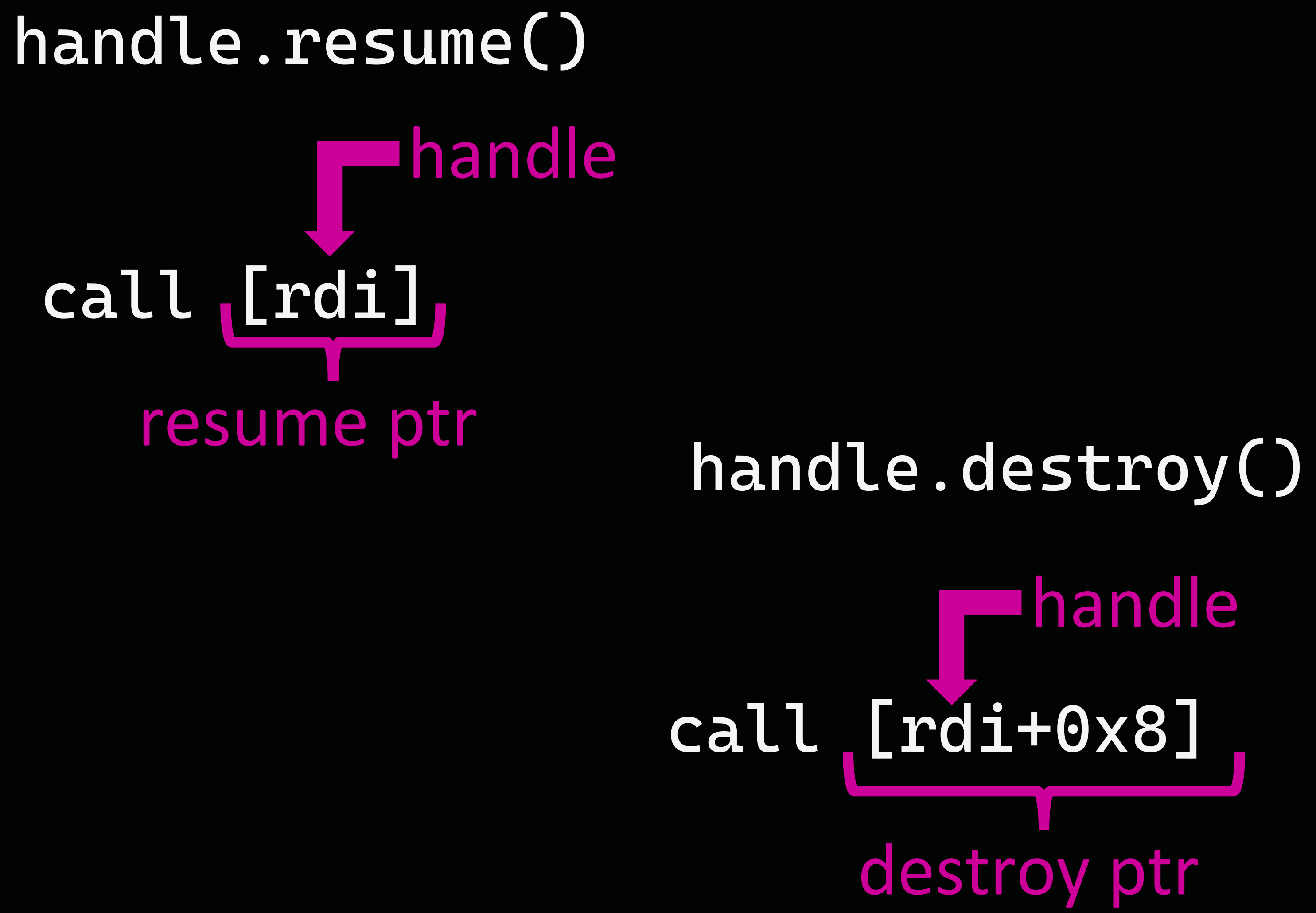
- Launching a DOA (and other CFOP attacks) requires either frame manipulation or frame injection
- Options:

1. Arbitrary memory write
2. Stack-based overflow overwriting handles
3. Stack-based overflow inside the coroutine
4. Heap-based overflow overwriting subsequent frames
5. Any combination of the previous or other bugs
   - DOAs -> arbitrary free() -> allocate one frame on top of the next one

Observations

- The resume and destroy pointers in the frame can be hijacked





Threat model

- Intel CET
- LLVM CFI (icall)
- LLVM KCFI
- Safe Dispatch
- Control Flow Guard
- MCFI/ piCFI
- ReCFI
- Path Armor
- VfGuard
- CFIXX
- PittyPat
- VTrust
- Typro

CFI Defenses in place

Coarse-grained

Finer-grained

- Intel CET
- LLVM CFI (icall)
- LLVM KCFI
- Safe Dispatch
- Control Flow Guard
- MCFI/ piCFI
- ReCFI
- Path Armor
- VfGuard
- CFIXX
- PittyPat
- VTrust
- Typro

(Disclaimer: This classification is a bit subjective)

Threat model

Intel CET

LLVM CFI (icall)

LLVM KCFI

Safe Dispatch

Control Flow Guard

MCFI/ piCFI

ReCFI

Path Armor

VfGuard

CFIXX

PittyPat

VTrust

Typro

Restrictions introduced

- No return address hijacking

Threat model

Intel CET

LLVM CFI (icall)

LLVM KCFI

Safe Dispatch

Control Flow Guard

MCFI/ piCFI

ReCFI

Path Armor

VfGuard

CFIXX

PittyPat

VTrust

Typro

Restrictions introduced

- No return address hijacking
- No vptr hijacking

Threat model

…But fine-grained schemes will protect every indirect jump, right?

Threat model

- The two problems with fine-grained CFI:

  1. Most fine-grained schemes are academic, and do not support modern features (like coroutines)

Threat model

Breaks with coroutines

- Intel CET
- LLVM CFI (icall)
- LLVM KCFI
- Safe Dispatch
- Control Flow Guard
- MCFI/ piCFI
- ReCFI
- Path Armor
- VfGuard
- CFIXX
- PittyPat
- VTrust
- Typro

Threat model

- The two problems with fine-grained CFI:

1. Most fine-grained schemes are academic, and do not support modern features (like coroutines)

2. New programming languages features break CFI, for which they were not prepared to deal with

Threat model

Failed to keep up-to-date with coroutines

- Control Flow Guard
- Intel CET
- LLVM CFI (icall)
- LLVM KCFI
- Safe Dispatch

- These fine-grained schemes do not generate instrumentation code for coroutine *resume* and *destroy* pointers.

(To be fair, SafeDispatch is for virtual calls, so unlike LLVM CFI, that was kinda expected)

Threat model

- The two problems with fine-grained CFI:

  1. Most fine-grained schemes are academic, and do not support modern features (like coroutines)

  2. New programming languages features break CFI, for which they were not prepared to deal with

- In the future, new programming features may be added that break CFI as well. New possibilities :)
- CFI cannot be static, it needs to evolve

black hat BRIEFINGS

Threat model

Intel CET

LLVM CFI (icall)

LLVM KCFI

Safe Dispatch

Control Flow Guard

MCFI/ piCFI

ReCFI

Path Armor

VfGuard

CFIXX

PittyPat

VTrust

Typro

Restrictions introduced

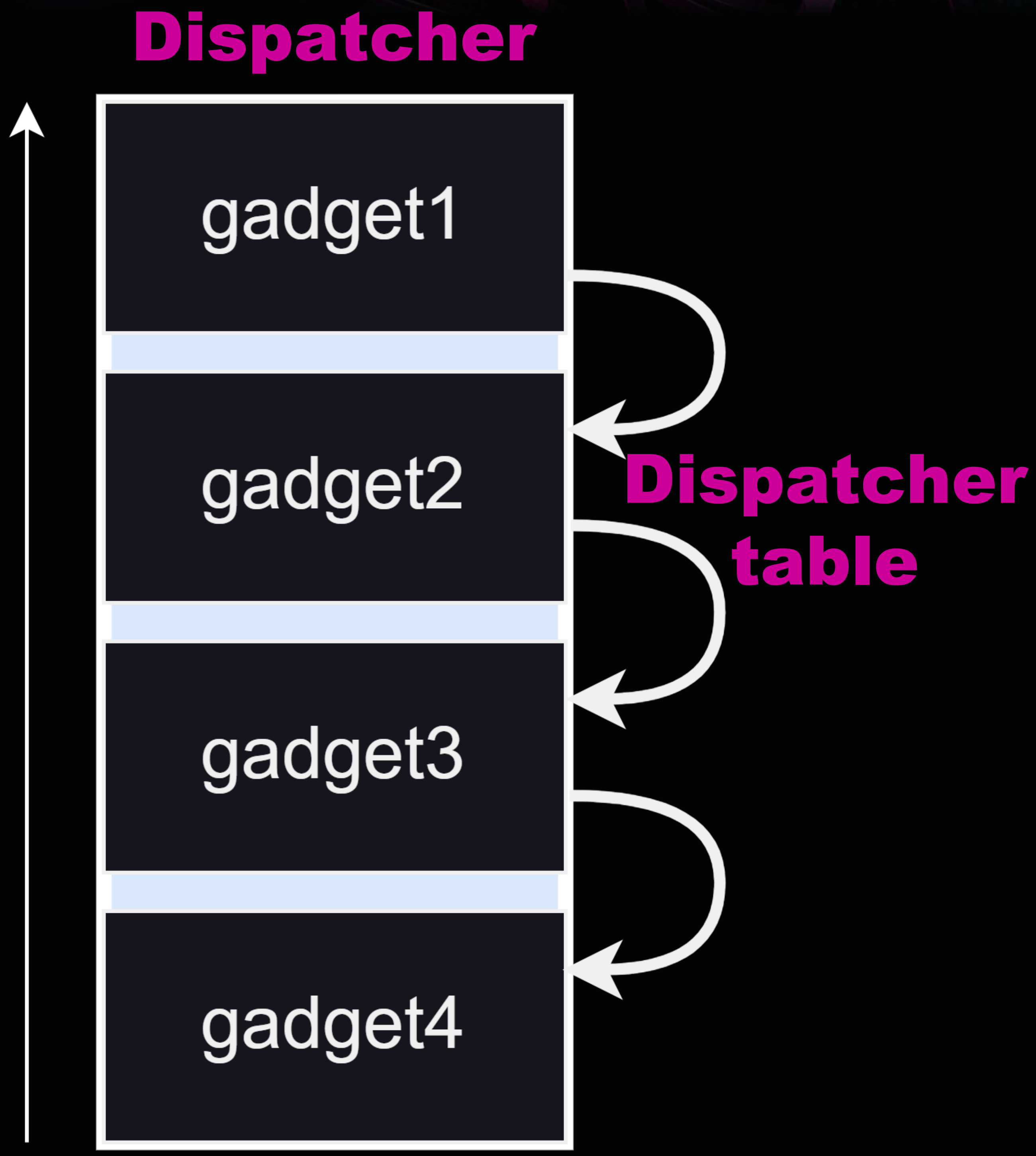
- No return address hijacking
- No vptr hijacking
- Only jump to the beginning of functions

Control Flow Hijacking

- What we have right now:
  - 1 arbitrary call with 0 arguments

- What we wish to have:
  - Infinitely many arbitrary calls with arbitrary arguments

Approach



Gadgets

```
…
inc rax
ret

…
pop rdi
ret

…
pop rsi
ret

…
mov rdx, rcx
ret
```

Control Flow Hijacking

- You do not need to overwrite the pointers for control flow hijacking! Just a *CFP*.
- A *Controlled Frame Pointer* (**CFP**) is any program pointer that indirectly or directly leads to control flow hijacking
- Also! There could be function pointers inside the frame, go for DOAs :)

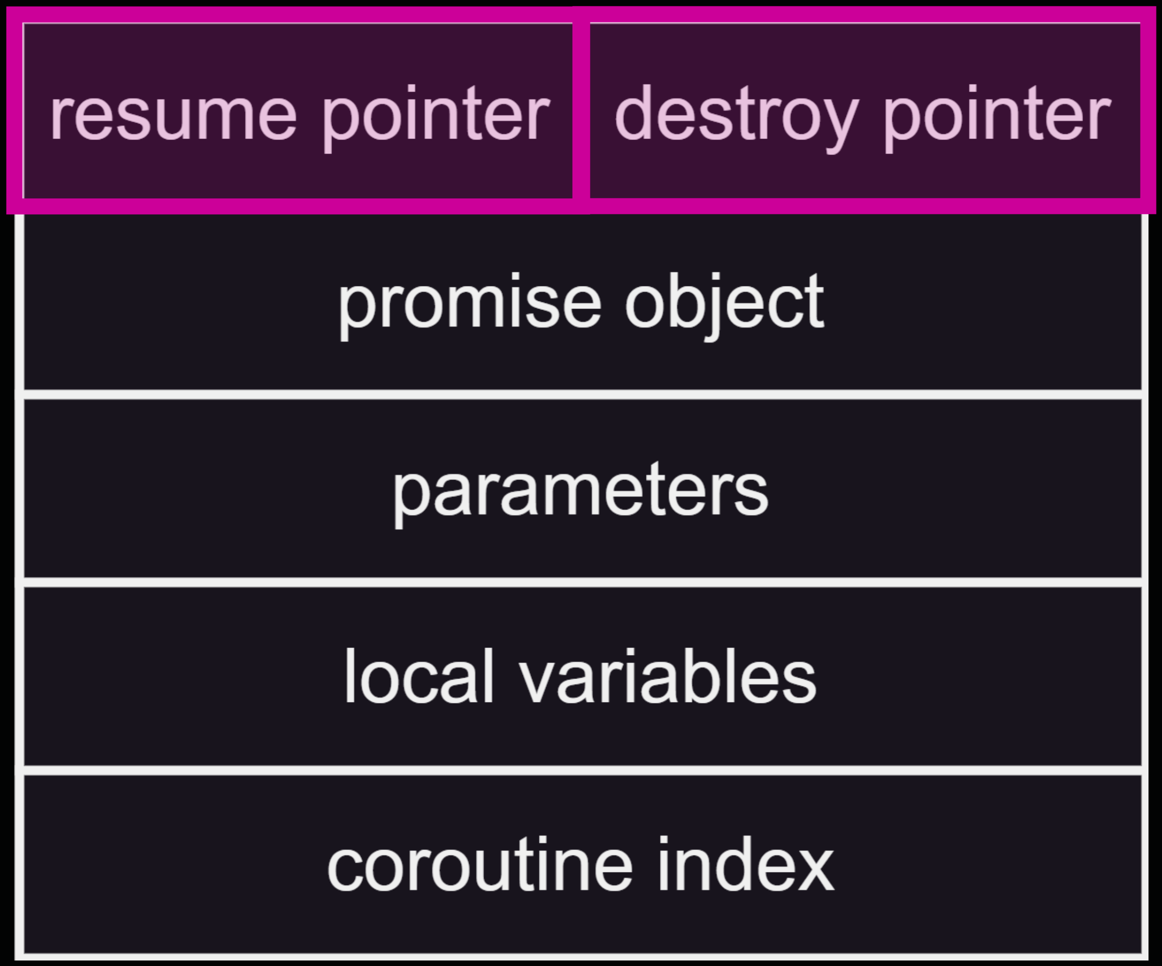
Control Flow Hijacking

- Sources of CFPs

    1. Overwriting the *resume* or *destroy* pointers

Control Flow Hijacking

- Sources of CFPs
  1. Overwriting the *resume* or *destroy* pointers

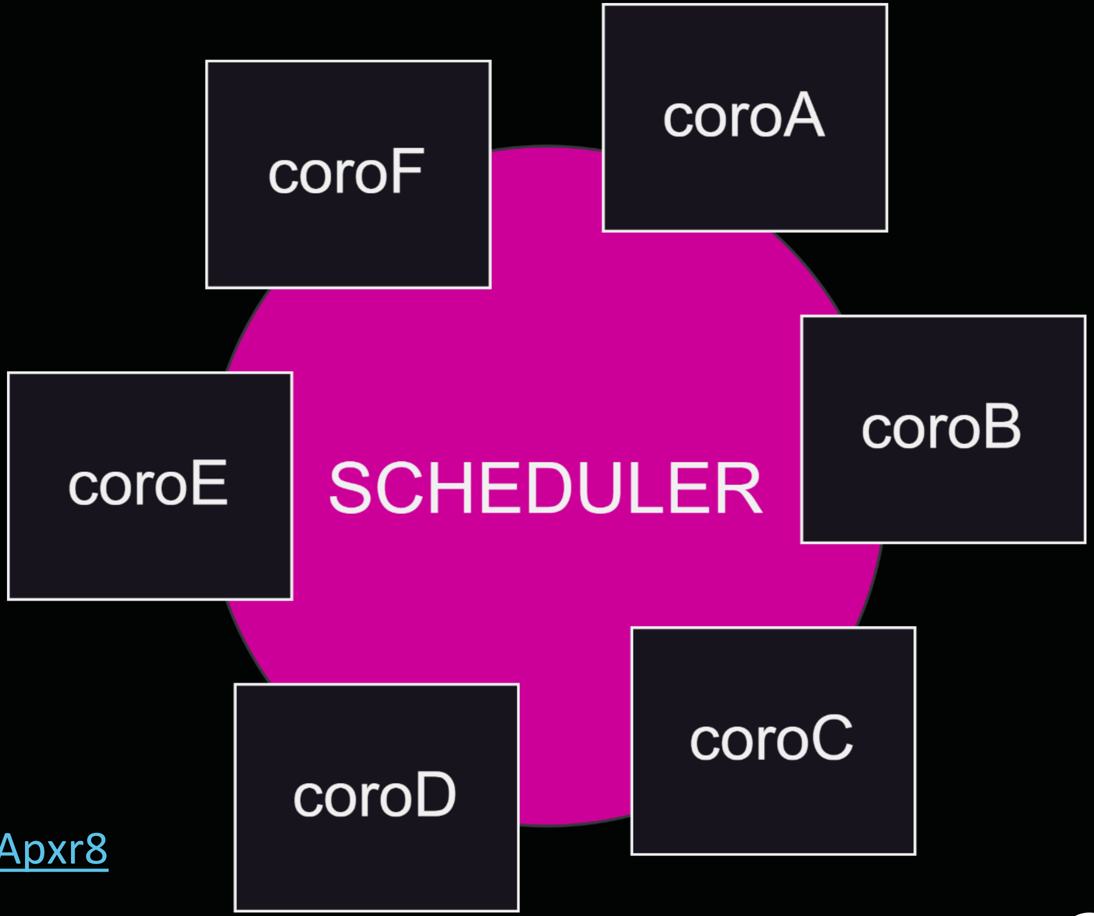


Control Flow Hijacking

- Sources of CFPs
  1. Overwriting the *resume* or *destroy* pointers
  2. Overwriting a coroutine handle. But where?
     - Schedulers
       - Look for databases

         ScyllaDB
       - Or browsers

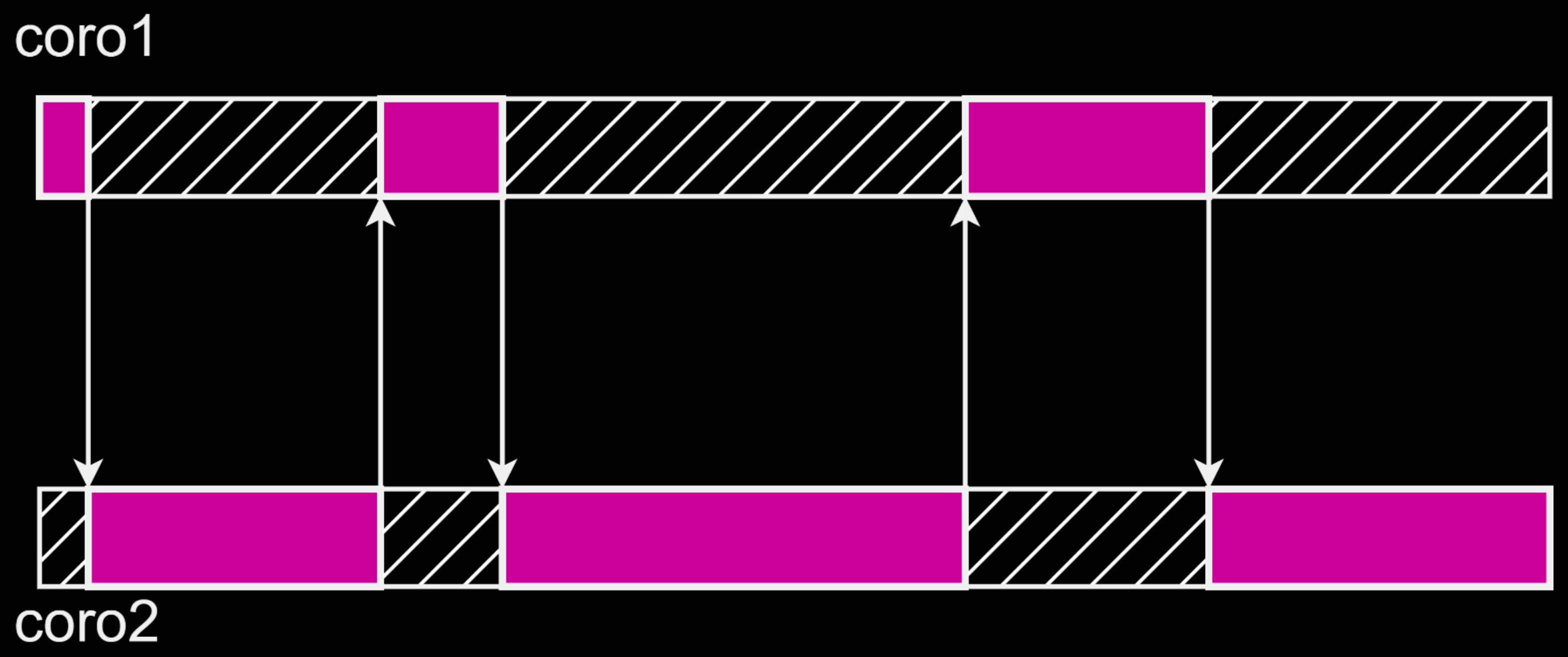
Ladybird Chromium



https://issues.chromium.org/issues/40251667
https://groups.google.com/a/chromium.org/g/cxx/c/ehMerLApxr8

Control Flow Hijacking

- Sources of CFPs
  1. Overwriting the *resume* or *destroy* pointers
  2. Overwriting a coroutine frame. But where?
  3. Overwriting internal frame coroutine pointers



The awesome world of Continuations

- Coroutines need to know how to resume the next coroutine

We know this

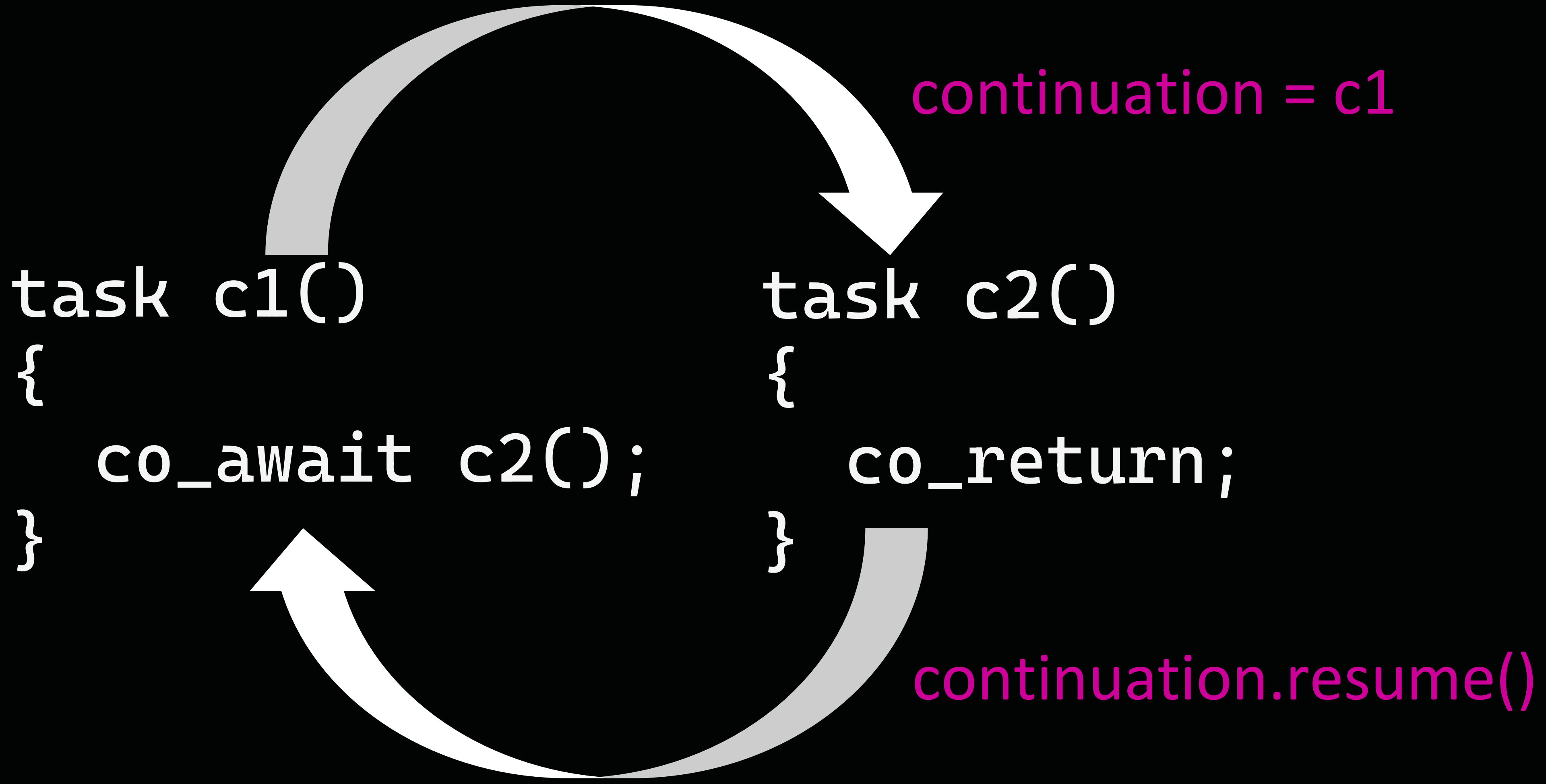
```
task c1()
{
   co_await c2();
}
```

```
task c2()
{
   co_return;
}
```

HOW??

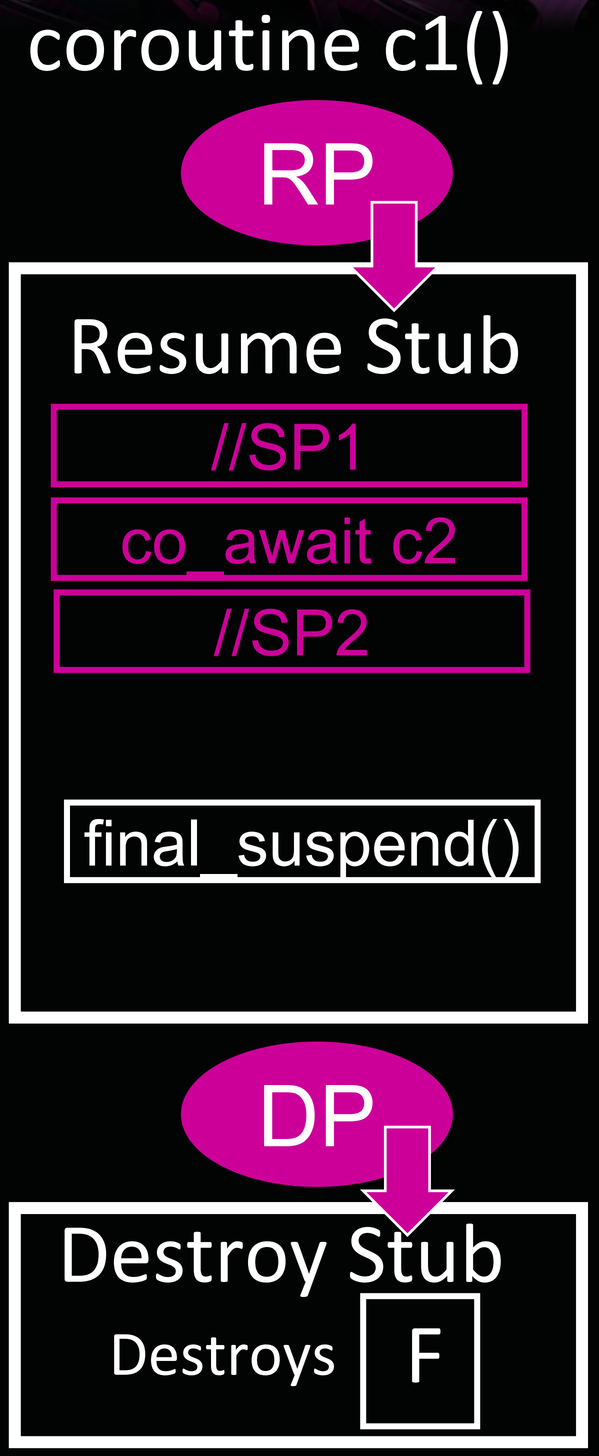
The awesome world of Continuations

- Coroutines need to know how to resume the next coroutine
- Coroutines set continuation points for the next coroutine



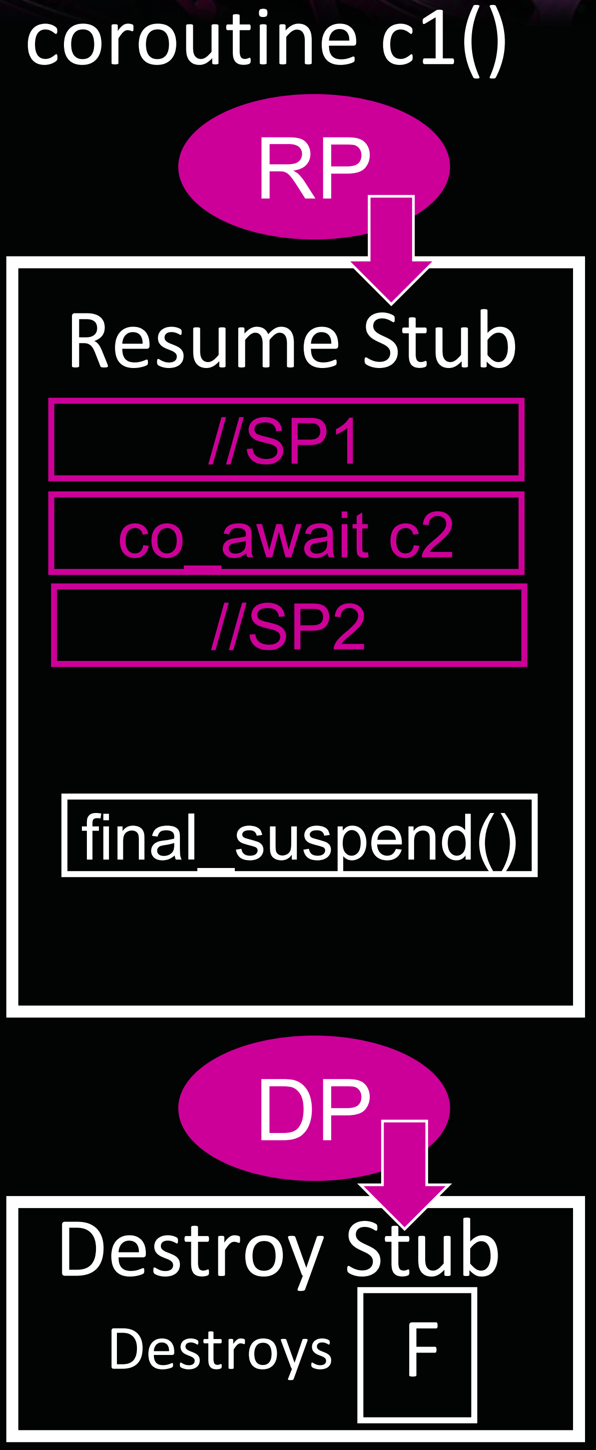
The awesome world of Continuations

```
task c1()
{
   //SP1
   co_await c2();
   //SP2
}

task c2()
{
   co_return;
}
```



The awesome world of Continuations

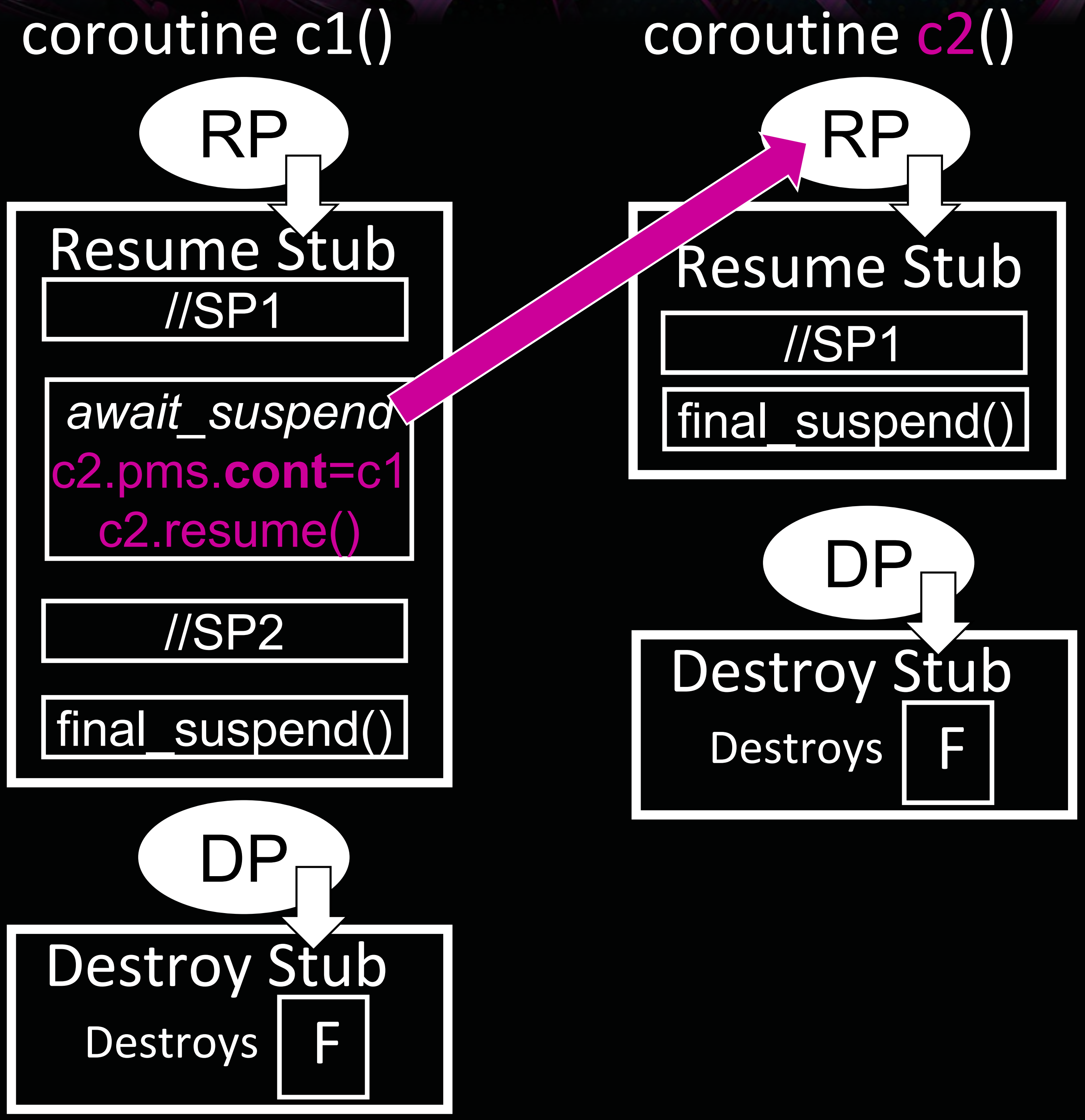
```
task c1()
{
   //SP1
   co_await c2();
   //SP2
}

task c2()
{
   co_return;
}
```




```
task
{
   handle h;
   struct promise_type
   {
      int return_value;
      suspend_always initial_suspend();
      suspend_always final_suspend();
   };
   struct awaiter
   {
      bool await_ready();
      void await_suspend(h);
      void await_resume();
   }
}
```

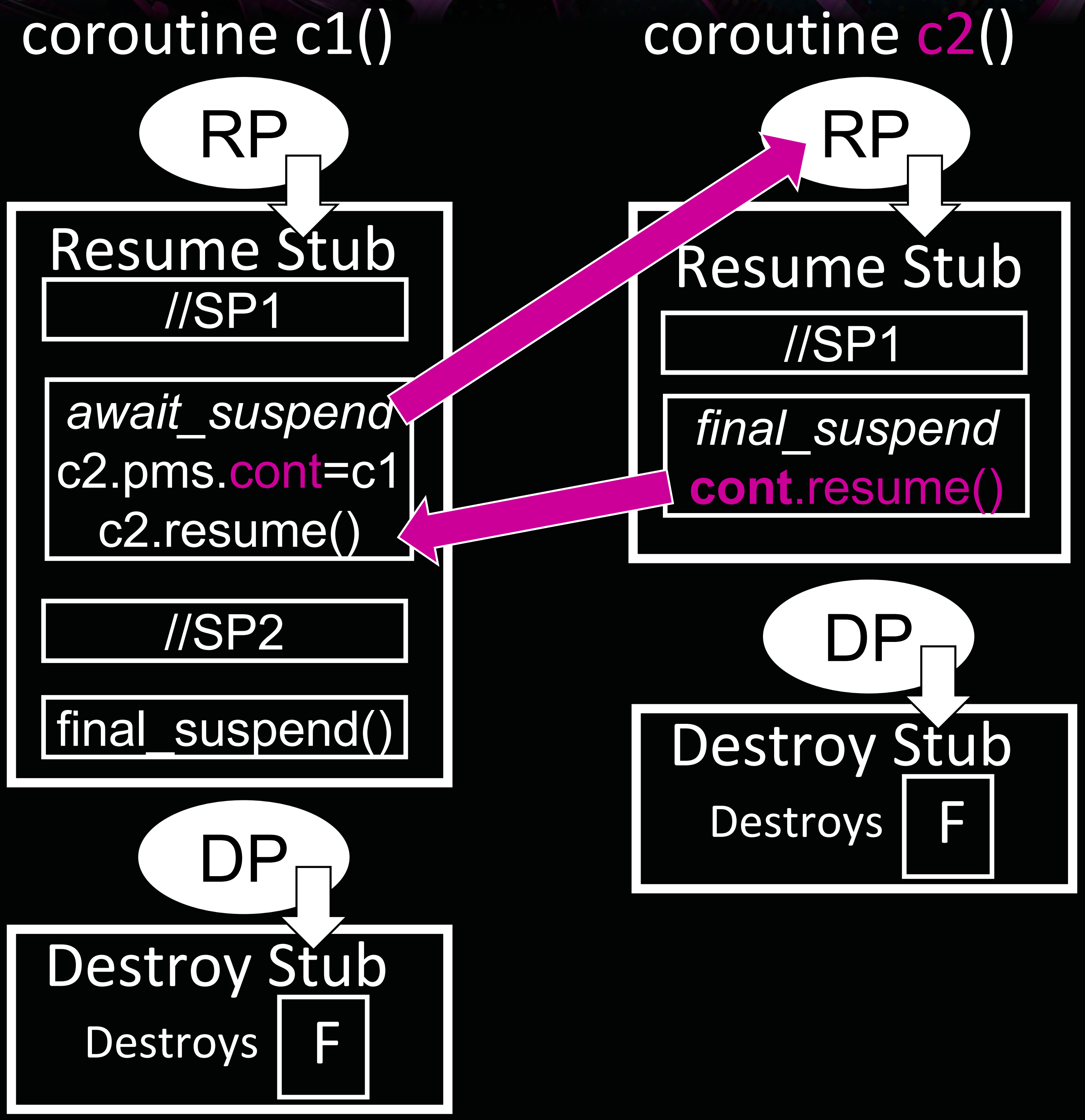
The awesome world of Continuations

```
task c1()
{
   //SP1
   co_await c2();
   //SP2
}

task c2()
{
   co_return;
}
```



The awesome world of Continuations

```
task c1()
{
   //SP1
   co_await c2();
   //SP2
}


task c2()
{
   co_return;
}
```



The awesome world of Continuations

```
task c1()
{
   //SP1
   co_await c2();
   //SP2
}


task c2()
{
   co_return;
}
```

```
task
{
    handle coro;
    struct promise_type
    {
        int return_value;
        suspend_always initial_suspend();
        final_awaiter final_suspend();
    };
    struct awaiter
    {
        bool await_ready();
        void await_suspend(h)
        {
            coro.continuation = h;
        }
        void await_resume();
    }
    struct final_awaiter
    {
        bool await_ready();
        void await_suspend(h)
        {
            if(continuation) continuation.resume();
        }
        void await_resume()
    }
}
```

The awesome world of Continuations

- Wait, where is c2() destroyed?

```
task c1()
{
  co_await c2();
}
```

```
task c2()
{
  co_return;
}
```

The awesome world of Continuations

- Wait, where is c2() destroyed?
  - Implicitly, right after co_await, as c2 goes out of scope

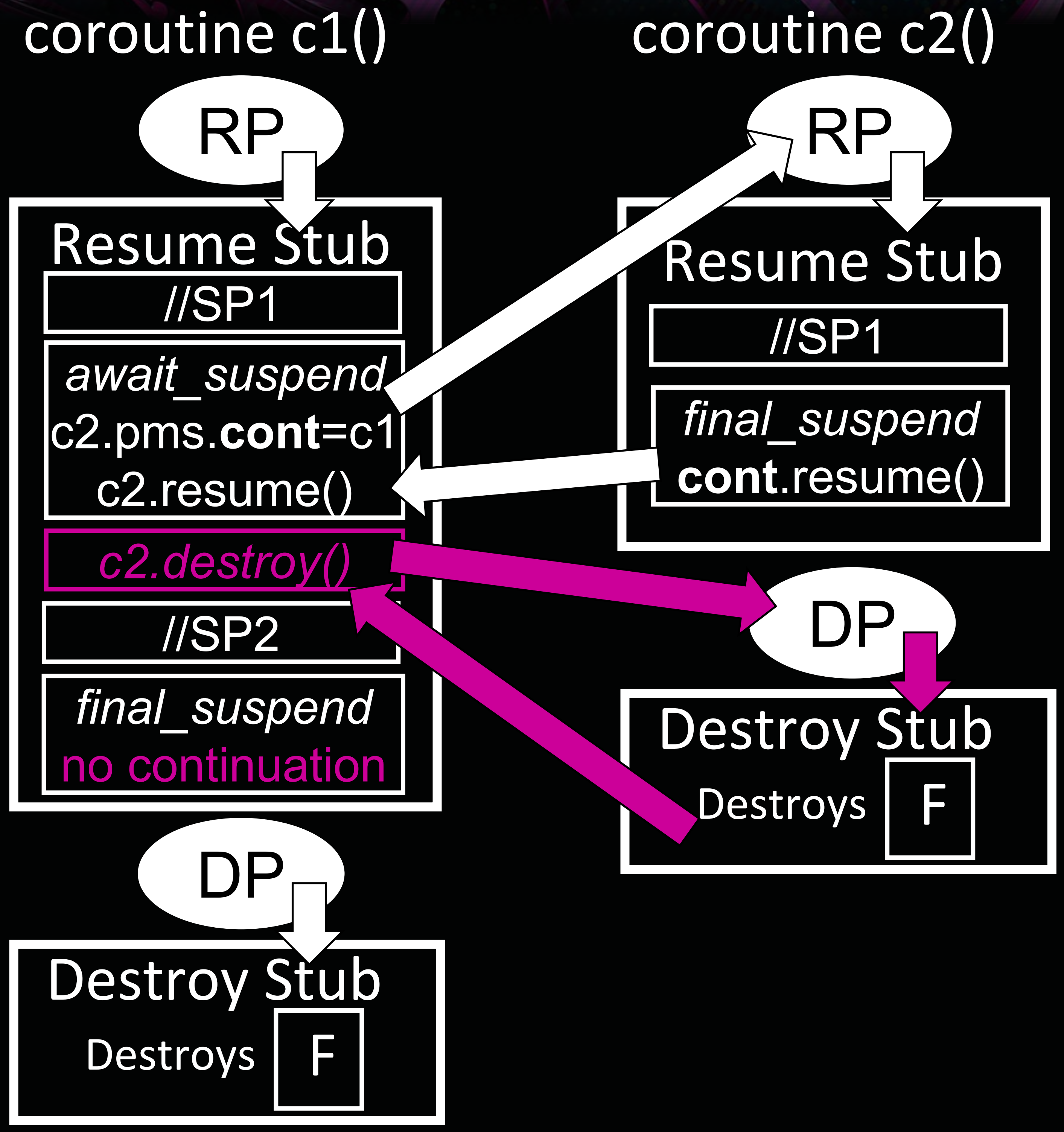
```
task c1()
{
  co_await c2();
  c2.destroy();
}
```

```
task c2()
{
  co_return;
}
```

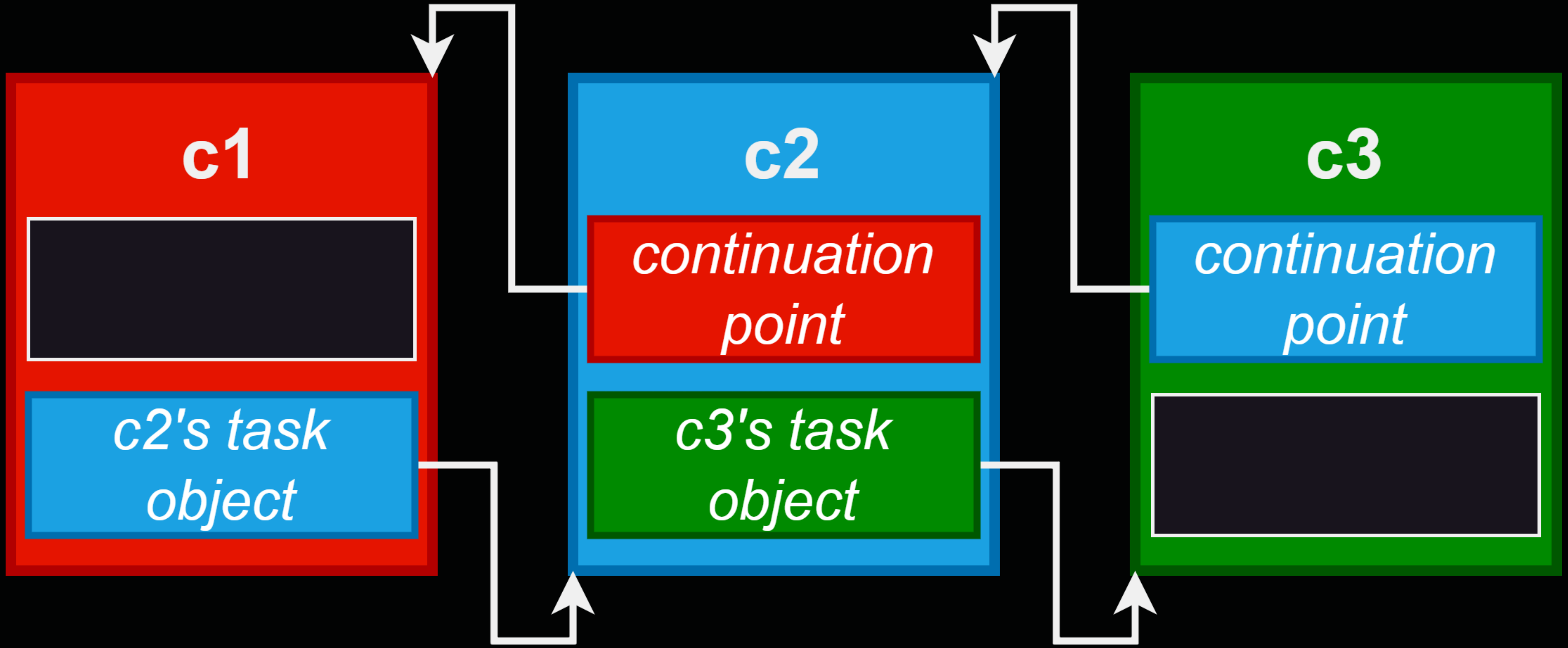
```
~task()
{
  if(coro)
   coro.destroy();
}
```

The awesome world of Continuations

```
task c1()
{
   //SP1
   co_await c2();
   //SP2
}

task c2()
{
   co_return;
}
```



The awesome world of Continuations



Infinite Coroutine Chaining

- Infinite Coroutine Chaining (**ICC**) allows you to call arbitrary functions while maintaining control flow control

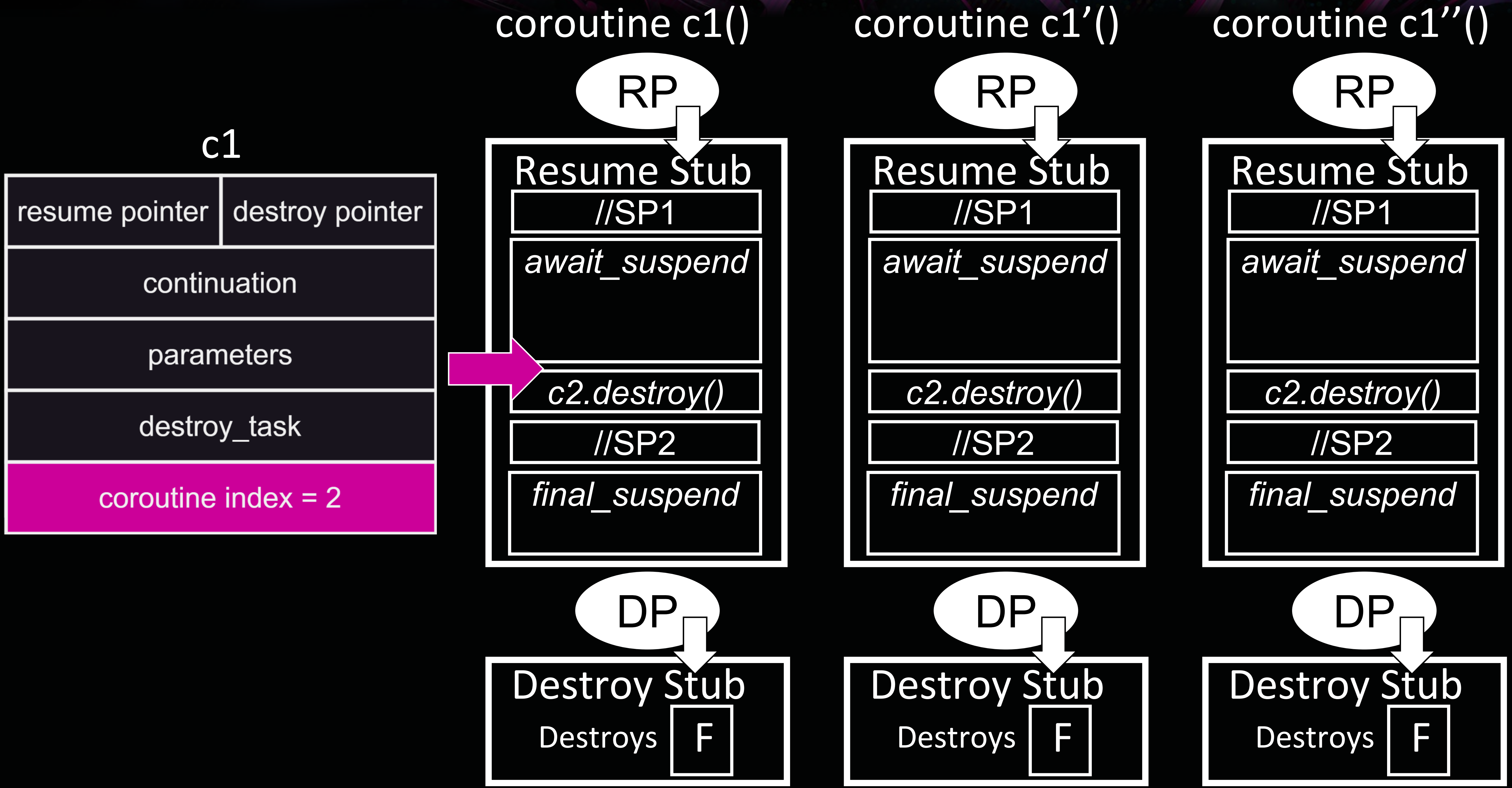
- If you have 2 CFPs, you can do ICC
  - First CFP: continuation point
  - Second CFP: task destroy

Infinite Coroutine Chaining

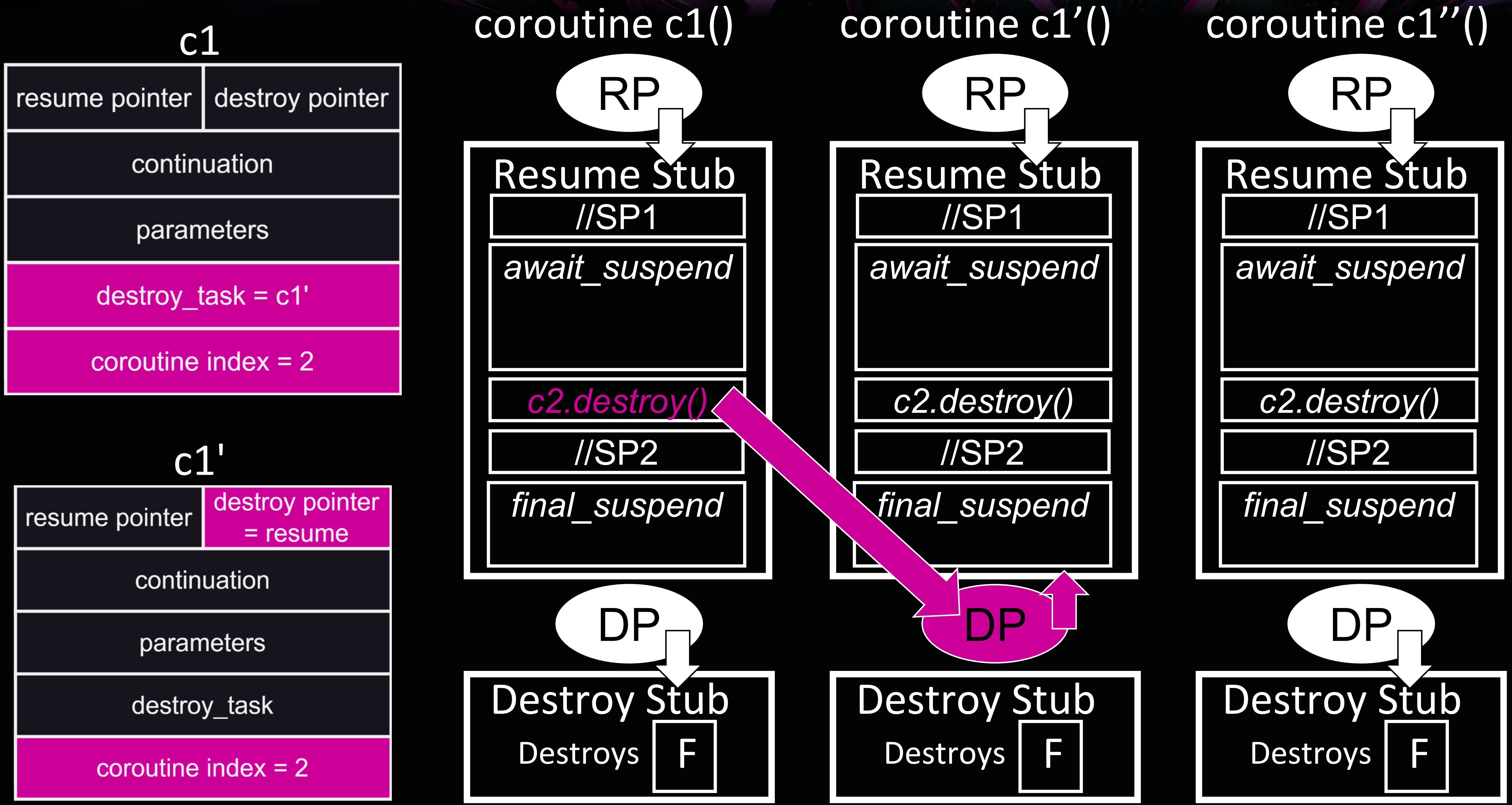
c1	c1'	c1''
resume pointer | destroy pointer	resume pointer | destroy pointer	resume pointer | destroy pointer
continuation	continuation	continuation
parameters	parameters	parameters
destroy_task	destroy_task	destroy_task
coroutine index = 2	coroutine index = 2	coroutine index = 2

trampoline frame 4		trampoline frame 3		trampoline frame 2		trampoline frame 1	
?	-	?	-	?	-	-	?

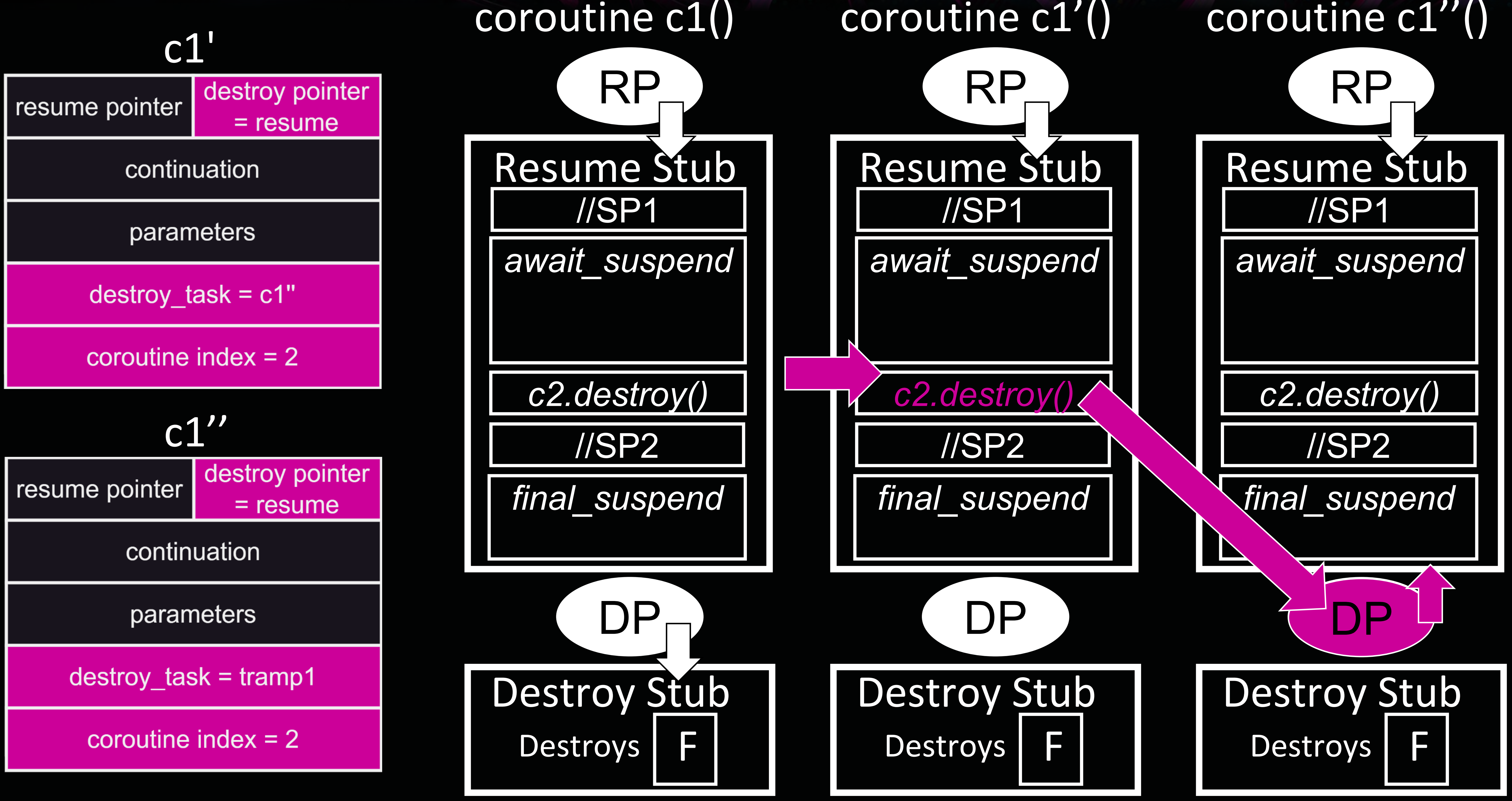
Infinite Coroutine Chaining



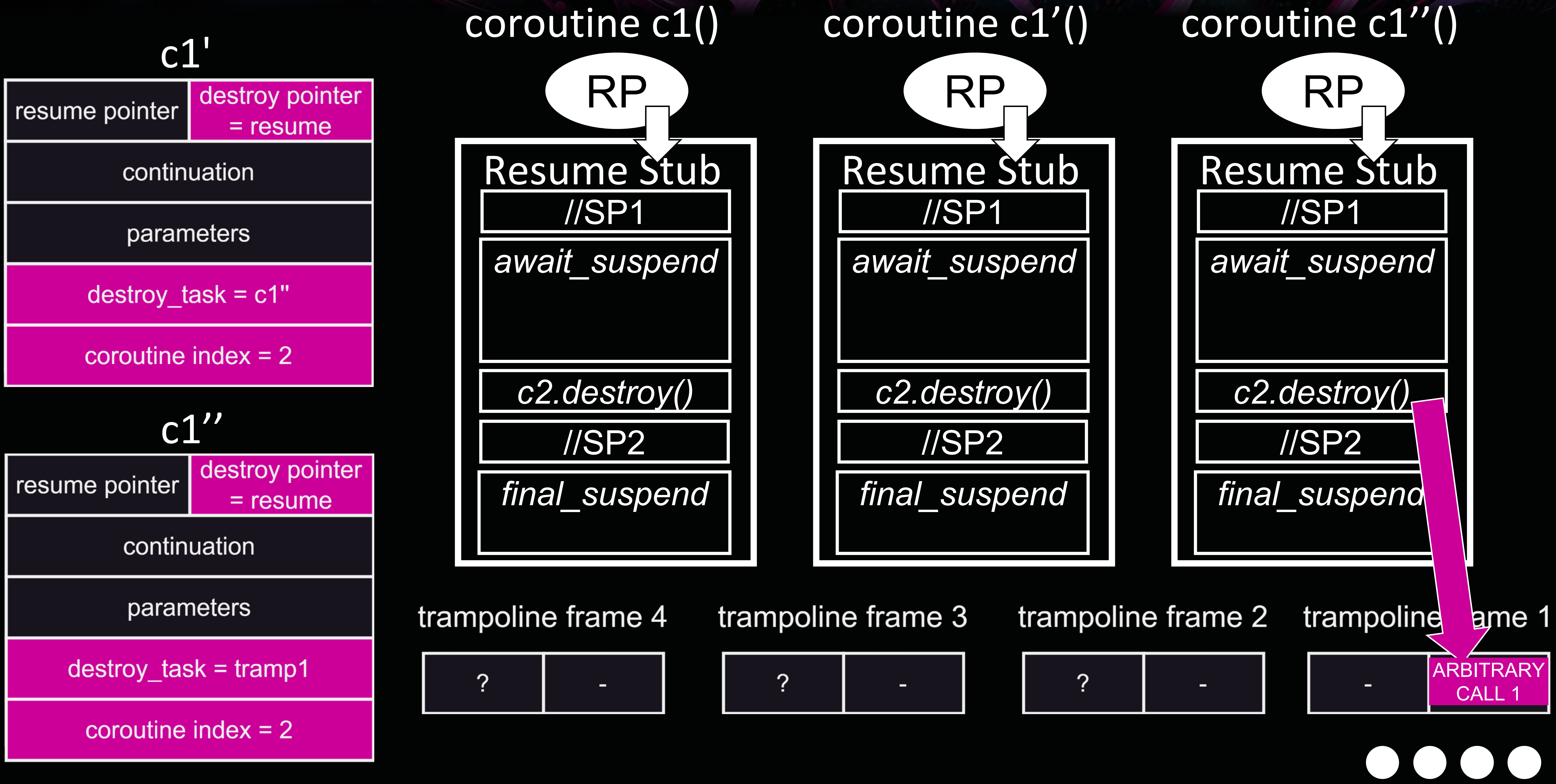
Infinite Coroutine Chaining



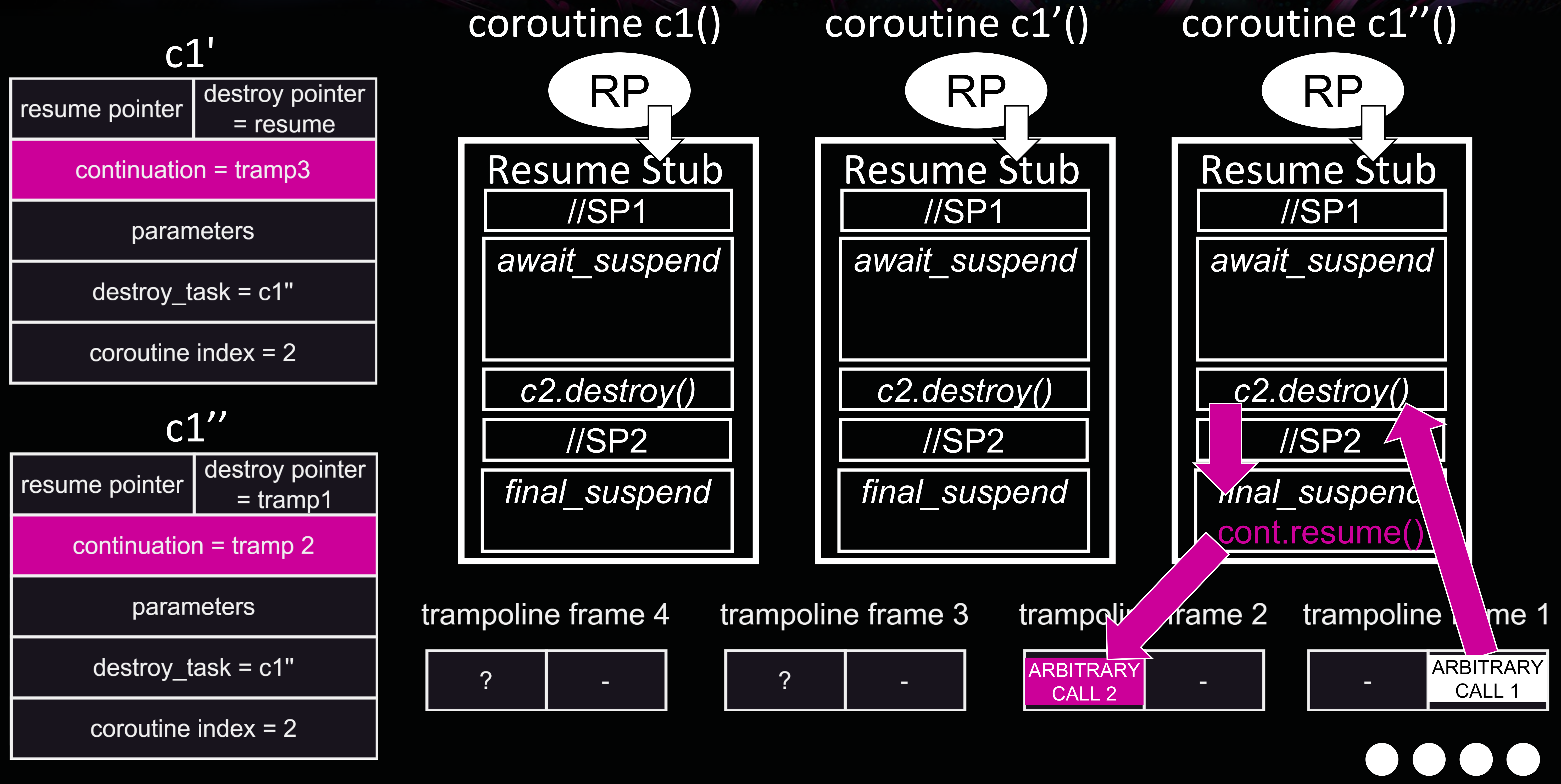
Infinite Coroutine Chaining



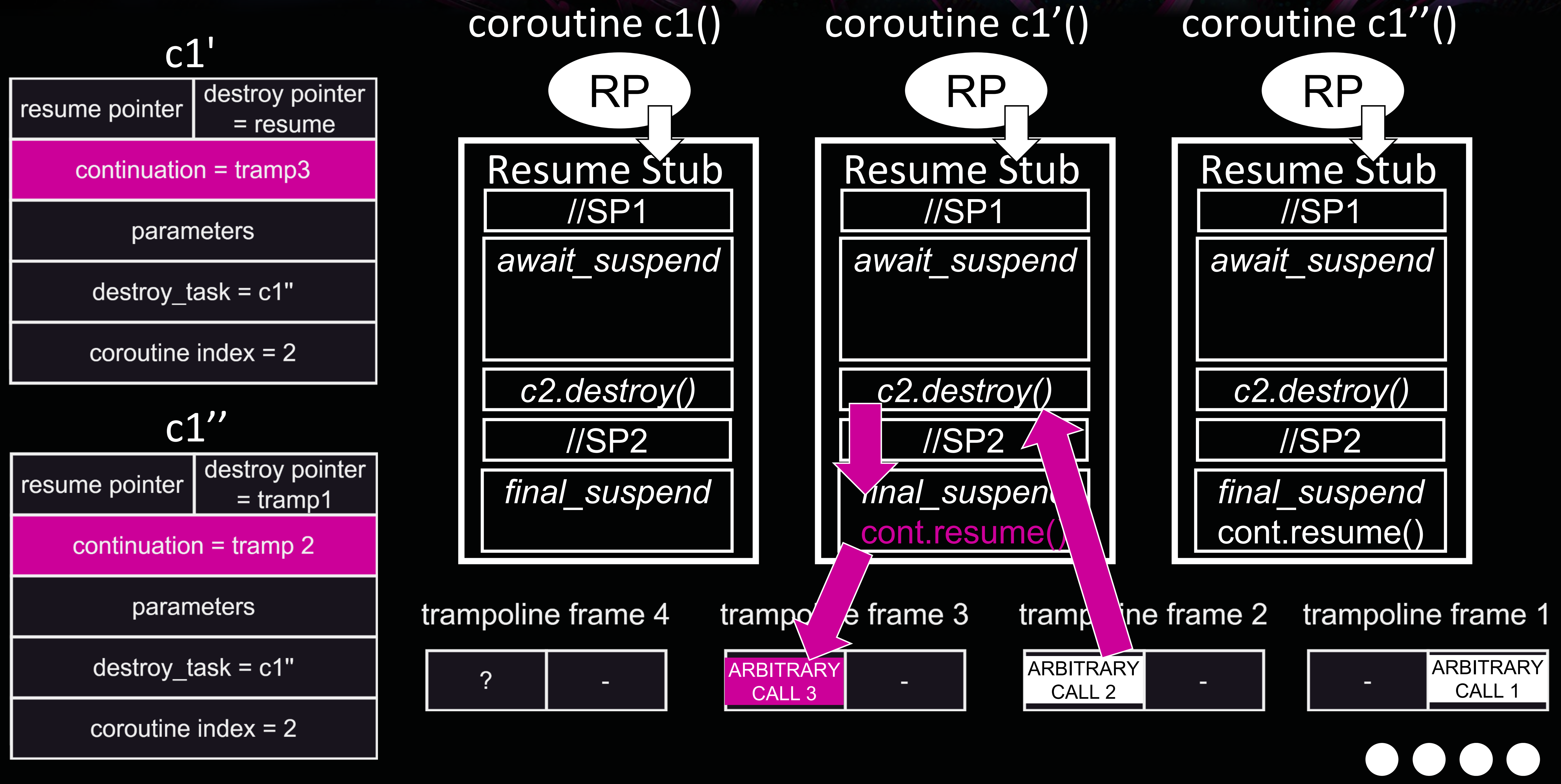
Infinite Coroutine Chaining



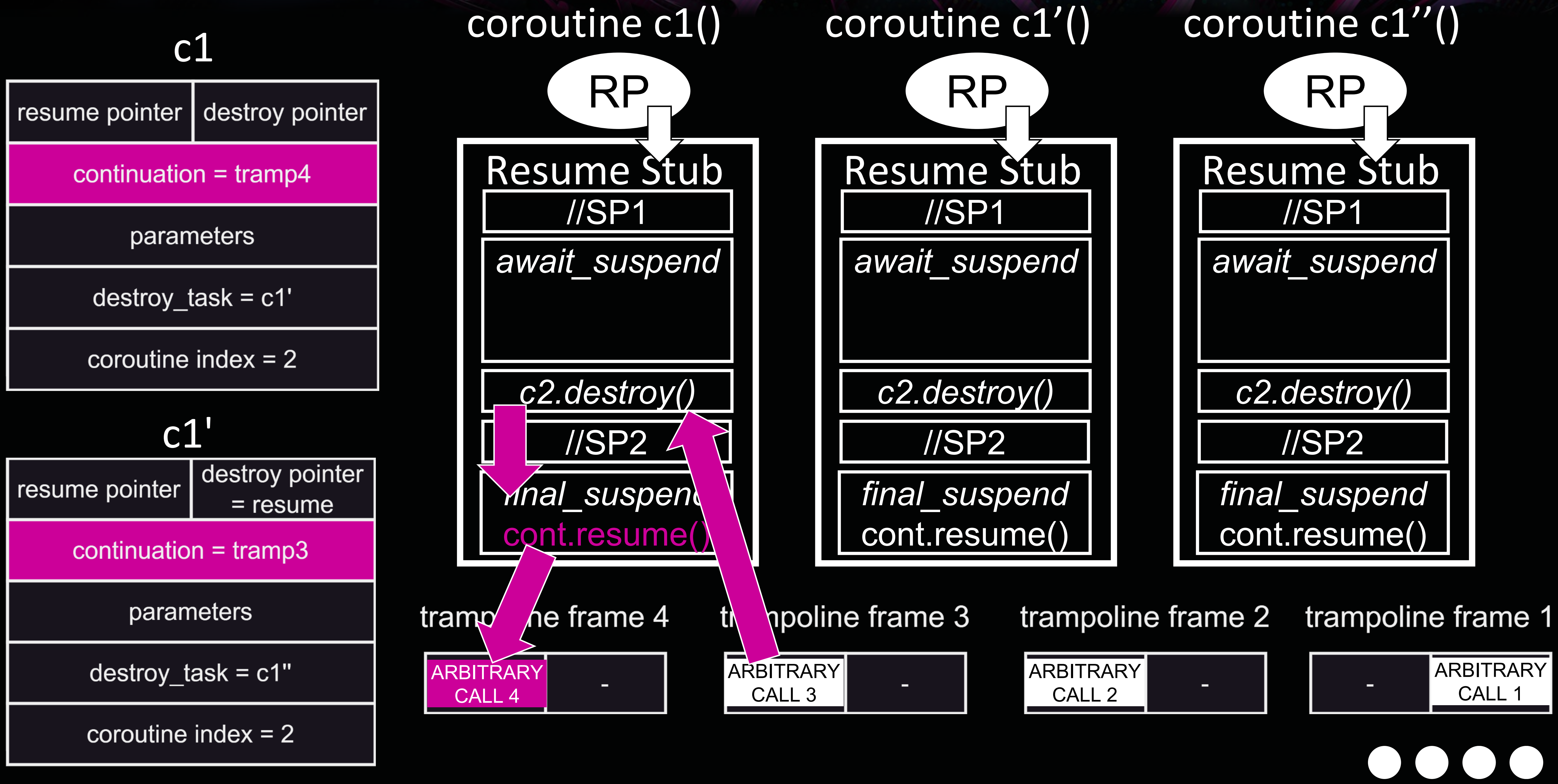
Infinite Coroutine Chaining



Infinite Coroutine Chaining



Infinite Coroutine Chaining



Argument Passing

- We now have infinite arbitrary calls
- What about setting arbitrary arguments in the registers?

Argument Passing

call [rdi] = call [call target]

arg0 = rdi = pointer to call target

call [rdi+0x8] = call [call target]

arg0 = rdi = pointer to argdata

Argument Passing

- So, *resume* and *destroy* have *rdi*=frame
- Is there anything else where *rdi* is always used?

Argument Passing

- So, *resume* and *destroy* have *rdi*=frame
- Is there anything else where *rdi* is always used?
  - Member functions address member variables as *rdi* offsets

```
class A:
   char[] buf;
   char* name;
   char* surname;
   void operate()
   {
     char* a = this.name;
     char* b = this.surname;
     ...
     func(a,b);
   }
```

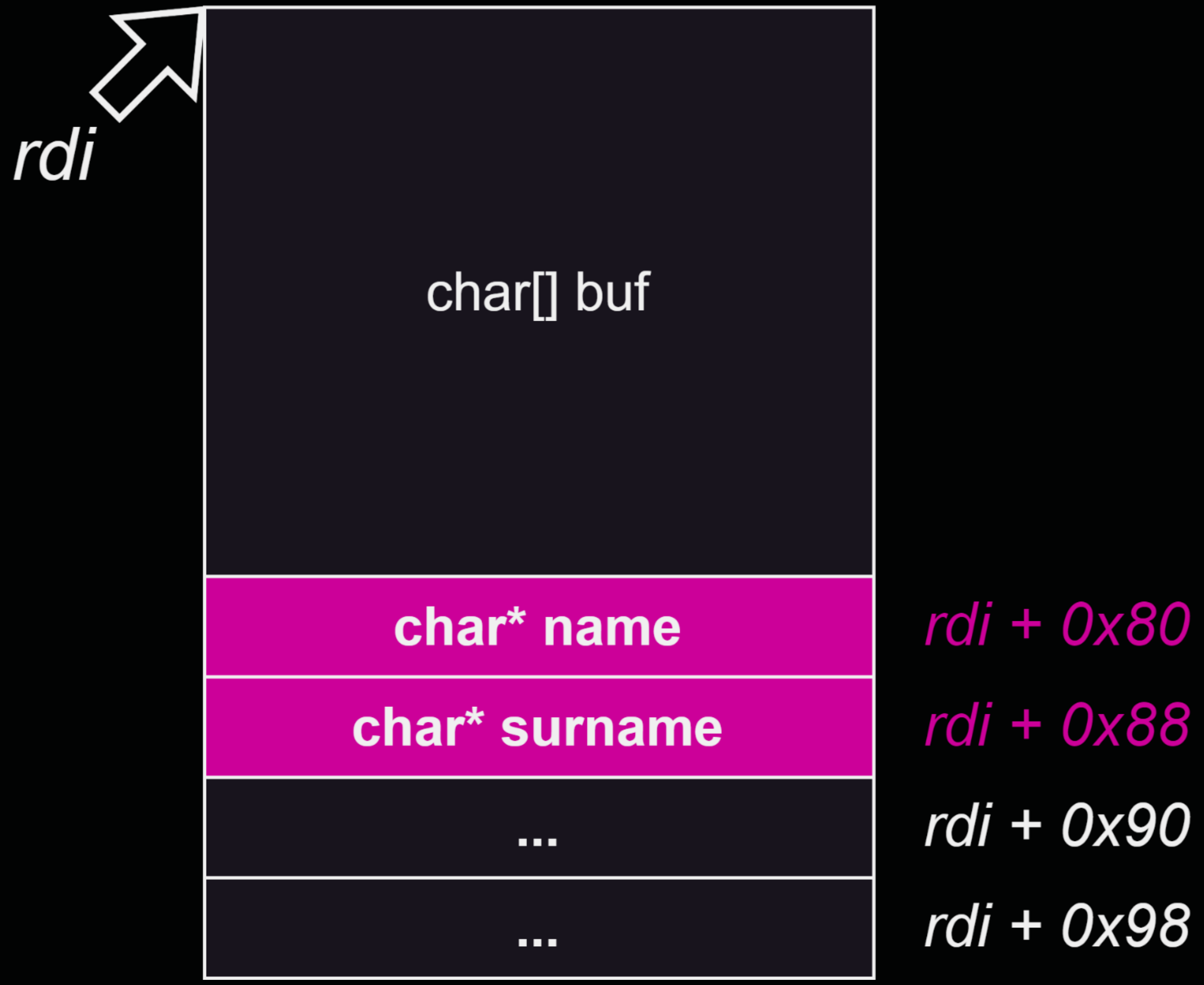
```
operate:
   endbr64
   mov rsi, [rdi+0x80]
   mov rdx, [rdi+0x88]
   ...
```

Argument Passing

- Coroutine frame & class collision

```
operate:
    endbr64
    mov rsi, [rdi+0x80]
    mov rdx, [rdi+0x88]
    ...
```

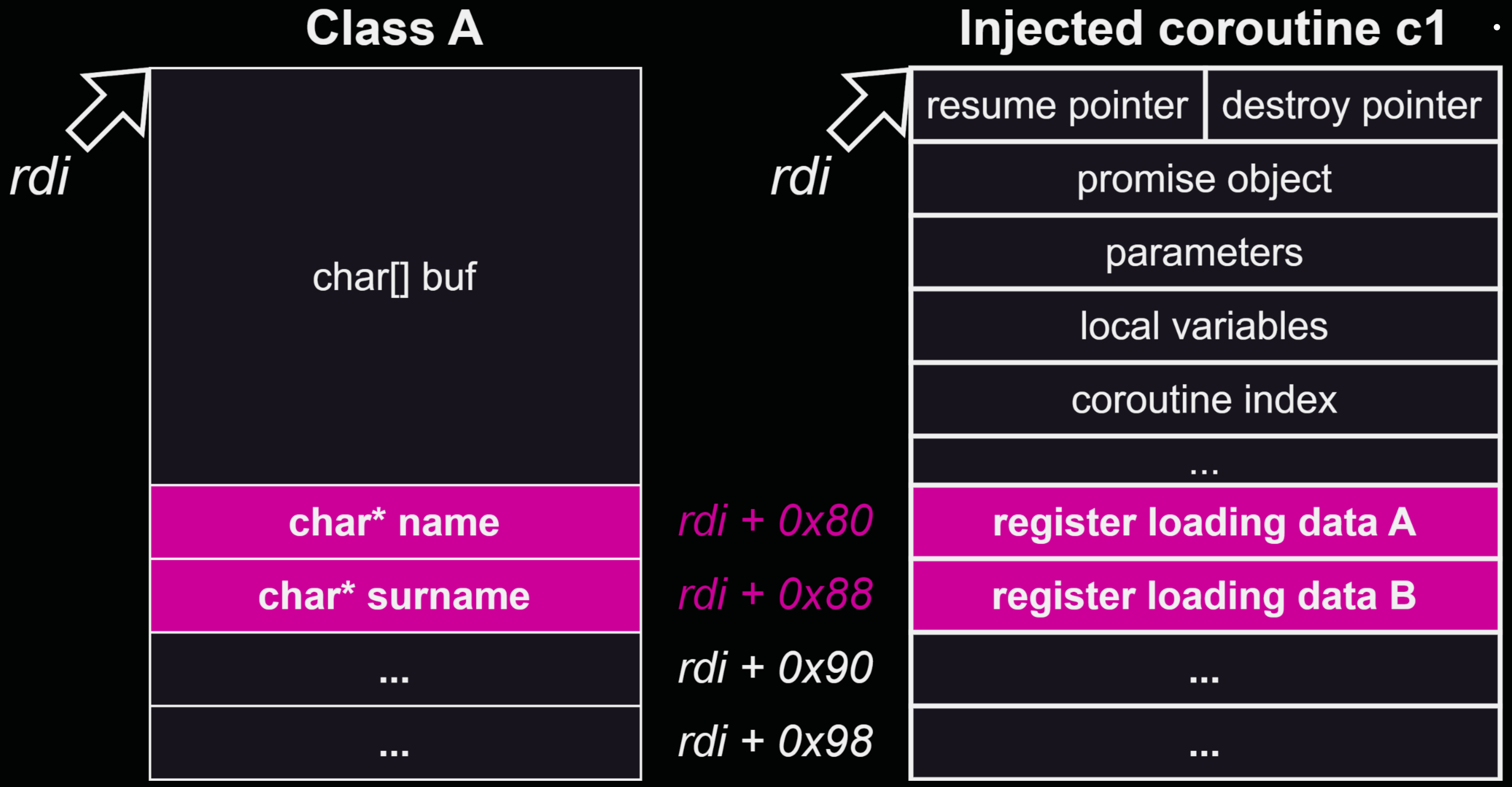
Class A



Argument Passing

- Coroutine frame & class collision

```
operate:
    endbr64
    mov rsi, [rdi+0x80]
    mov rdx, [rdi+0x88]
    ...
```



Argument Passing

- Golden gadget
  - Sets registers and controlled call
- Silver gadget
  - Only sets registers and returns, needs to leverage another CFP for the call

```
endbr64
mov rax, rsi
mov rcx, [rdi+0x90] ;ctrl rcx
mov esi, [rdi+0x80] ;ctrl rsi
mov edx, [rdi+0x98] ;ctrl rdx
mov rdi,rax         ;ctrl rdi
jmp rcx             ;arbitrary call
```

Argument Passing

- Golden gadget
  - Sets registers and controlled call
- Silver gadget
  - Only sets registers and returns, needs to leverage another CFP for the call

```
endbr64
mov rax, rsi
mov rcx, [rdi+0x90] ;ctrl rcx
mov esi, [rdi+0x80] ;ctrl rsi
mov edx, [rdi+0x98] ;ctrl rdx
mov rdi,rax         ;ctrl rdi
ret
```

DEMO TIME

CFOP in Windows

- MSVC supports coroutines from MSVC 19 (and Clang8, gcc 10)
- The coroutine frame, handler and every other internal also exists
  - Still subject to frame manipulation and frame injection

CFOP in Windows

- MSVC supports coroutines from MSVC 19
- The coroutine frame, handler and every other internal also exists
  - Still subject to frame manipulation and frame injection

- Frame injection harder than ptmalloc, LFH chunks are randomized
  - But if you find one frame, you can overwrite its inner CFPs, or overwrite a handler in the stack, and point to known locations

CFOP in Windows

- MSVC supports coroutines from MSVC 19
- The coroutine frame, handler and every other internal also exists
  - Still subject to frame manipulation and frame injection

- Frame injection harder than ptmalloc, LFH chunks are randomized
  - But if you find one frame, you can overwrite its inner CFPs, or overwrite a handler in the stack, and point to known locations

- Bypassing CET SHSTK and CFG is parallel to SHSTK & IBT

CFOP in Windows

- MSVC supports coroutines from MSVC 19
- The coroutine frame, handler and every other internal also exists
  - Still subject to frame manipulation and frame injection
- Frame injection harder than ptmalloc, LFH chunks are randomized
  - But if you find one frame, you can overwrite its inner CFPs, or overwrite a handler in the stack, and point to known locations
- Bypassing CET SHSTK and CFG is parallel to SHSTK & IBT
- The *rdi* = *this* convention turns into *rcx* = *this*, account for other regs

Defense Proposal

- Move the *resume* and *destroy* pointers to read-only memory
- Add a new *coroutine identifier* to search the corresponding pointers

handler

coroutine identifier
promise object
parameters
local variables
coroutine index

```
coro_resume(handler){
  switch (handler.coroutine_identifier){
    case coroA:
      jmp coroA_ResumeStub();
      break;
    case coro_B:
      jmp coroB_ResumeStub();
      break;
    case coro_C:
      jmp coroC_ResumeStub();
      break;
    default:
      exception();
  }
}
```

coroutine jumptable (read-only)

coroA_ResumeStub
coroB_ResumeStub
coroC_ResumeStub
coroA_DestroyStub
coroB_DestroyStub
coroC_DestroyStub

Heap Allocation Elision Optimization

- Heap Allocation Elision Optimization (HALO) moves the coroutines from the heap to the stack
- As an accidental byproduct, it also stops using the *resume* and *destroy* pointers completely. DOAs still good (*megaframes*)

Heap Allocation Elision Optimization

- Heap Allocation Elision Optimization (HALO) moves the coroutines from the heap to the stack
- As an accidental byproduct, it also stops using the *resume* and *destroy* pointers completely. DOAs still good (*megaframes*)
- In practice, getting HALO on your coroutines is *hard*
  - The compiler must be sure that the coroutine is created and destroyed in a certain scope (e.g., a function). Therefore:
    - All boilerplate needs to be inlinable (await_suspend, constructors, etc…)
    - Not possible if code is at different translation units (no LTO)
    - Indirect calls inside the coroutine may break HALO
    - Accessing coroutine objects outside the the coroutine (e.g., return value)
    - Works well if program is simple and/or you prepare the program for HALO, otherwise it is almost guaranteed you will not get it

Heap Allocation Elision Optimization

Does my compiler support HALO at all?

- GCC
  - No.
- Clang
  - Yes (with the mentioned restrictions), but Clang 19 and 20 are slightly broken – this is a bug, HALO was not discarded
- MSVC
  - Since MSVC 19.43, from VS 17.13, dating February 2025 (after our report)
  - However, requires compiling without exception support (EHsc), which is enabled by default in VS

black hat BRIEFINGS

AUGUST 6-7, 2025

MANDALAY BAY / LAS VEGAS

https://github.com/coroutine-cfop/cfop

Marcos Bajo *h3xduck*

Christian Rossow