CriptoRed /Rooted CON

Binary Exploitation in the Control Flow Integrity Era

Coroutine Frame-Oriented Programming

Marcos Bajo (*h3xduck*) | CriptoredCON | RootedCON 2026
CISPA Helmholtz Center for Information Security

The Old Ages

1972

Buffer overflows 1st mentioned

ESD-TR-73-51, Vol. II

COMPUTER SECURITY TECHNOLOGY PLANNING STUDY

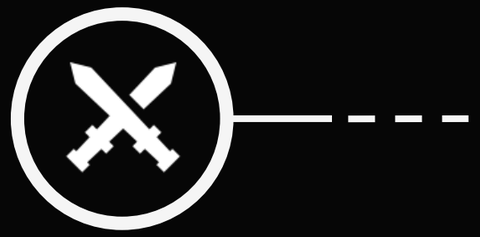
James P. Anderson

October 1972

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

The Old Ages

1972



Buffer overflows 1st mentioned

2000 2010 2020

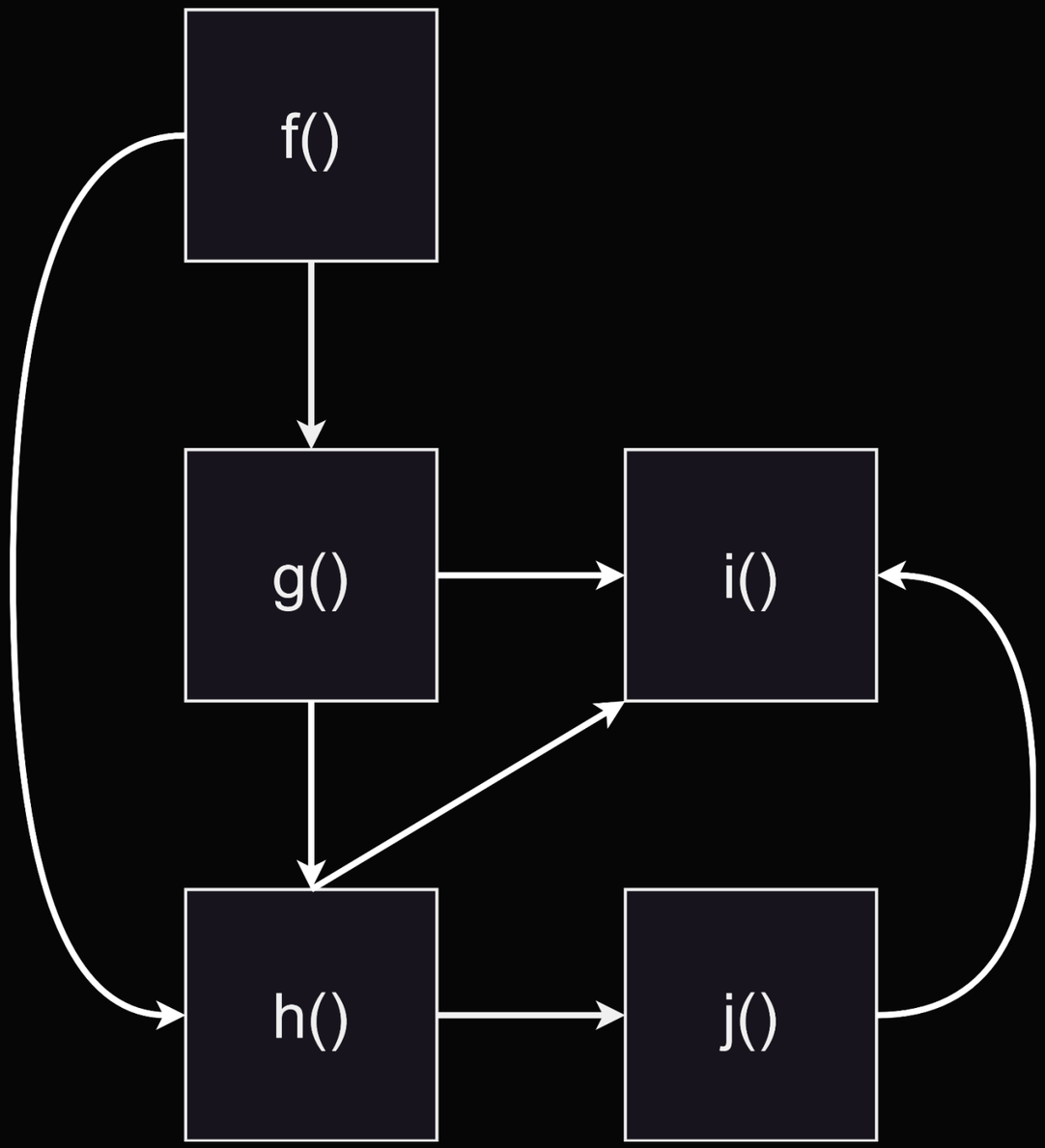
MEMORY CORRUPTION

CODE EXECUTION

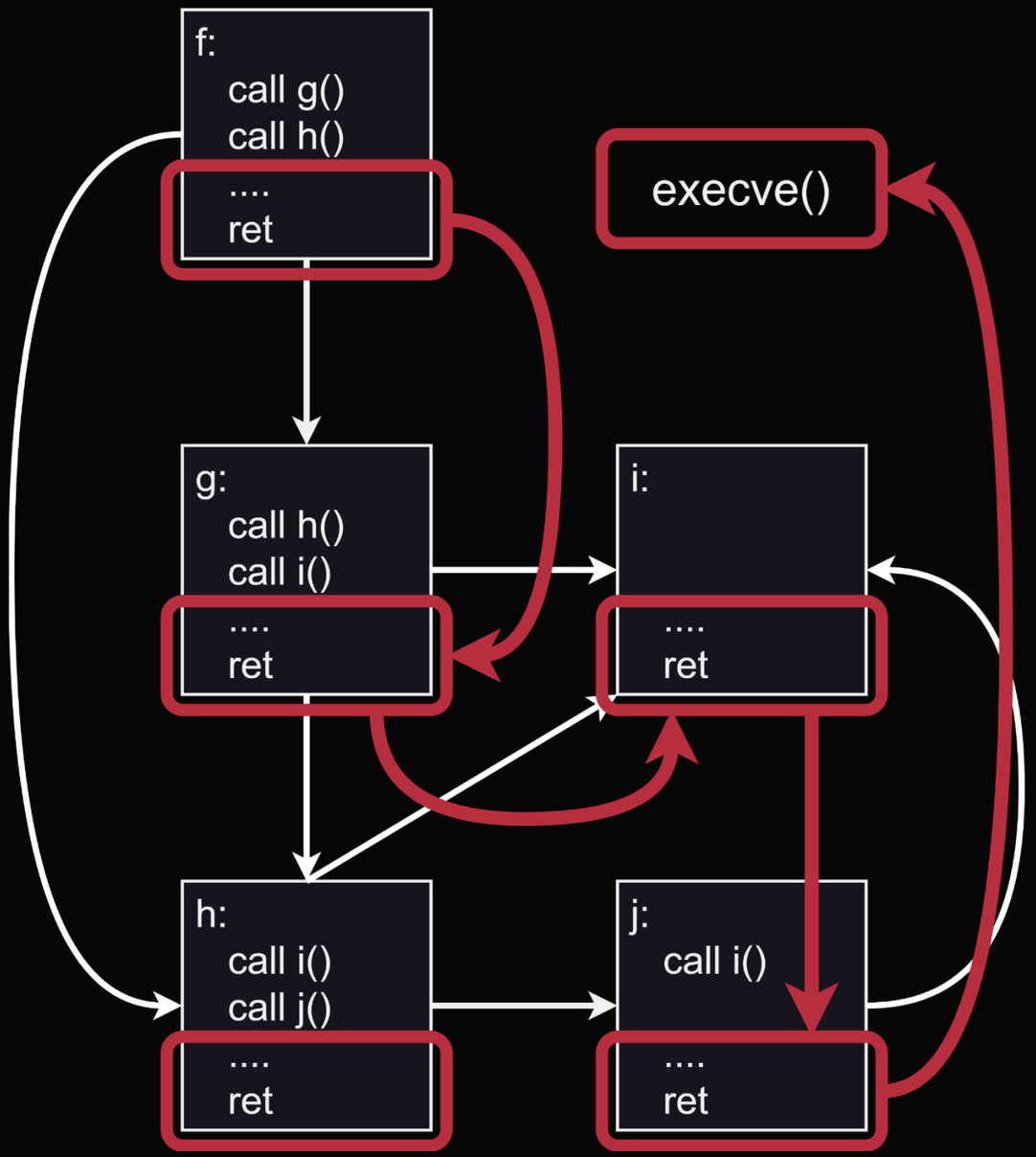
The Old Ages

1972 2000 2010 2020

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

Code-Reuse

- Code-Reuse uses small code pieces: gadgets

- **ROP** gadgets:
  - End with a *ret* instruction

```
pop rax
ret
```

- **JOP** gadgets:
  - End with a jmp instruction

```
mov rax, [rsp]
mov rdi, [rsp+0x8]
jmp rdi
```

The Modern Ages

1972 2000 2010 2020

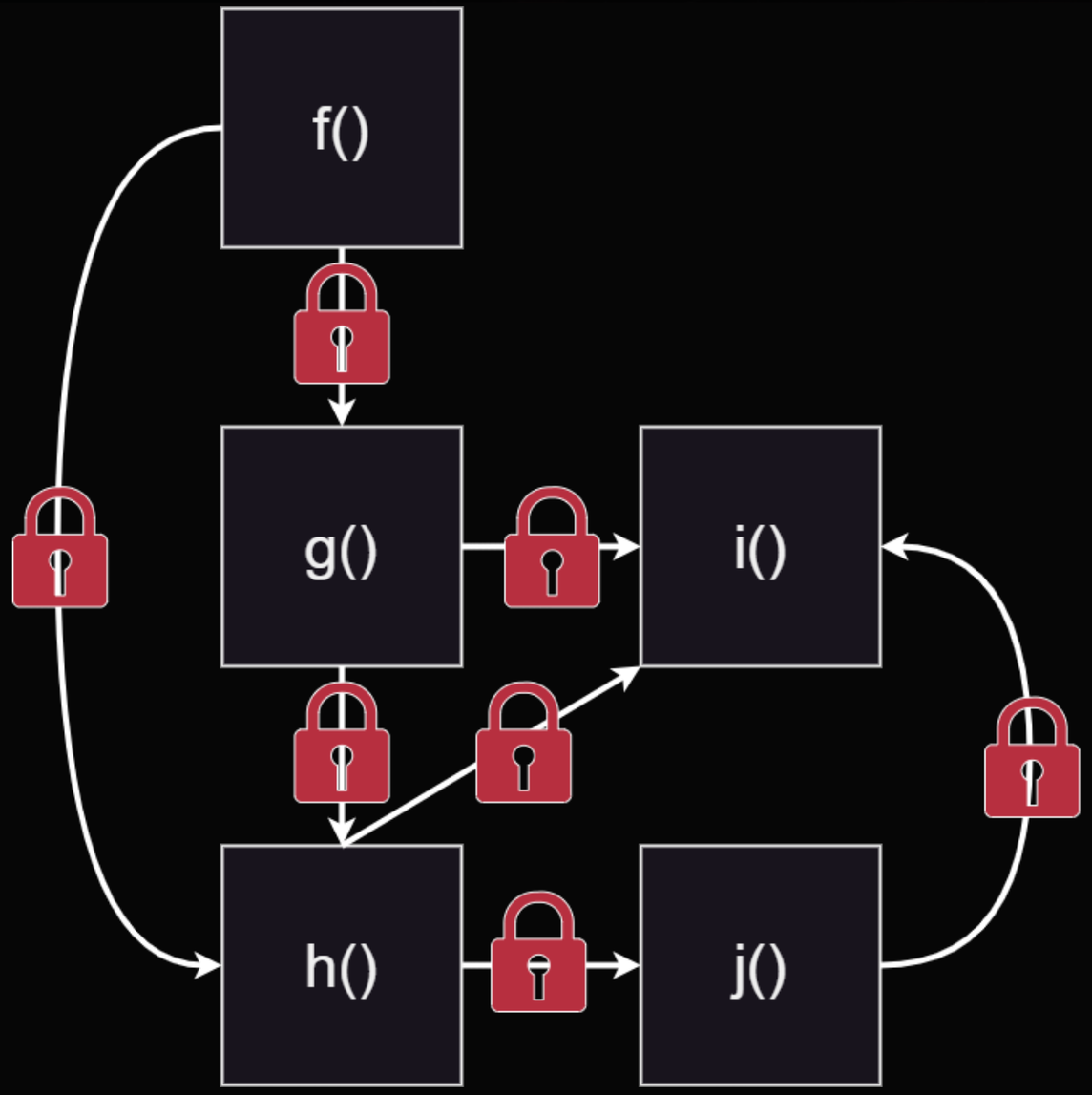
CFI 1st mentioned

The Modern Ages

1972 2000 2010 2020

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 I Am

Marcos Bajo

aka ***h3xduck***

https://h3xduck.github.io

marcos.sanchez-bajo@cispa.de

CISPA

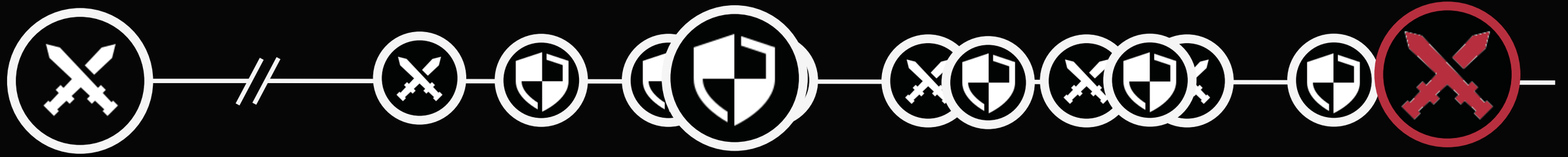
HELMHOLTZ CENTER FOR INFORMATION SECURITY

- PhD Student at CISPA (Germany)
- Three things I love:
  - Malware
  - Exploits
  - Ducks

- Speaker at:
  - USENIX Security 2025
  - Black Hat USA 2025
  - Here! 2023 :)

The Modern Ages

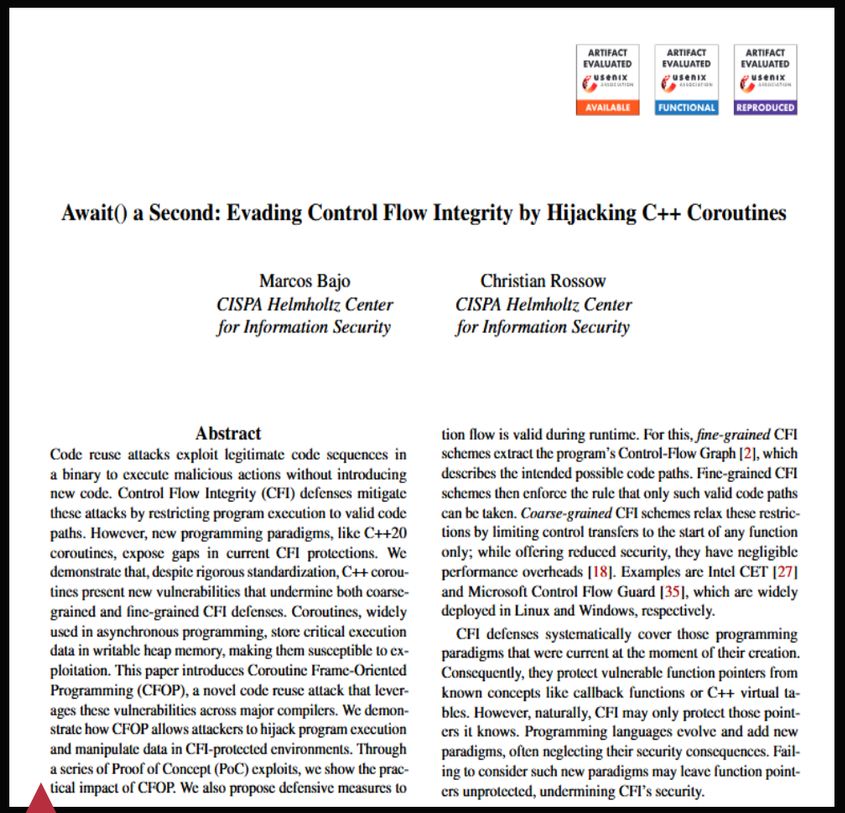
1972 2000 2010 2020



CFOP

What We Will Learn

CFOP



ARTIFACT EVALUATED USENIX ASSOCIATION AVAILABLE

ARTIFACT EVALUATED USENIX ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED USENIX ASSOCIATION REPRODUCED

Await() a Second: Evading Control Flow Integrity by Hijacking C++ Coroutines

Marcos Bajo
CISPA Helmholtz Center for Information Security

Christian Rossow
CISPA Helmholtz Center for Information Security

Abstract

Code reuse attacks exploit legitimate code sequences in a binary to execute malicious actions without introducing new code. Control Flow Integrity (CFI) defenses mitigate these attacks by restricting program execution to valid code paths. However, new programming paradigms, like C++20 coroutines, expose gaps in current CFI protections. We demonstrate that, despite rigorous standardization, C++ coroutines present new vulnerabilities that undermine both coarse-grained and fine-grained CFI defenses. Coroutines, widely used in asynchronous programming, store critical execution data in writable heap memory, making them susceptible to exploitation. This paper introduces Coroutine Frame-Oriented Programming (CFOP), a novel code reuse attack that leverages these vulnerabilities across major compilers. We demonstrate how CFOP allows attackers to hijack program execution and manipulate data in CFI-protected environments. Through a series of Proof of Concept (PoC) exploits, we show the practical impact of CFOP. We also propose defensive measures to tion flow is valid during runtime. For this, *fine-grained* CFI schemes extract the program's Control-Flow Graph [2], which describes the intended possible code paths. Fine-grained CFI schemes then enforce the rule that only such valid code paths can be taken. *Coarse-grained* CFI schemes relax these restrictions by limiting control transfers to the start of any function only; while offering reduced security, they have negligible performance overheads [18]. Examples are Intel CET [27] and Microsoft Control Flow Guard [35], which are widely deployed in Linux and Windows, respectively.

CFI defenses systematically cover those programming paradigms that were current at the moment of their creation. Consequently, they protect vulnerable function pointers from known concepts like callback functions or C++ virtual tables. However, naturally, CFI may only protect those pointers it knows. Programming languages evolve and add new paradigms, often neglecting their security consequences. Failing to consider such new paradigms may leave function pointers unprotected, undermining CFI's security.

SFOP

Under review ~2026

CCCC

Current research ~2027

Marcos Bajo and Christian Rossow, *USENIX Security 2025*

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.

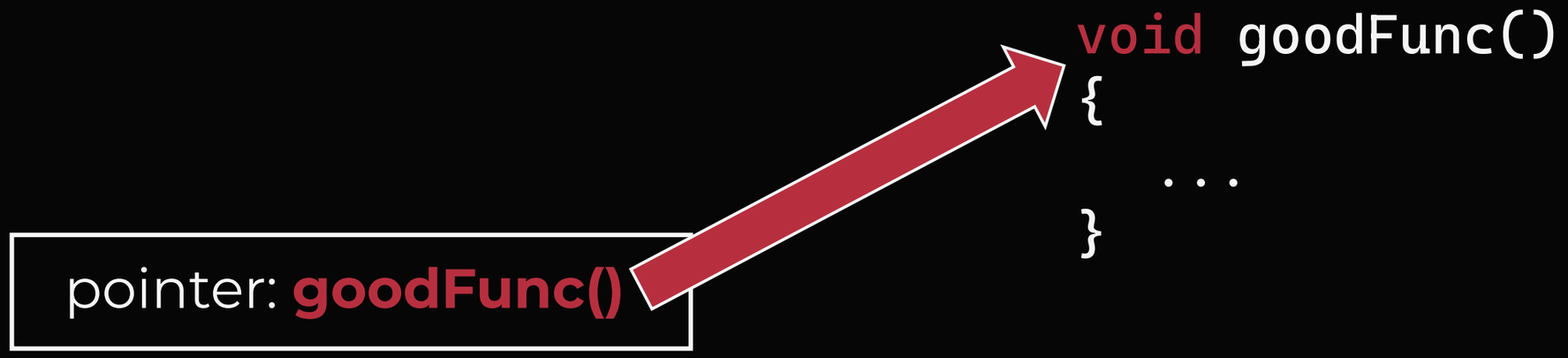
4. Coroutine Frame-Oriented Programming

Using coroutines to bypass CFI.

1 Userspace CFI Defenses

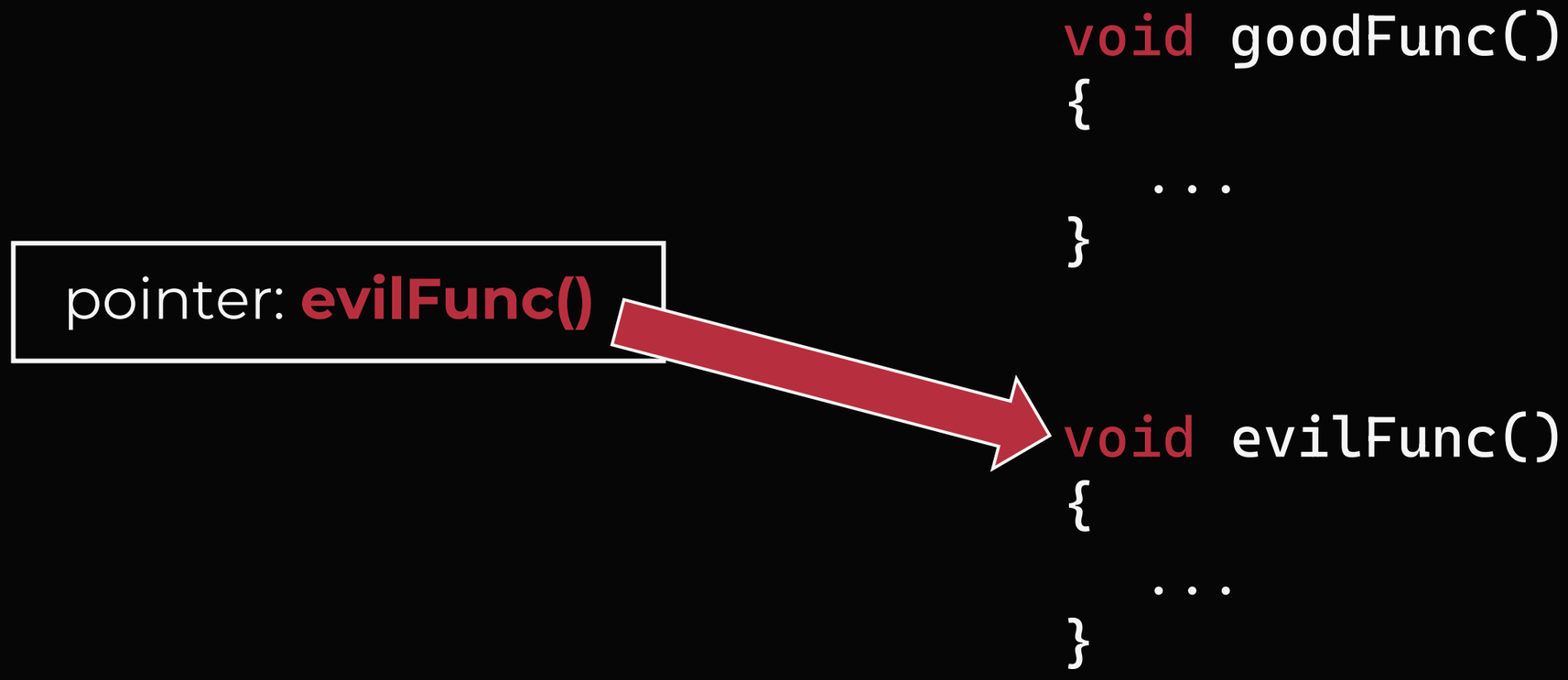
Memory Corruption

- Memory corruption vulnerabilities overwrite code pointers



Memory Corruption

- Memory corruption vulnerabilities overwrite code pointers



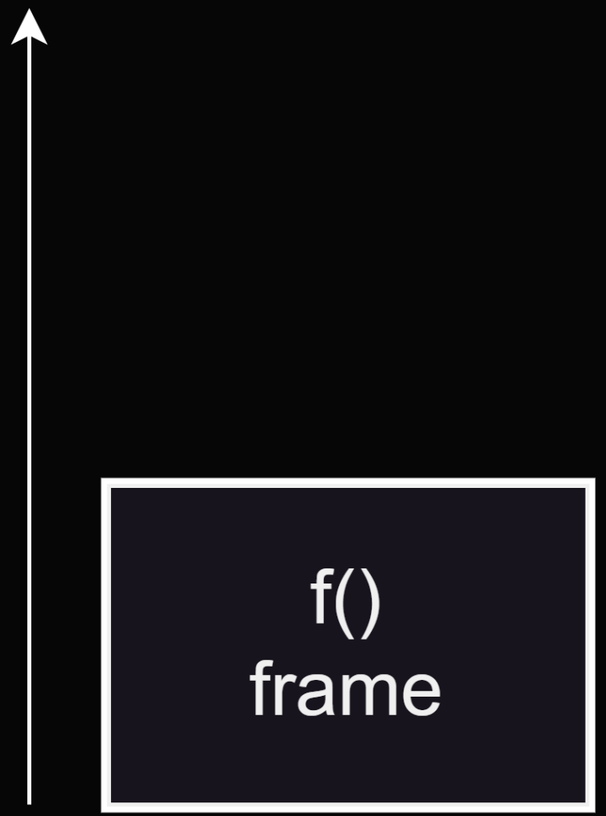
Memory Corruption

- Memory corruption vulnerabilities overwrite code pointers

- Two main types of code pointers:
  - Backward-edge
  - Forward-edge

What is CFI

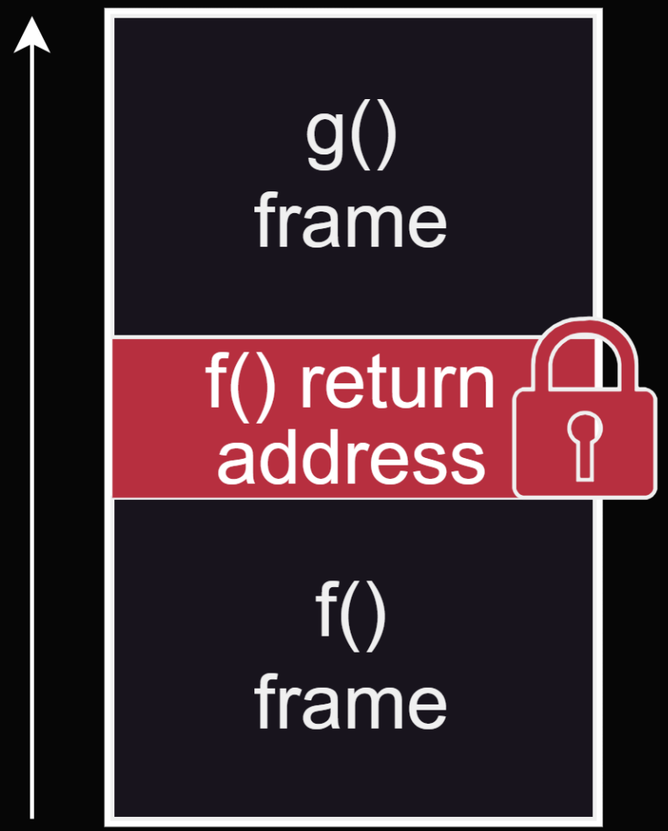
Backward-edge




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

What is CFI

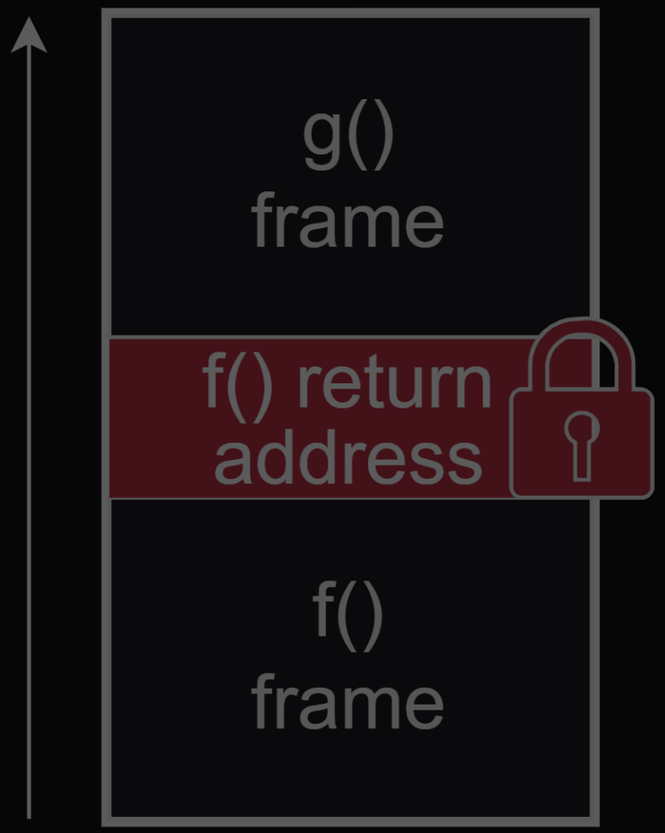
Backward-edge




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

What is CFI

Backward-edge

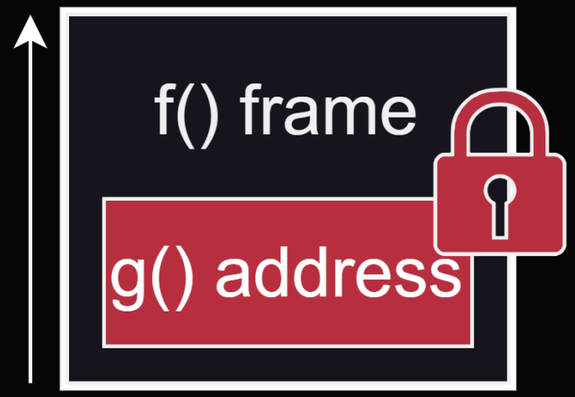



```
void g()
{
    ret;
}
void f()
{
    g();
}
```

Forward-edge

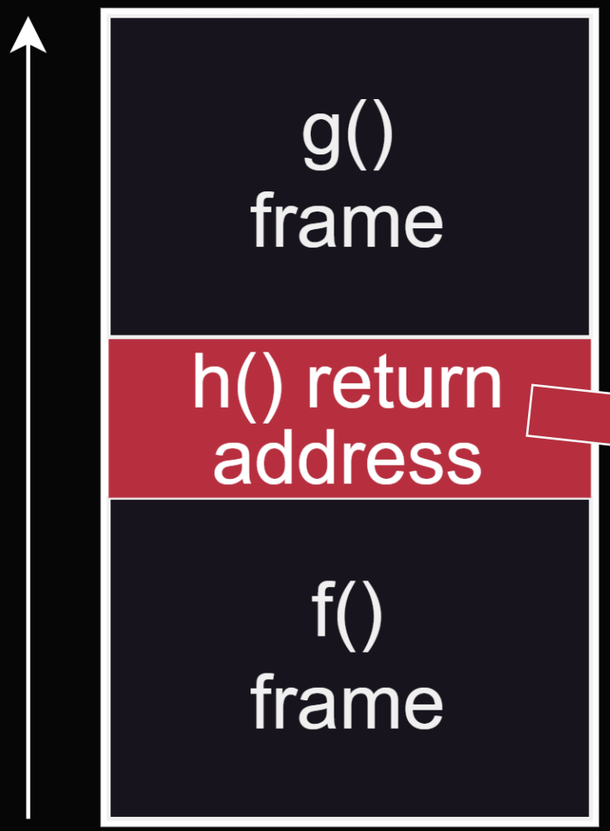
```
void f()
{
   void *ptr = &g;
   ptr();
}
```

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



What is CFI

Backward-edge




```
void g()
{
    ret;
}
void f()
{
    g();
}
void h()
{
  <win>
}
```

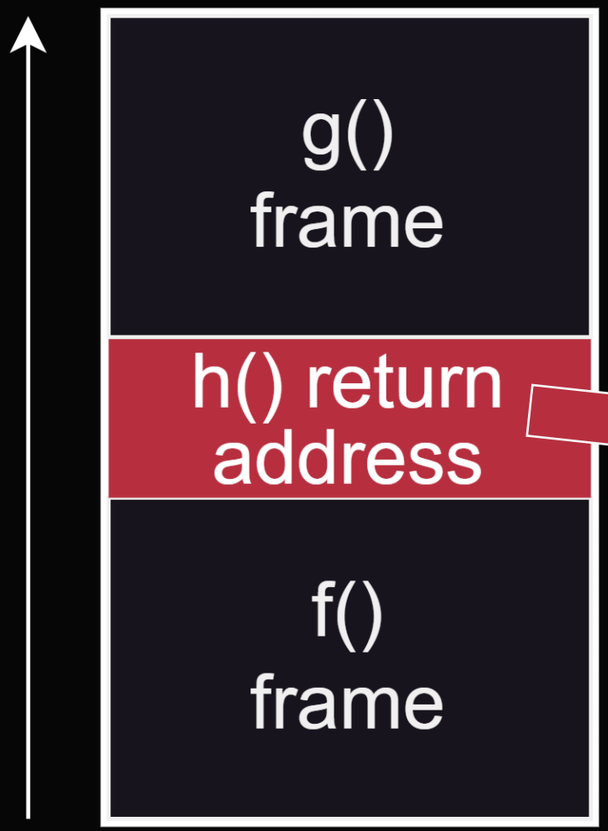
Forward-edge

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


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

What is CFI

Backward-edge




```
void g()
{
    ret;
}
void f()
{
    g();
}
void h()
{
  <win>
}
```

Forward-edge

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


```
void g()
{
    ret;
}
void h()
{
  <win>
}
```

What is CFI

Backward-edge

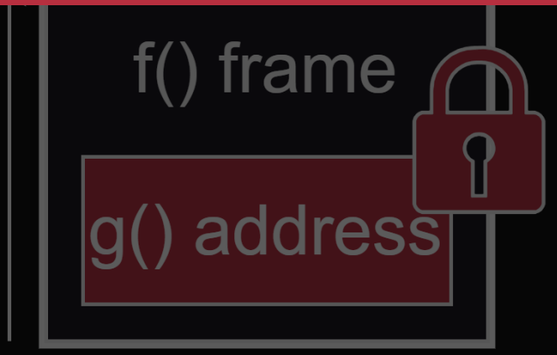
```
void g()
{
    ret;
}
```

Backward-edge Control Flow Integrity

Forward-edge

```
void f()
{
```

Forward-edge Control Flow Integrity



CFI Types

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

- Security + Security

+ Performance - Performance

		x86_64 LINUX		x86_64 WINDOWS		ARM ANDROID	Other x86 Software (Chrome)
		Userspace	Kernel	Userspace	Kernel		
Backward-edge CFI		Shadow Stack (CET)	Shadow Stack (CET)	Shadow Stack (CET)	Shadow Stack (CET)	GCS	Shadow Stack (CET)
Forward-edge CFI	Coarse-grained						
	Medium-grained						
	Fine-grained						

Always by-default

Opt-in / sometimes by-default

Soon incorporation

		x86_64 LINUX		x86_64 WINDOWS		ARM ANDROID	Other x86 Software (Chrome)
		Userspace	Kernel	Userspace	Kernel		
Backward-edge CFI		Shadow Stack (CET)	Shadow Stack (CET)	Shadow Stack (CET)	Shadow Stack (CET)	GCS	Shadow Stack (CET)
Forward-edge CFI	Coarse-grained	IBT (CET)	IBT (CET)	MSCFG	MSCFG	PAC BTI	IBT (CET)
	Medium-grained						
	Fine-grained						

Always by-default

Opt-in / sometimes by-default

Soon incorporation

		x86_64 LINUX		x86_64 WINDOWS		ARM ANDROID	Other x86 Software (Chrome)
		Userspace	Kernel	Userspace	Kernel		
Backward-edge CFI		Shadow Stack (CET)	Shadow Stack (CET)	Shadow Stack (CET)	Shadow Stack (CET)	GCS	Shadow Stack (CET)
Forward-edge CFI	Coarse-grained	IBT (CET)	IBT (CET)	MSCFG	MSCFG	PAC BTI	IBT (CET)
	Medium-grained		KCFI LLVM CFI			LLVM CFI	LLVM CFI
	Fine-grained						

Always by-default | Opt-in / sometimes by-default | Soon incorporation

		x86_64 LINUX		x86_64 WINDOWS		ARM ANDROID	Other x86 Software (Chrome)
		Userspace	Kernel	Userspace	Kernel		
Backward-edge CFI		Shadow Stack (CET)	Shadow Stack (CET)	Shadow Stack (CET)	Shadow Stack (CET)	GCS	Shadow Stack (CET)
Forward-edge CFI	Coarse-grained	IBT (CET)	IBT (CET)	MSCFG	MSCFG	PAC BTI	IBT (CET)
	Medium-grained		KCFI LLVM CFI			LLVM CFI	LLVM CFI
	Fine-grained		FineIBT				

Always by-default | Opt-in / sometimes by-default | Soon incorporation

		x86_64 LINUX	x86_64 WINDOWS	ARM ANDROID	Other x86 Software (Chrome)
Backward-edge CFI		Shadow Stack (CET)	Shadow Stack (CET)	GCS	Shadow Stack (CET)
Forward-edge CFI	Coarse-grained	IBT (CET)	CFG	PAC BTI	IBT (CET)
	Medium-grained	KCFI		LLVM CFI	LLVM CFI
	Fine-grained	FineIBT			

MCFI/ piCFI

Safe Dispatch

ReCFI

Path Armor

VfGuard

CFIXX

PittyPat

VTrust

Typro

Always by-default

Opt-in / sometimes by-default

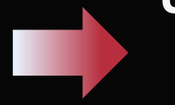
Soon incorporation

Intel CET

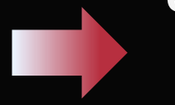
- Coarse-grained CFI
- Hardware-assisted
- Two protections in one:
  - Backward-edge: Shadow Stack
  - Forward-edge: Indirect Branch Tracking (IBT)

Intel CET

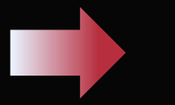
```
void f()
{
    g();
    ret;
}
```



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

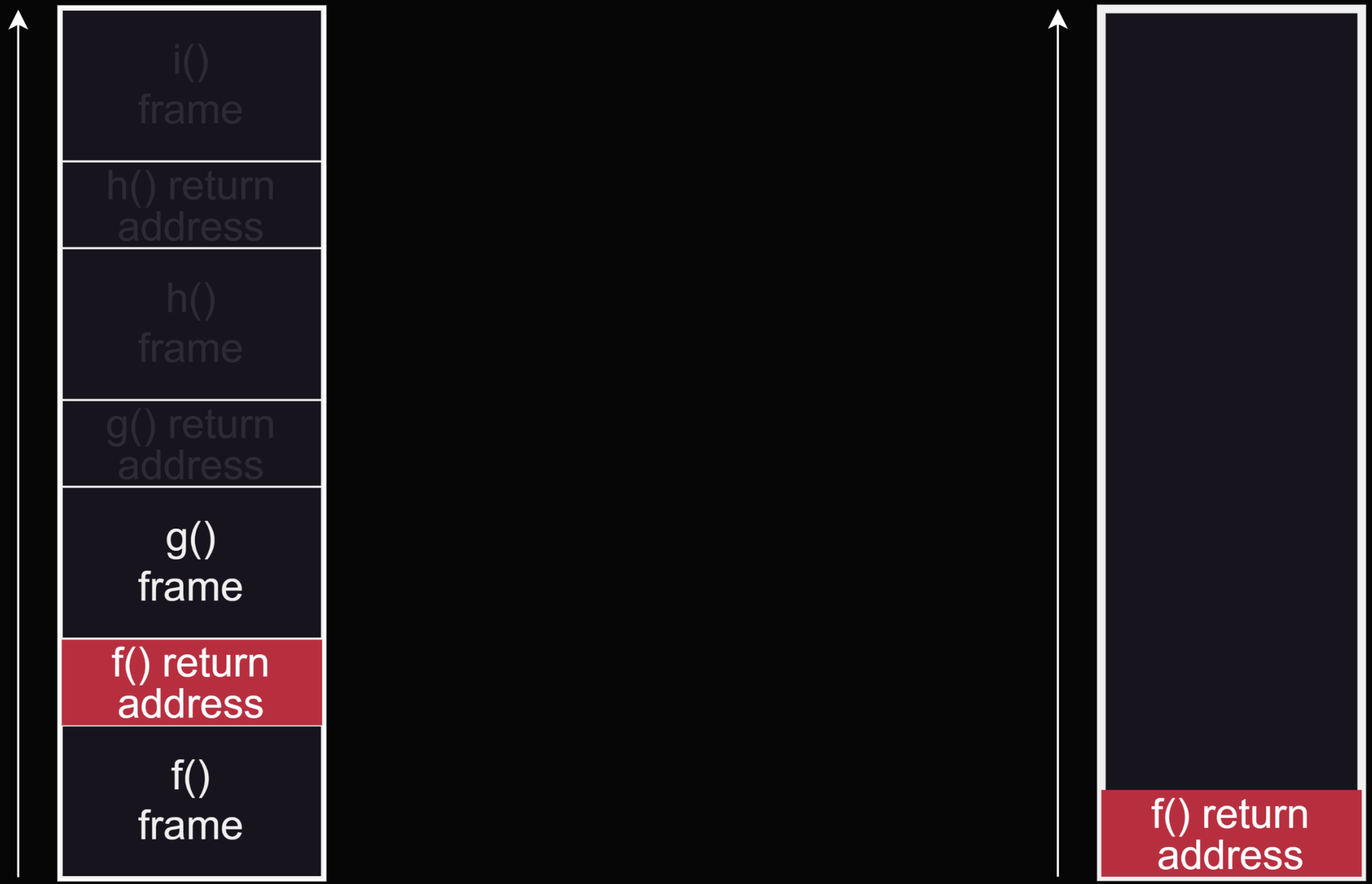


```
void h()
{
    i();
    ret;
}
```

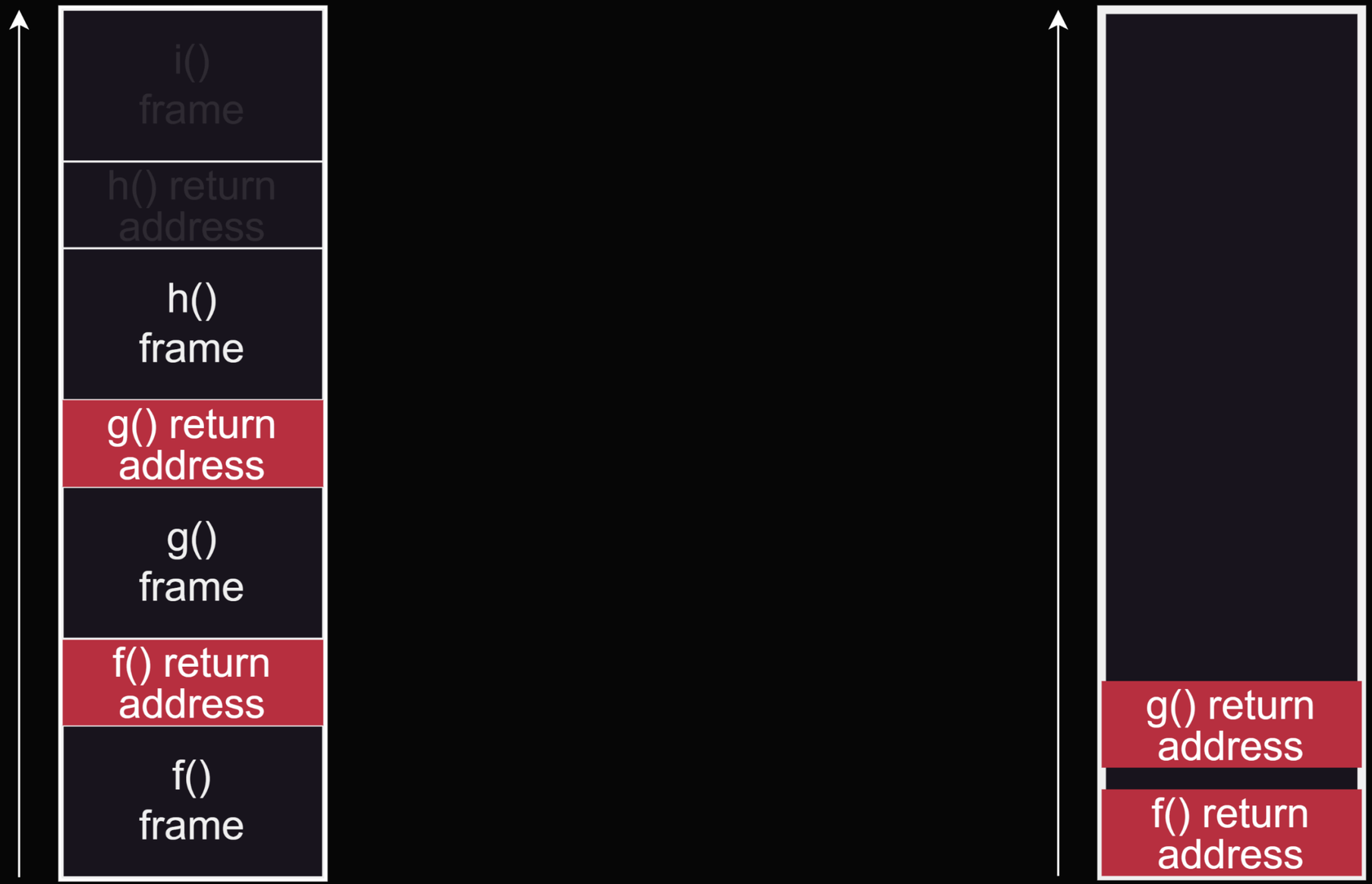


```
void i()
{
    ret;
}
```

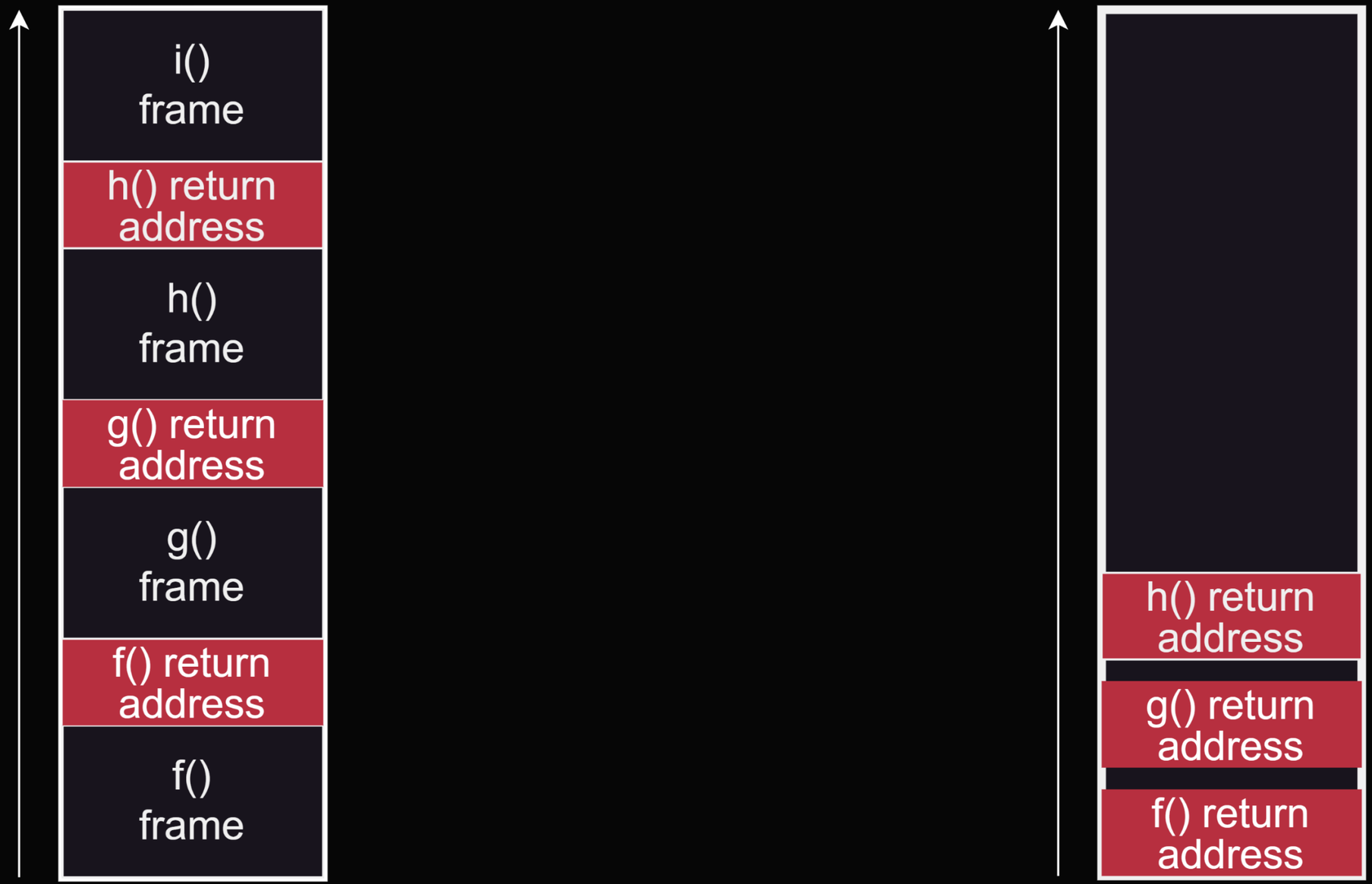
Intel CET (Shadow Stack)



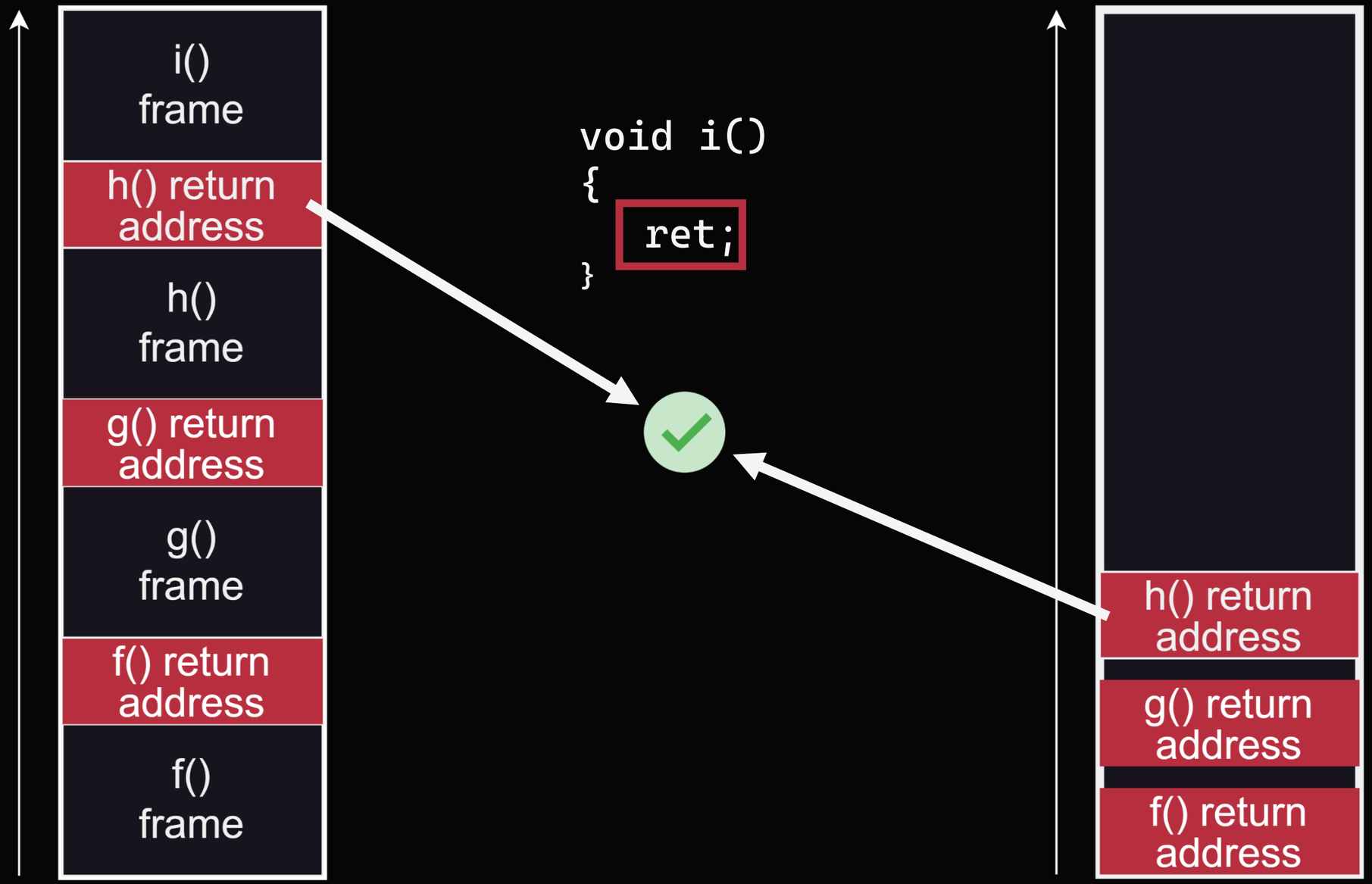
Intel CET (Shadow Stack)



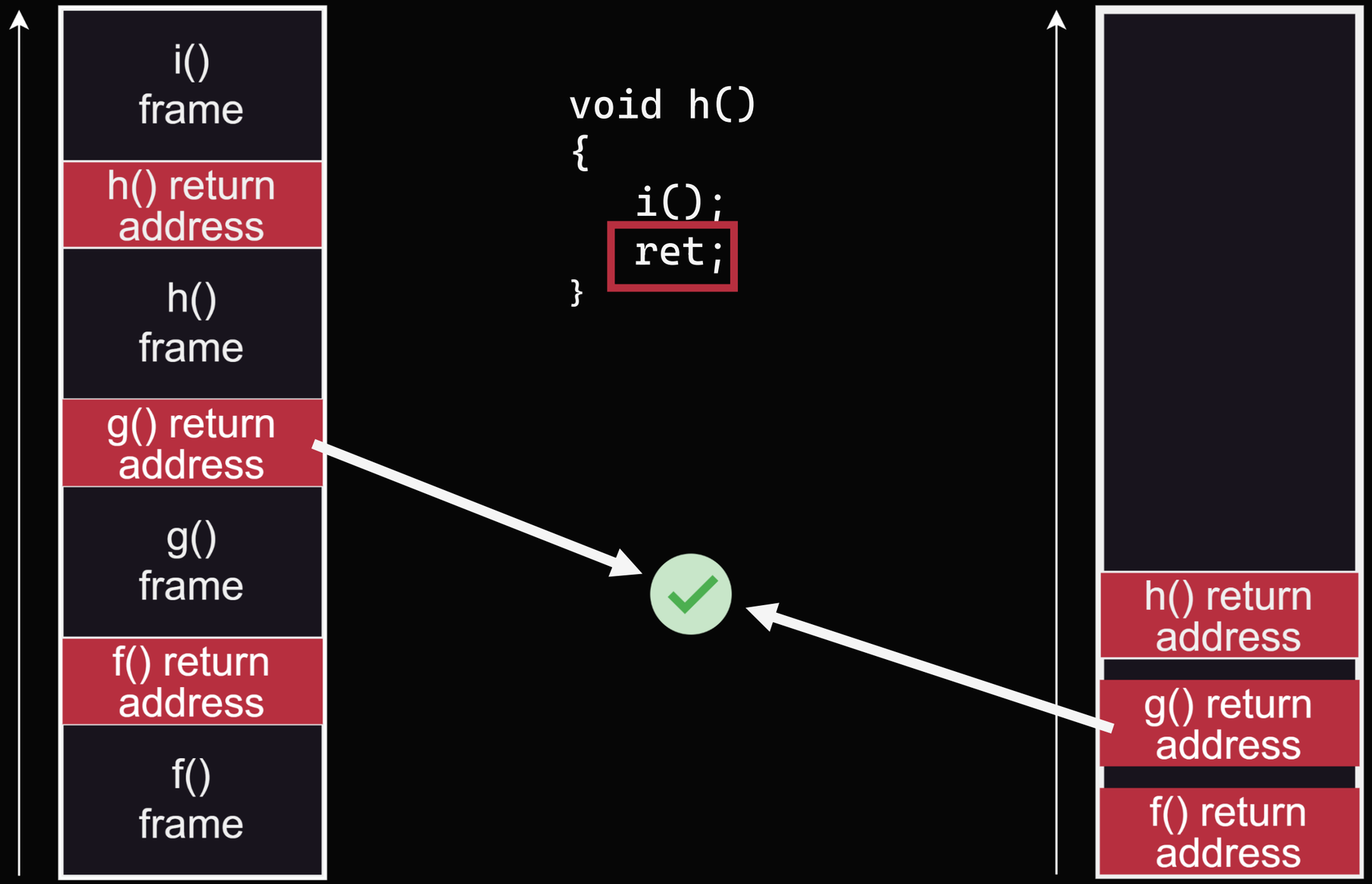
Intel CET (Shadow Stack)



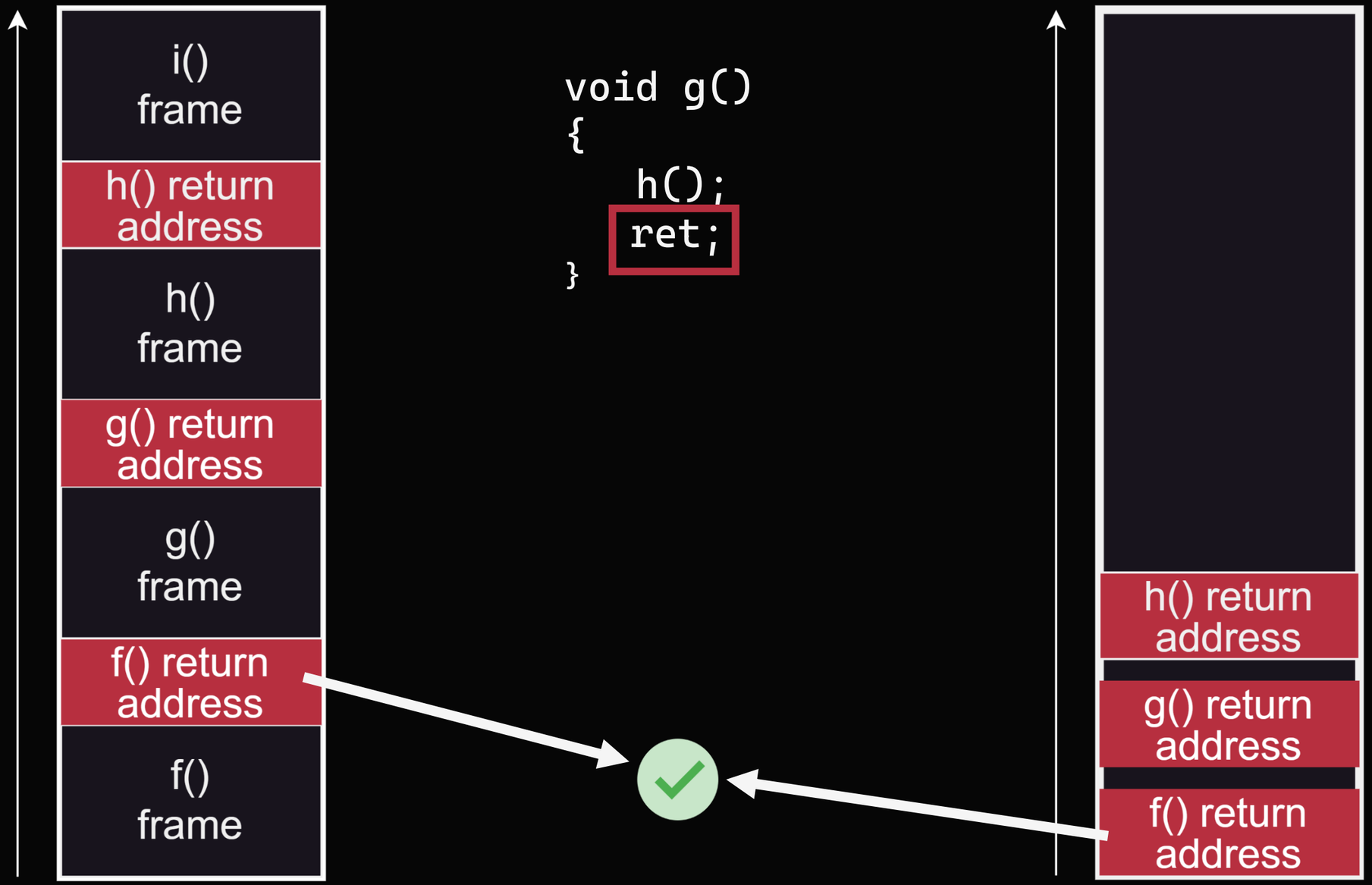
Intel CET (Shadow Stack)



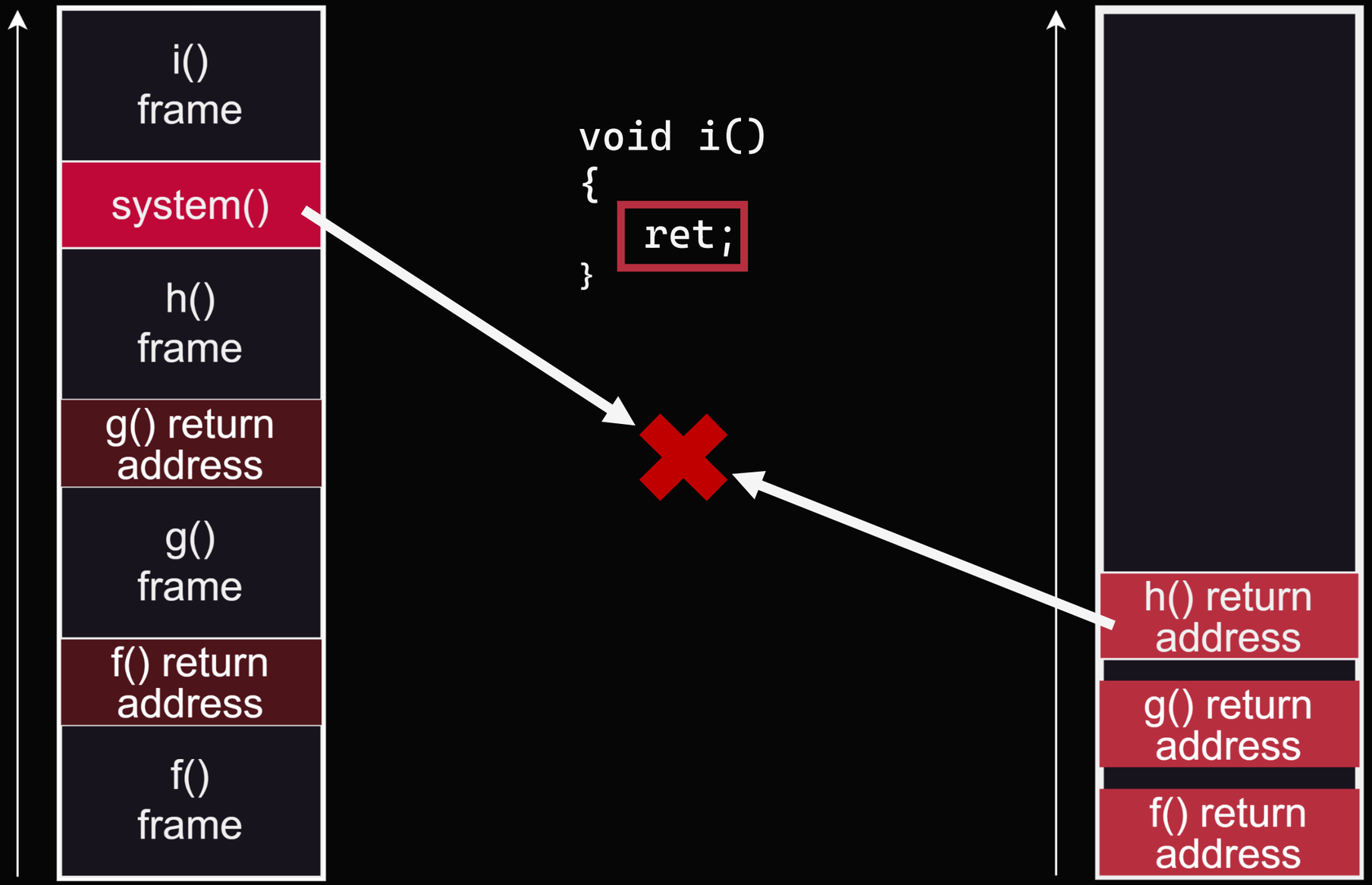
Intel CET (Shadow Stack)



Intel CET (Shadow Stack)



Intel CET (Shadow Stack)



Intel CET (Shadow Stack)

- Processes allocate a dedicated memory region before calling *main*
  - Function *_dl_cet_setup_features*
  - Instructs the kernel to allocate a shadow stack

    arch_prctl ➡️ do_arch_prctl_64 ➡️ shstk_prctl ➡️ shstk_setup ➡️ alloc_shstk

- There is one shadow stack per thread
- This shadow stack region is mapped, but the process cannot read or write into it
  - Kernel VMA: VM_SHADOW_STACK
  - PTE: ~_PAGE_RW & _PAGE_SAVED_DIRTY

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 (Shadow Stack)

- Does CET Shadow Stack protect programs against all kind of attacks?

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

Why is Coarse-Grained CFI useful?

- Coarse-grained CFI still prevents classic code-reuse

**(supposedly)*

  - ROP

```
pop rax
ret
```

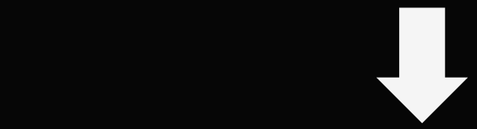
  - JOP

```
mov rax, [rsp]
mov rdi, [rsp+0x8]
jmp rdi
```

CFG: Control Flow Guard

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

```
call qword ptr [rdi]
```



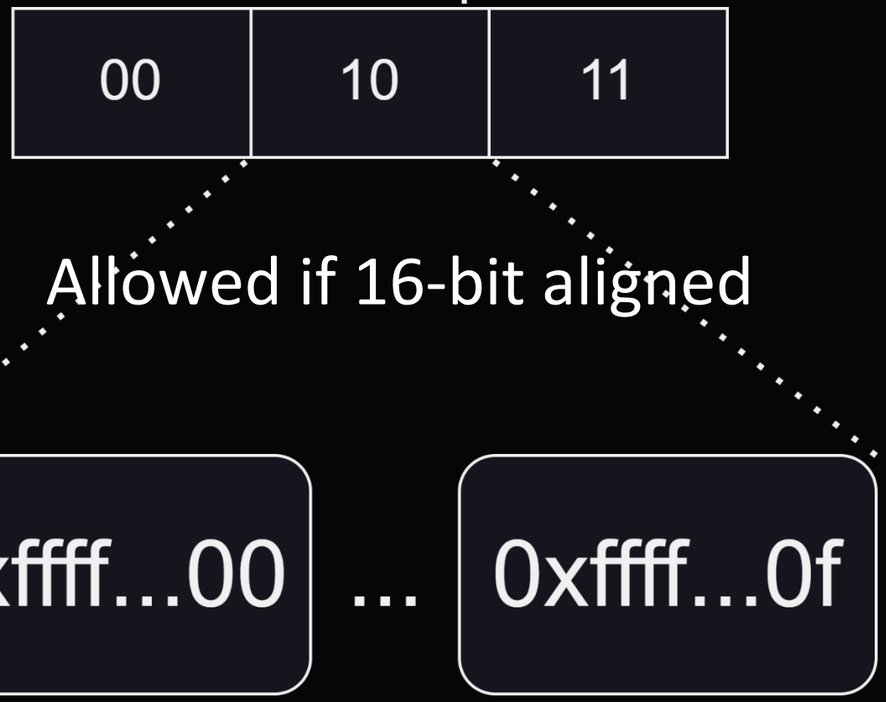
```
call qword ptr [binary!__guard_dispatch_icall_fptr]
```

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

```
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)

- Medium-grade CFI: label based
- Flag *–fsanitize=cfi-icall* in Clang/LLVM
- Each function is assigned a dynamic type
  - Signature: return type, argument types

```
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)

- Medium-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)
```

SIGNATURE: only functions with…

return type = *int*

argument1 type = *int*

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
```

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)

- Medium-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

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?

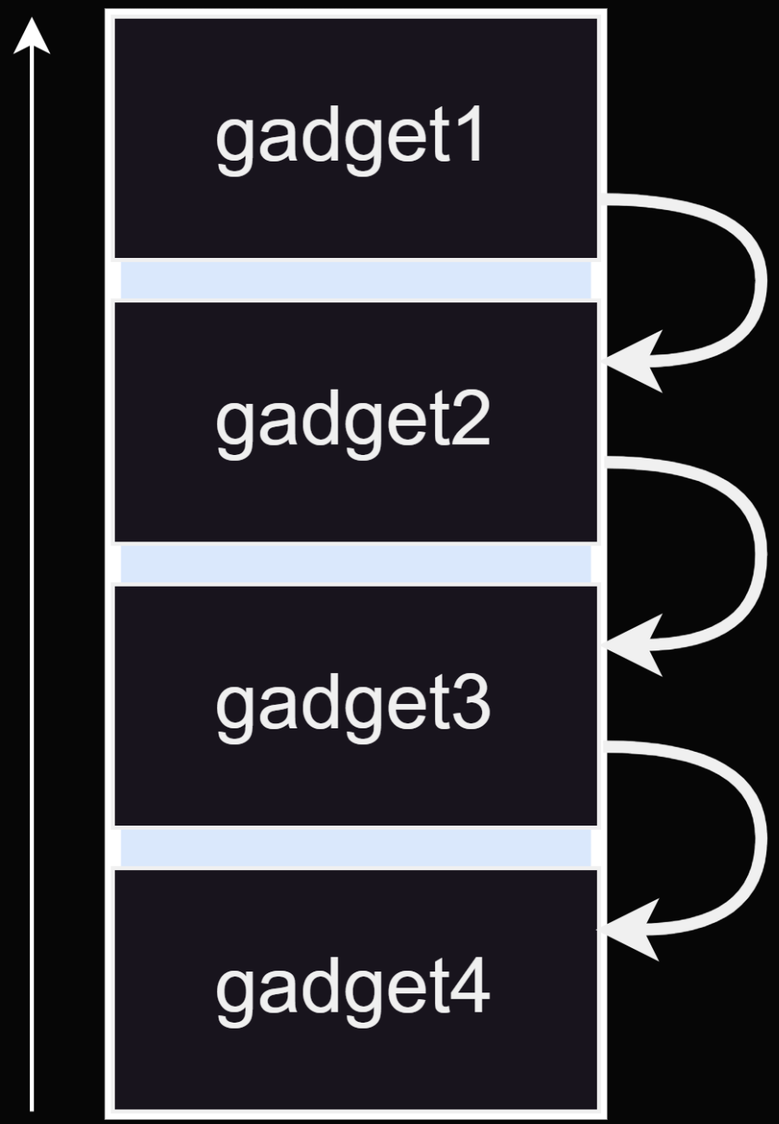
Approach

Gadgets

```
…
inc rax
ret

…
pop rdi
ret

…
pop rsi
ret

…
mov rdx, rcx
ret
```

Approach

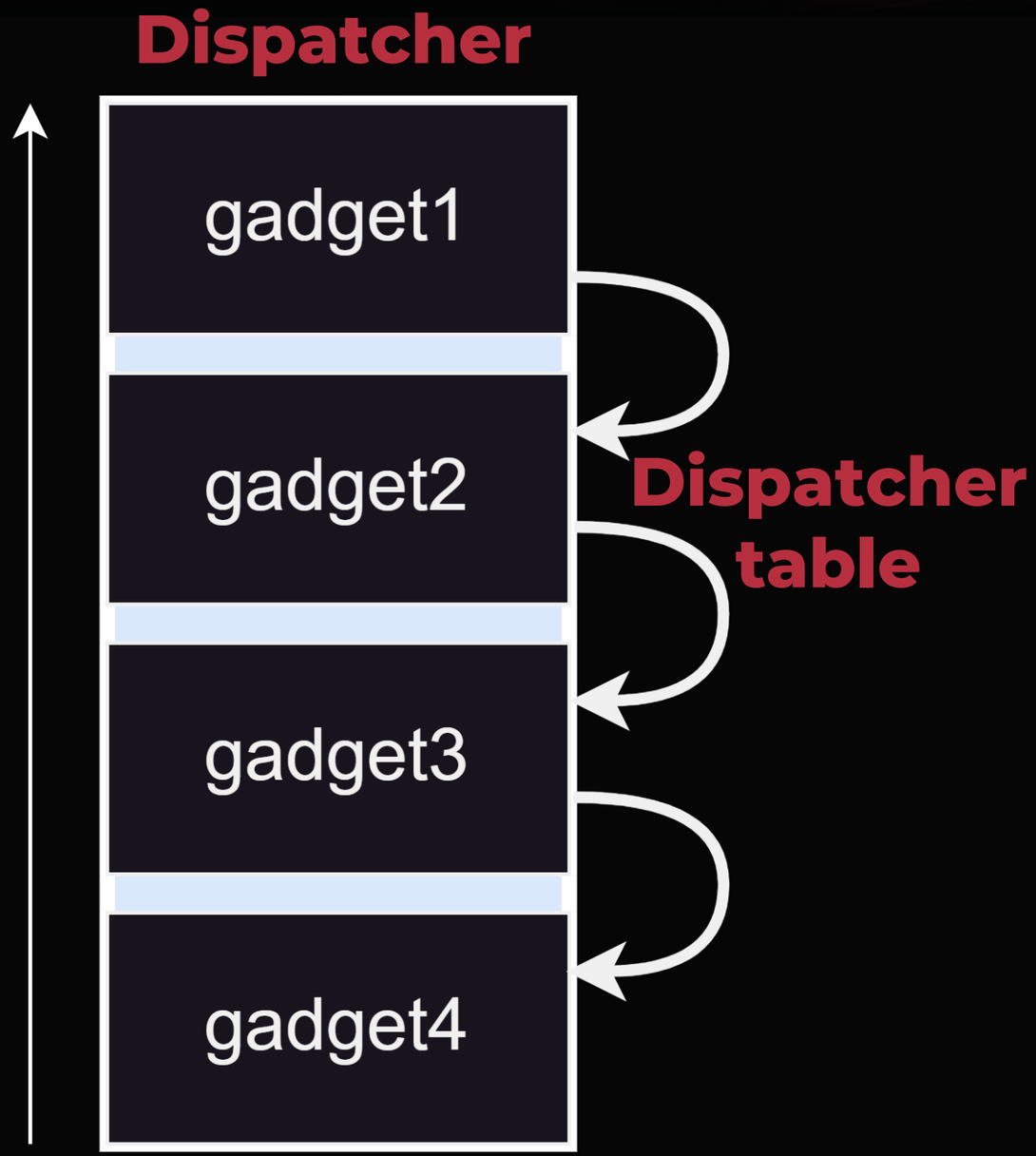
Dispatcher



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

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 medium/fine-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 medium/fine-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

- A coroutine is a function that can suspend and resume

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

   bar();
}
```

Function

```
void bar()
{

   …

}
```

What is a Coroutine

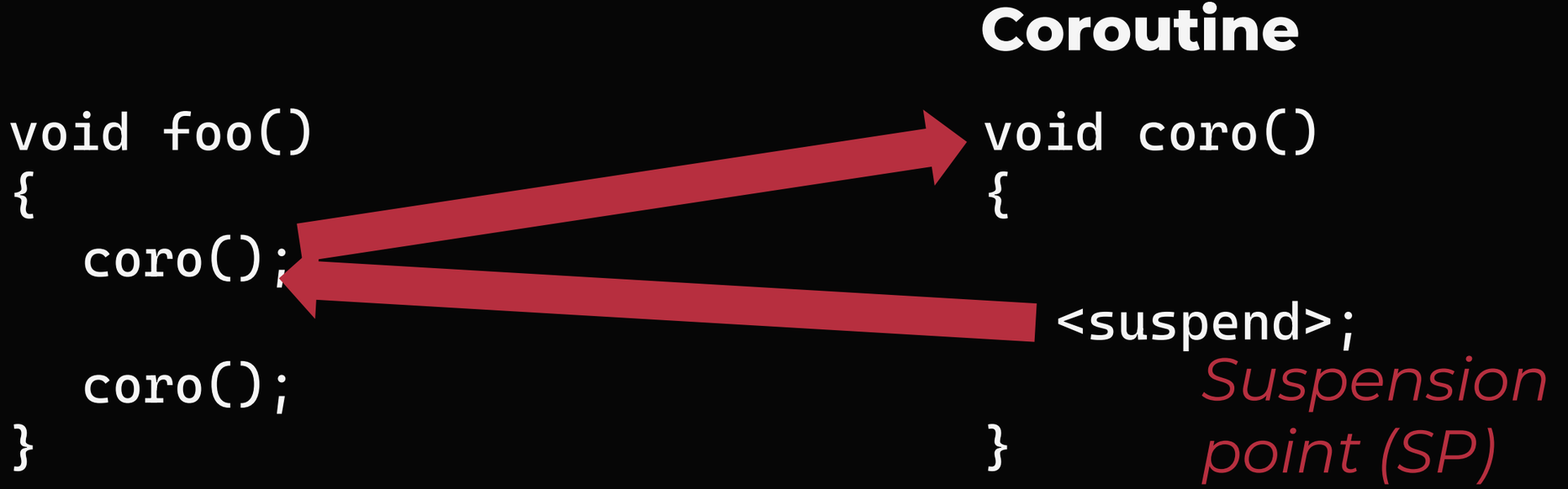
- A coroutine is a function that can suspend and resume

What is a Coroutine

- A coroutine is a function that can suspend and resume

What is a Coroutine

- A coroutine is a function that can suspend and resume



What is a Coroutine

- 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()
{
   coroutine_handle<> h1 = coro().handle;
}
```

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

Coroutine Handle

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

```
void foo()
{
   coroutine_handle<> h1 = 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;

   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:

```
task coro()
{

   <suspend>;

}
```

co_yield

co_return

co_await

What is a Coroutine

- *co_yield* suspends and returns a value

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

   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_return* suspends and returns a value

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

   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();
      };
}
```

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;
}
```

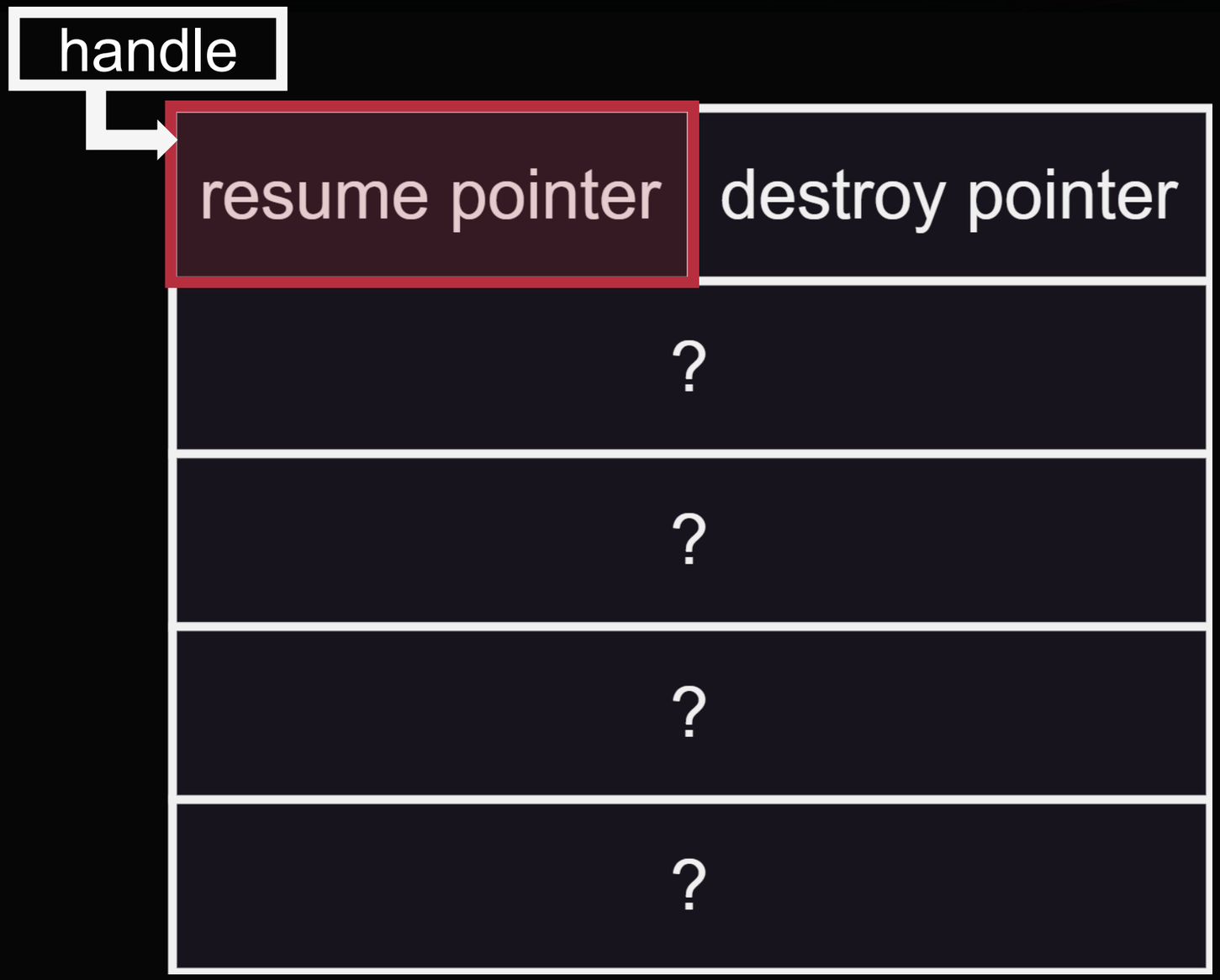
2 allocated frames

The Coroutine Frame

handle

resume pointer	destroy pointer
?	
?	
?	
?	

The Coroutine Frame




```
handle.resume()
```

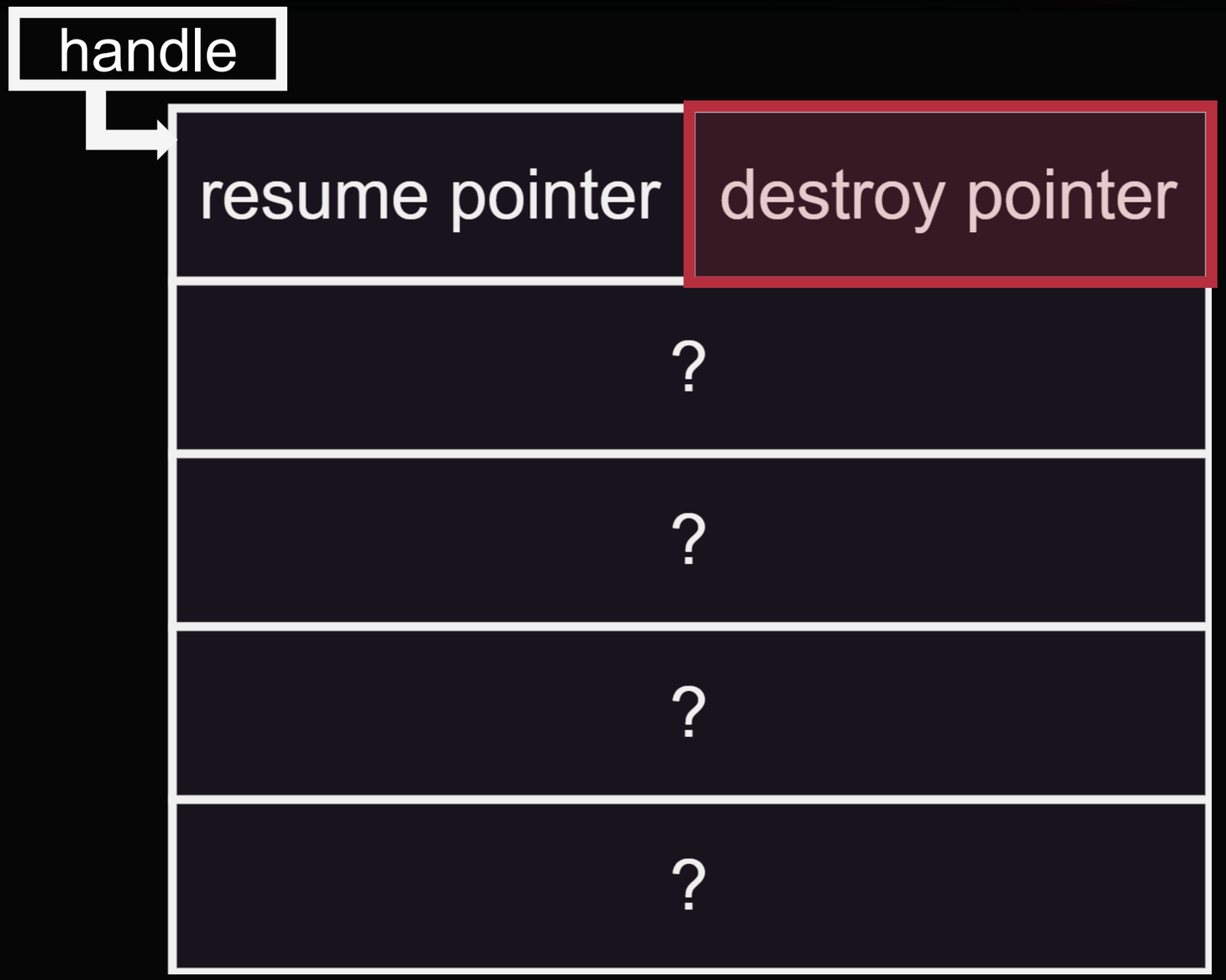
```
call [rdi]
```

handle

resume ptr

- Points to the resume stub

The Coroutine Frame




```
handle.destroy()
```

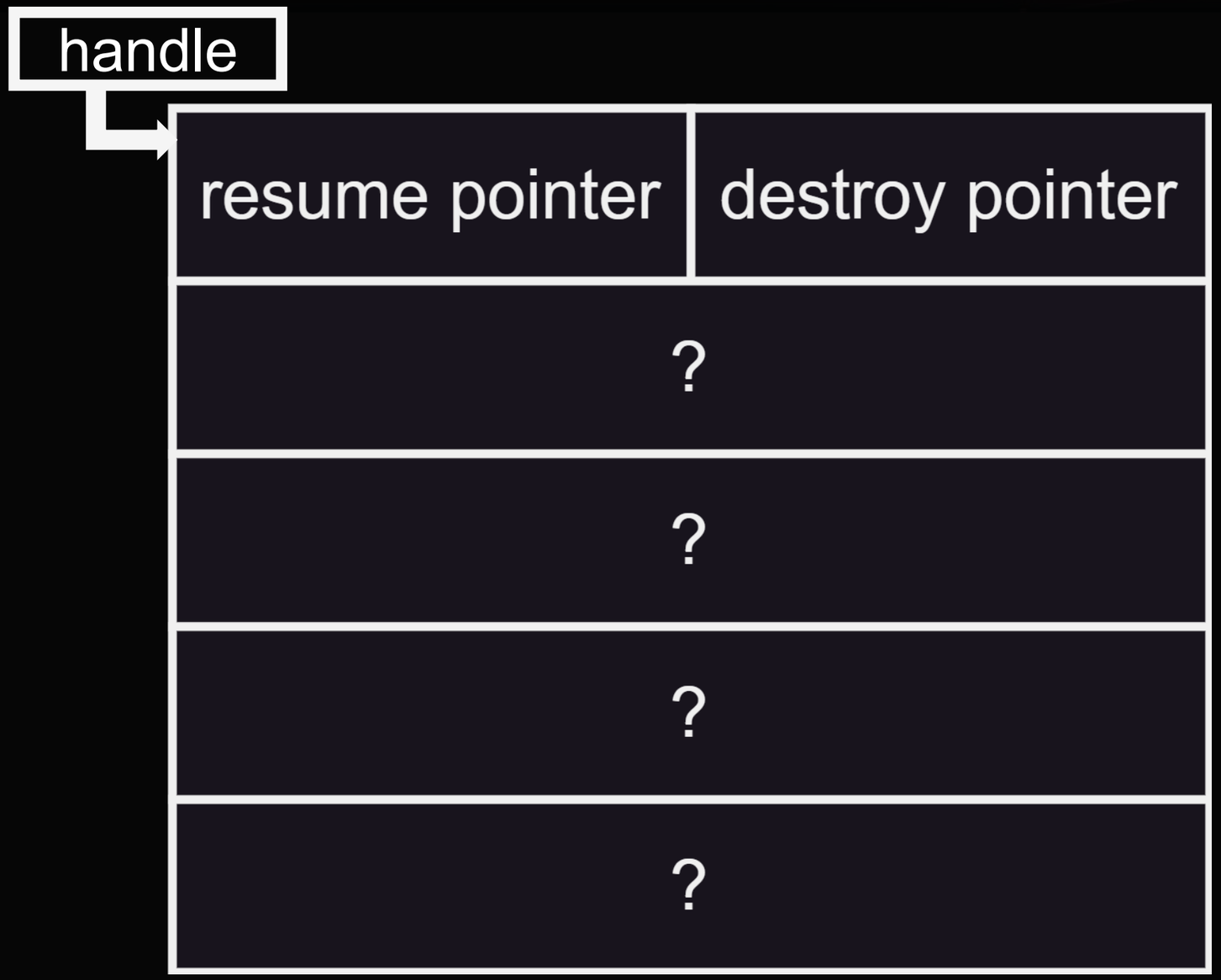
```
call [rdi+0x8]
```

handle

destroy ptr

- 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

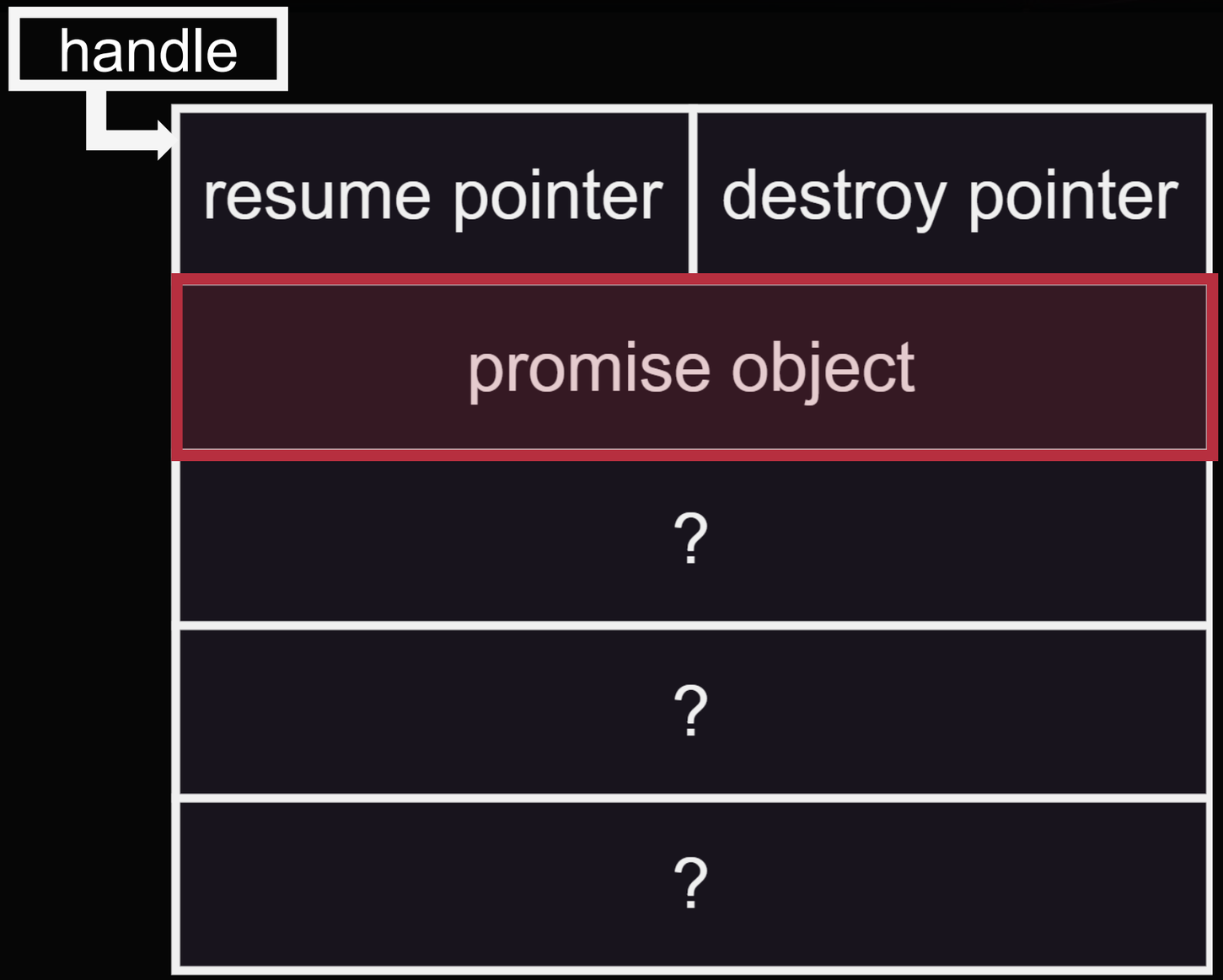
resume pointer	destroy pointer
?	
?	
?	
?	

resume pointer	destroy pointer
?	
?	
?	
?	

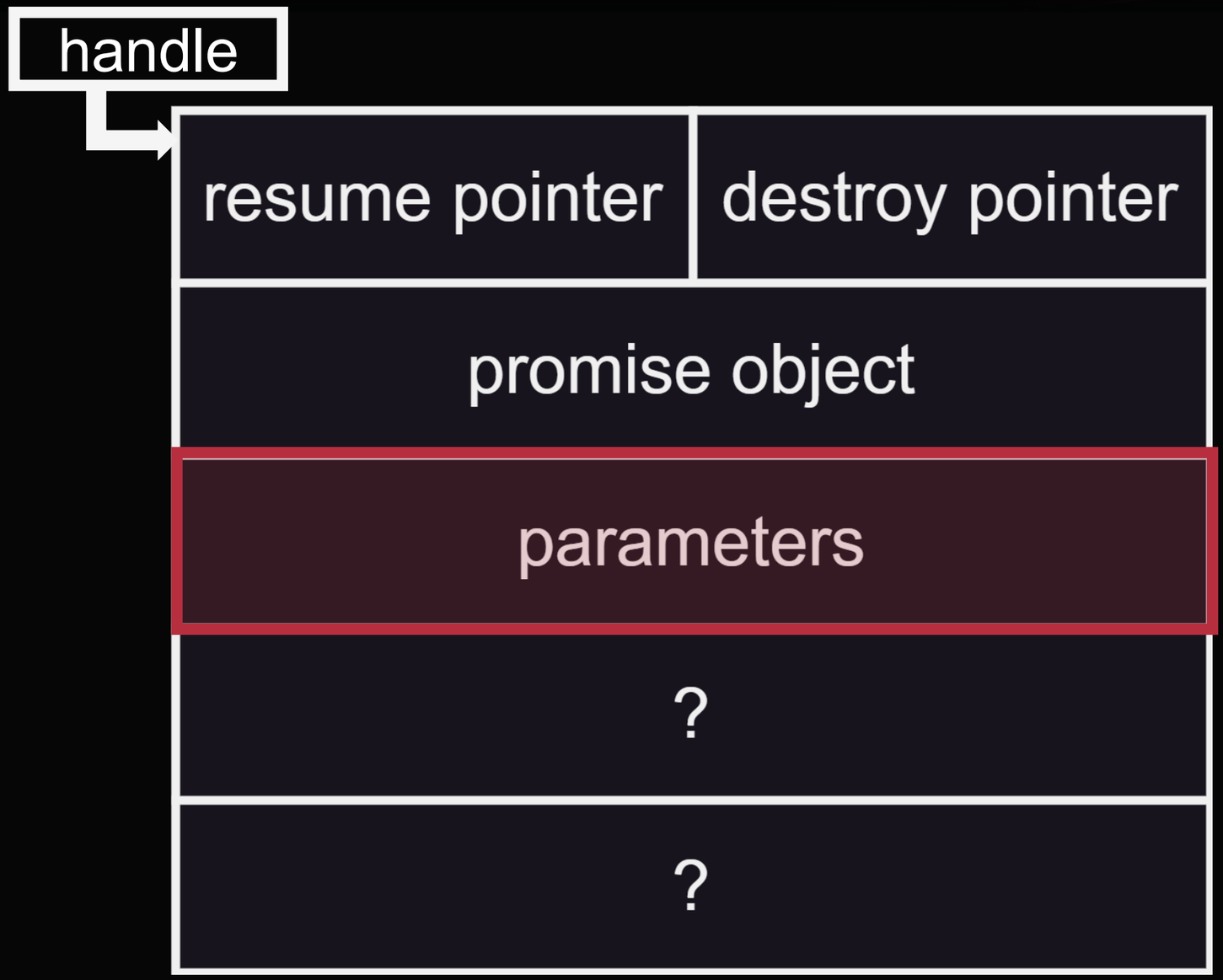
resume pointer	destroy pointer
?	
?	
?	
?	

resume pointer	destroy pointer
?	
?	
?	
?	

The Coroutine Frame

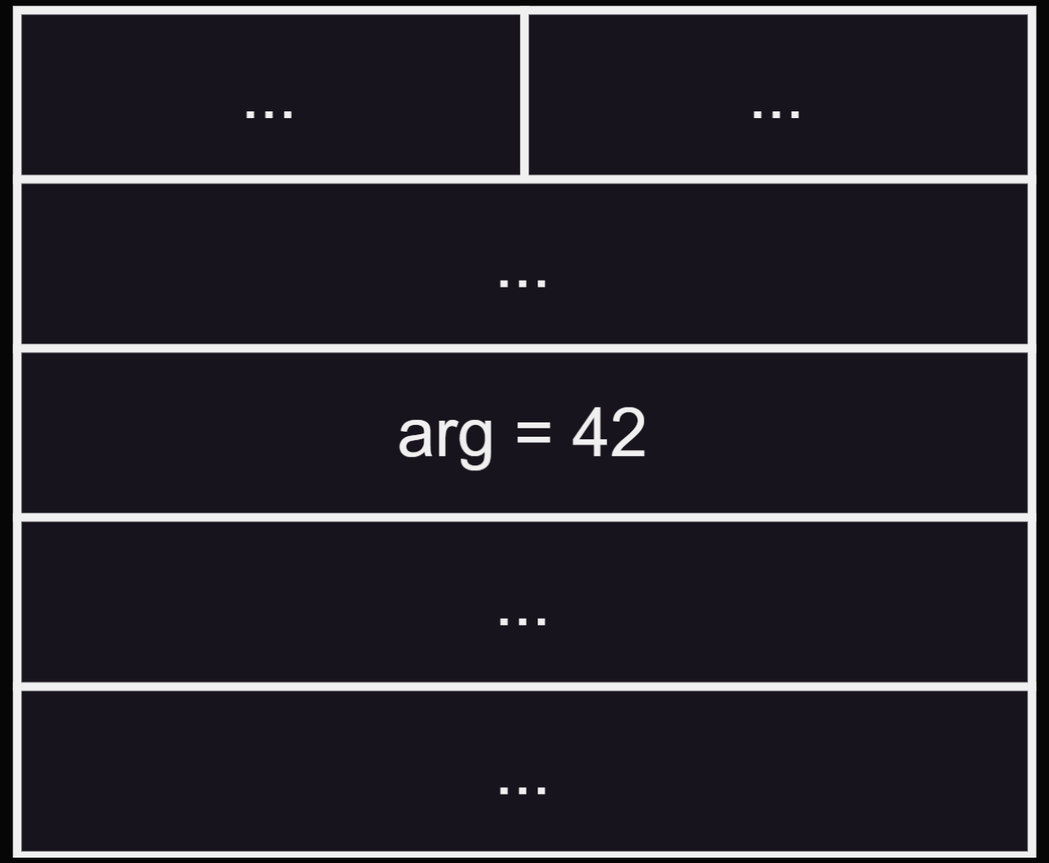


The Coroutine Frame



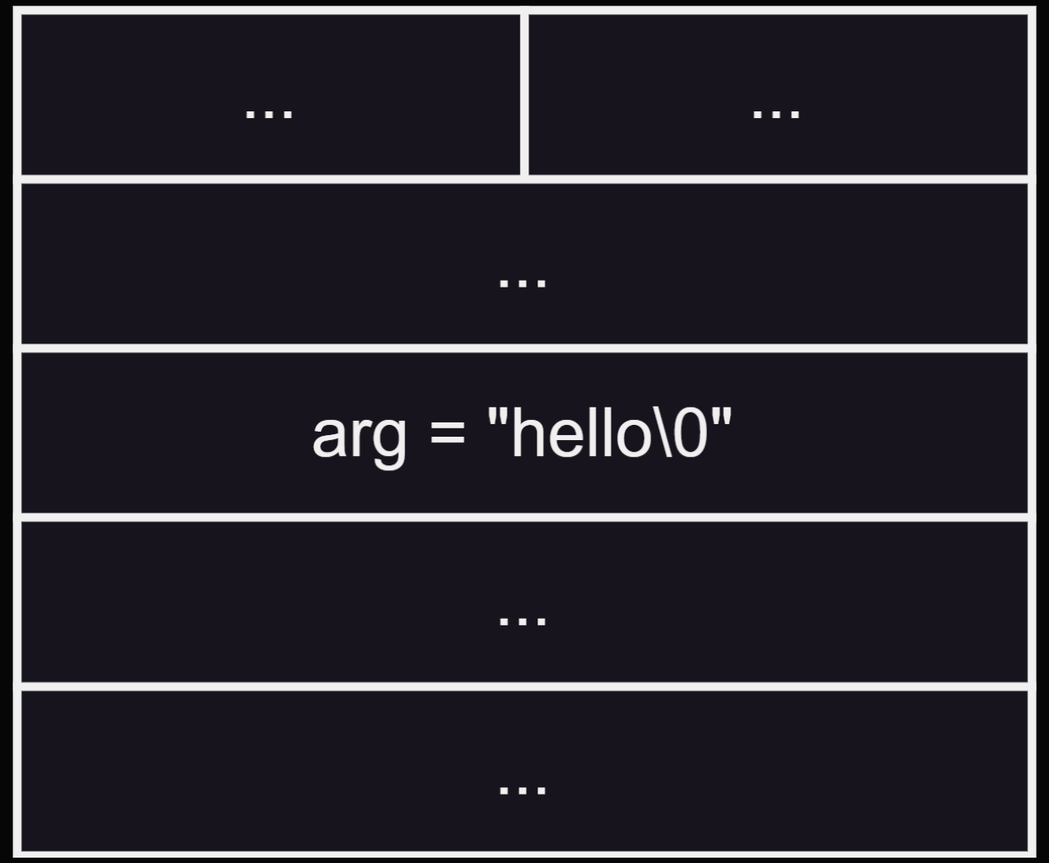
The Coroutine Frame

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

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



The Coroutine Frame

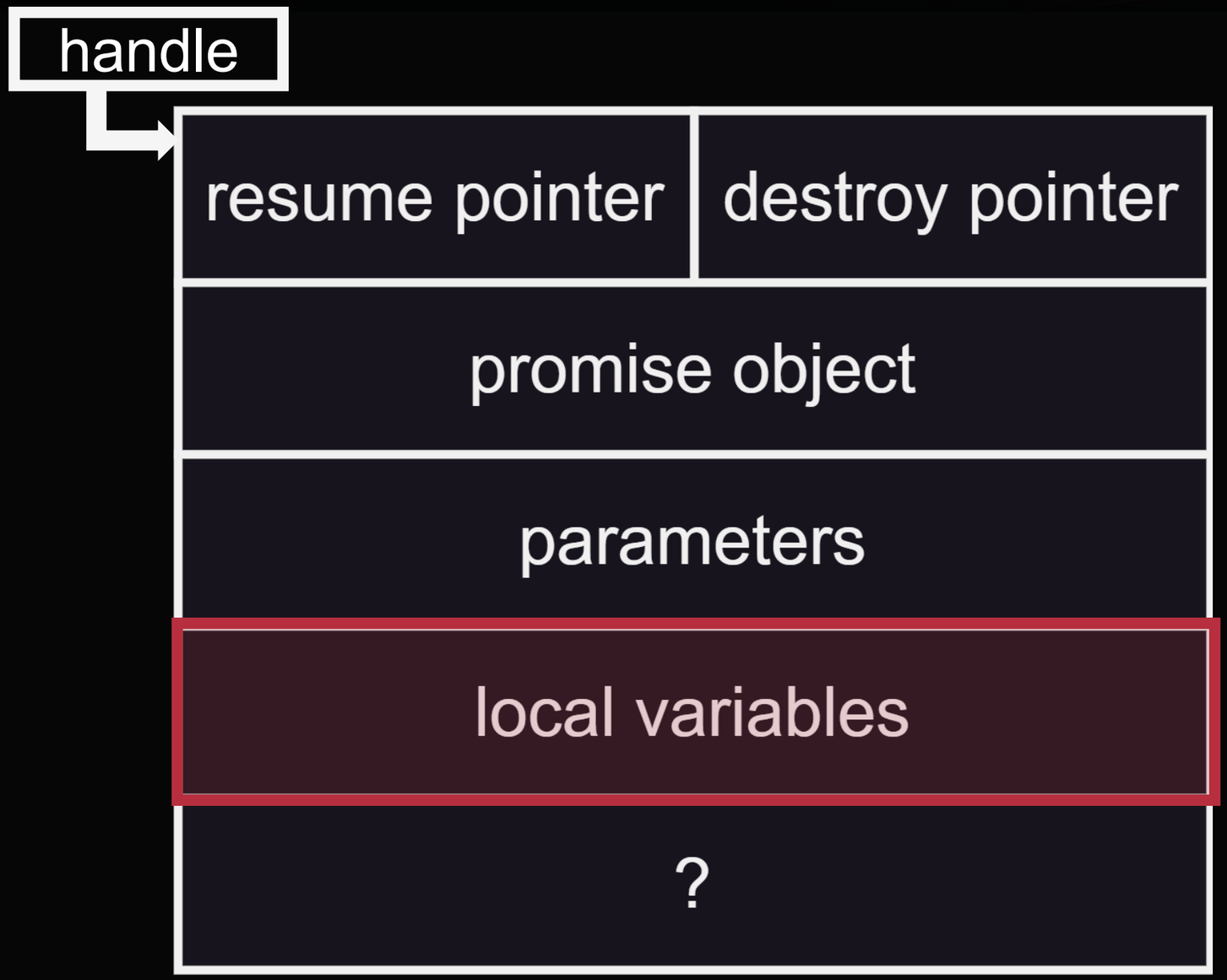
```
void main()
{
   string s = “hello”;
   coro(s);
}
task coro(string arg)
{
   co_return;
}
```



The Coroutine Frame

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

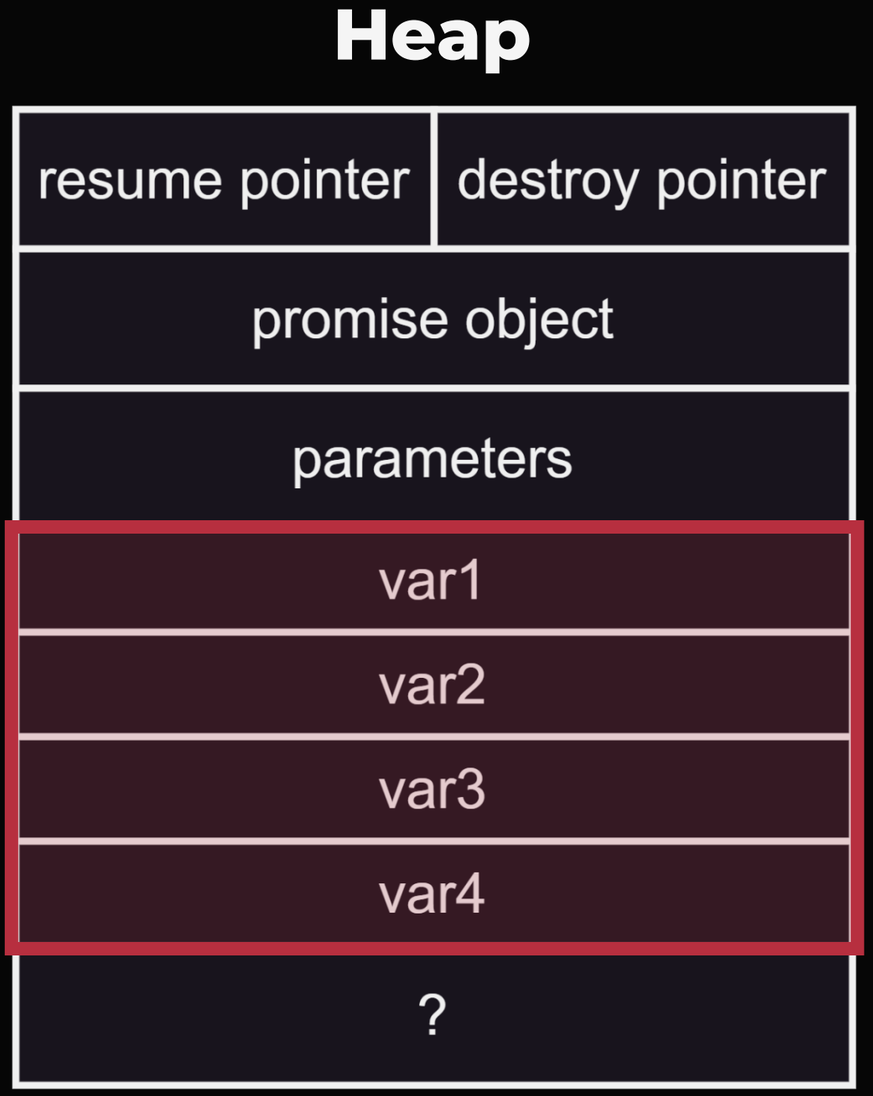
The Coroutine Frame



The Coroutine Frame


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

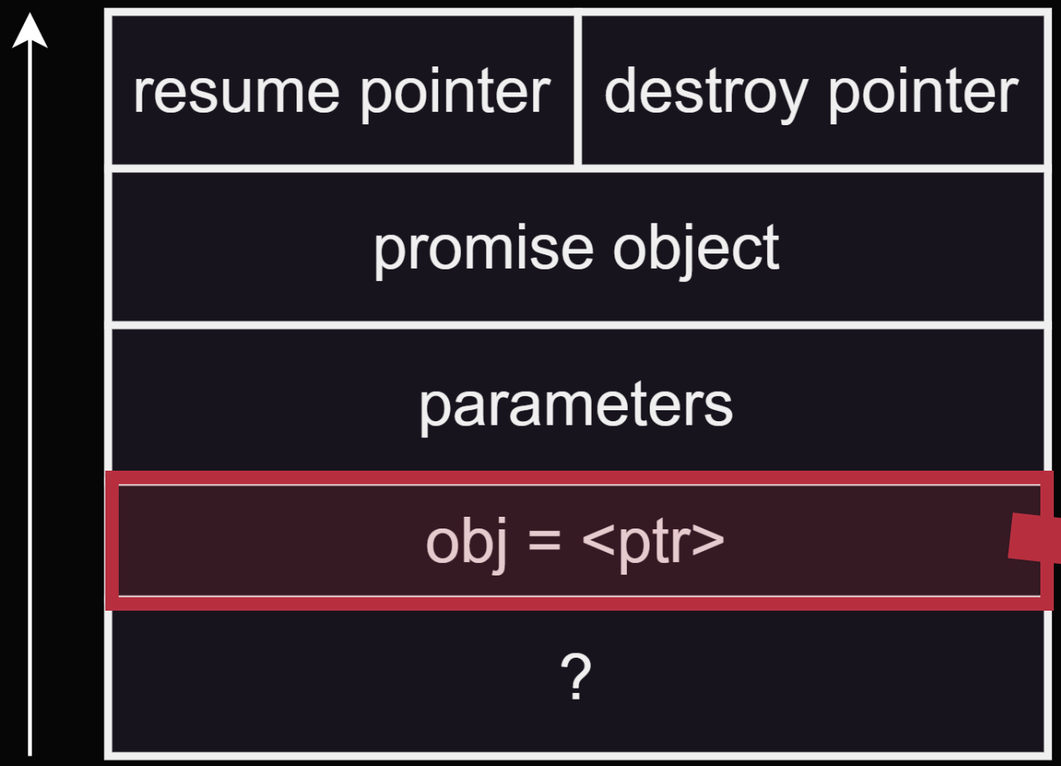
The Coroutine Frame




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

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

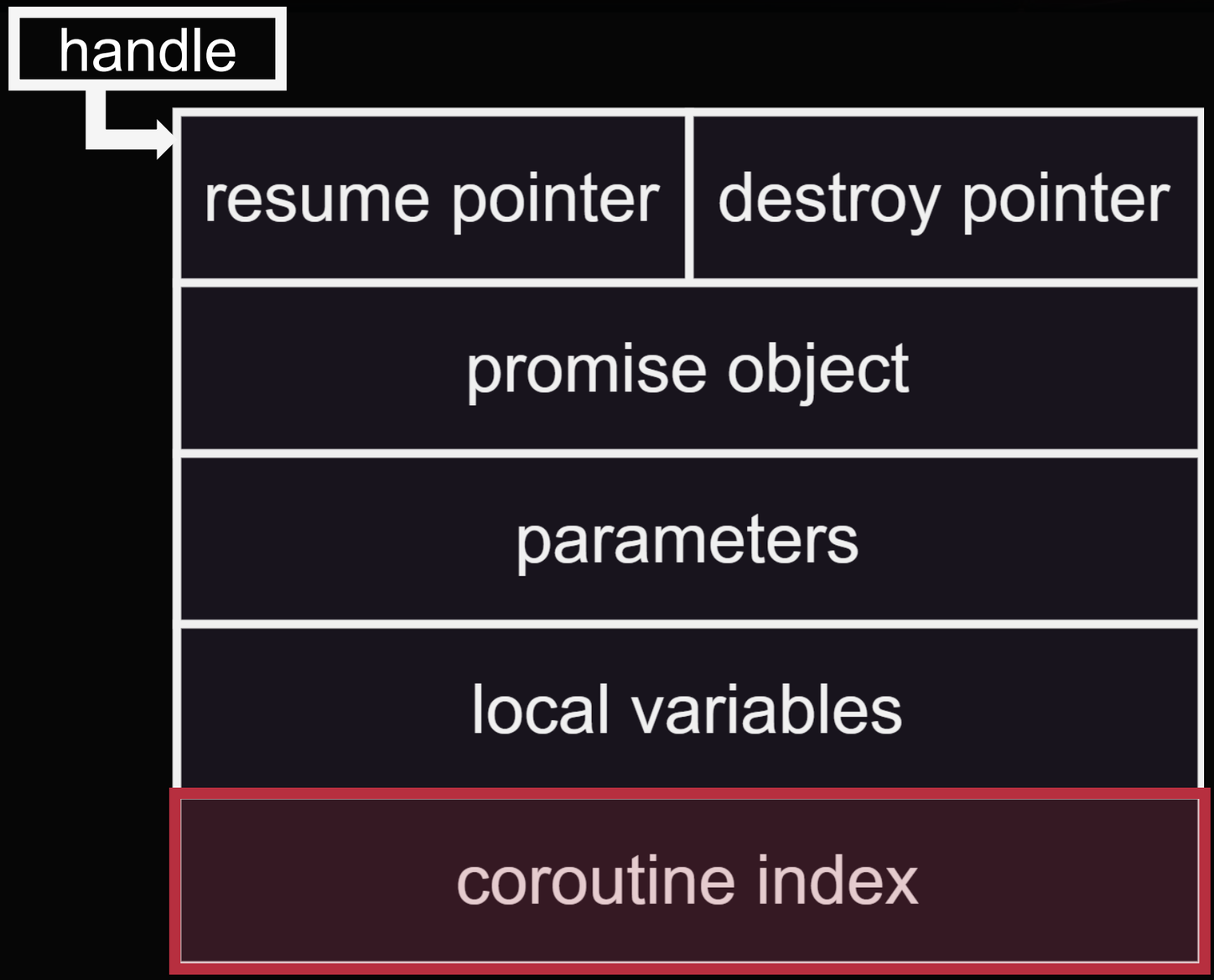
The Coroutine Frame




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

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

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 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

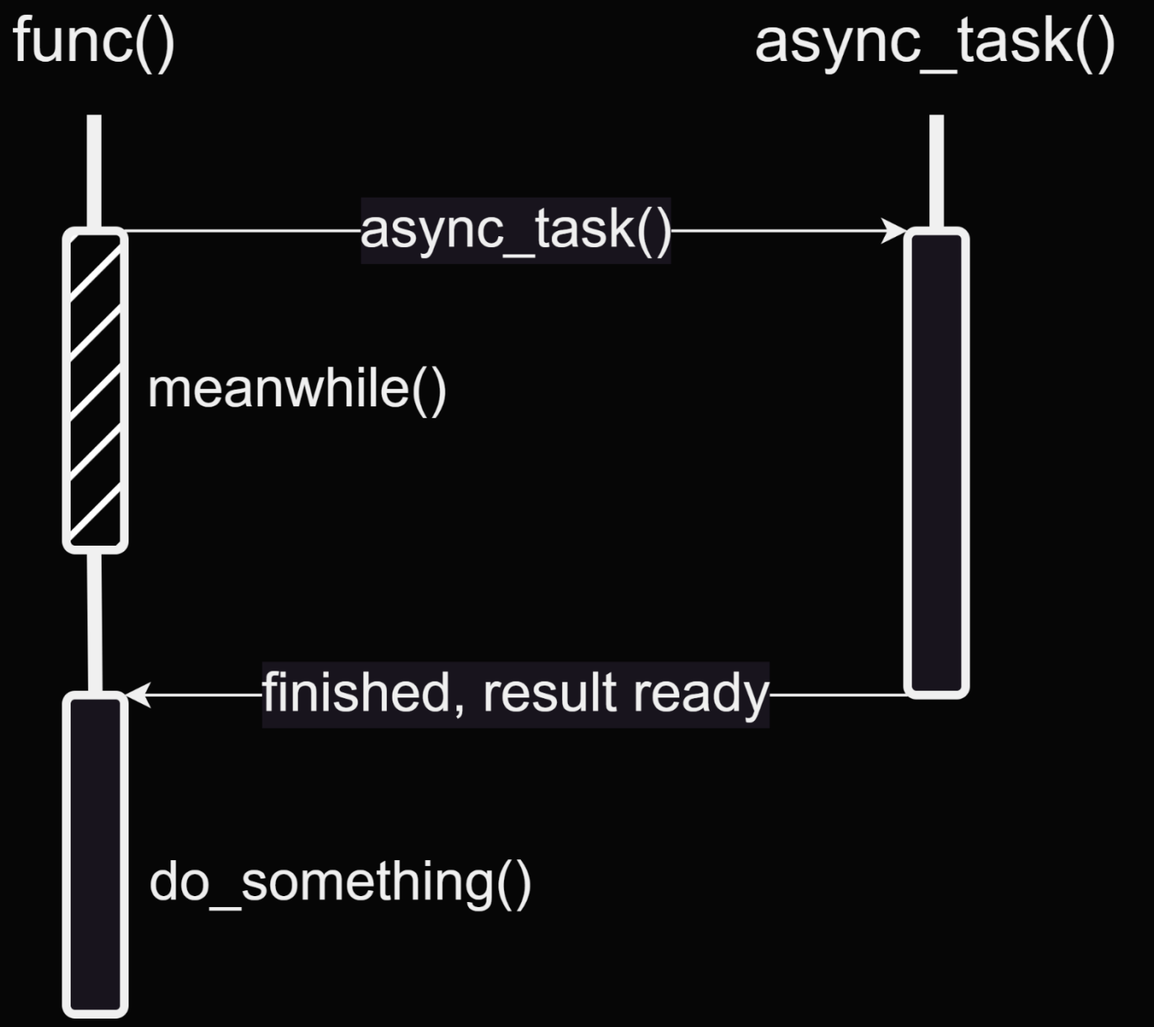
co_return

Coroutine Awaiting

- *co_await* evaluates an awaitable

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

Coroutine Awaiting



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…

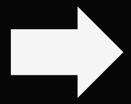
```
struct Awaitable
{
   Awaiter operator co_await()
   {
      return{};
   }
}
```

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

   void await_suspend(…);

   void await_resume();
}
```

co_await ➡ Awaitable ➡ Awaiter

Coroutine Awaiting

```
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?

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
}
```

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

co_await Example

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

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

   <something else>
}
```

t1

t2

```
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();

   <join>
}
```

t2 t1

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

Co_awaiting Coroutines

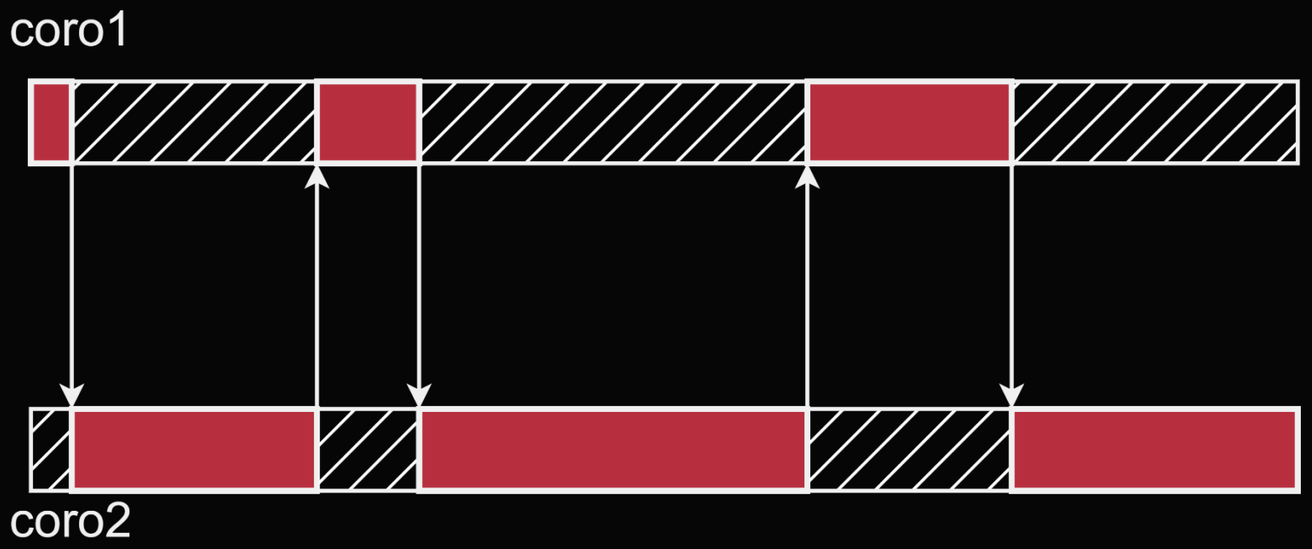
```
task
{
      handle h;
      struct promise_type
      {
         int return_value;
         suspend_always initial_suspend();
         suspend_always final_suspend();
      };
      struct awaiter
      {
         bool await_ready();
         void await_suspend();
         void await_resume();
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

Co_awaiting Coroutines

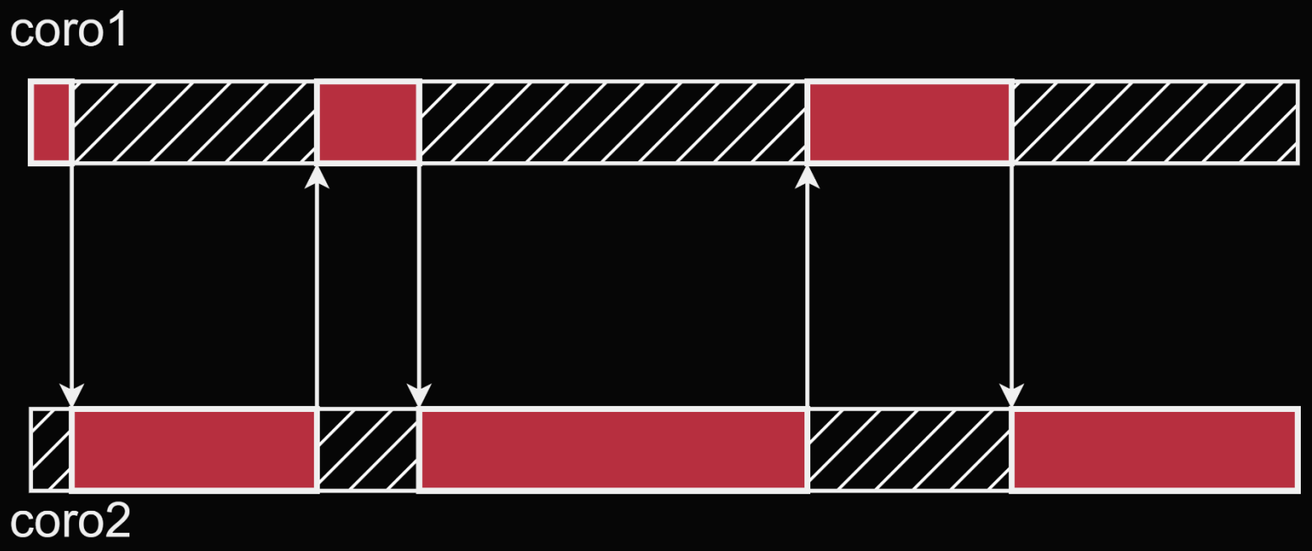
```
task coro2()
{
   co_return;
}

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

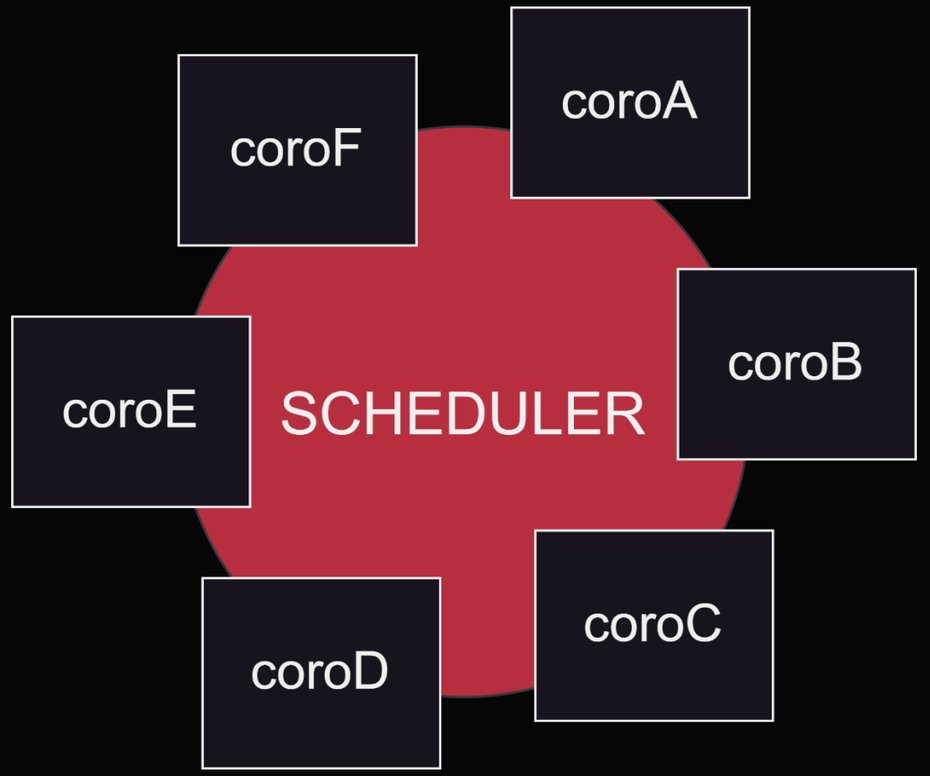
```
task
{
      handle h;
      struct promise_type
      {
         int return_value;
         suspend_always initial_suspend();
         suspend_always final_suspend();
      };
      struct awaiter
      {
         bool await_ready();
         void await_suspend();
         void await_resume();
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

Co_awaiting Coroutines



Co_awaiting Coroutines





4 CFOP

Threat model

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

Threat model

- CFI threat model:
  - ASLR bypassed
  - Arbitrary 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

CFI Defenses Considered

Intel CET

LLVM CFI (icall)

LLVM KCFI

Safe Dispatch

Control Flow Guard

MCFI/ piCFI

ReCFI

Path Armor

VfGuard

CFIXX

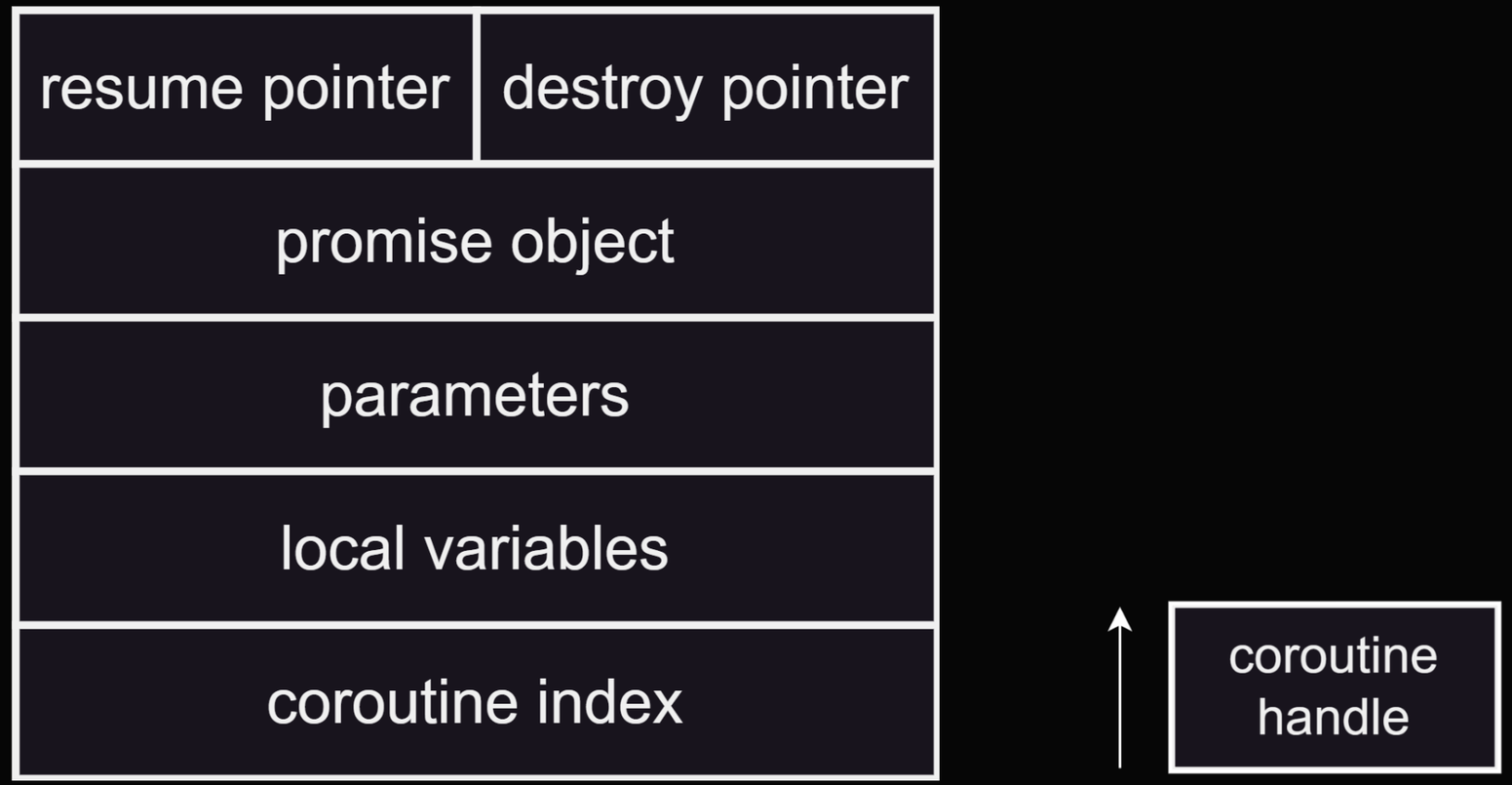
PittyPat

VTrust

Typro

Observations

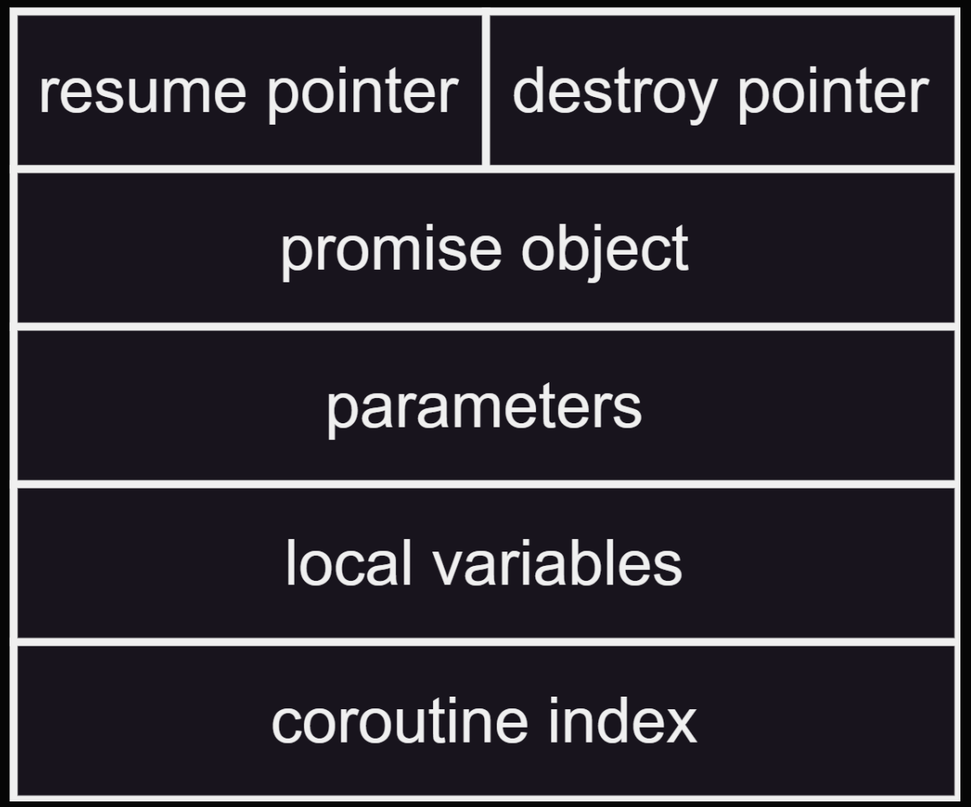
- The coroutine handles and frames are writable



Attack primitives

FRAME MANIPULATION

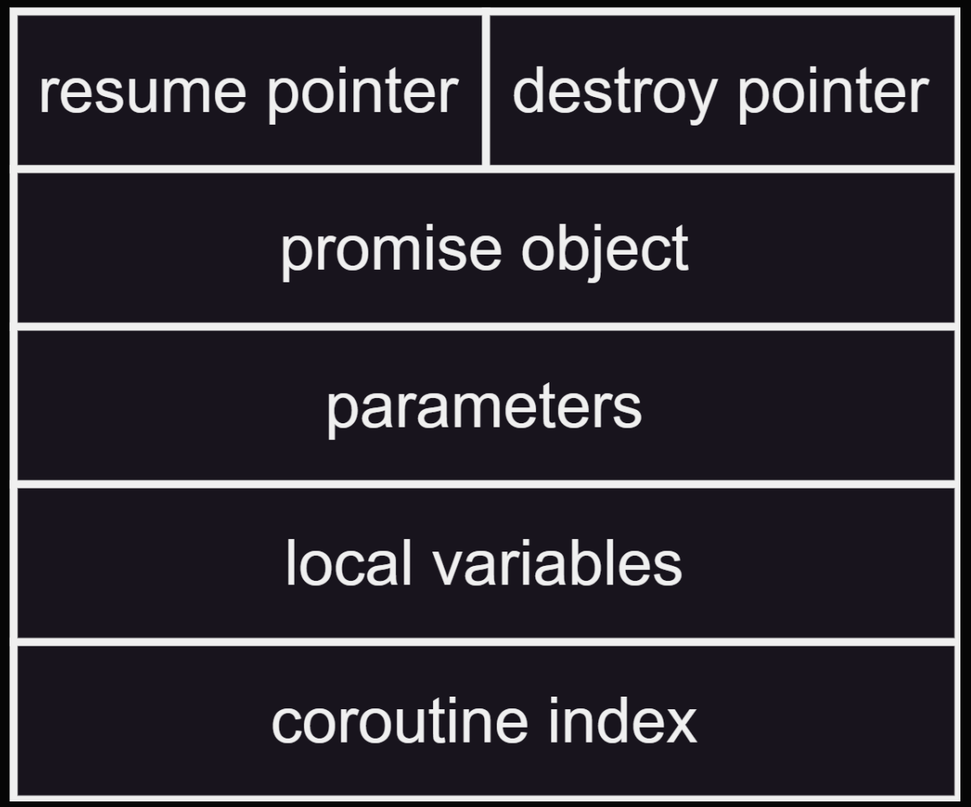
- Modifying existing frames



Attack primitives

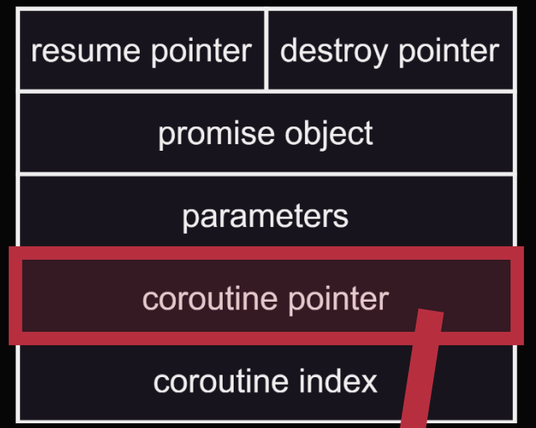
FRAME MANIPULATION

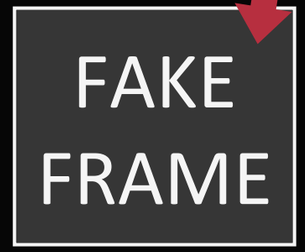
- Modifying existing frames

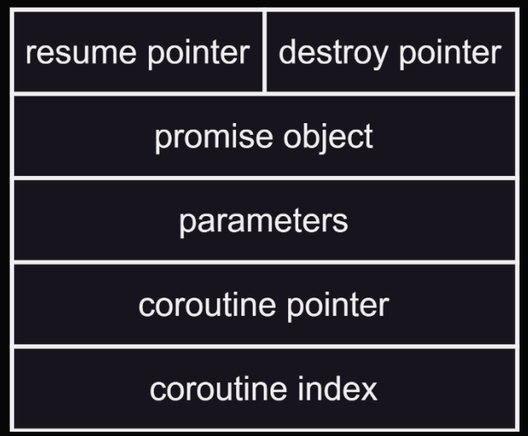


FRAME INJECTION

- Inserting new frames







DOA: Data Only Attack

- Modifying the runtime data of a program can lead to arbitrary code execution
- Data-Only Attacks (DOA) modify existing frames

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);
}
```

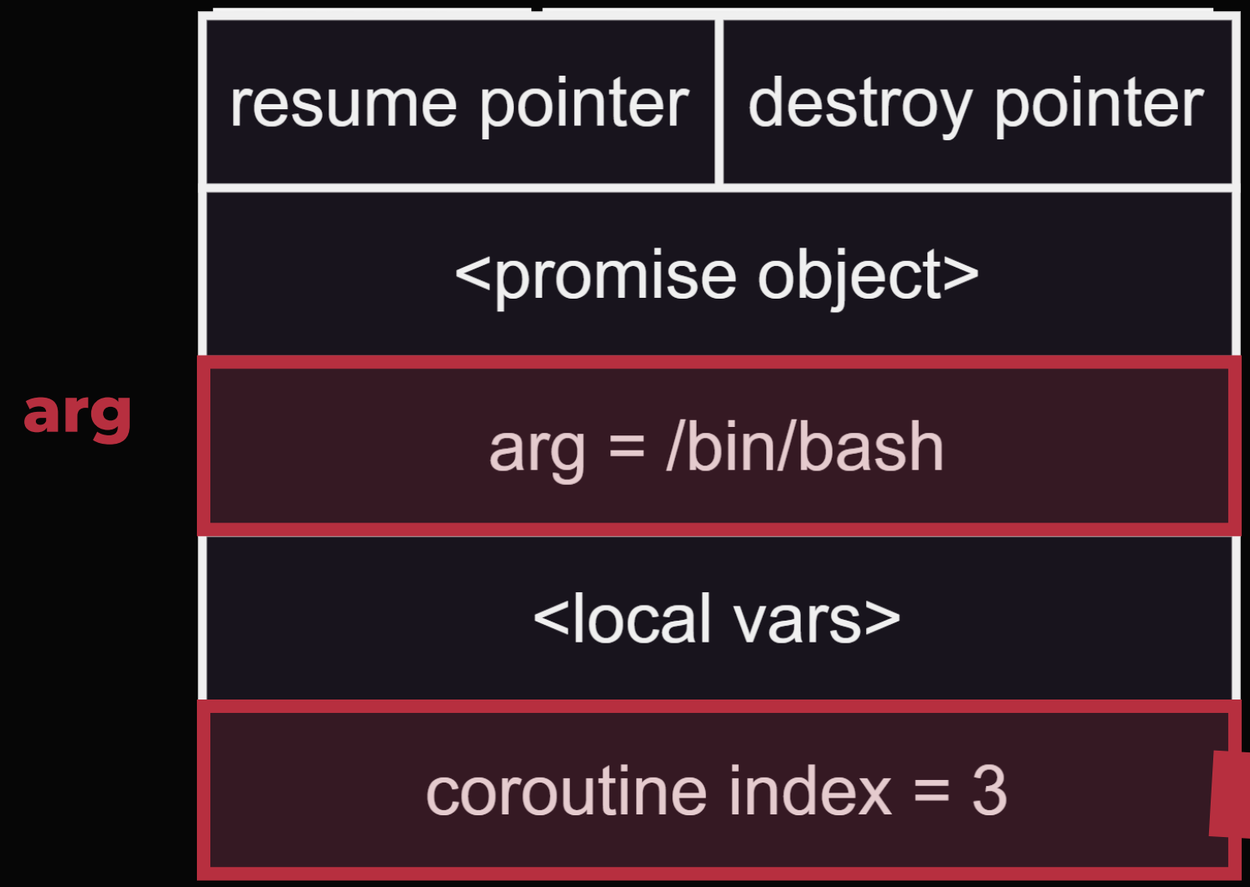
DOA: Data Only Attack

arg Init Vuln Vuln Vuln

```
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.

Advanced DOAs

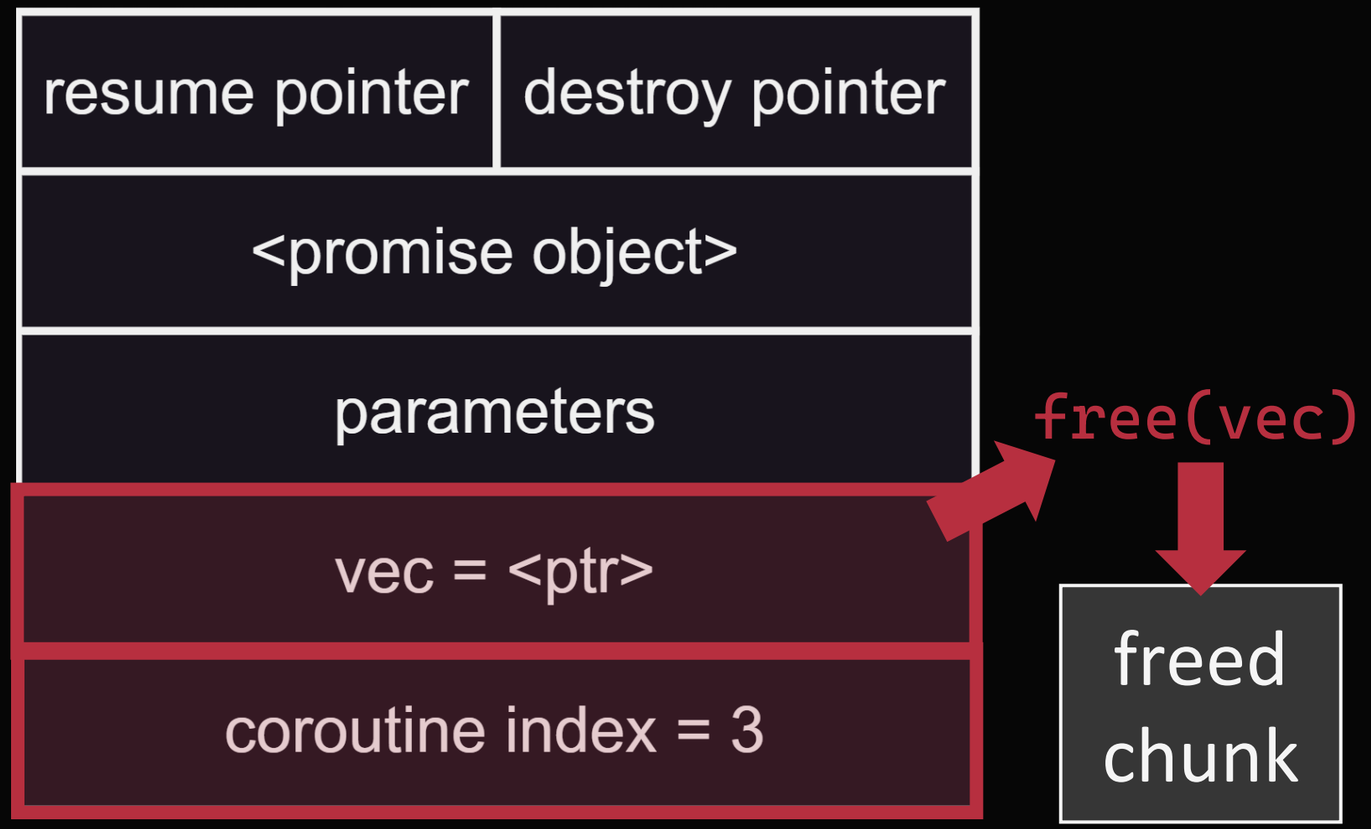
```
task coro(char* arg)
{
★ //SP1//
  vector<int> vec;
  vec.push_back(1);
  co_await some_task;
★ //SP2//
  co_await some_task;
★ //SP3//
}
```

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



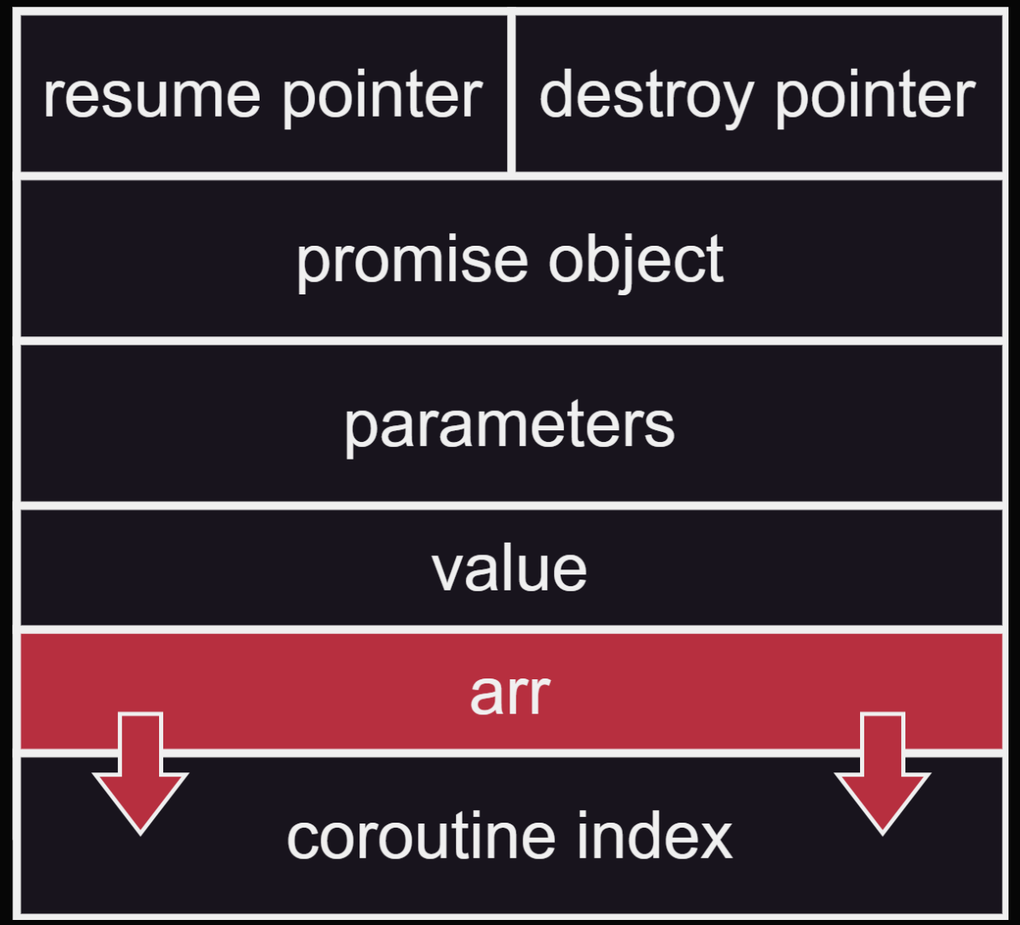

```
task coro(char* arg)
{
★ //SP1//
  initial_suspend();
  vector<int> vec;
  vec.push_back(1);
  co_await some_task;
★ //SP2//
  co_await some_task;
★ //SP3//
  <FREE ALL VARIABLES>
  final_suspend();
}
```

Advanced DOAs

clang++-19 -O3

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

Safe reordering



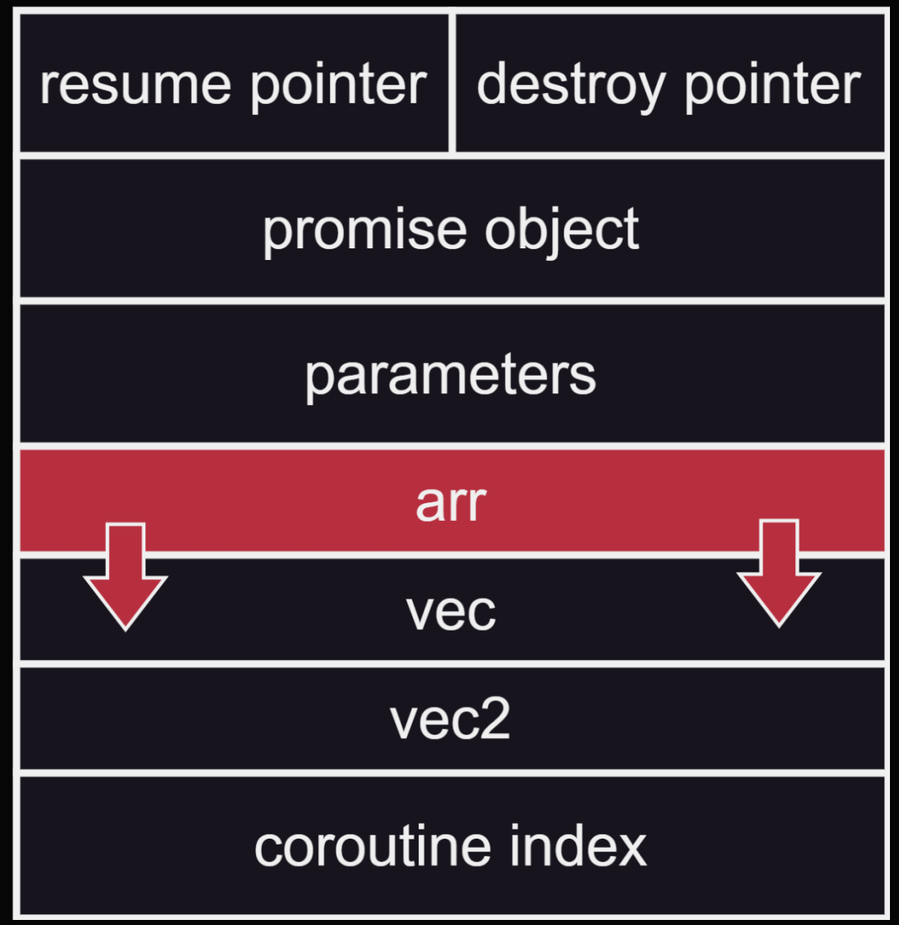
Advanced DOAs

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



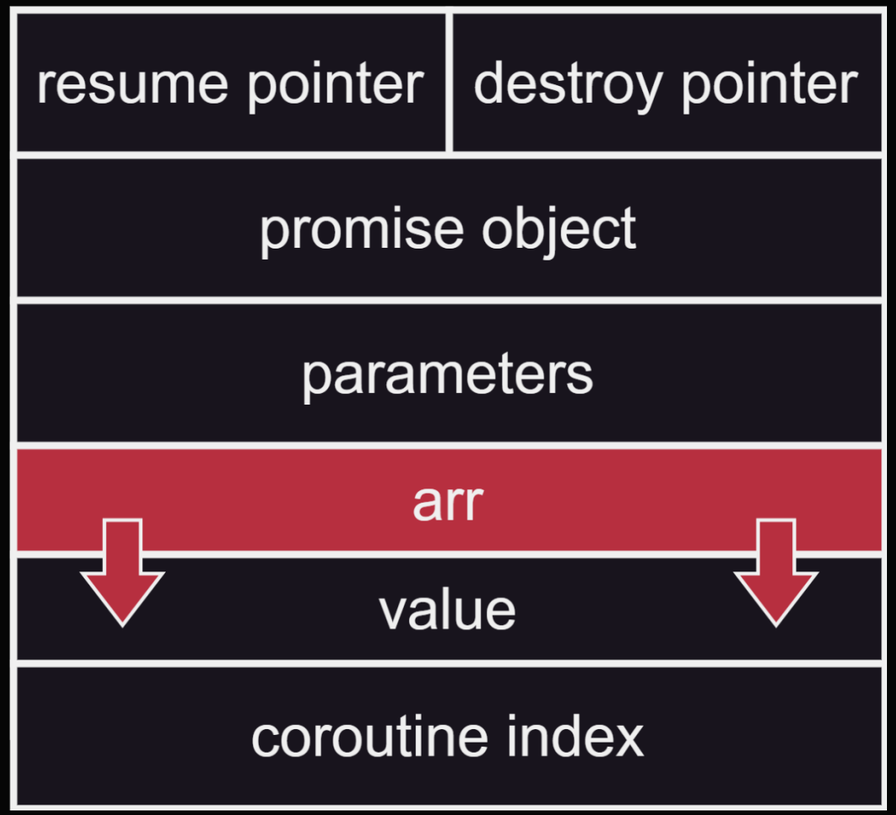
Advanced DOAs

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

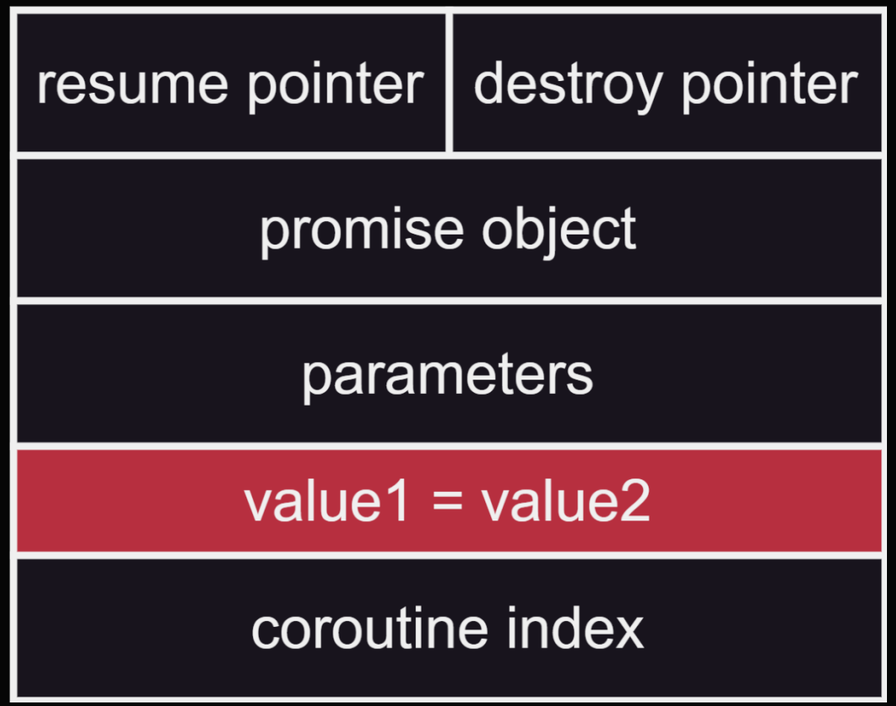
Ok, that was weird



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;
}
```

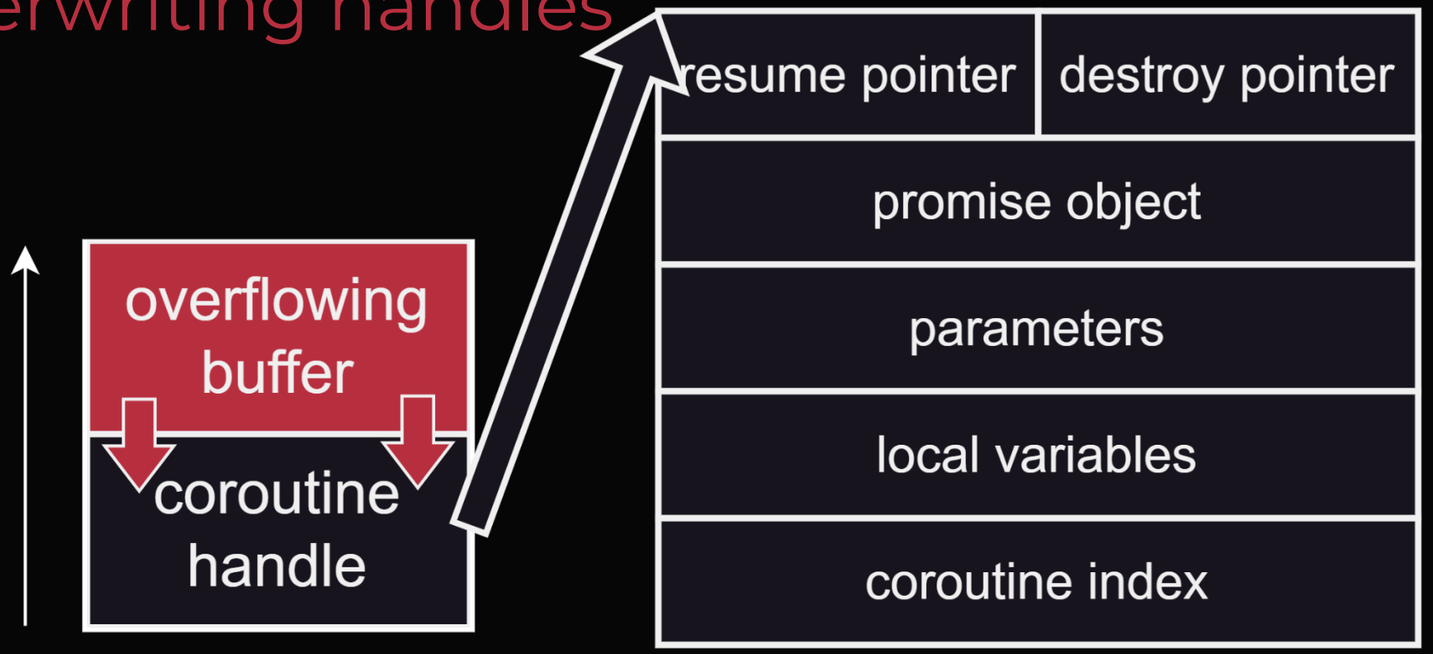


Revisiting the Threat model

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

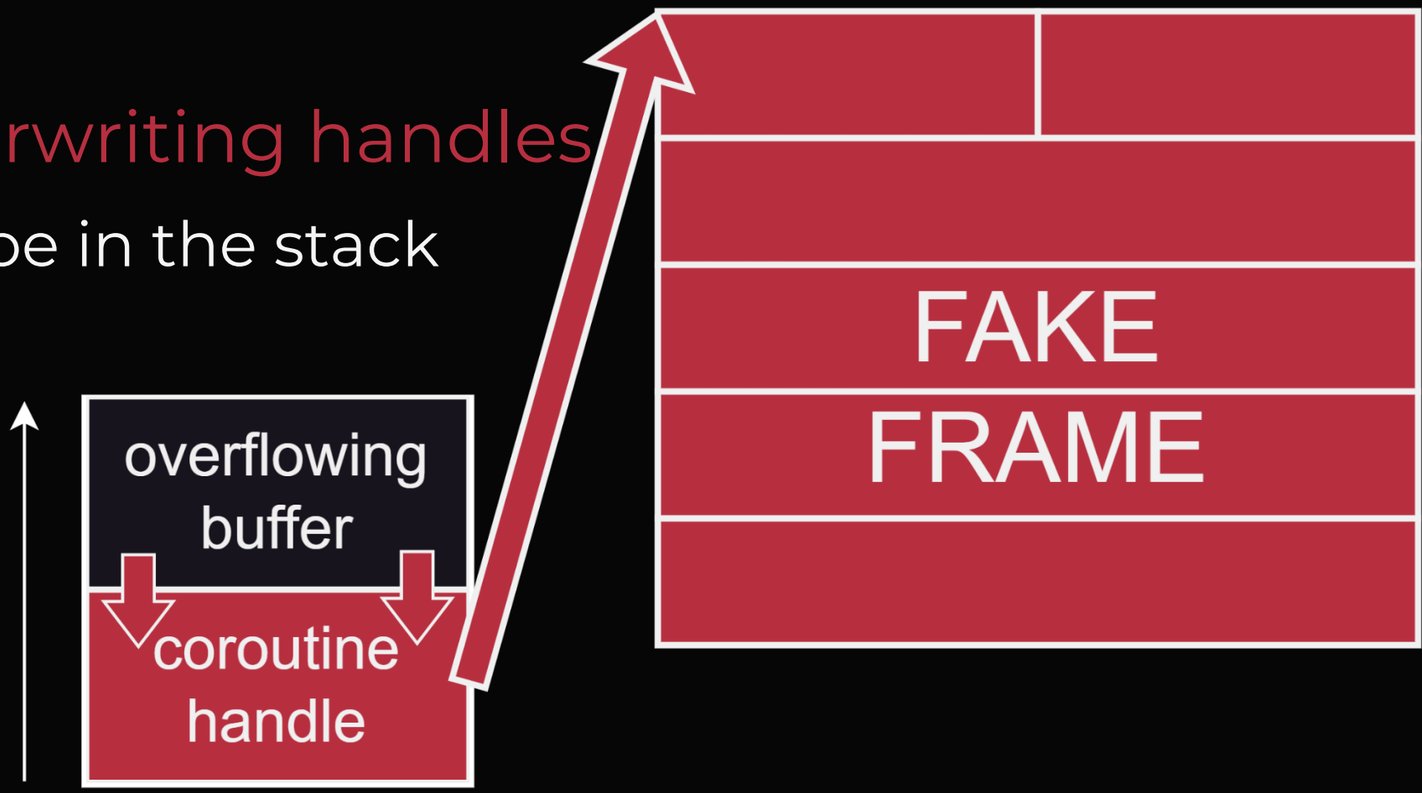
Revisiting the Threat model

- 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

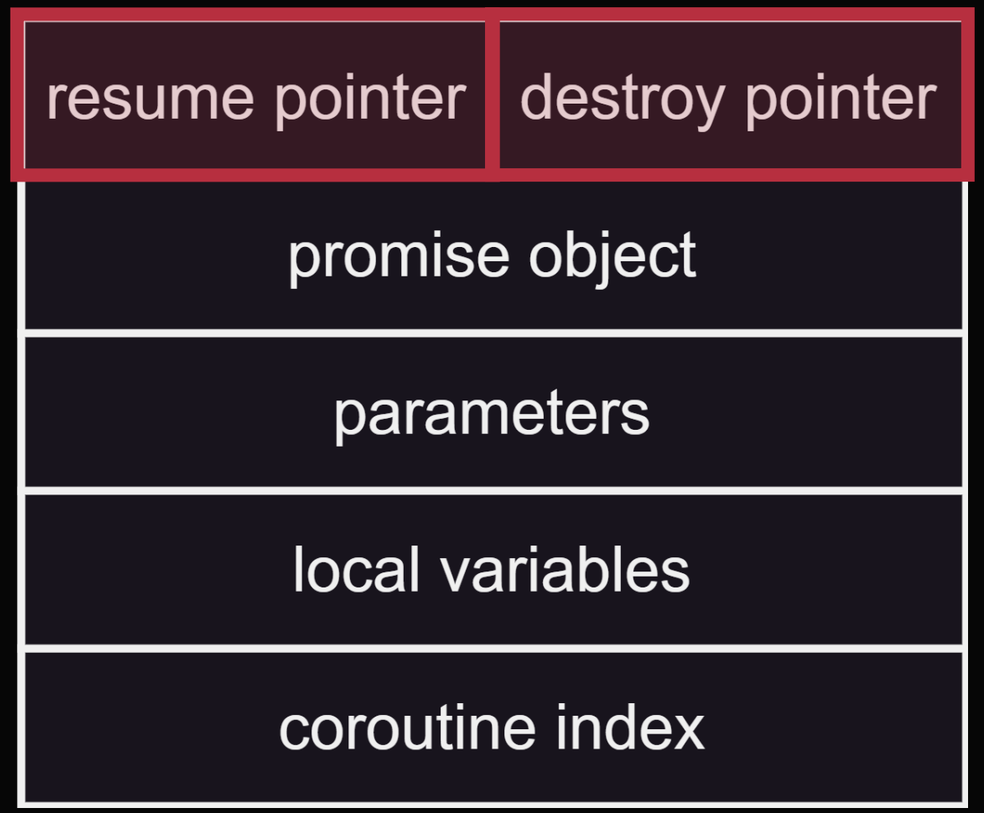
- Launching a DOA (and other CFOP attacks) requires either frame manipulation or frame injection
- Options:
  1. Arbitrary memory write
  2. Stack-based overflow overwriting handles
  3. Stack-based overflow inside the coroutine
  4. Heap-based overflow overwriting subsequent frames

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




```
handle.resume()
```

handle

```
call [rdi]
```

resume ptr

```
handle.destroy()
```

handle

```
call [rdi+0x8]
```

destroy ptr

Overview of CFI Defenses

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

Overview of CFI Defenses

Intel CET

LLVM CFI (icall)

LLVM KCFI

Control Flow Guard

MCFI/ piCFI

Safe Dispatch

ReCFI

Path Armor

VfGuard

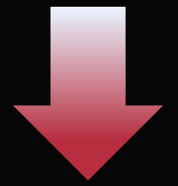
CFIXX

PittyPat

VTrust

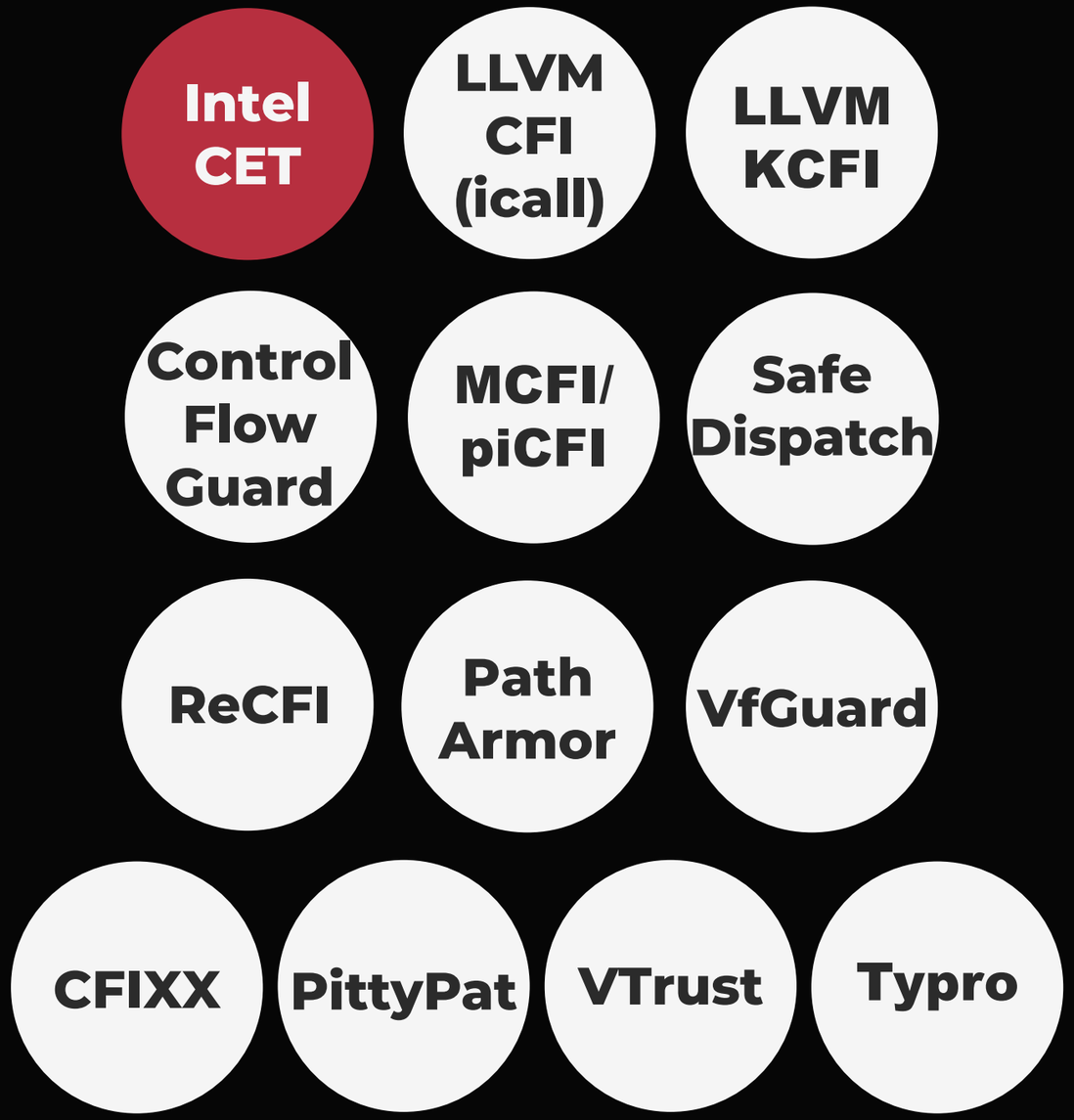
Typro

Coarse-grained



Finer-grained

Overview of CFI Defenses



Restrictions introduced

- No return address hijacking

Overview of CFI Defenses

Intel CET

LLVM CFI (icall)

LLVM KCFI

Control Flow Guard

MCFI/ piCFI

Safe Dispatch

ReCFI

Path Armor

VfGuard

CFIXX

PittyPat

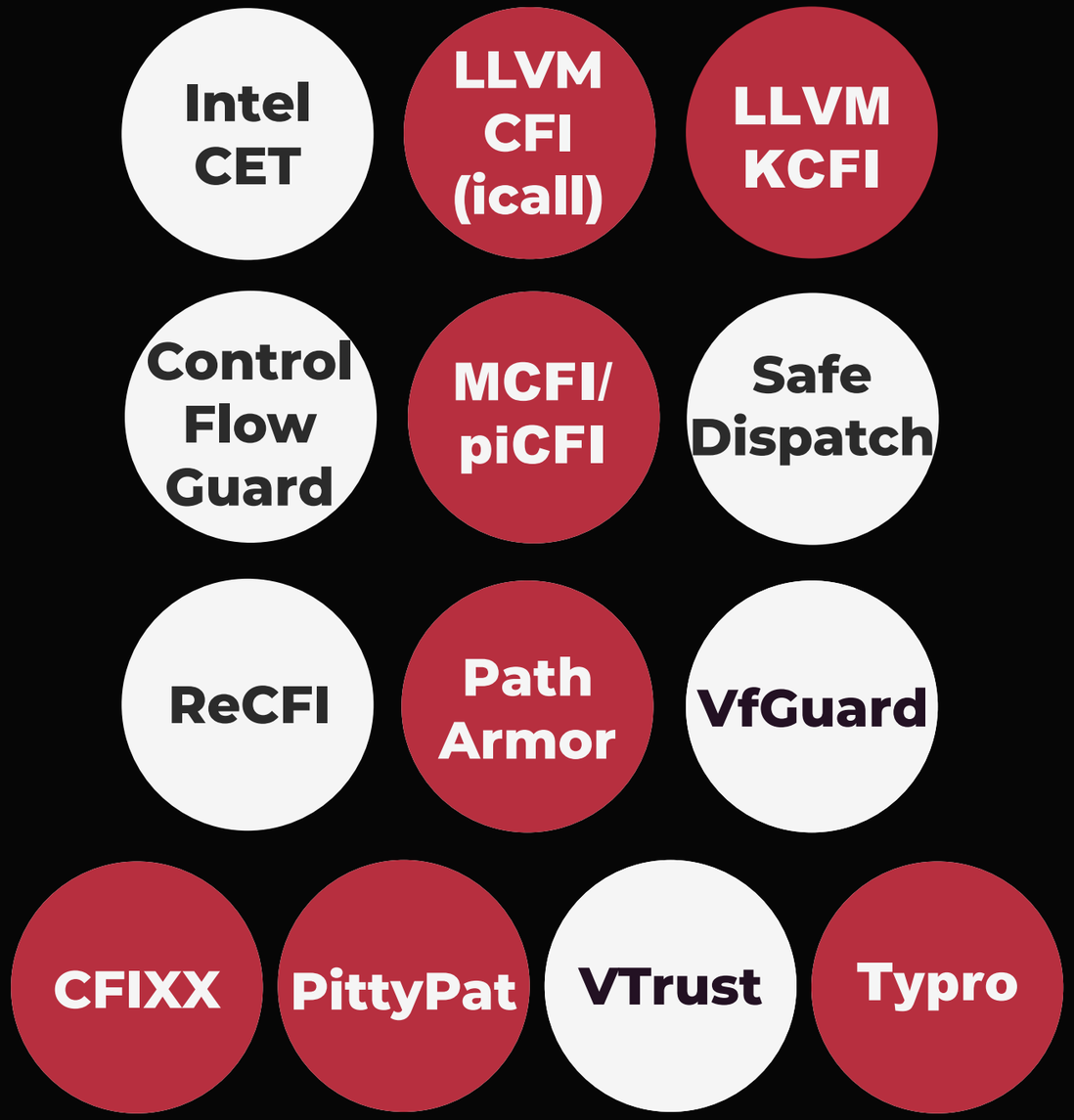
VTrust

Typro

Restrictions introduced

- No return address hijacking
- No vptr hijacking

Overview of CFI Defenses



Restrictions introduced

- No return address hijacking
- No vptr hijacking
- No arbitrary call targets

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

- 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

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

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

Overview of CFI Defenses

Intel CET

LLVM CFI (icall)

LLVM KCFI

Control Flow Guard

MCFI/ piCFI

Safe Dispatch

ReCFI

Path Armor

VfGuard

CFIXX

PittyPat

VTrust

Typro

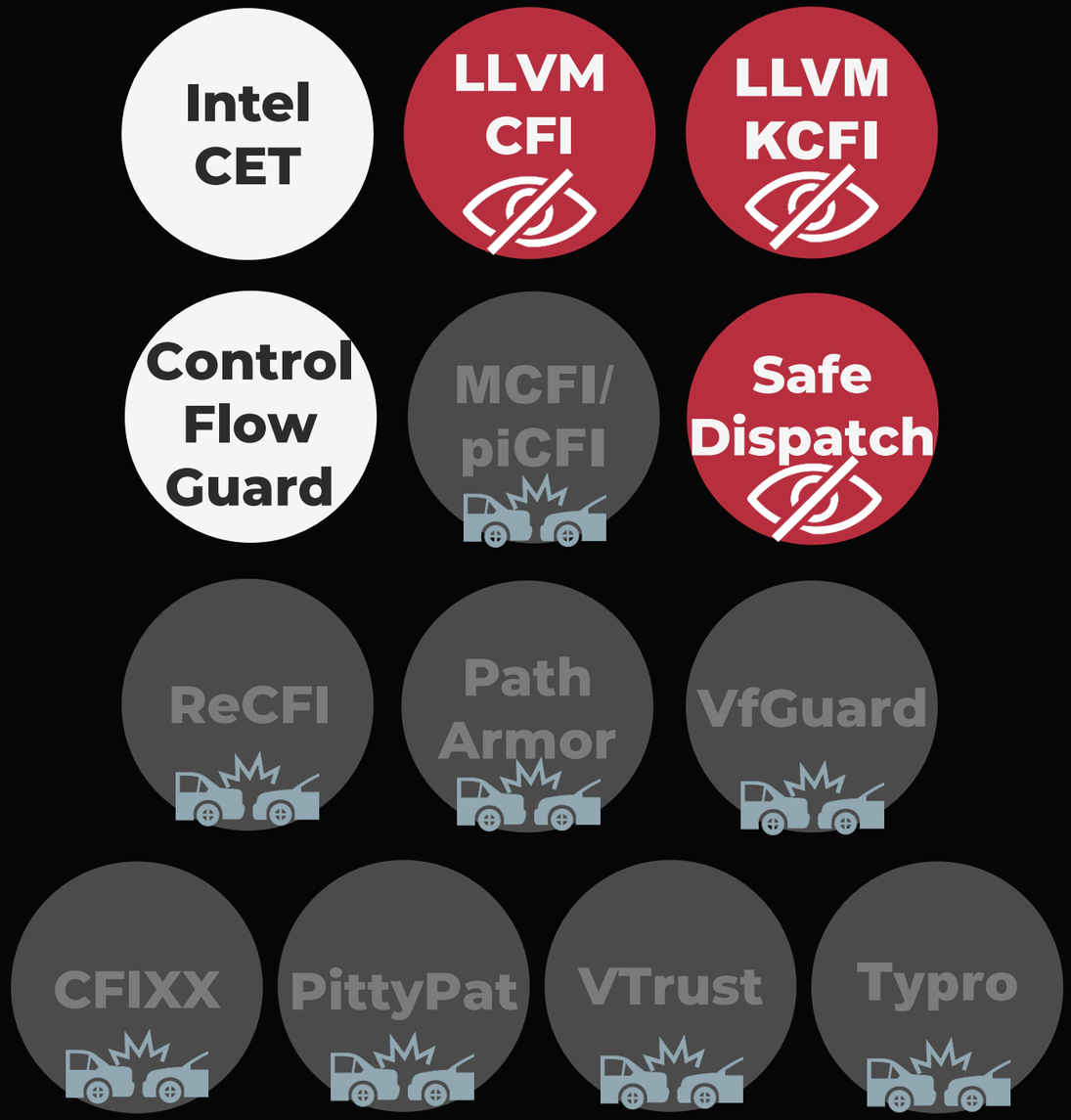
Restrictions introduced

- No return address hijacking
- No vptr hijacking
- No arbitrary call targets

Issues with CFI schemes

- CFI breaks with coroutines

Overview of CFI Defenses



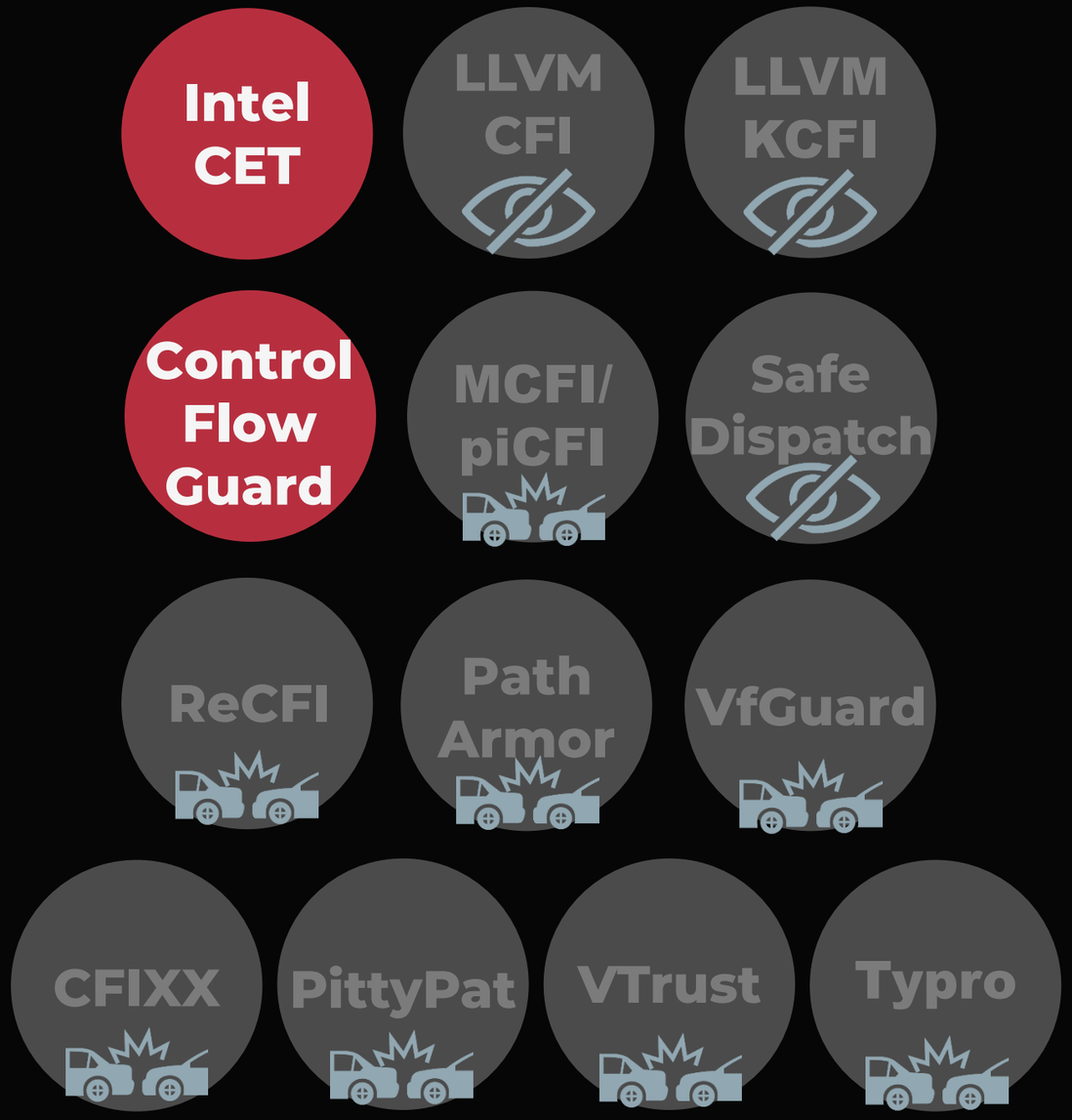
Restrictions introduced

- No return address hijacking
- No vptr hijacking
- No arbitrary call targets

Issues with CFI schemes

- CFI breaks with coroutines
- Does not instrument coroutine *resume* & *destroy* calls

Overview of CFI Defenses



Restrictions introduced

- No return address hijacking
- No vptr hijacking
- No arbitrary call targets

Issues with CFI schemes

- CFI breaks with coroutines
- Does not instrument coroutine *resume* & *destroy* calls
- Jumps to the beginning of functions still allowed

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

Control Flow Hijacking

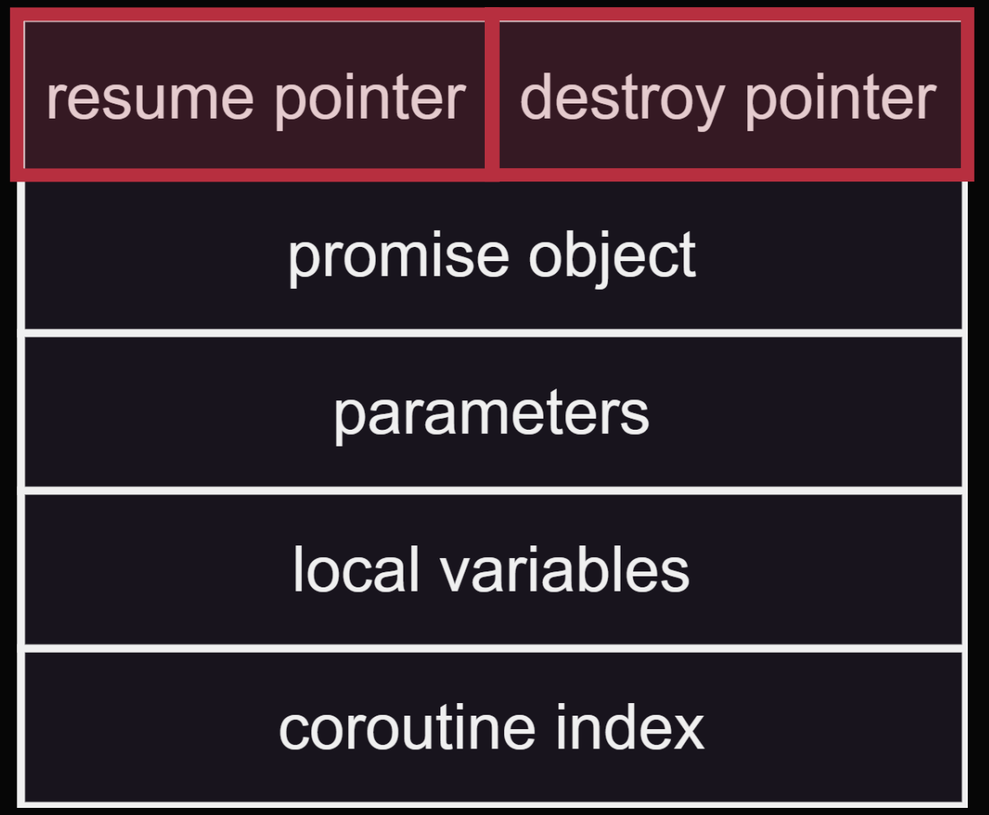
- You do not need to overwrite the pointers for control flow hijacking! Just a *CFP*.

- 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




```
handle.resume()
```
handle

```
call [rdi]
```
resume ptr

```
handle.destroy()
```
handle

```
call [rdi+0x8]
```
destroy ptr

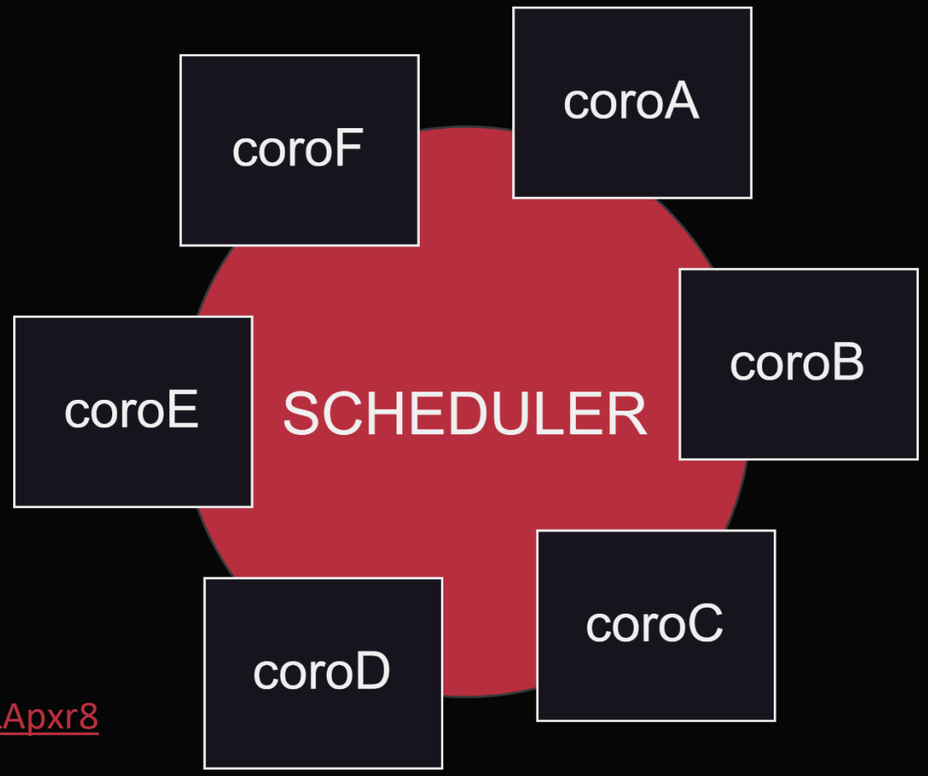
Control Flow Hijacking

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

ScyllaDB

       - Or browsers

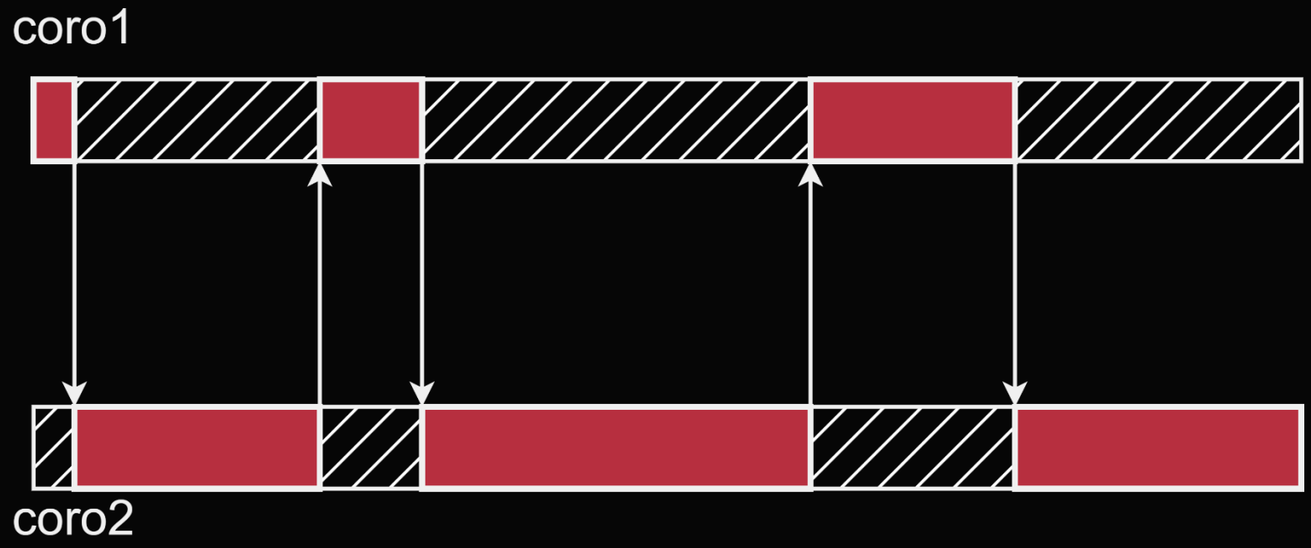
Ladybird Chromium



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

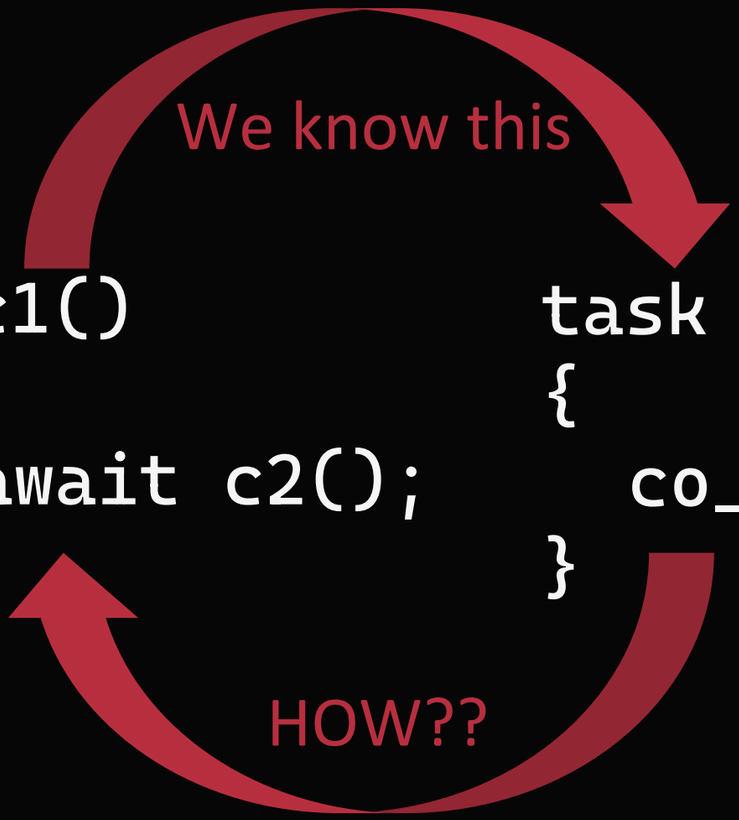
Control Flow Hijacking

- Vulnerable pointers:
  1. Overwriting the *resume* or *destroy* pointers
  2. Overwriting a coroutine handle. But where?
  3. Overwriting internal frame coroutine pointers



The awesome world of Continuations

- Coroutines need to know how to resume the next one

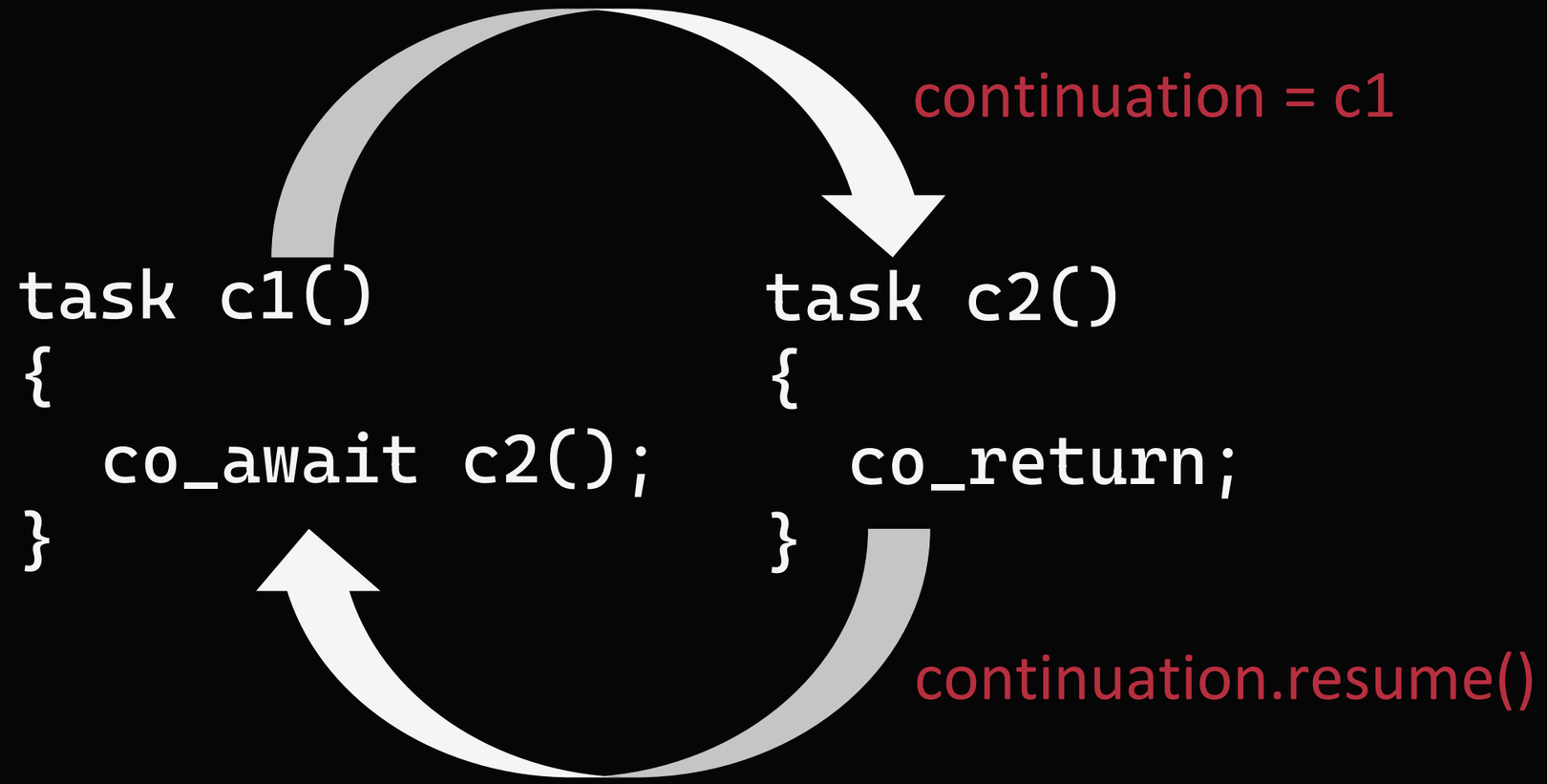



```
task c1()
{
  co_await c2();
}
```

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

The awesome world of Continuations

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



The awesome world of Continuations

- Wait, where is c2() destroyed?

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

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

The awesome world of Continuations

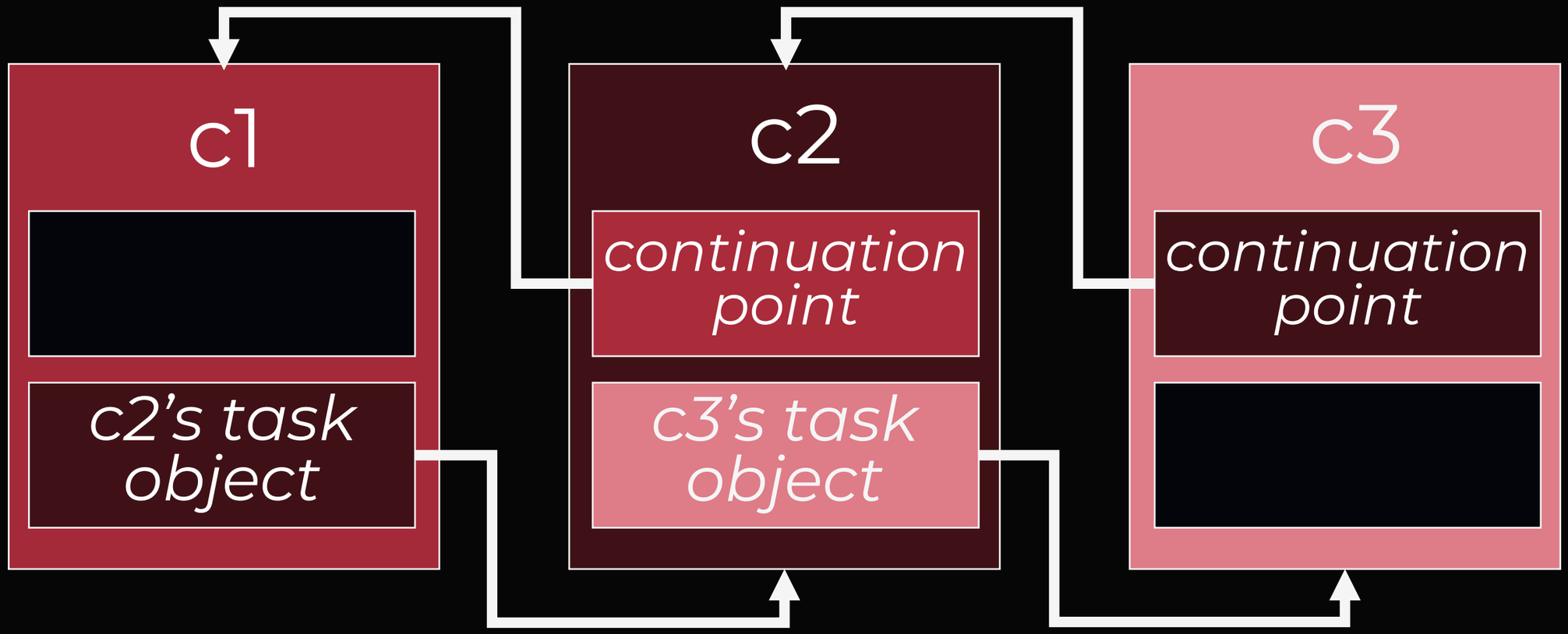
- Wait, where is c2() destroyed?
  - Implicitly, right after co_await, as c2 goes out of scope

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

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

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

The awesome world of Continuations



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

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

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

Infinite Coroutine Chaining

c1

resume pointer	destroy pointer
continuation	
parameters	
destroy_task	
coroutine index = 2	

c1'

resume pointer	destroy pointer
continuation	
parameters	
destroy_task	
coroutine index = 2	

c1''

resume pointer	destroy pointer
continuation	
parameters	
destroy_task	
coroutine index = 2	

trampoline frame 4

?	-

trampoline frame 3

?	-

trampoline frame 2

?	-

trampoline frame 1

-	?

Infinite Coroutine Chaining

c1

resume pointer	destroy pointer
continuation	
parameters	
destroy_task	
coroutine index = 2	

coroutine c1()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

DP

Destroy Stub

Destroys F

coroutine c1’()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

DP

Destroy Stub

Destroys F

coroutine c1’’()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

DP

Destroy Stub

Destroys F

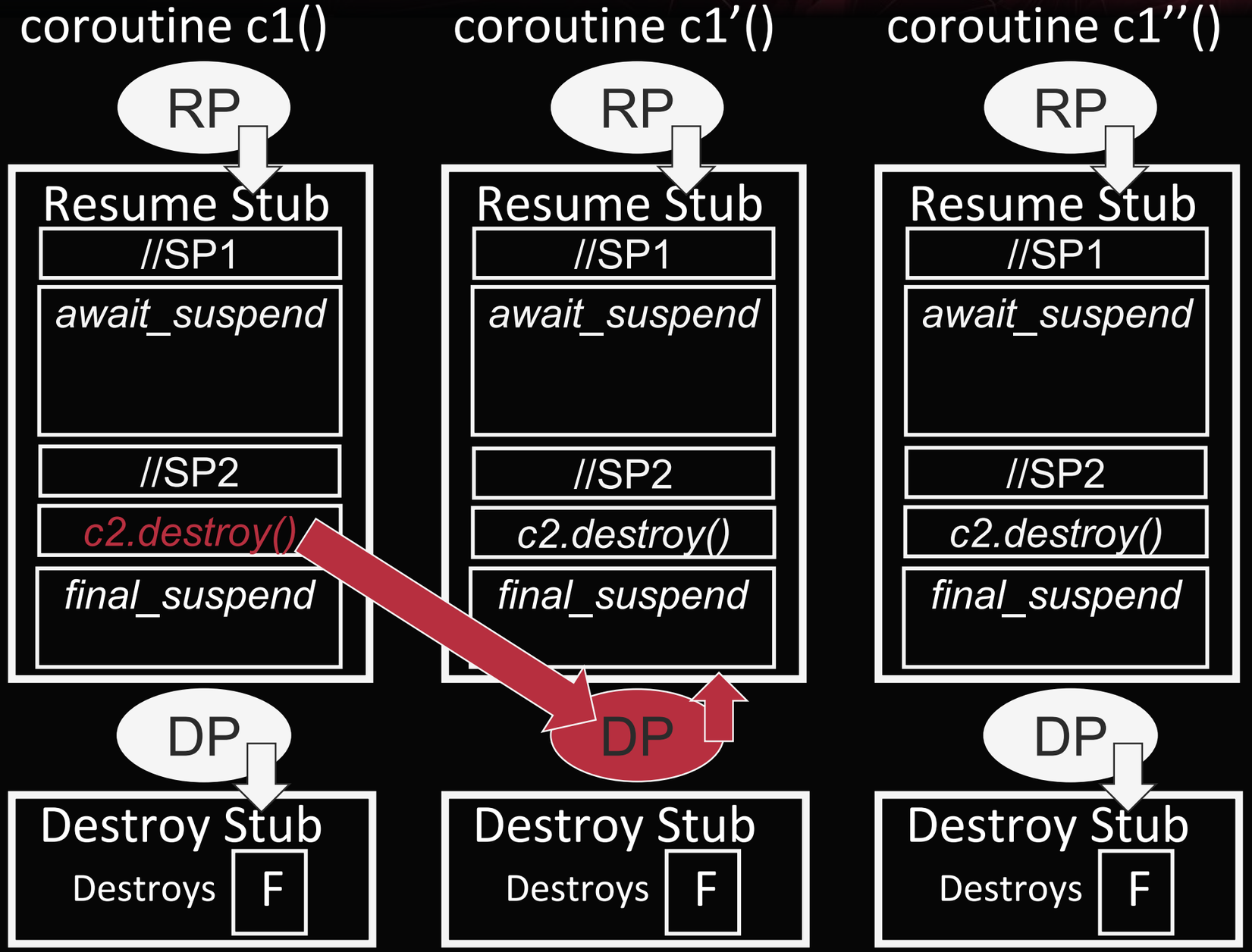
Infinite Coroutine Chaining

c1

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

c1'

resume pointer	destroy pointer = resume
continuation	
parameters	
destroy_task	
coroutine index = 2	



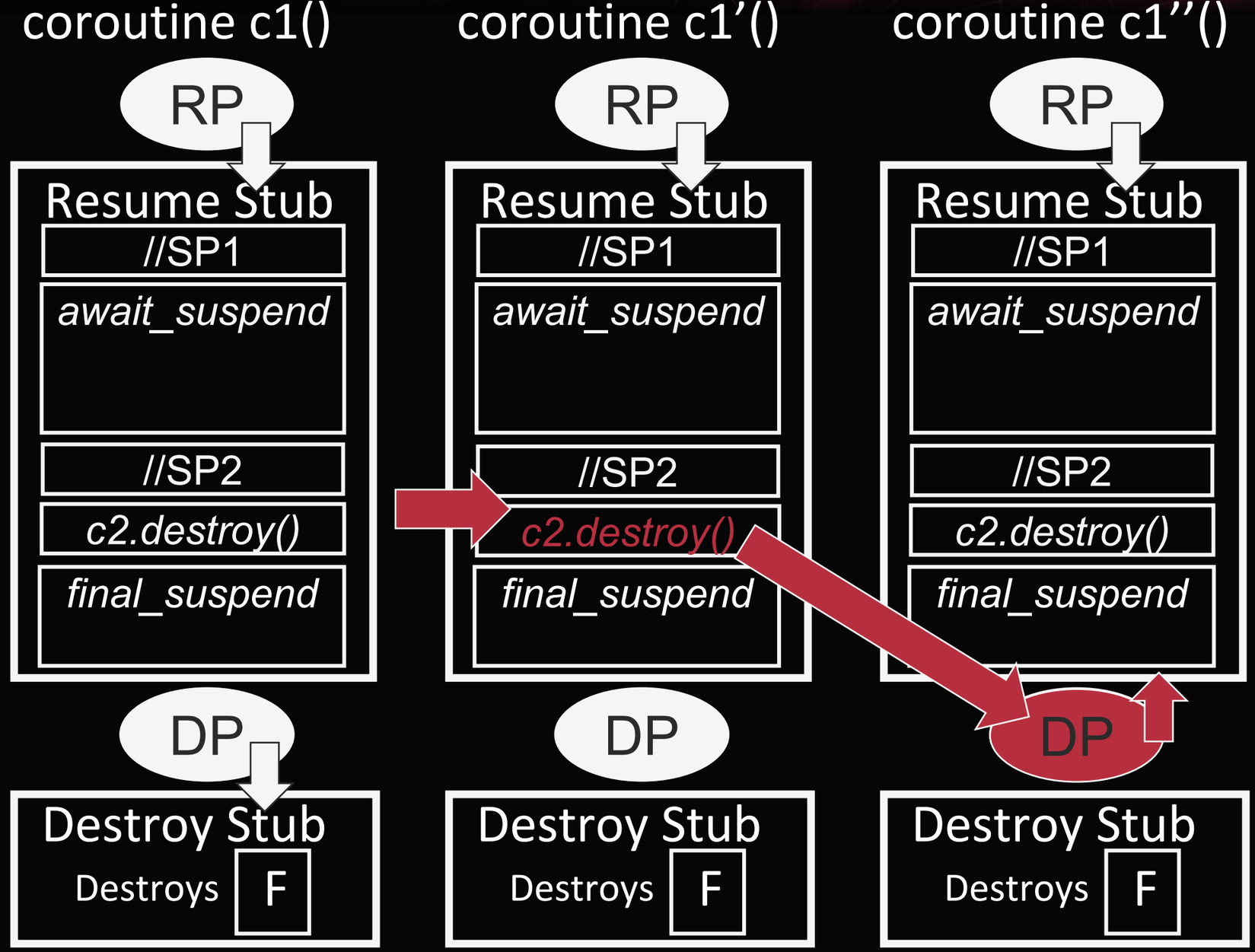
Infinite Coroutine Chaining

c1'

resume pointer	destroy pointer = resume
continuation	
parameters	
destroy_task = c1"	
coroutine index = 2	

c1’’

resume pointer	destroy pointer = resume
continuation	
parameters	
destroy_task = tramp1	
coroutine index = 2	



Infinite Coroutine Chaining

c1'

resume pointer	destroy pointer = resume
continuation	
parameters	
destroy_task = c1"	
coroutine index = 2	

c1’’

resume pointer	destroy pointer = resume
continuation	
parameters	
destroy_task = tramp1	
coroutine index = 2	

coroutine c1()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

coroutine c1’()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

coroutine c1’’()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

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

Infinite Coroutine Chaining

c1'

resume pointer	destroy pointer = resume
continuation = tramp3	
parameters	
destroy_task = c1"	
coroutine index = 2	

c1’’

resume pointer	destroy pointer = tramp1
continuation = tramp 2	
parameters	
destroy_task = c1"	
coroutine index = 2	

coroutine c1()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

coroutine c1’()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

coroutine c1’’()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

cont.resume()

trampoline frame 4

?	-

trampoline frame 3

?	-

trampoline frame 2

ARBITRARY CALL 2	-

trampoline frame 1

-	ARBITRARY CALL 1

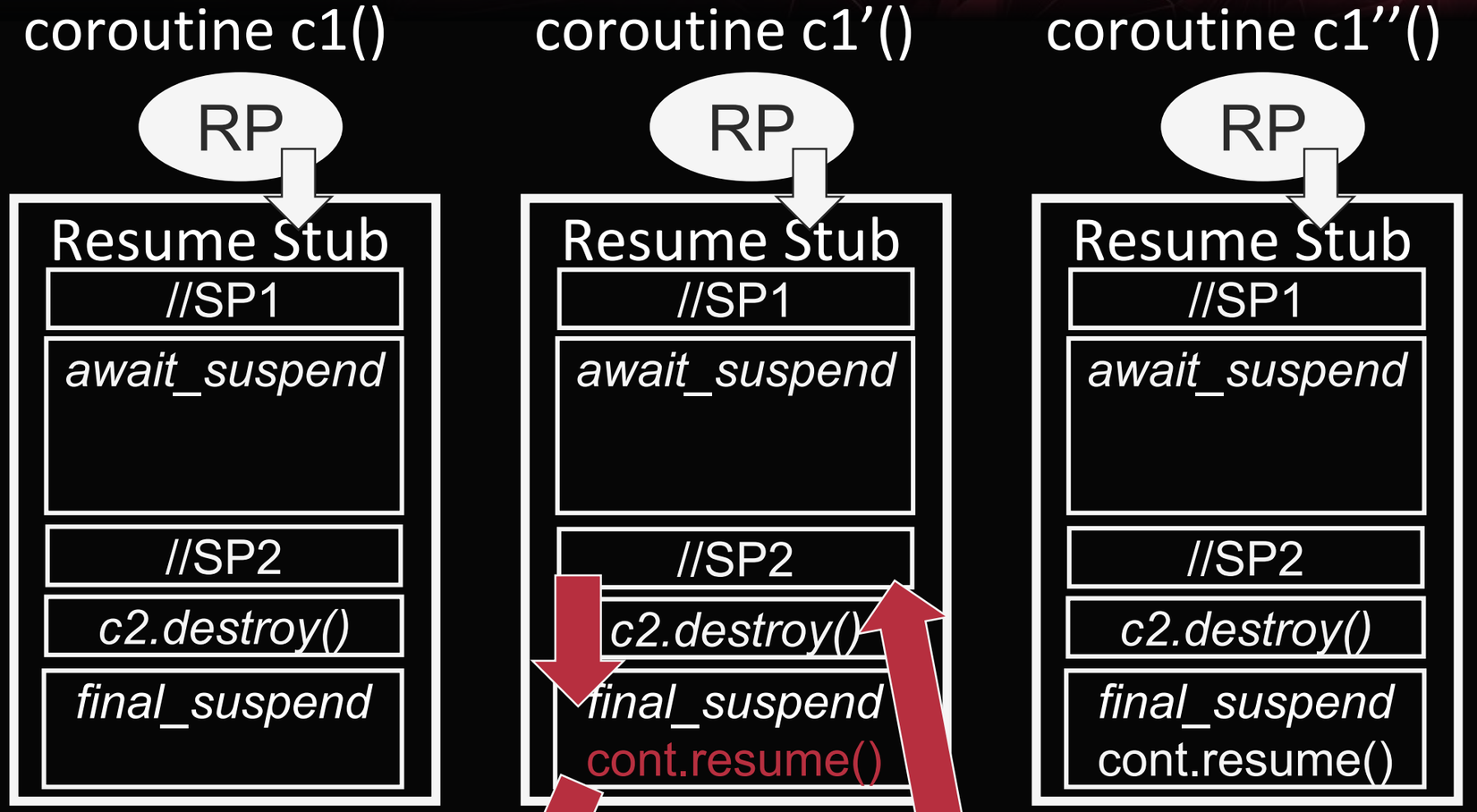
Infinite Coroutine Chaining

c1'

resume pointer	destroy pointer = resume
continuation = tramp3	
parameters	
destroy_task = c1"	
coroutine index = 2	

c1’’

resume pointer	destroy pointer = tramp1
continuation = tramp 2	
parameters	
destroy_task = c1"	
coroutine index = 2	



Infinite Coroutine Chaining

c1

resume pointer	destroy pointer
continuation = tramp4	
parameters	
destroy_task = c1'	
coroutine index = 2	

c1'

resume pointer	destroy pointer = resume
continuation = tramp3	
parameters	
destroy_task = c1''	
coroutine index = 2	

coroutine c1()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

cont.resume()

coroutine c1'()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

cont.resume()

coroutine c1''()

RP

Resume Stub

//SP1

await_suspend

//SP2

c2.destroy()

final_suspend

cont.resume()

trampoline frame 4

ARBITRARY CALL 4	-

trampoline frame 3

ARBITRARY CALL 3	-

trampoline frame 2

ARBITRARY CALL 2	-

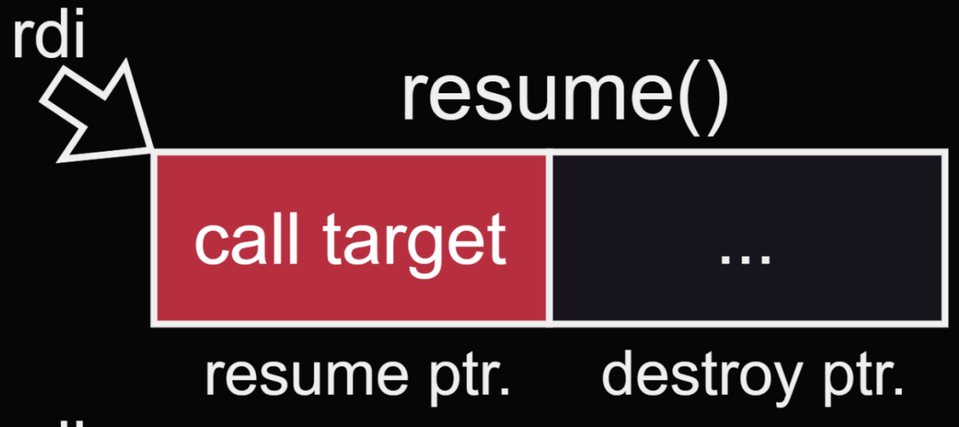
trampoline frame 1

-	ARBITRARY CALL 1

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

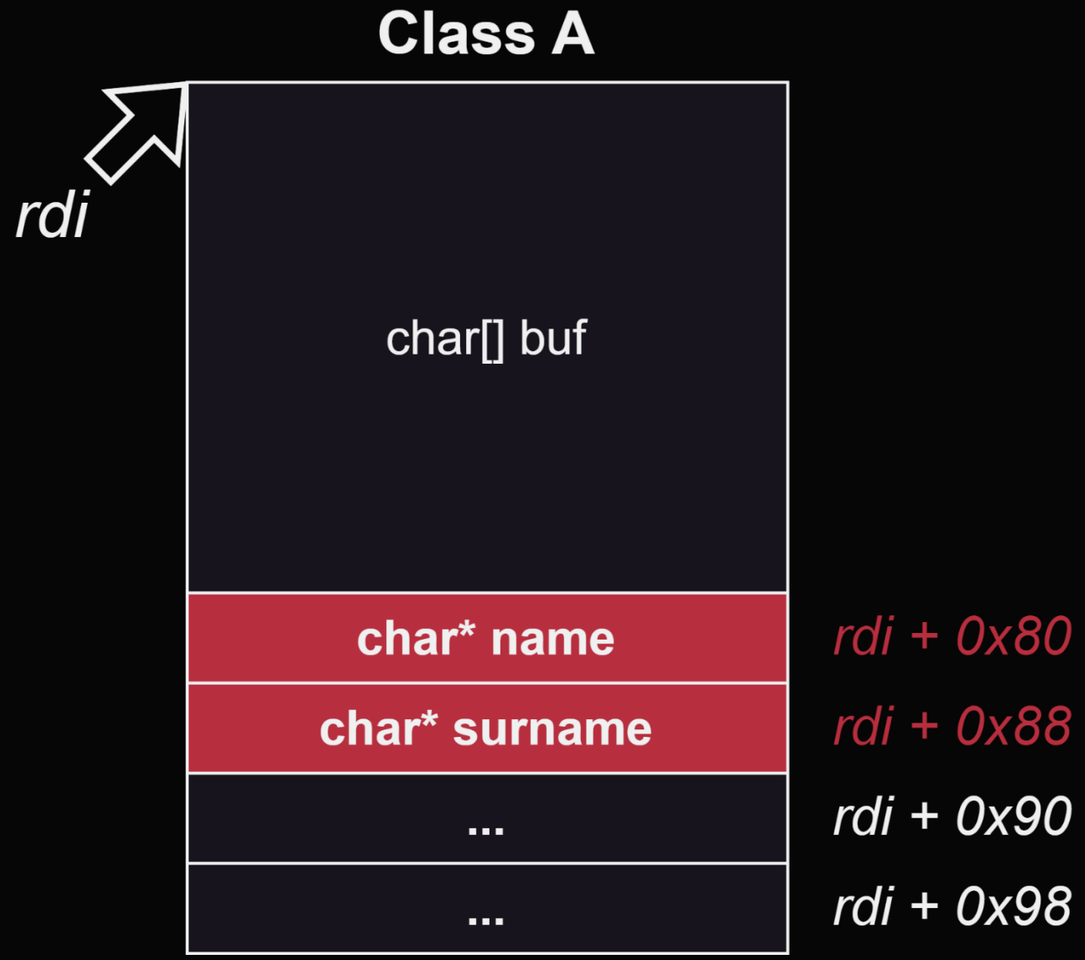
- So, *resume* and *destroy* have *rdi*=frame
- Is there anything else where *rdi* is always used?
  - By convention *rdi=this* in an object

```
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

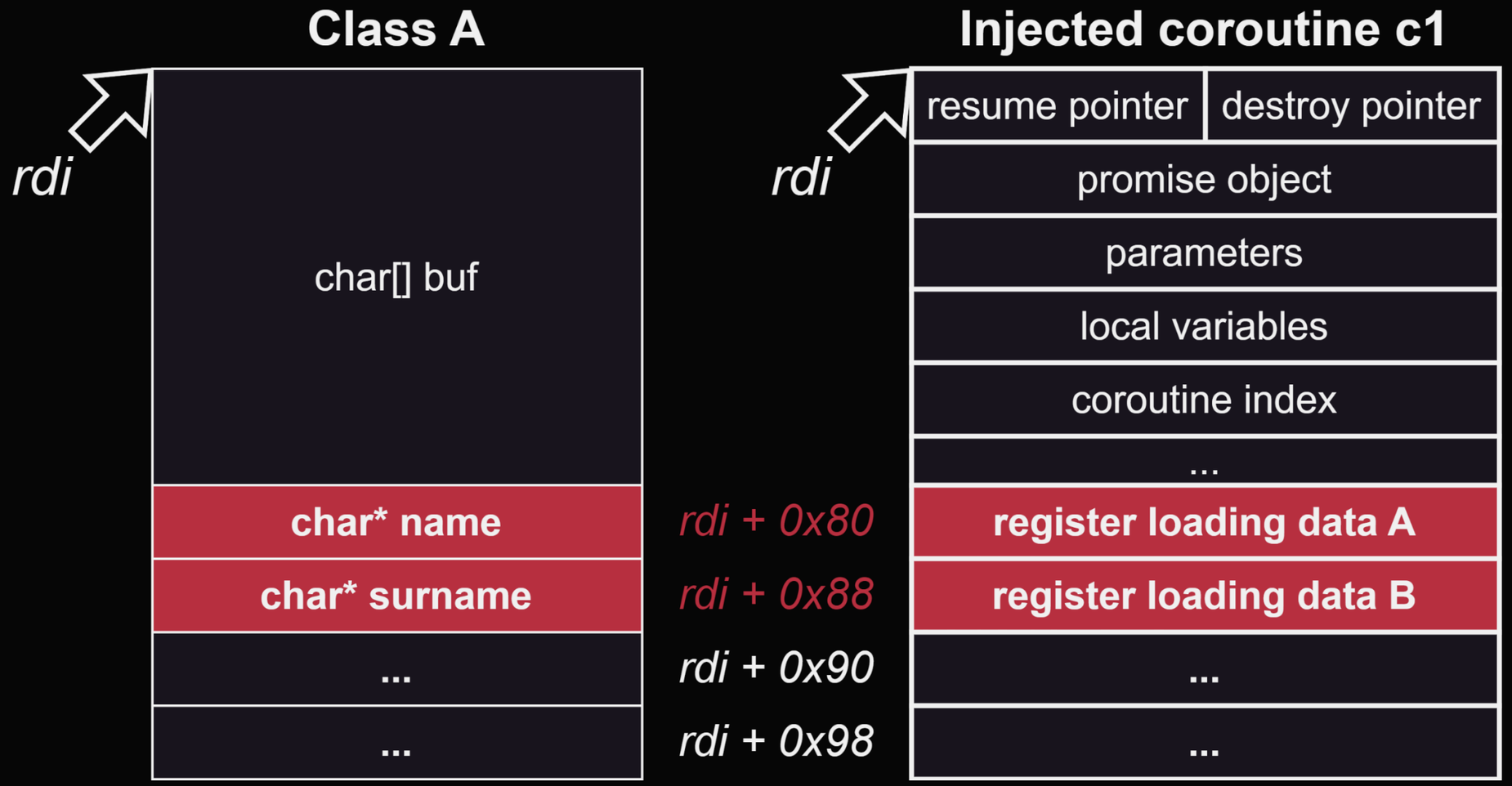
- Coroutine frame & class collision




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

Argument Passing

- What about setting arbitrary arguments in the registers?
- Coroutine frame & class collision



Argument Passing

- Golden gadget
  - Sets registers and controlled 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
```

- Silver gadget
  - Only sets registers and returns, needs to leverage another CFP for the 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
```

SerenityOS / serenity

Type / to search

Code | Issues 726 | Pull requests 16 | Actions | Security | Insights

serenity

Public

Watch 361 | Fork 3.3k | Star 32.1k

master | 1 Branch | 0 Tags | Go to file t | Add file | Code

LucasChollet and nico LibGfx/JPEG: Don't assume that app segments always st... 59d2426 · 1 hour ago 65,566 Commits

.devcontainer	Meta: Bump the pre-commit feature version to 2 in dev-cont...	2 months ago
.github	CI: Set CTest timeout to 30 minutes	2 weeks ago
AK	AK: Extract a `shift` variable in sign_extend()	3 weeks ago
Base	Base: Add "Ladyball" desktop background	last month
Documentation	Everywhere: Replace discord.gg/serenityos with serenityos.or...	2 months ago
Kernel	Kernel/MM: Remove "Virtual" from "ContiguousPhysicalVirtu...	2 days ago
Ladybird	Ladybird/AppKit: Use sRGB color space when blitting web co...	3 months ago

About

The Serenity Operating System

serenityos.org

c-plus-plus | unix | browser | kernel | os | operating-system | desktop-environment

Readme

BSD-2-Clause license

Security policy

Activity

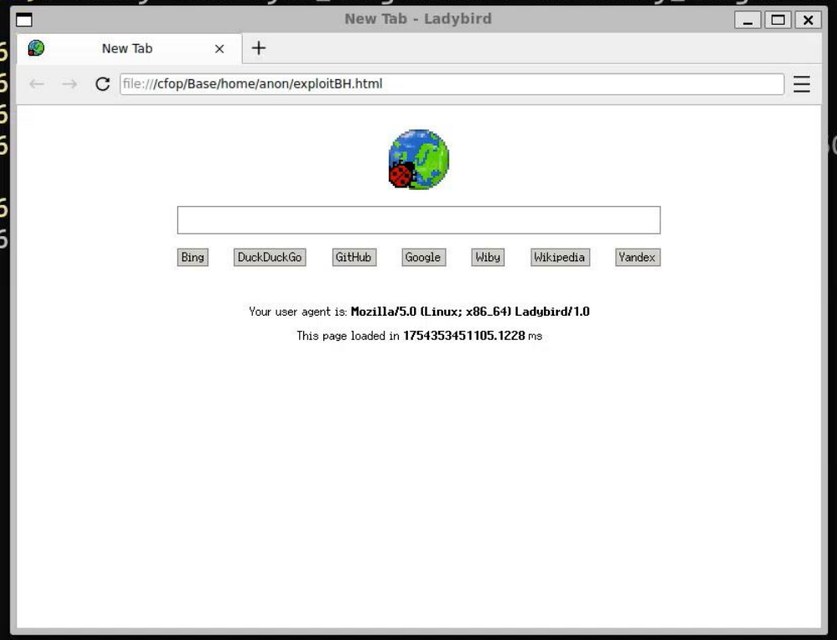
Custom properties

32.1k stars

```cpp
void EventLoop::adopt_coroutine(Coroutine<void>&& coroutine)
{
    class OrphanedCoroutine {
        struct PromiseType;

    public:
        using promise_type = PromiseType;

    private:
        struct Destroyer {
            bool await_ready() const noexcept { return false; }
            void await_suspend(std::coroutine_handle<> handle) const noexcept { handle.destroy(); }
            void await_resume() const noexcept { }
        };

        struct PromiseType {
            OrphanedCoroutine get_return_object() { return {}; }
            AK::Detail::SuspendNever initial_suspend() { return {}; }
            Destroyer final_suspend() noexcept { return {}; }
            void return_void() { }
        };
    };

    [](Coroutine<void>&& coroutine) mutable -> OrphanedCoroutine {
        auto saved_coroutine = move(coroutine);
        co_await saved_coroutine;
    }(move(coroutine));
}
```

```
void EventLoop::adopt_coroutine(Coroutine<void>&& coroutine)
{
    class OrphanedCoroutine {
        struct PromiseType;

    public:
        using promise_type = PromiseType;

    private:
        struct Destroyer {
            void await_suspend(std::coroutine_handle<> handle) const noexcept { handle.destroy(); }
        };

        struct PromiseType {
            OrphanedCoroutine get_return_object() { return {}; }
            AK::Detail::SuspendNever initial_suspend() { return {}; }
            Destroyer final_suspend() noexcept { return {}; }
            void return_void() { }
        };
    };

    co_await saved_coroutine;
    move(coroutine));
```

```
[+] Starting local process '/usr/bin/gdbserver': pid 830
[*] running in new terminal: ['/usr/bin/gdb', '-q', '../Build/lagom/bin/Ladybird
', '-x', '/tmp/pwnlib-gdbscript-sd2_8i6o.gdb']
[*] Switching to interactive mode
1613512.936 Ladybird(833): Debugger is attached, ignoring SIGINT
Detaching from process 849
Detaching from process 851
Detaching from process 833
1613533.049 WebContent(861): FontDatabase::load_all_fonts_from_uri('file:///home
/ubuntu/.local/share/fonts'): stat: No such file or directory (errno=2)
1613533.062 WebContent(861): array_buffer size:3524
1613533.062 WebContent(861): Array initially at data:0x0000555555997440
1613533.062 WebContent(861): Array with byte_length:3524 and array_length 3524 c
reated! Offset:0
1613533.062 WebContent(86
1613533.070 WebContent(86
1613533.070 WebContent(86                                                    05 c
1613533.070 WebContent(86
reated! Offset:0
1613533.070 WebContent(86
Detaching from process 86
$
```




```
0x00007fffffffdde8|+0x0030: 0x00005555557722c0  →  0x0000000000000000
0x00007fffffffddf0|+0x0038: 0x0000000000000001
```

code:x86:64

```
   0x7ffff4e8b48b <poll+000b>      je     0x7ffff4e8b4a0 <__GI___poll+32>
   0x7ffff4e8b48d <poll+000d>      mov    eax, 0x7
   0x7ffff4e8b492 <poll+0012>      syscall
 → 0x7ffff4e8b494 <poll+0014>      cmp    rax, 0xfffffffffffff000
   0x7ffff4e8b49a <poll+001a>      ja     0x7ffff4e8b4e8 <__GI___poll+104>
   0x7ffff4e8b49c <poll+001c>      ret
   0x7ffff4e8b49d <poll+001d>      nop    DWORD PTR [rax]
   0x7ffff4e8b4a0 <poll+0020>      push   rbp
   0x7ffff4e8b4a1 <poll+0021>      mov    rbp, rsp
```

threads

```
[#0] Id 1, Name: "WebContent", stopped 0x7ffff4e8b494 in __GI___poll (), reas
on: SIGINT
```

trace

```
[#0] 0x7ffff4e8b494 → __GI___poll(fds=0x555555d0d4e0, nfds=0x4, timeout=0x1f4
)
[#1] 0x7ffff396c68e → mov rdi, rbx
[#2] 0x7ffff390ca63 → g_main_context_iteration()
[#3] 0x7ffff796b5ef → QEventDispatcherGlib::processEvents(QFlags<QEventLoop::
ProcessEventsFlag>)()
[#4] 0x7ffff777c9a3 → QEventLoop::exec(QFlags<QEventLoop::ProcessEventsFlag>)
()
[#5] 0x555555672b4e → Ladybird::EventLoopImplementationQt::exec(this=<optimiz
ed out>)
[#6] 0x7ffff5909bcb → Core::EventLoop::exec(this=0x7fffffffdf90)
[#7] 0x555555680f7a → serenity_main(arguments={
  argc = 0x8,
  argv = 0x7fffffffe2f8,
  strings = {
    <AK::Detail::Span<AK::StringView>> = {
      m_values = 0x55555572b460,
      m_size = 0x8
    }, <No data fields>}
})
[#8] 0x5555556864b7 → main(argc=0x8, argv=0x7fffffffe2f8)
```

```
(remote) gef➤  gef config context.nb_lines_backtrace 1
(remote) gef➤  c
Continuing.
```

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

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

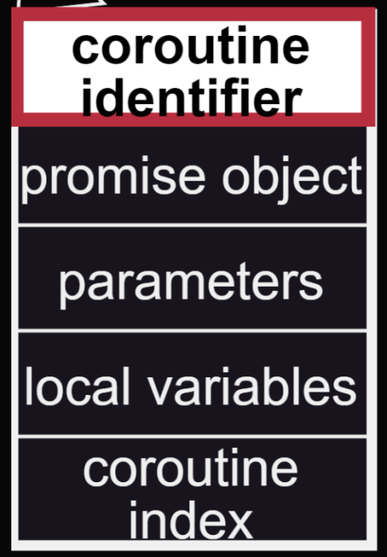
- Bypassing CET SHSTK and CFG is parallel to SHSTK & IBT

- 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




```
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 work
- In practice, getting HALO on your coroutines is *hard*
- Mostly works well if program is very simple

Heap Allocation Elision Optimization

Does my compiler support HALO at all?

- GCC
  - No.
- Clang
  - Yes (with the mentioned restrictions), but Clang 19 and 20 now do not – this is a bug, it 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
  - We discussed with Microsoft this and got an acknowledgment 😵‍💫

WHAT'S NEXT?

To Sum It Up

- CFI defenses protect code pointers in the program
  - Both *forward* and *backward*

To Sum It Up

- CFI defenses protect code pointers in the program
  - Both *forward* and *backward*

- Intel CET (SHSTK + IBT), CFG, LLVM CFI are some of the most prevalent
  - Prevent code-reuse (ROP, JOP)

To Sum It Up

- CFI defenses protect code pointers in the program
  - Both *forward* and *backward*
- Intel CET (SHSTK + IBT), CFG, LLVM CFI are some of the most prevalent
  - Prevent code-reuse (ROP, JOP)
- To bypass CFI, a dispatcher, dispatcher table and gadgets are needed

To Sum It Up

- CFI defenses protect code pointers in the program
  - Both *forward* and *backward*
- Intel CET (SHSTK + IBT), CFG, LLVM CFI are some of the most prevalent
  - Prevent code-reuse (ROP, JOP)
- To bypass CFI, a dispatcher, dispatcher table and gadgets are needed
- Coroutines are just one example that can bypass CFI
  - DOAs, control flow hijacking, ICC, frame collisions

Research to Come

SFOP

Under review ~2026

- **SFOP**: ********-Oriented Programming
- Bypasses Intel CET (SHSTK+IBT) in every Linux system, no special features needed
- Features kernel vulnerabilities (currently under embargo)

Await() a Second: Evading Control Flow Integrity by Hijacking C++ Coroutines

Marcos Bajo
CISPA Helmholtz Center for Information Security

Christian Rossow
CISPA Helmholtz Center for Information Security

https://syssec.cispa.io/coroutine-cfop/

CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

Marcos Bajo *h3xduck*

https://h3xduck.github.io