

STKTOKENS: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities

Lau Skorstengaard¹ Dominique Devriese² Lars Birkedal¹

¹Aarhus University

²Vrije Universiteit Brussel

POPL, January 16, 2019

Abstractions all the way down



Abstractions all the way down

```
main:  
    .cfi_startproc  
# BB#0:  
    pushq %rbp  
.Ltmp0:  
    .cfi_offset %rbp, -16  
.Ltmp1:  
    .cfi_offset %rbp, -16  
    movq %rsp, %rbp  
.Ltmp2:  
    .cfi_offset %rbp, -16  
    subq $16, %rsp  
    movabsq $.L.str, %rdi  
    movl $0, -4(%rbp)  
    movb $0, %al  
    callq printf  
    xorl %ecx, %ecx  
    movl %eax, -8(%rbp)  
    movl %ecx, %eax  
    addq $16, %rsp  
    popq %rbp  
    retq  
.Lfunc_end0:  
    .size main, .Lfunc_end0-main  
    .cfi_endproc
```



Abstractions all the way down

```
#include <stdio.h>
int main()
{
    int t = 5;
    printf("Hello, World!");

    return 0;
}

main:
.cfi_startproc
# BB#0:
    pushq %rbp
.Ltmp0:
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
.Ltmp1:
    .cfi_offset %rbp, -16
    movq %rbp, %rsp
    movabsq $.L.str, %rdi
    movl $0, -4(%rbp)
    movb $0, %al
    callq printf
    xorl %ecx, %ecx
    movl %eax, -8(%rbp)
    movl %ecx, %eax
    addq $16, %rsp
    popq %rbp
    retq
.Lfunc_end0:
.size main, .Lfunc_end0-main
.cfi_endproc
```



Abstractions all the way down

```
#include <stdio.h>
int main()
{
    int t = 5;
    printf("Hello, World!");
    return 0;
}
```

compilation

```
main:
.cfi_startproc
# BB#0:
pushq %rbp
.Ltmp0:
.cfi_offset %rbp, -16
.Ltmp1:
.cfi_offset %rbp, -16
movq %rsp, %rbp
.Ltmp2:
.cfi_offset %rbp, -16
subq $16, %rsp
movabsq $.L.str, %rdi
movl $0, -4(%rbp)
movb $0, %al
callq printf
xorl %ecx, %ecx
movl %eax, -8(%rbp)
movl %ecx, %eax
addq $16, %rsp
popq %rbp
retq
.Lfunc_end0:
.size main, .Lfunc_end0-main
.cfi_endproc
```



Abstractions all the way down

secure
compilation

```
#include <stdio.h>
int main()
{
    int t = 5;
    printf("Hello, World!");
    return 0;
}
```

```
main:
    .cfi_startproc
# BB#0:
    pushq %rbp
.Ltmp0:
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
.Ltmp1:
    .cfi_offset %rbp, -16
    movq %rbp, %rsp
    movabsq $.L.str, %rdi
    movl $0, -4(%rbp)
    movb $0, %al
    callq printf
    xorl %ecx, %ecx
    movl %eax, -8(%rbp)
    movl %ecx, %eax
    addq $16, %rsp
    popq %rbp
    retq
.Lfunc_end0:
    .size main, .Lfunc_end0-main
    .cfi_endproc
```

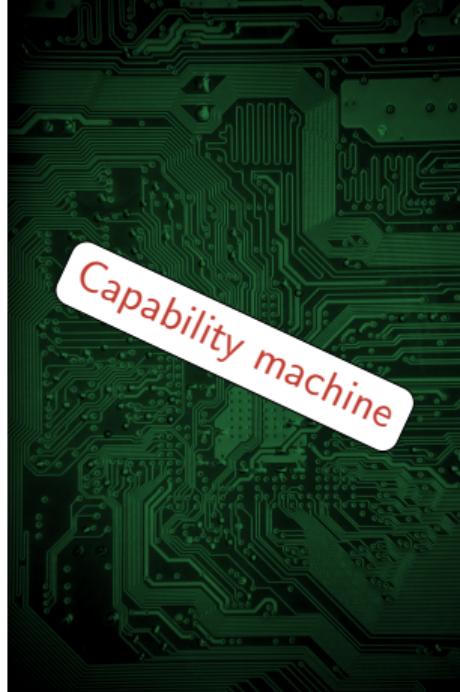


Abstractions all the way down

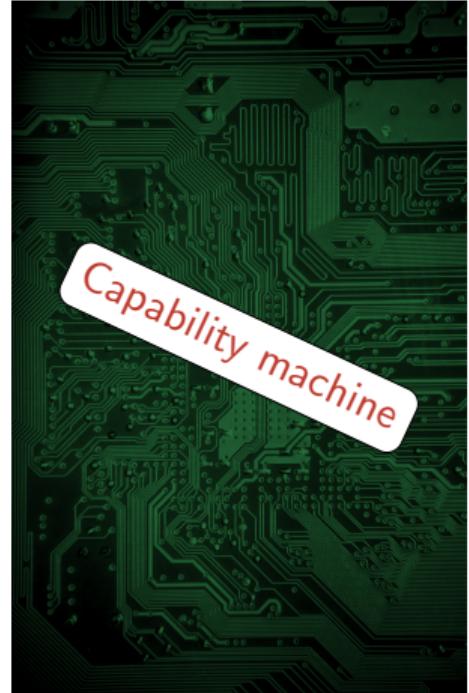
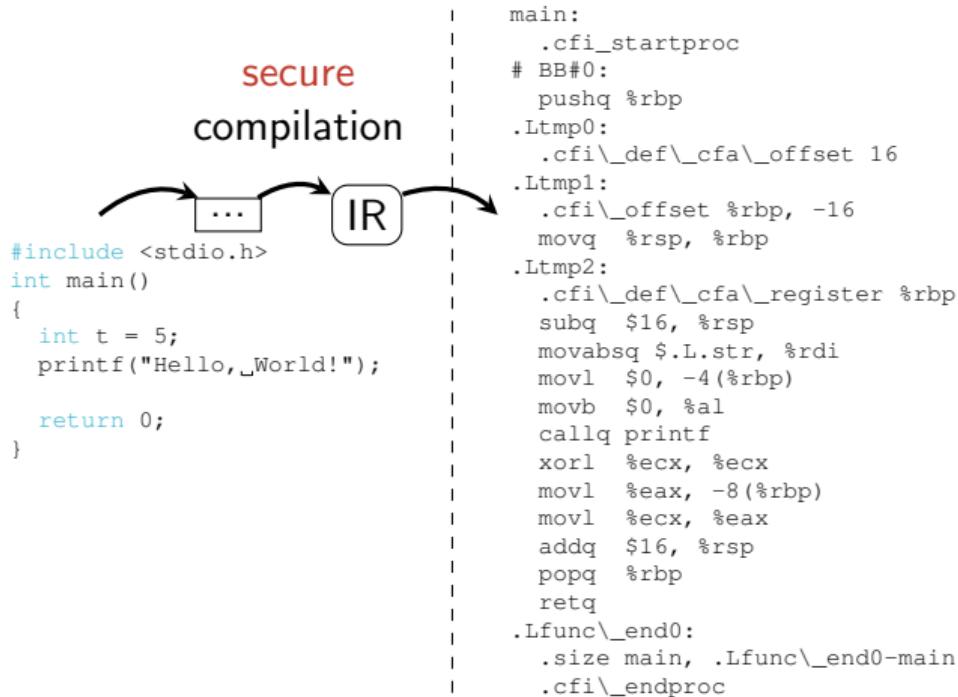
secure
compilation

```
#include <stdio.h>
int main()
{
    int t = 5;
    printf("Hello, World!");
    return 0;
}
```

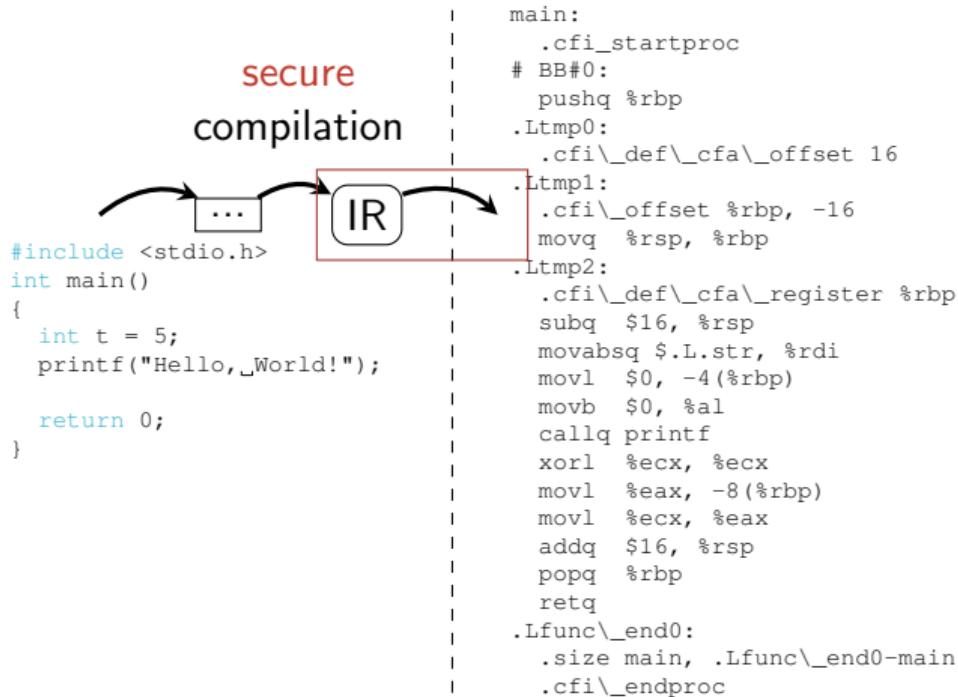
```
main:
    .cfi_startproc
# BB#0:
    pushq %rbp
.Ltmp0:
    .cfi_offset %rbp, -16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
.Ltmp2:
    .cfi_offset %rbp, -16
    subq $16, %rsp
    movabsq $.L.str, %rdi
    movl $0, -4(%rbp)
    movb $0, %al
    callq printf
    xorl %ecx, %ecx
    movl %eax, -8(%rbp)
    movl %ecx, %eax
    addq $16, %rsp
    popq %rbp
    retq
.Lfunc_end0:
    .size main, .Lfunc_end0-main
    .cfi_endproc
```



Abstractions all the way down



Abstractions all the way down



Well-bracketed control flow and local state encapsulation

```
void a()
{
    ...
    return;
}

void b()
{
    int x = 5;
    a();
    ...
    a();
    return;
}
```

Well-bracketed control flow and local state encapsulation

```
void a()  
{  
    ...  
    return;  
}
```

} Function a cannot
access variable x

```
void b()  
{  
    int x = 5;  
    a();  
    ...  
    a();  
    return;  
}
```

Local-state encapsulation (LSE)

Well-bracketed control flow and local state encapsulation

```
void a()
{
    ...
    return;
}

void b()
{
    int x = 5;
    a();
    ...
    → a();
    return;
}
```

Well-bracketed control flow (WBCF)

Well-bracketed control flow and local state encapsulation

```
→ void a()
{
    ...
    return;
}
```

```
void b()
{
    int x = 5;
    a();
    ...
    a();
    return;
}
```

Well-bracketed control flow (WBCF)

Well-bracketed control flow and local state encapsulation

```
void a()  
{  
→ ...  
    return;  
}
```

```
void b()  
{  
    int x = 5;  
    a();  
    ...  
    a();  
    return;  
}
```

Well-bracketed control flow (WBCF)

Well-bracketed control flow and local state encapsulation

```
void a()
{
    ...
→ return;
}
```

```
void b()
{
    int x = 5;
    a();
    ...
    a();
    return;
}
```

Well-bracketed control flow (WBCF)

Well-bracketed control flow and local state encapsulation

```
void a()  
{  
    ...  
    return;  
}  
  
void b()  
{  
    int x = 5;  
    a();  
    ...  
    a();  
    return;  
}
```



Well-bracketed control flow (WBCF)

Well-bracketed control flow and local state encapsulation

```
void a()  
{  
    ...  
    return;  
}  
  
void b()  
{  
    int x = 5;  
    a();  
    ...  
    a();  
    → return;  
}
```

Well-bracketed control flow (WBCF)

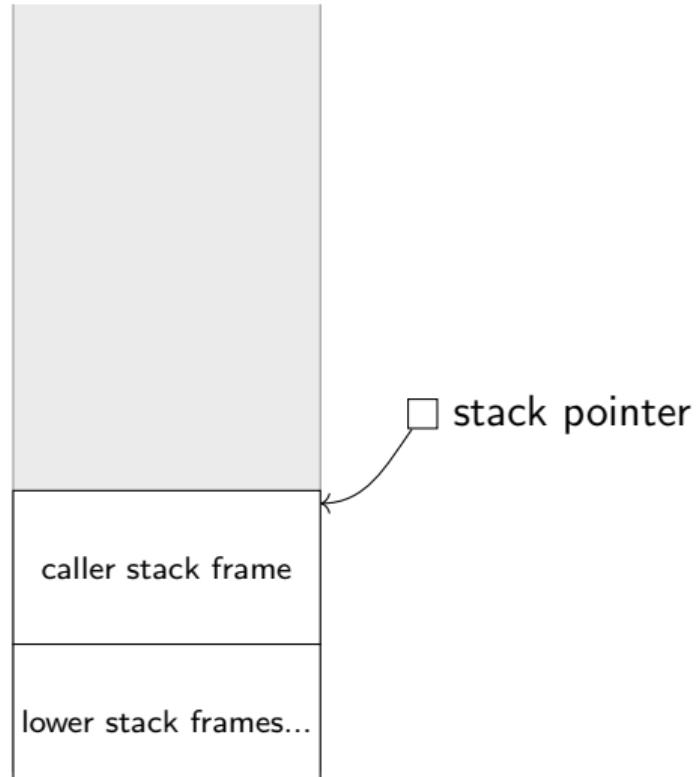
Well-bracketed control flow and local state encapsulation

```
void a()  
{  
    ...  
    return;  
}  
  
void b()  
{  
    int x = 5;  
    a();  
    → ...  
    a();  
    return;  
}
```

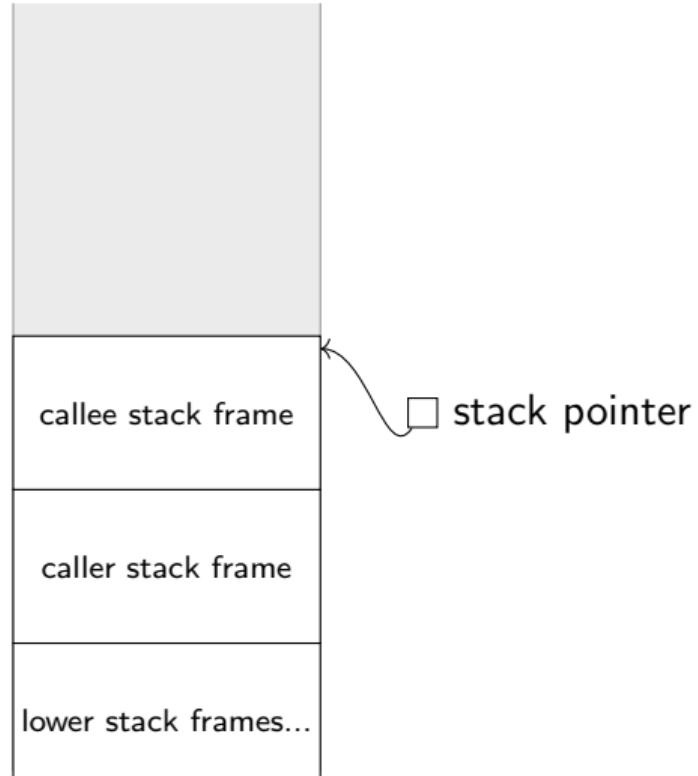


Well-bracketed control flow (WBCF)

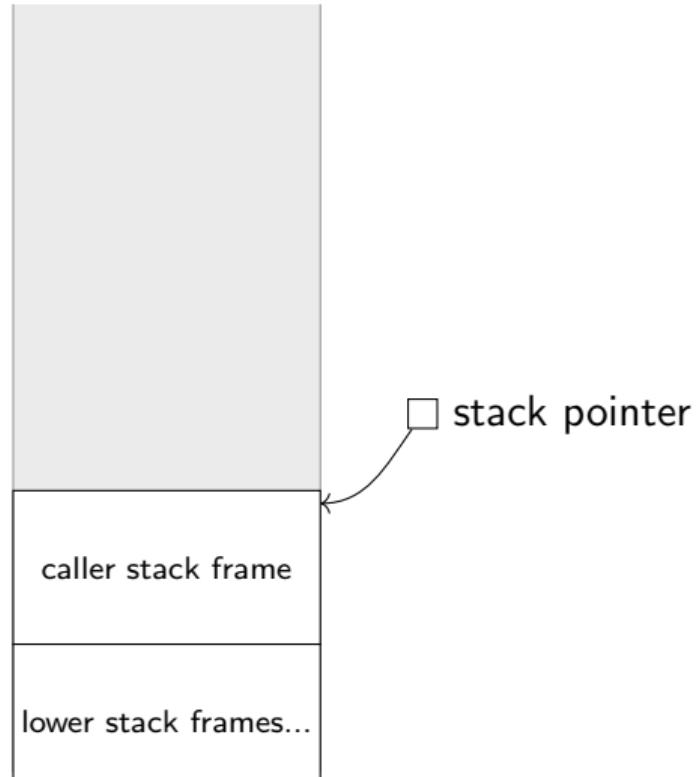
Traditional stack pointers



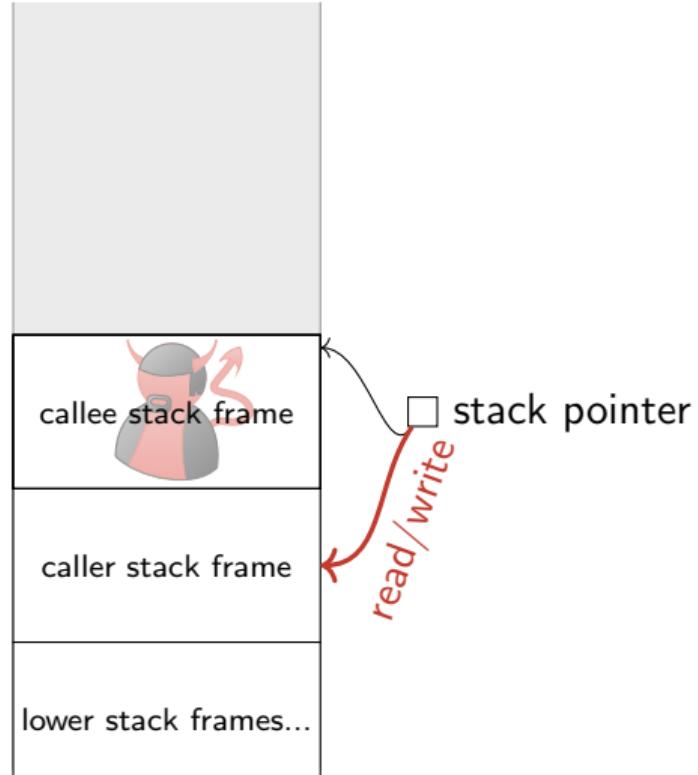
Traditional stack pointers



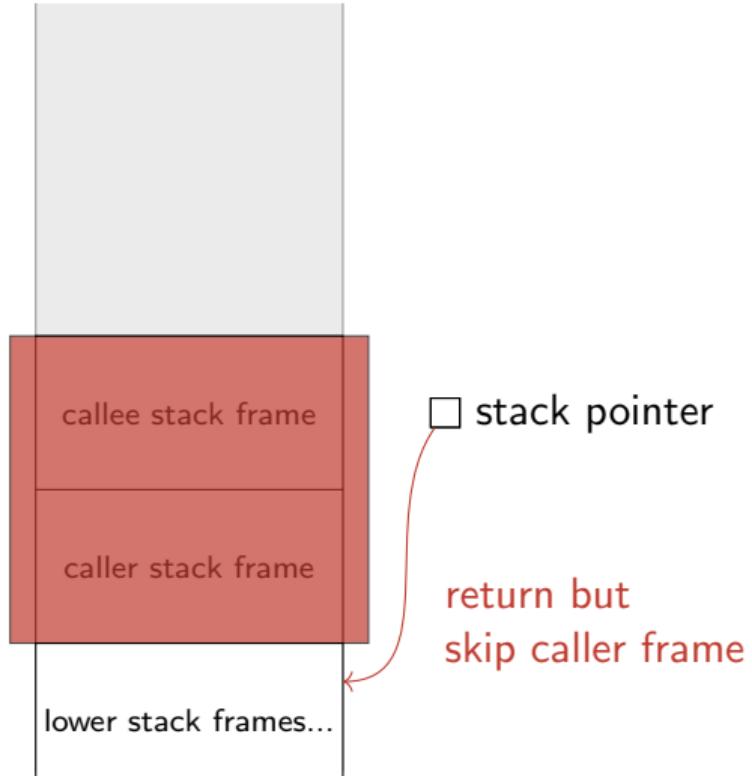
Traditional stack pointers



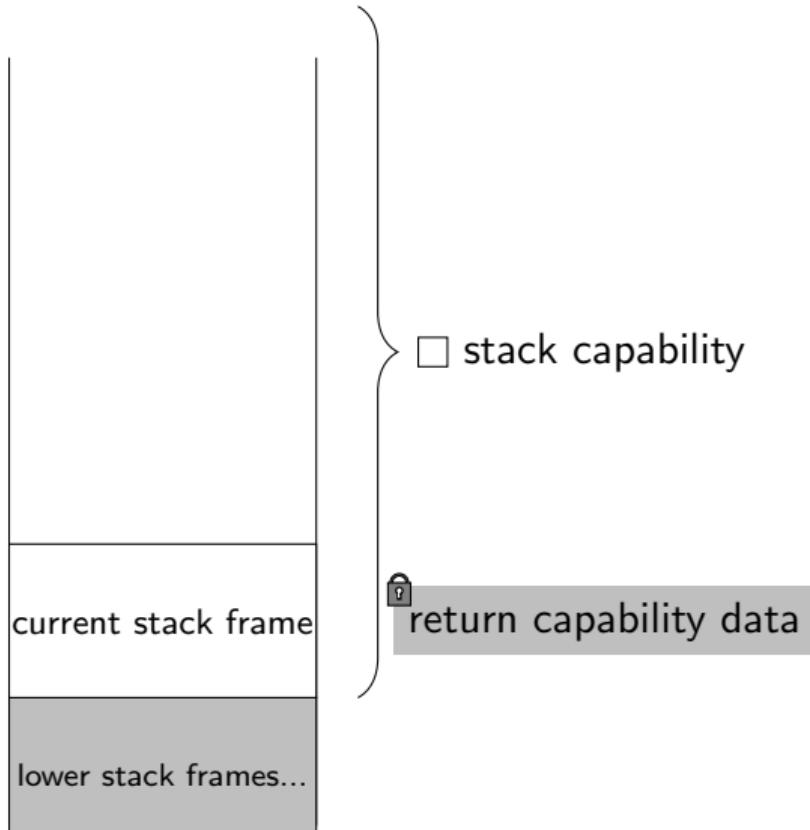
Traditional stack pointers



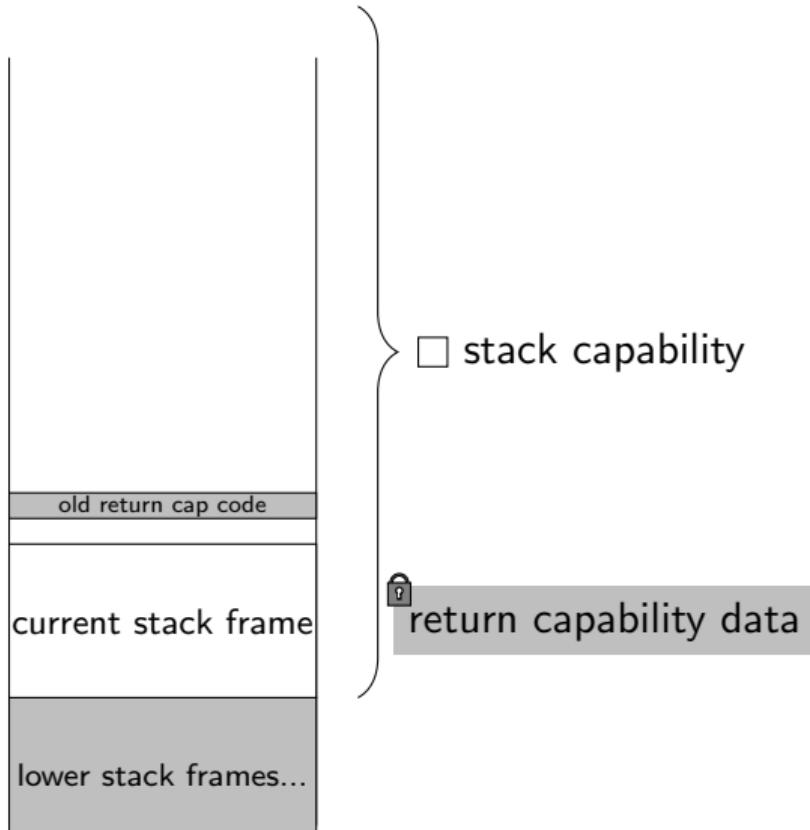
Traditional stack pointers



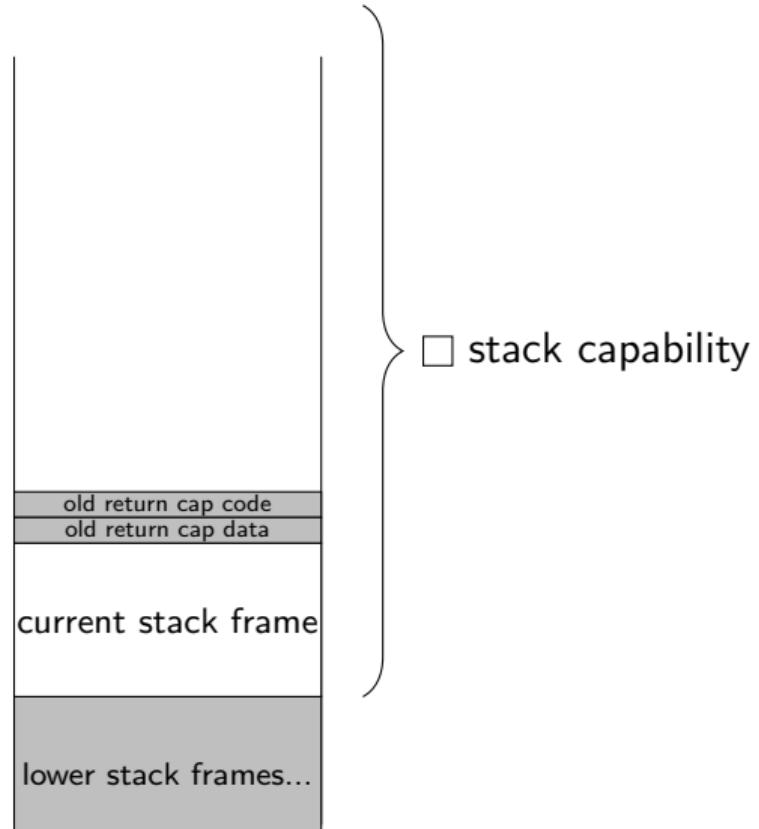
Naive stack and return capabilities



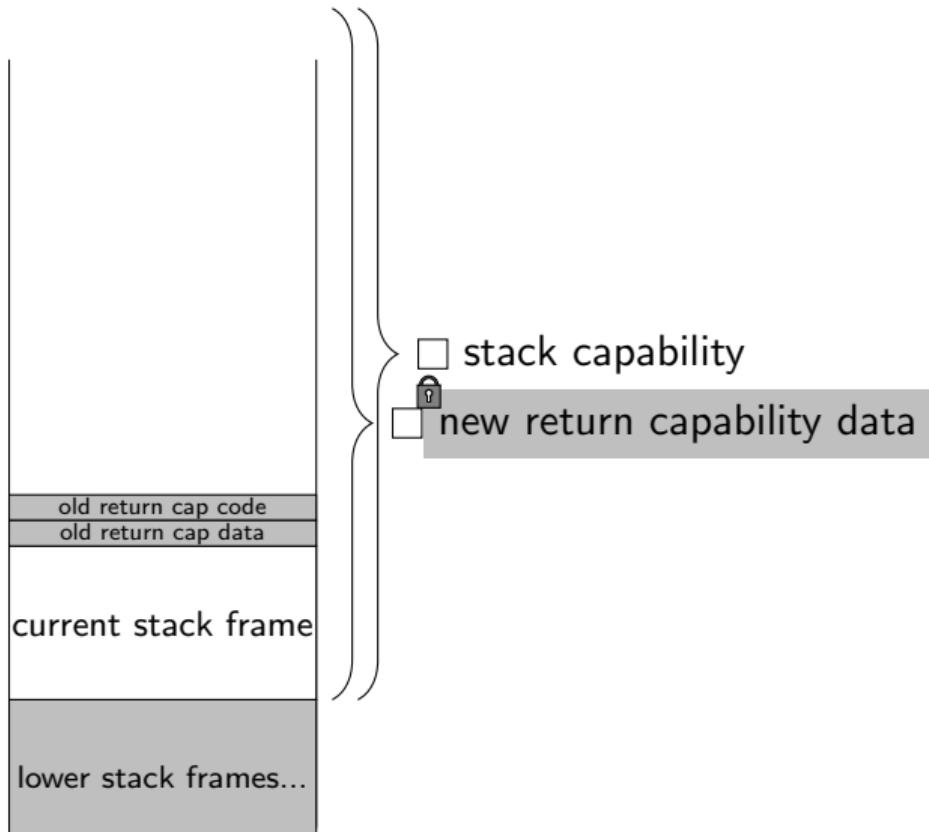
Naive stack and return capabilities



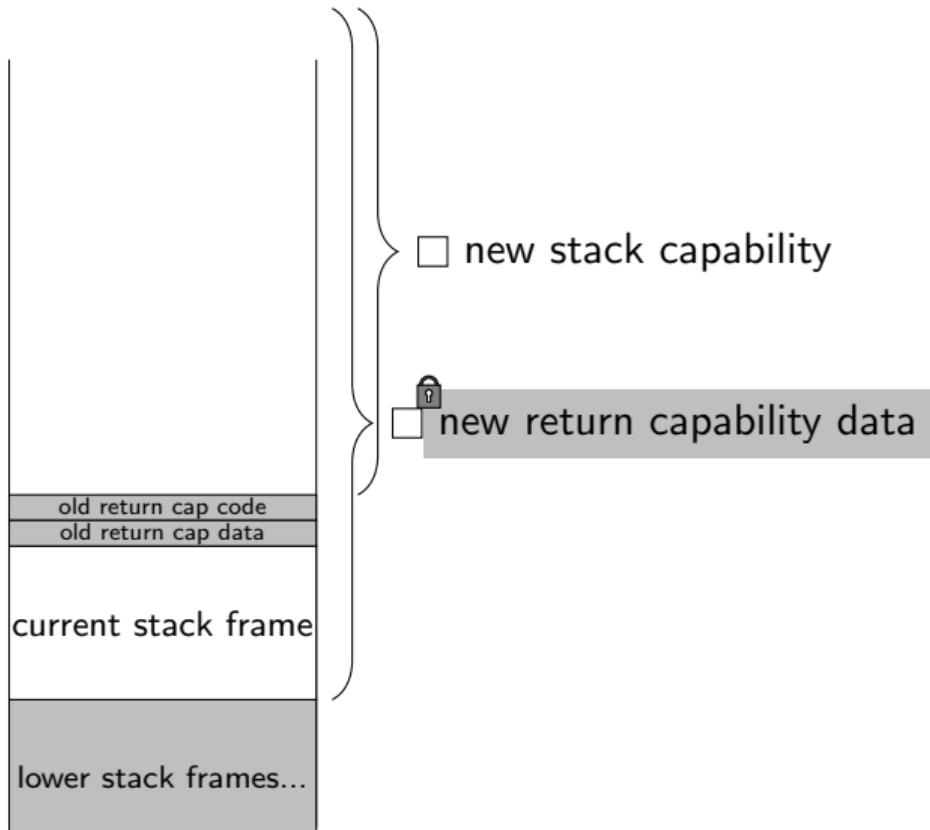
Naive stack and return capabilities



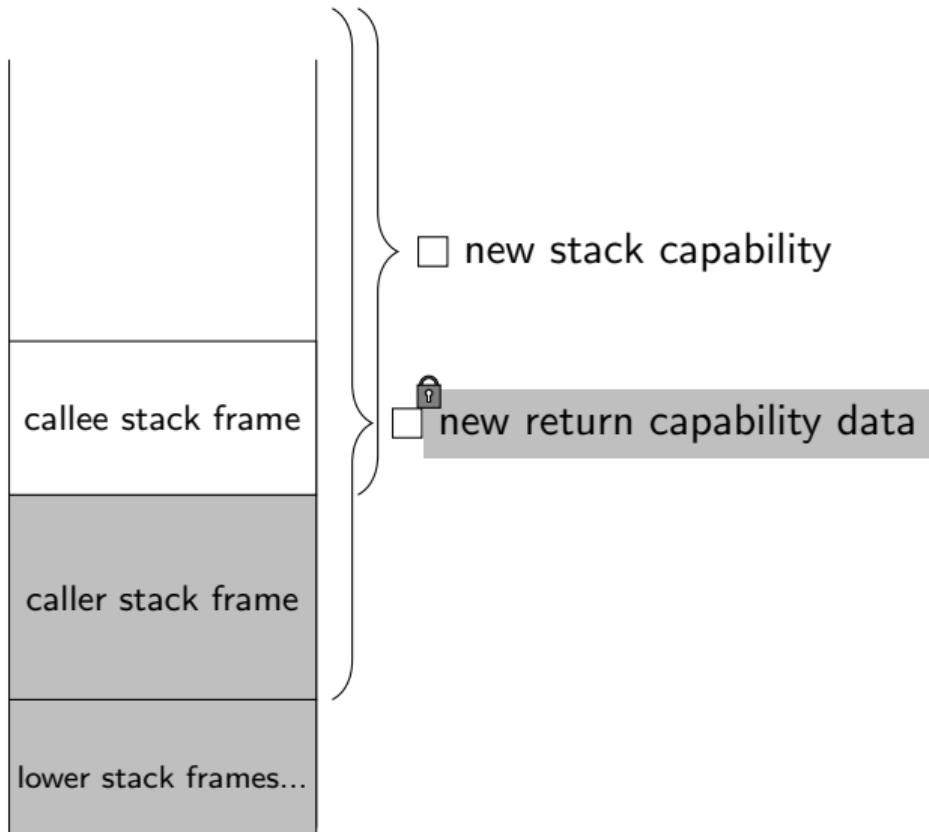
Naive stack and return capabilities



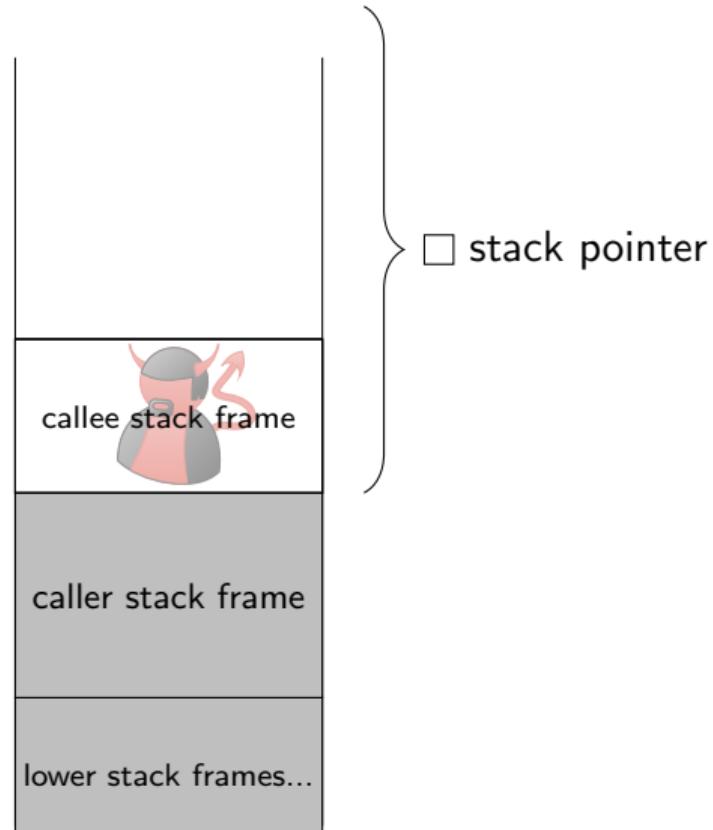
Naive stack and return capabilities



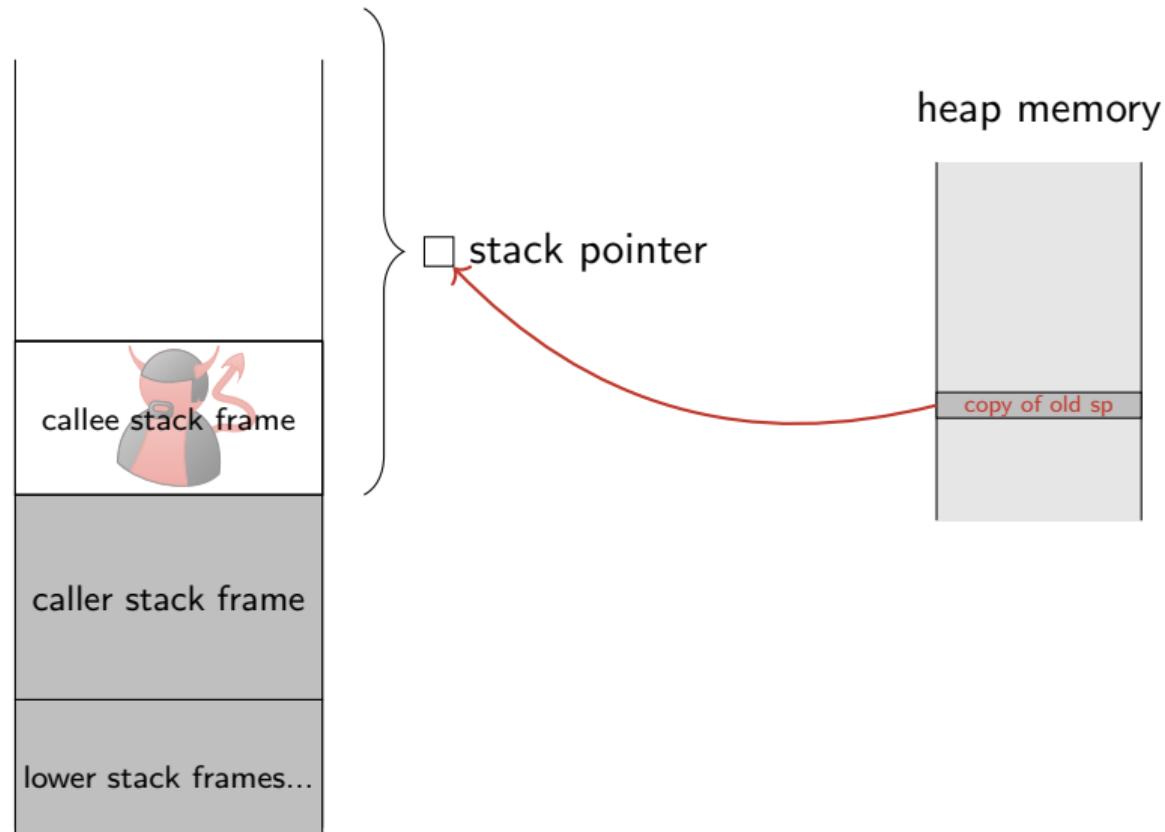
Naive stack and return capabilities



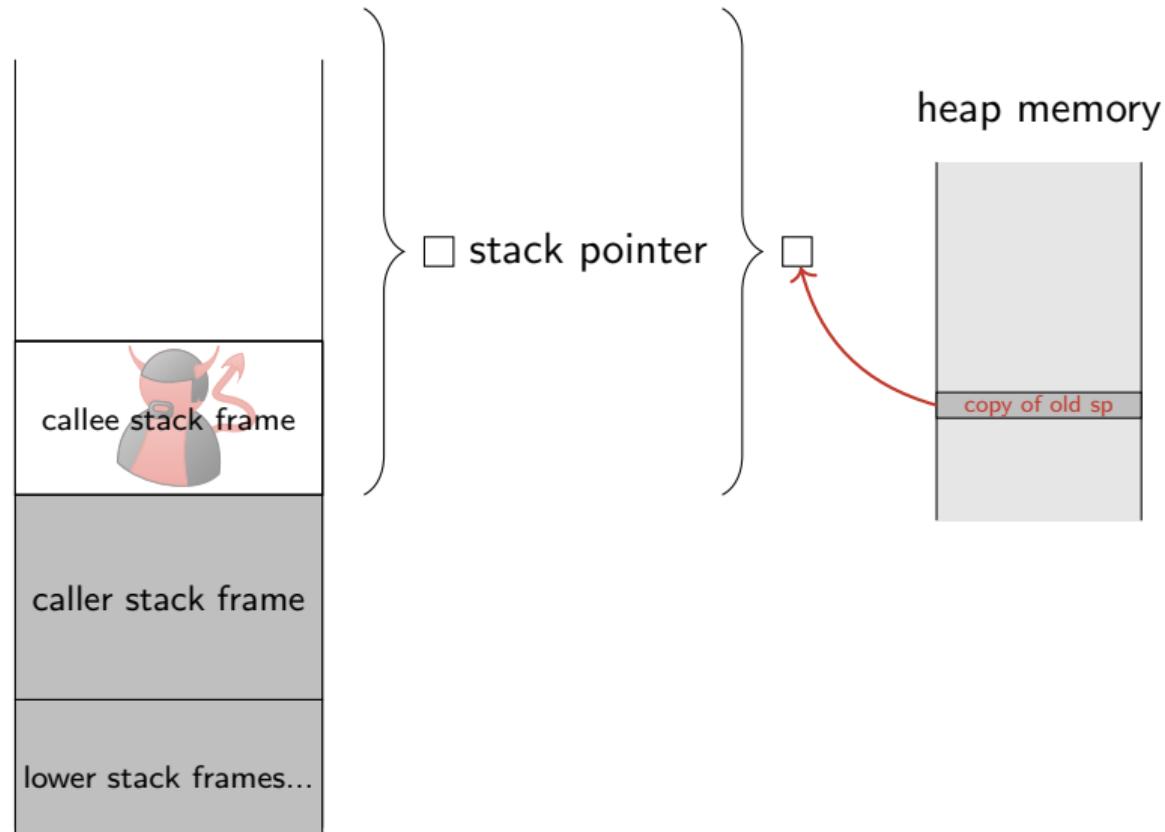
Attack 1 on naive stack and return capabilities



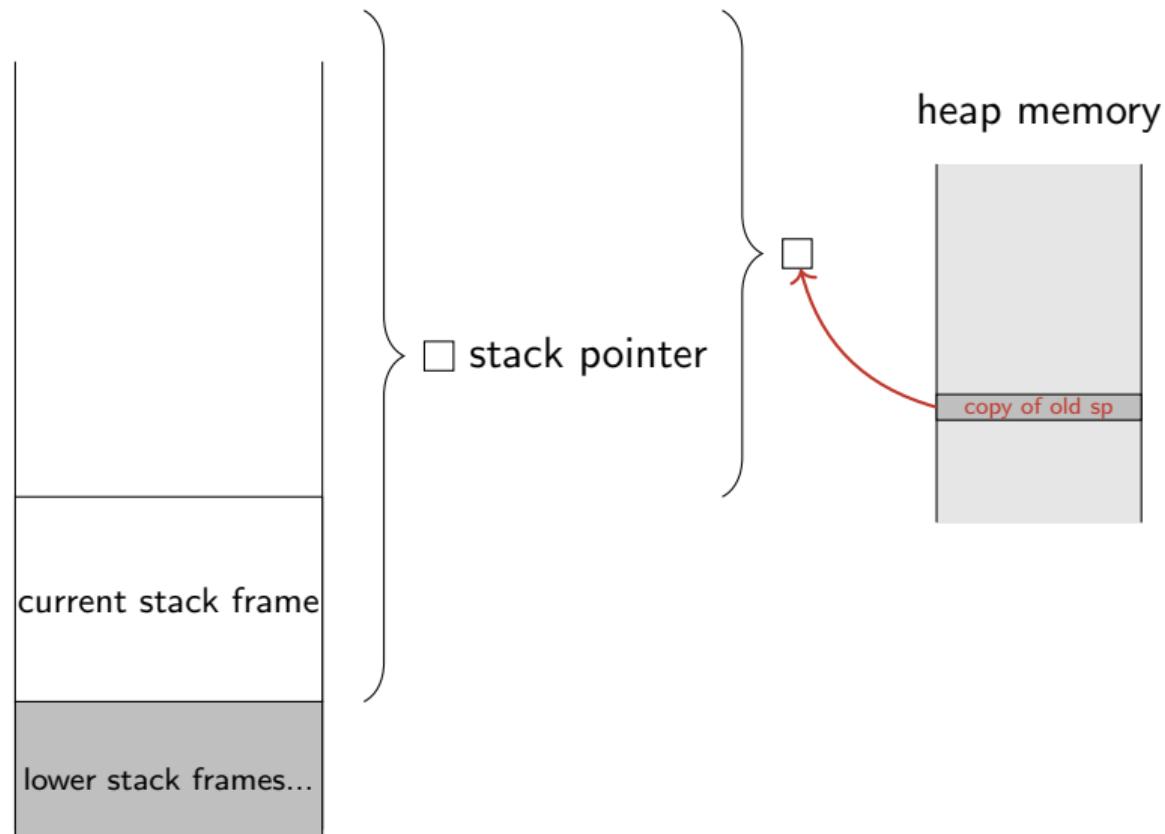
Attack 1 on naive stack and return capabilities



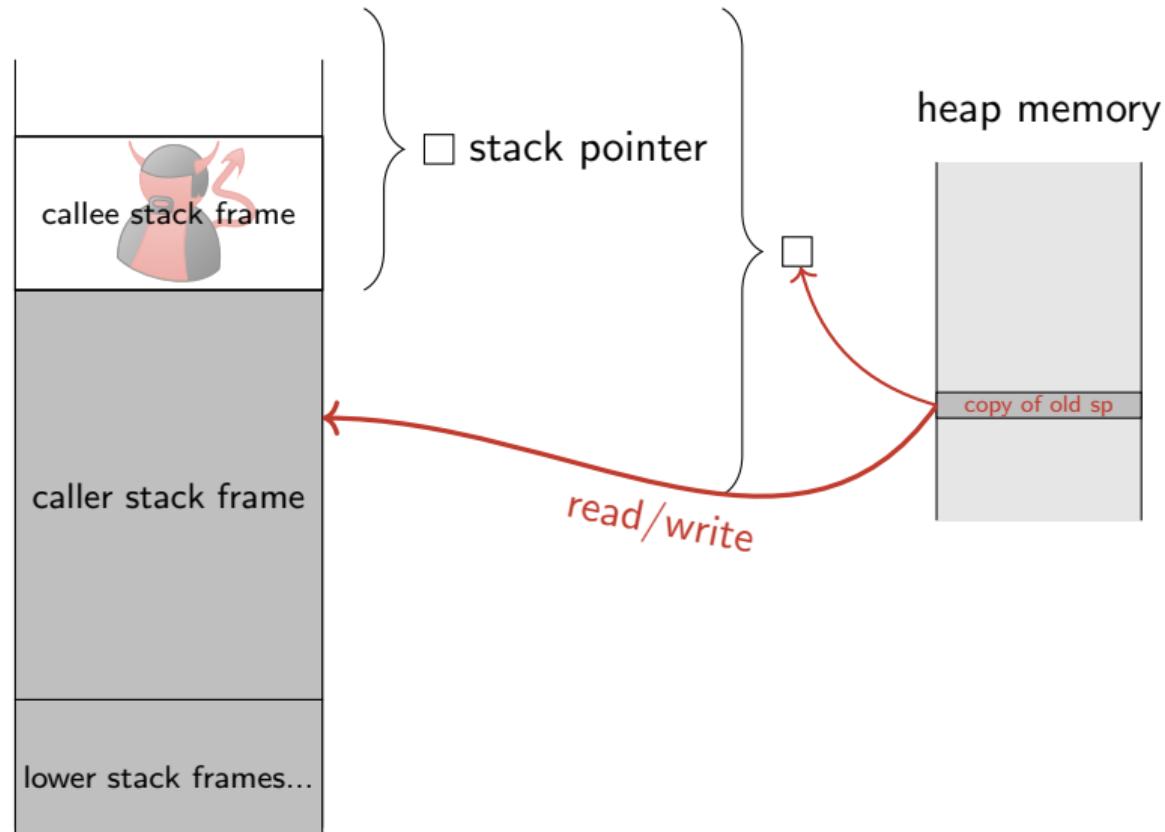
Attack 1 on naive stack and return capabilities



Attack 1 on naive stack and return capabilities



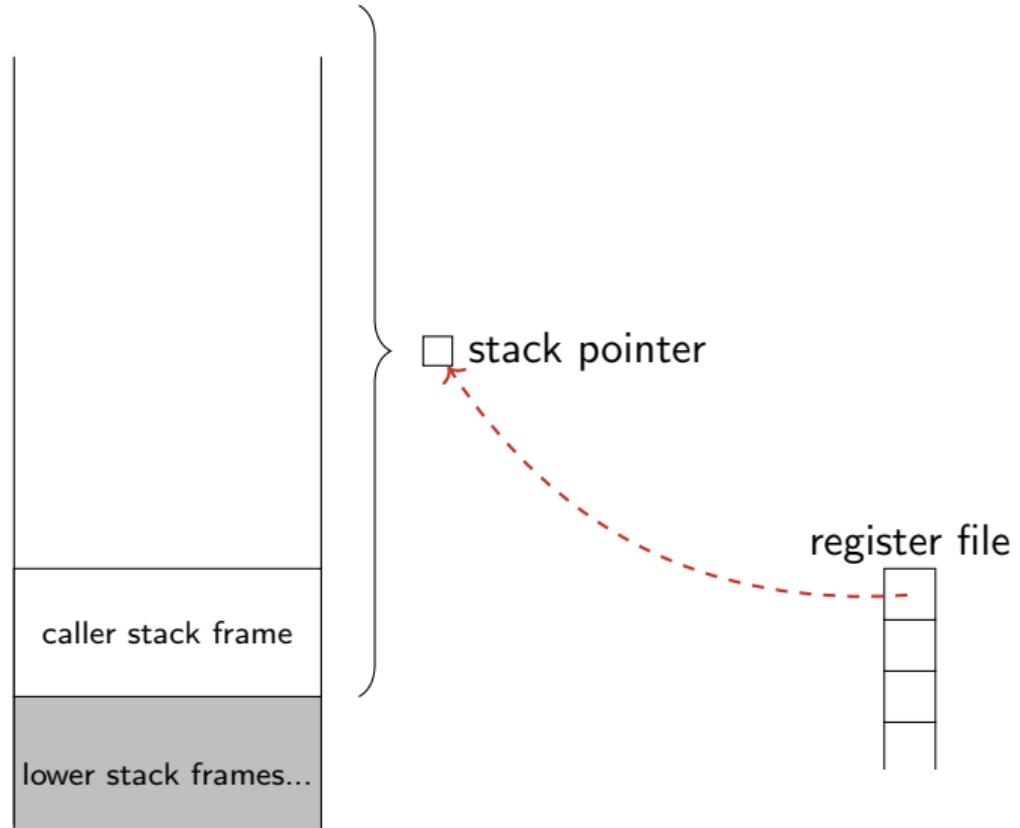
Attack 1 on naive stack and return capabilities



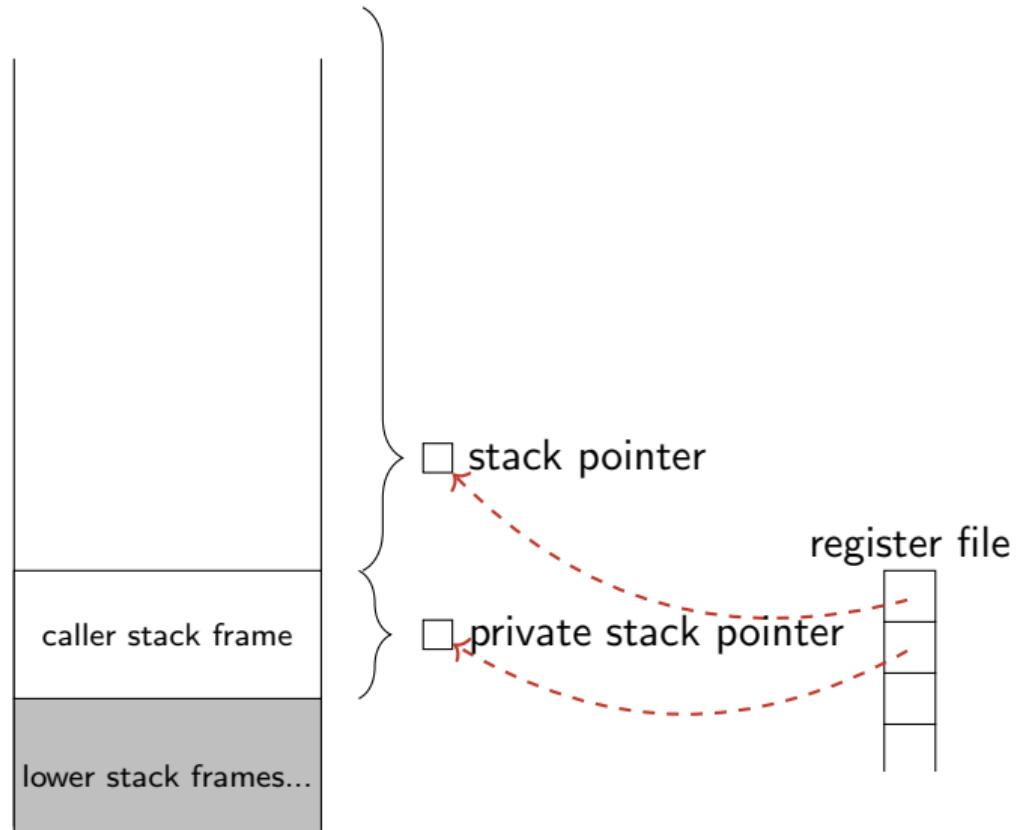
STKTOKENS

- ▶ Check the base address of the stack capability before and after calls.
- ▶ Make sure that local stack frames are non-empty.
- ▶ Create token and data return capability on call: split the stack capability in two to get a stack capability for your local stack frame and a stack capability for the unused part of the stack. The former is sealed and used for the data part of the return pair.
- ▶ Create code return capability on call: Seal the old program pointer.
- ▶ Reasonable use of seals: Return seals are only used to seal old program pointers, every return seal is only used for one call site, and they are not leaked.

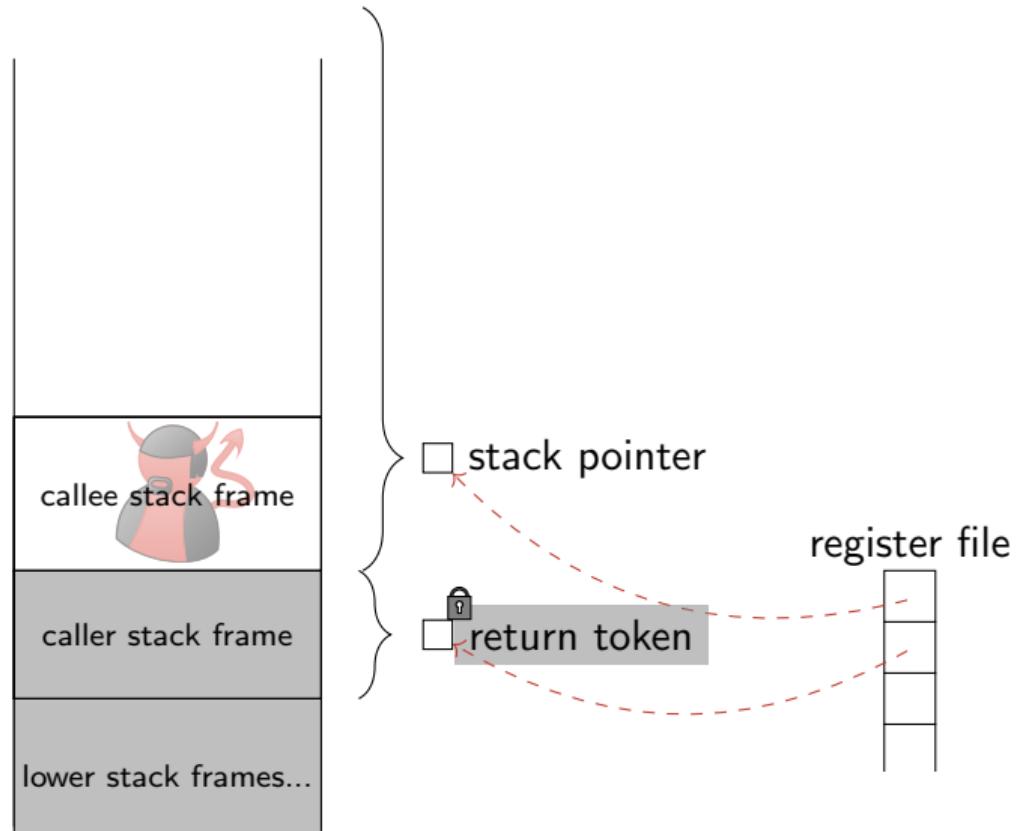
STKTOKENS prevent attack 1



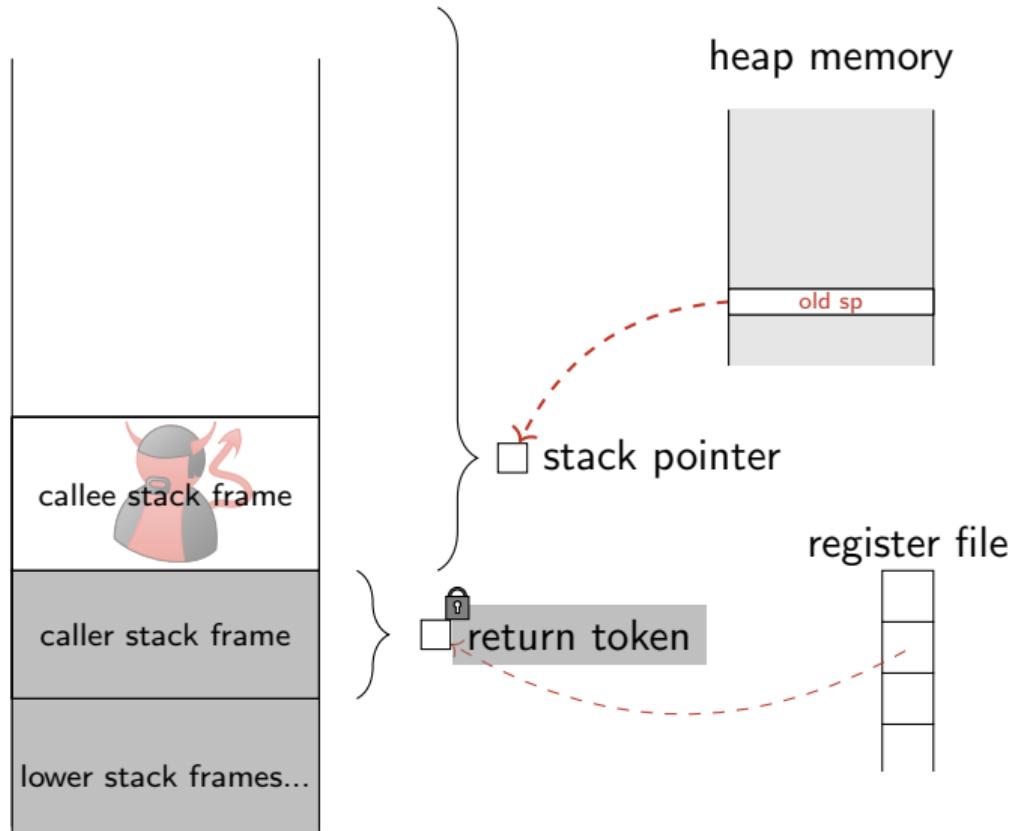
STKTOKENS prevent attack 1



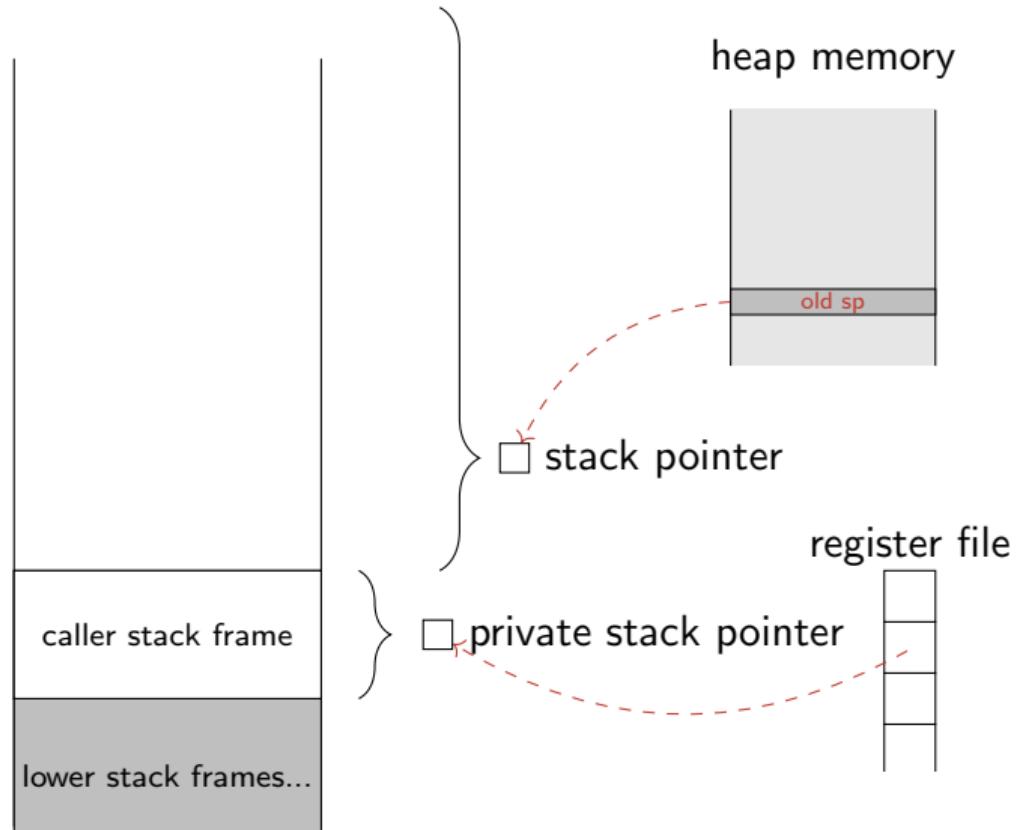
STKTOKENS prevent attack 1



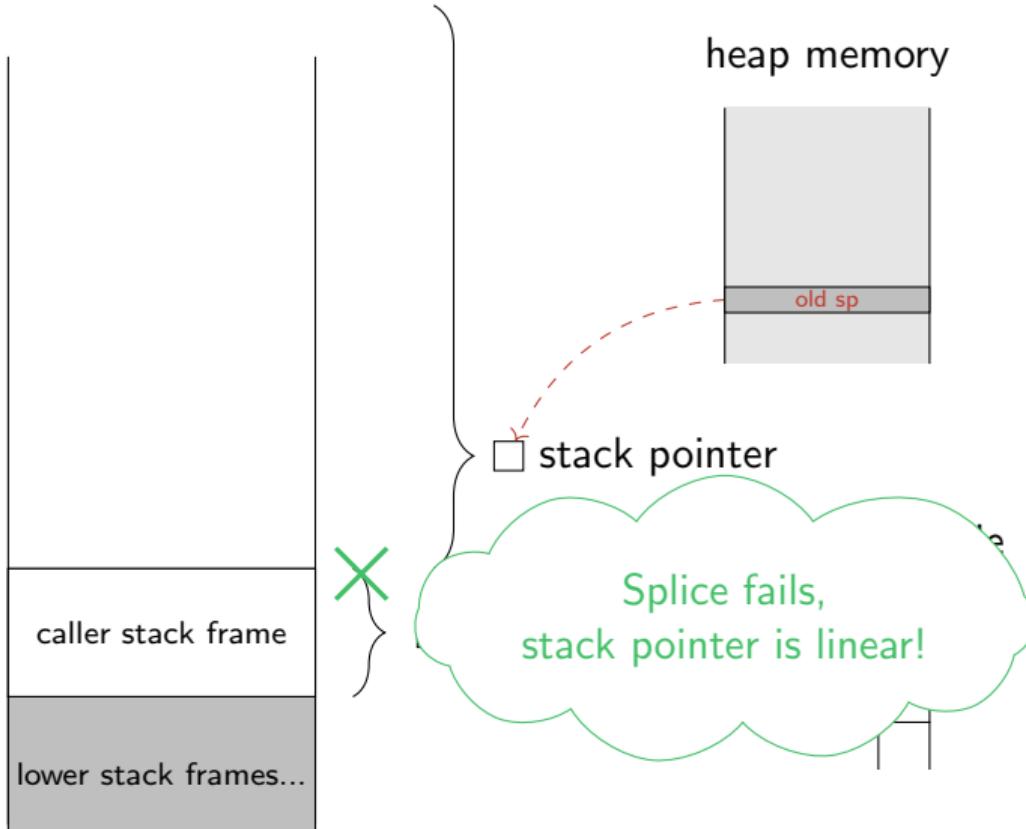
STKTOKENS prevent attack 1



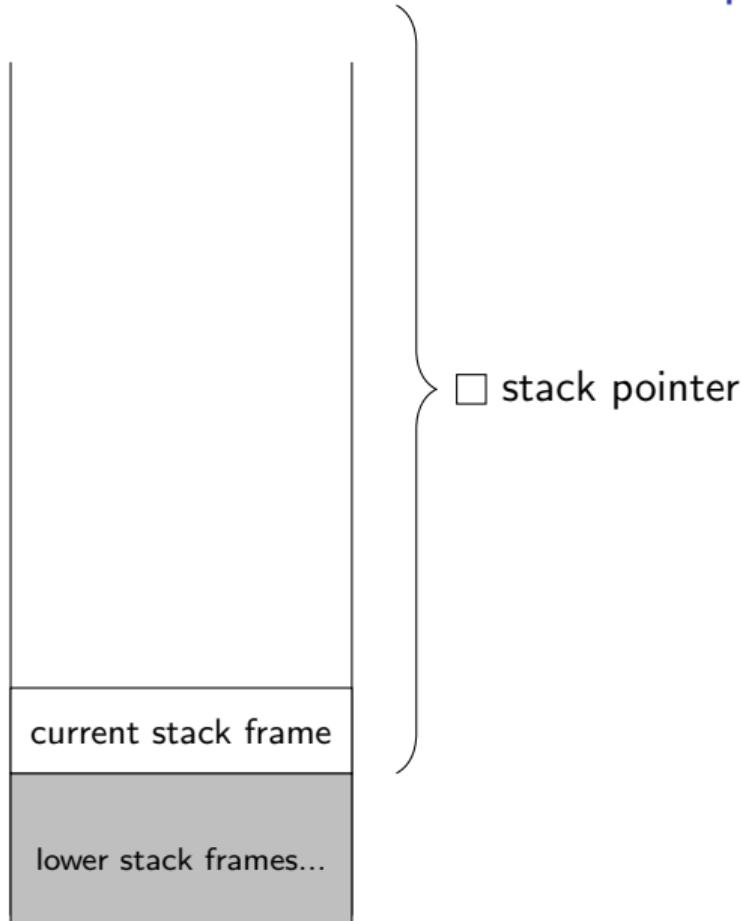
STKTOKENS prevent attack 1



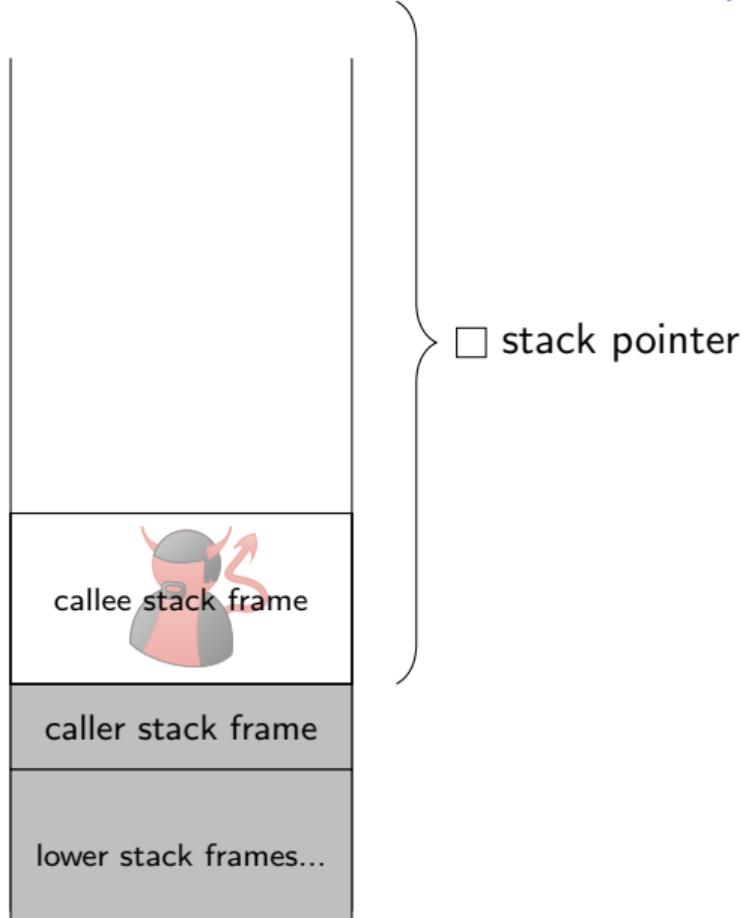
STKTOKENS prevent attack 1



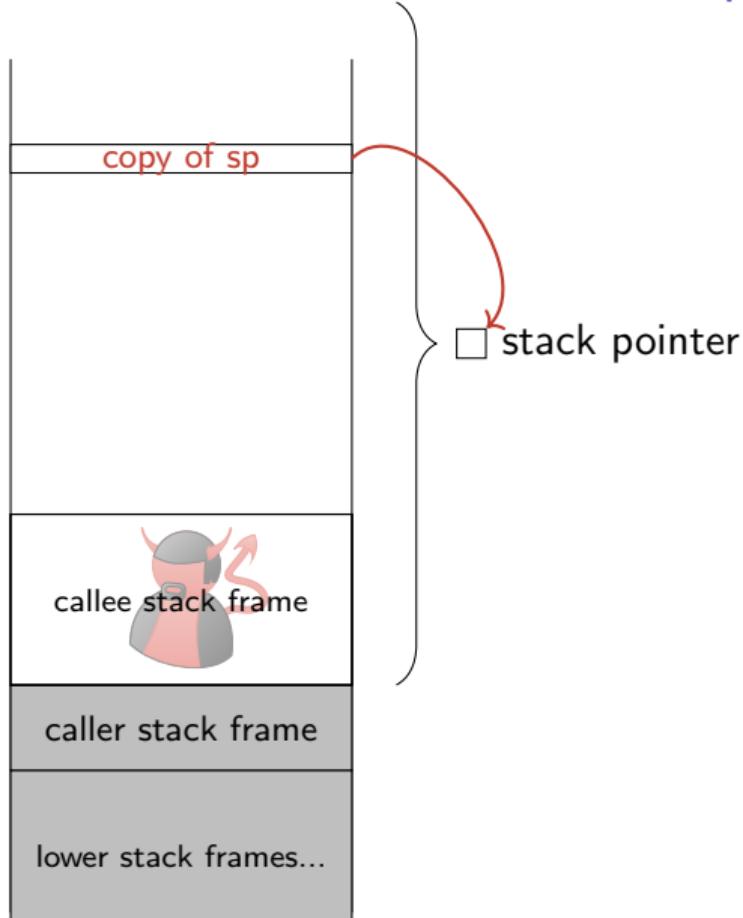
Attack 2 on naive stack and return capabilities



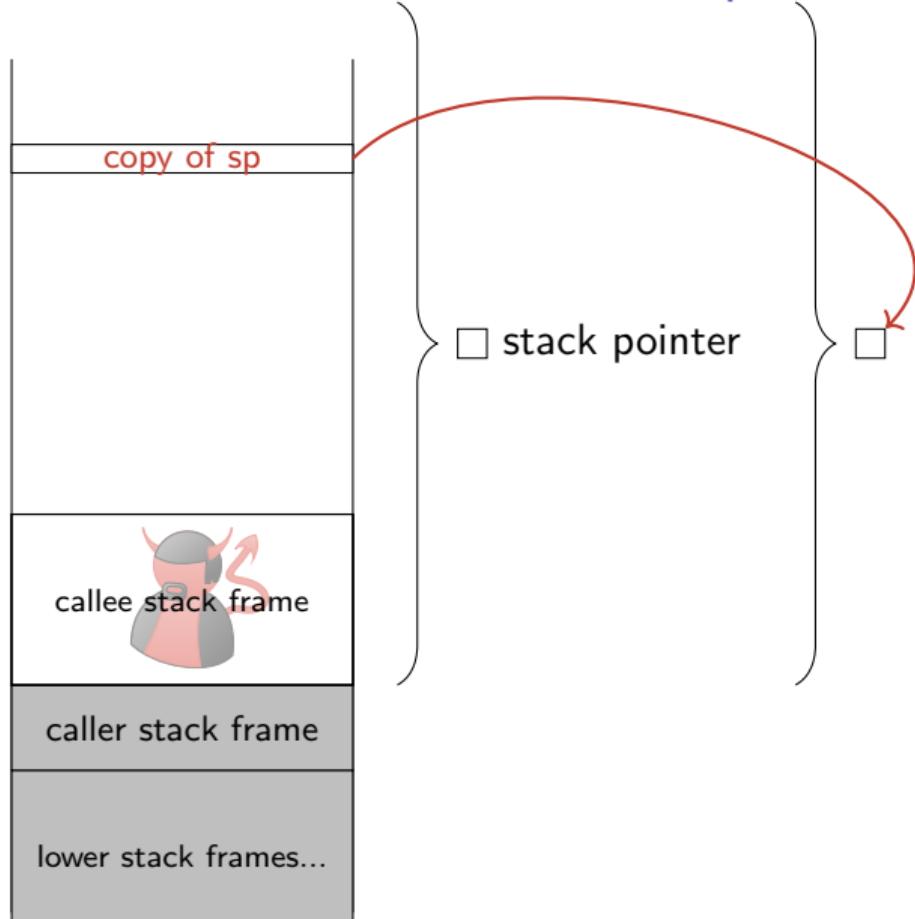
Attack 2 on naive stack and return capabilities



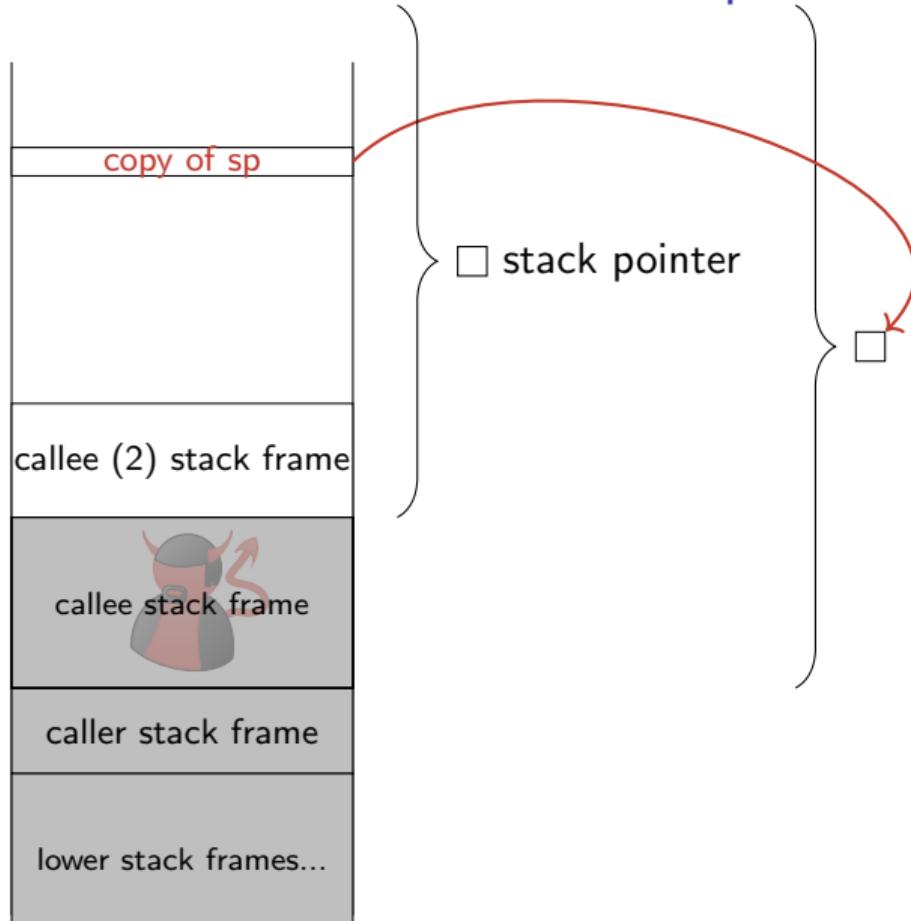
Attack 2 on naive stack and return capabilities



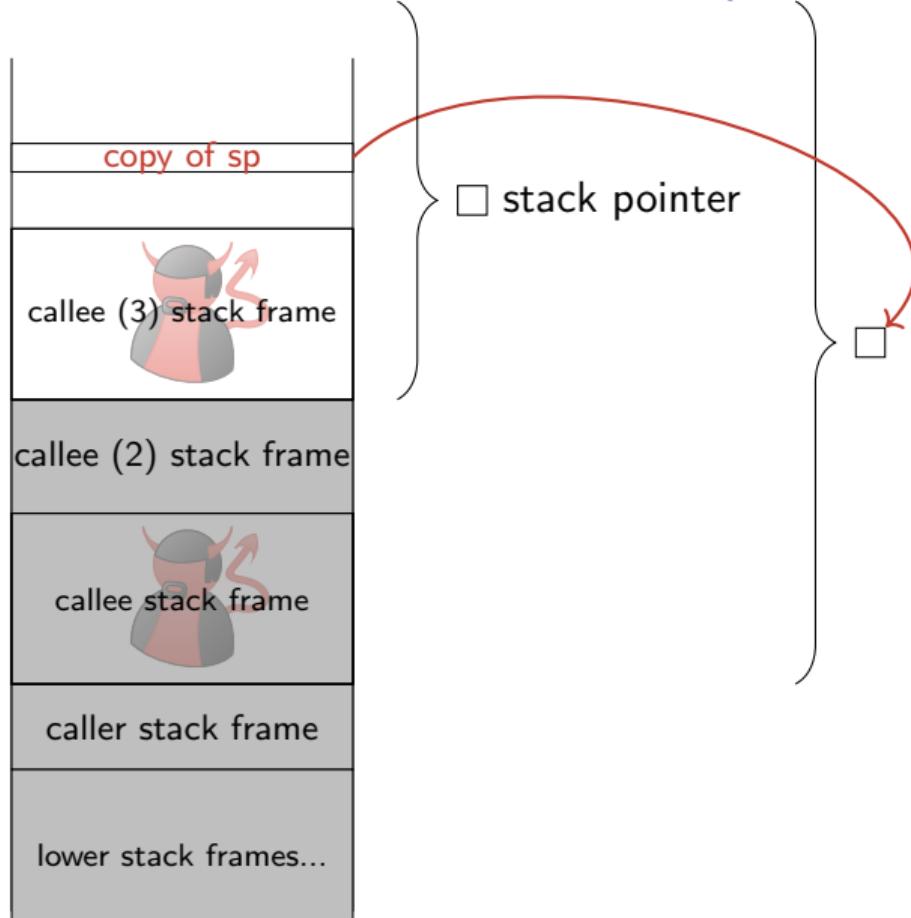
Attack 2 on naive stack and return capabilities



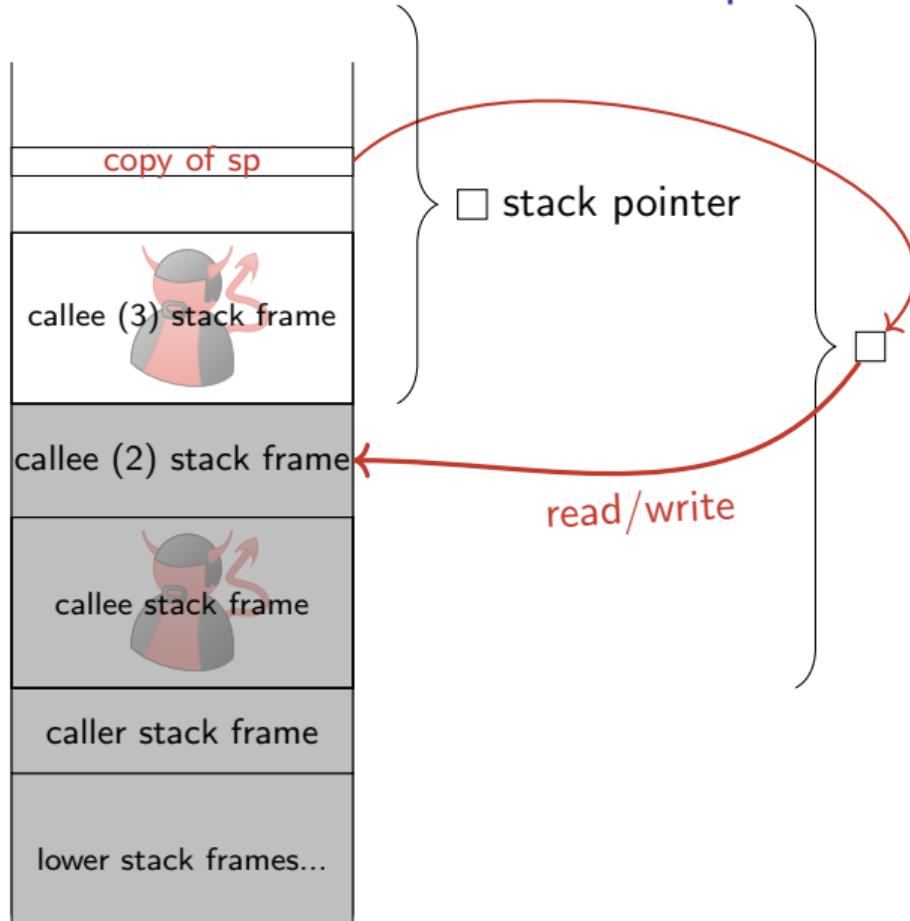
Attack 2 on naive stack and return capabilities



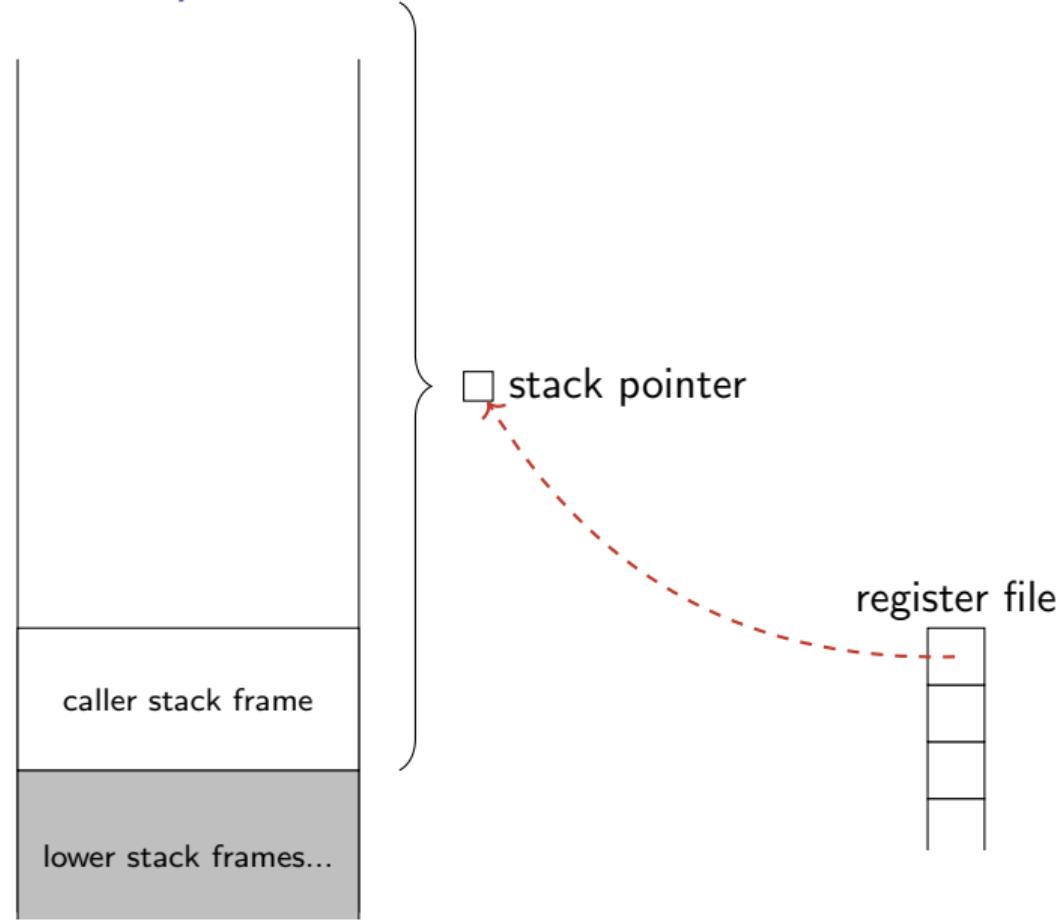
Attack 2 on naive stack and return capabilities



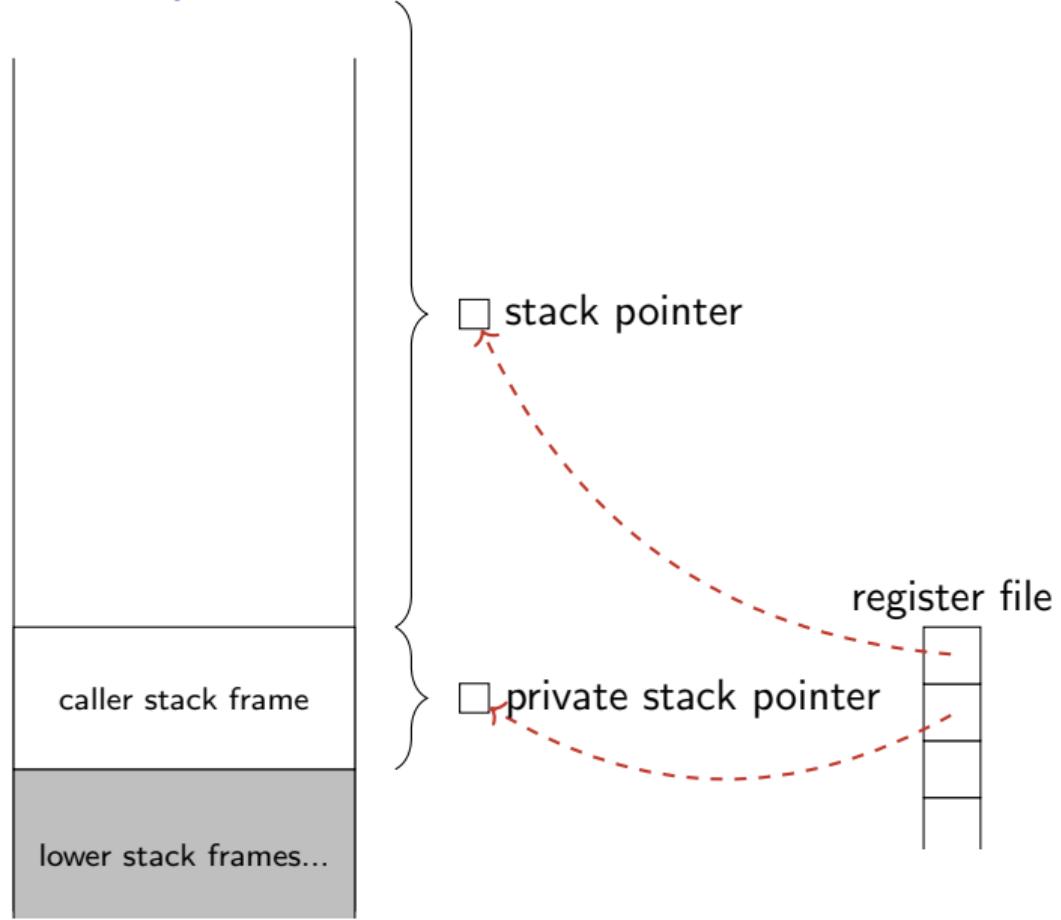
Attack 2 on naive stack and return capabilities



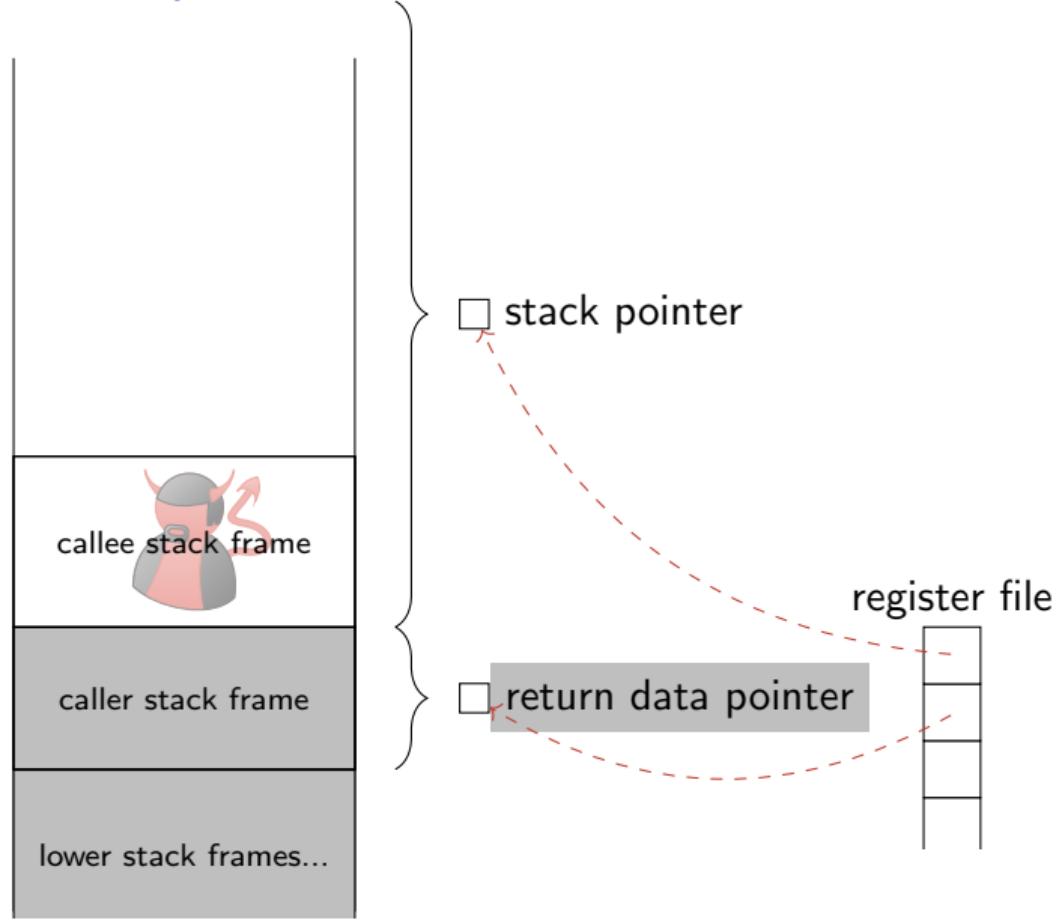
STKTOKENS prevent attack 2



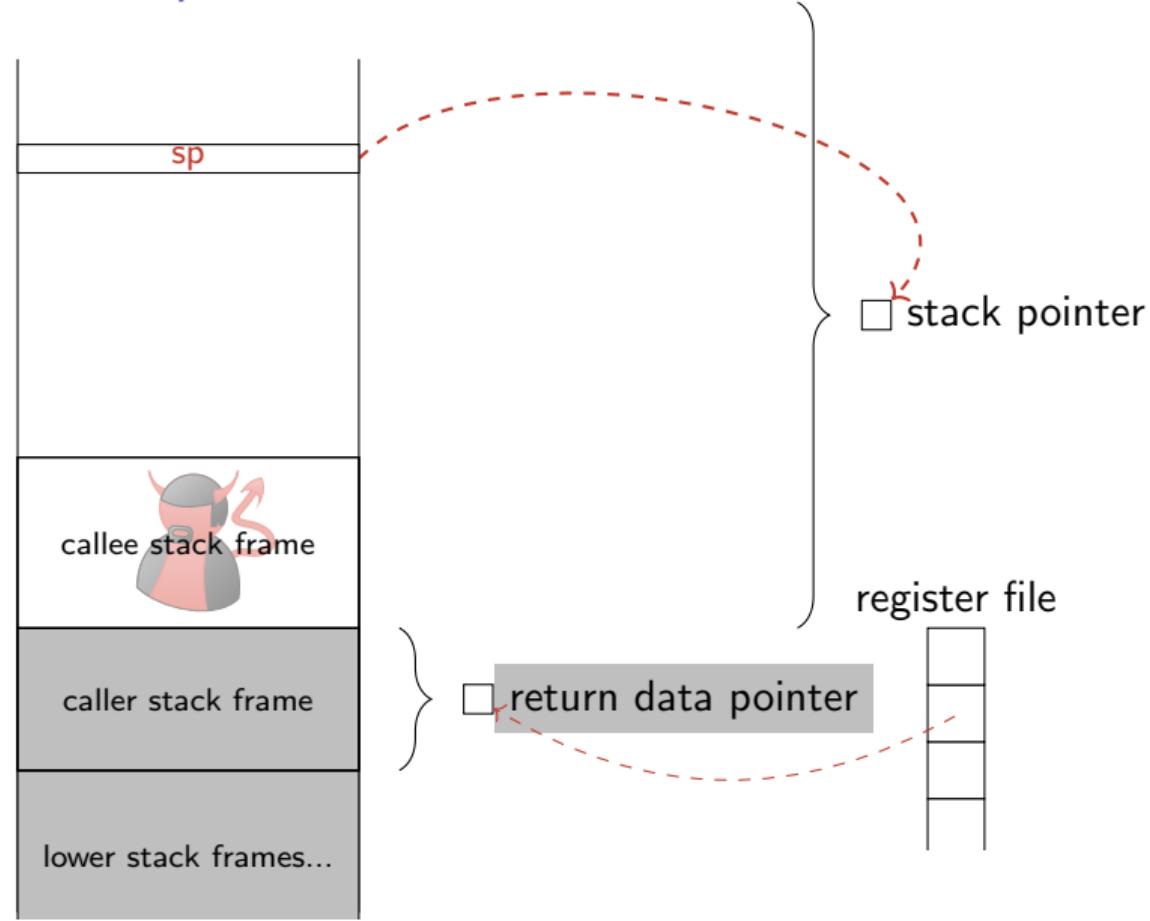
STKTOKENS prevent attack 2



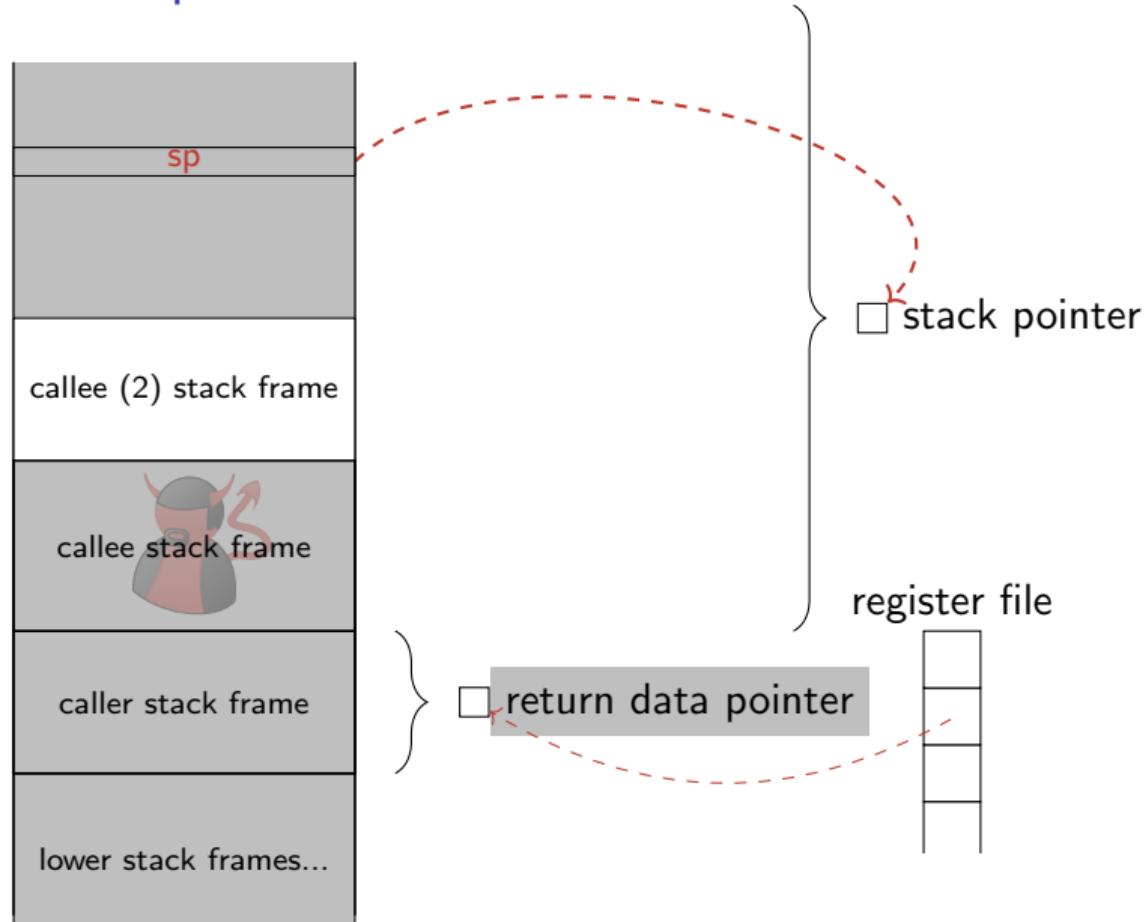
STKTOKENS prevent attack 2



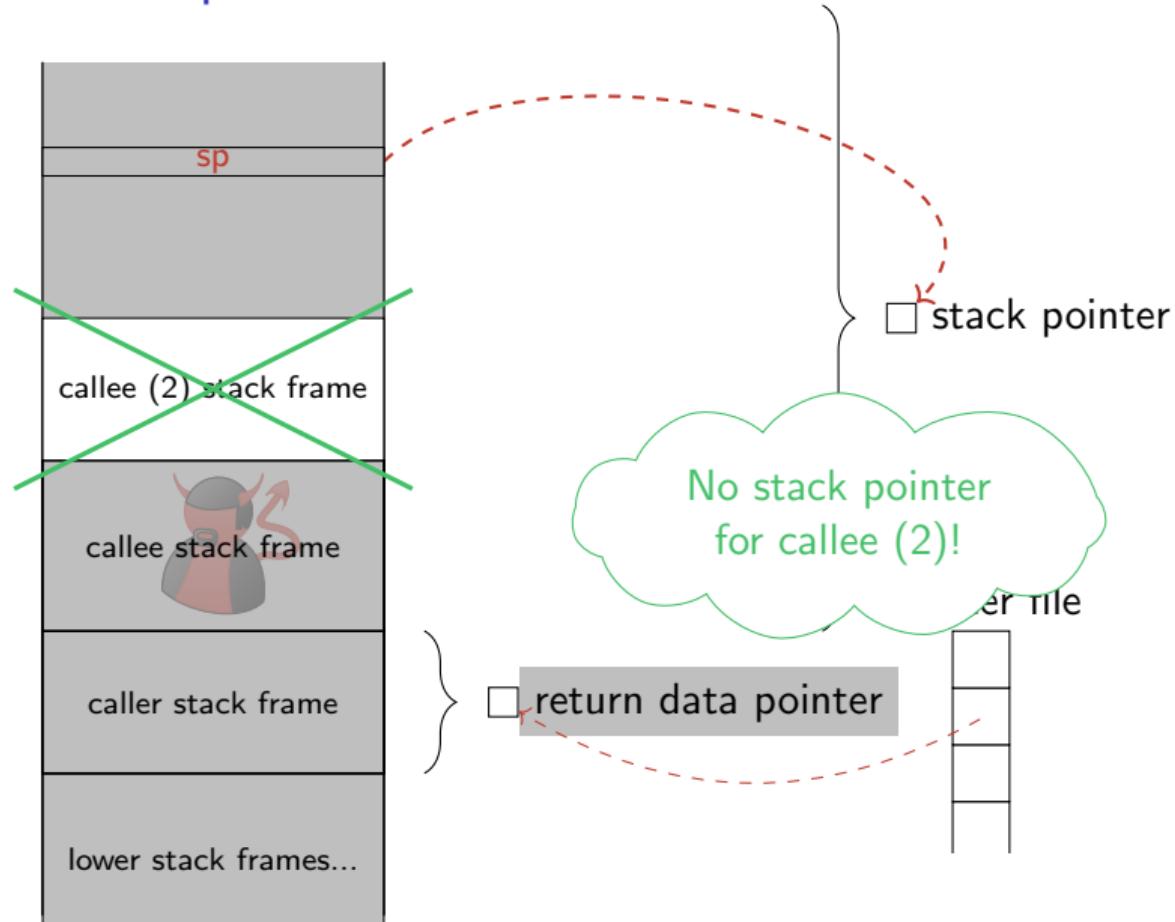
STKTOKENS prevent attack 2



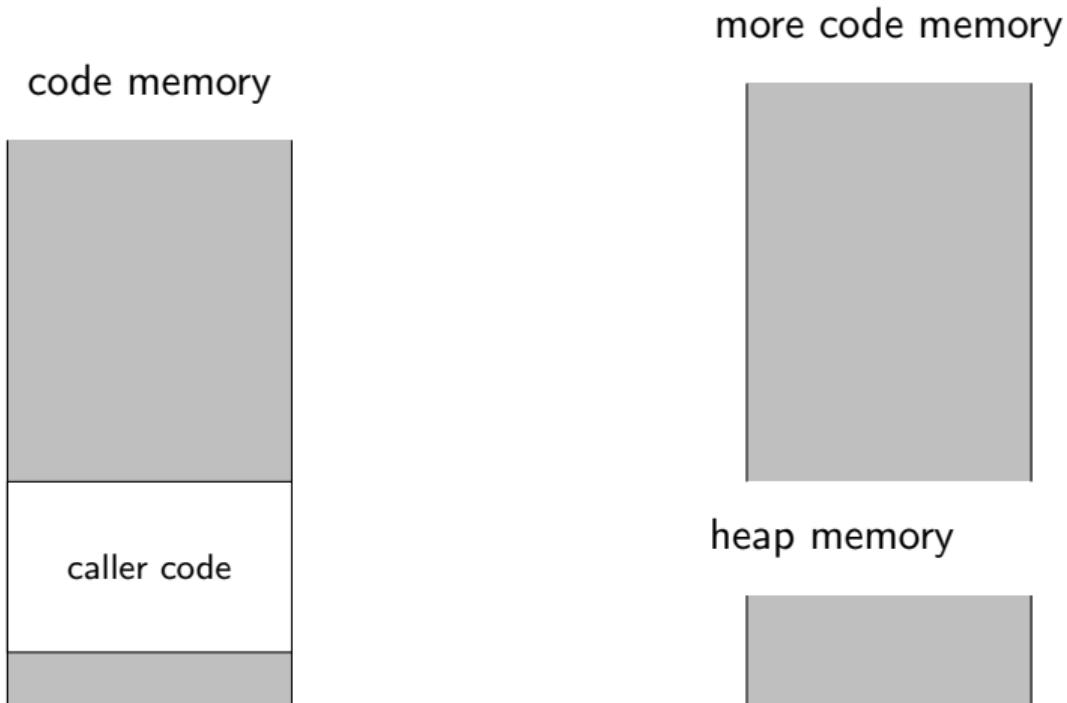
STKTOKENS prevent attack 2



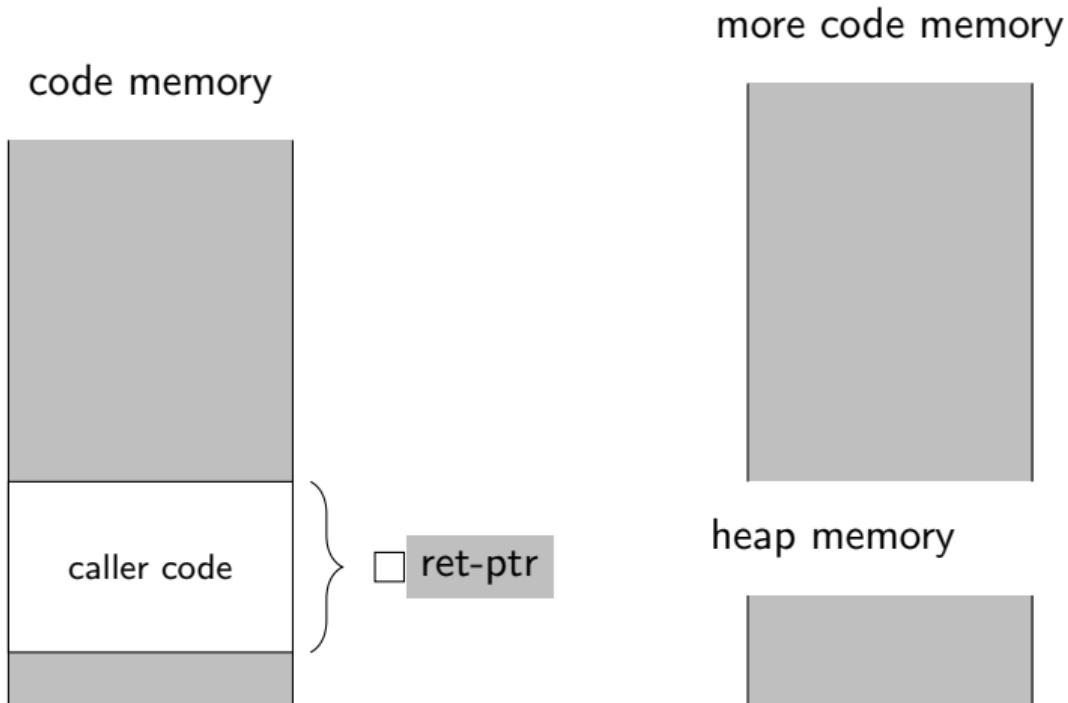
STKTOKENS prevent attack 2



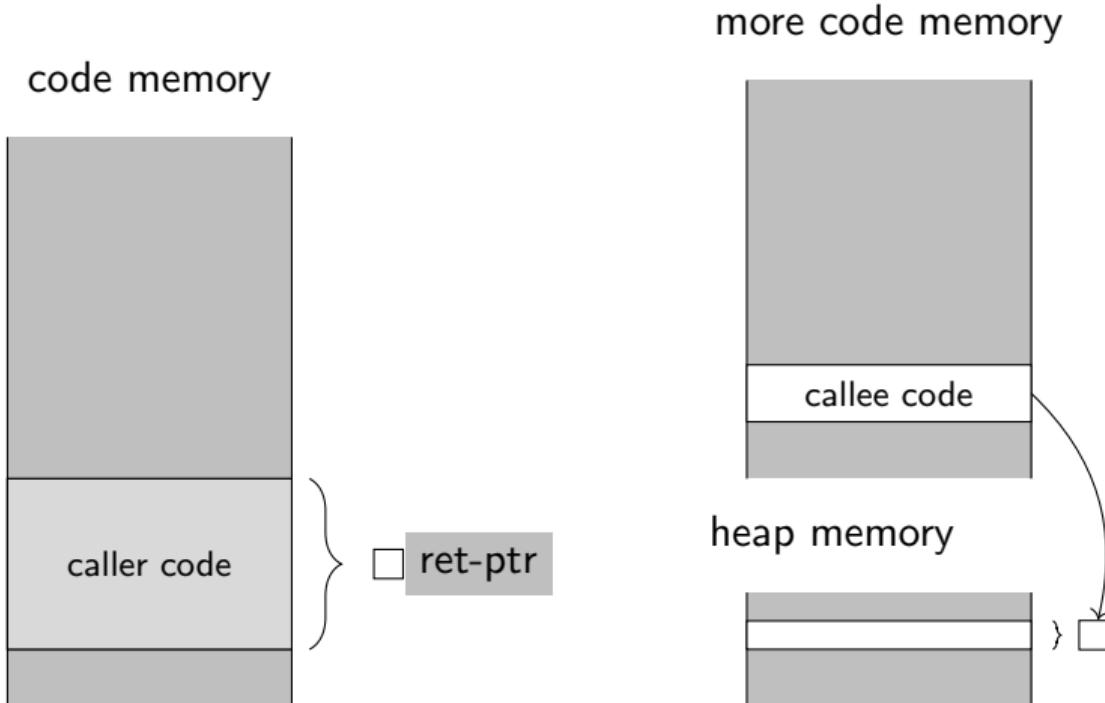
Attack 3 on naive stack and return capabilities



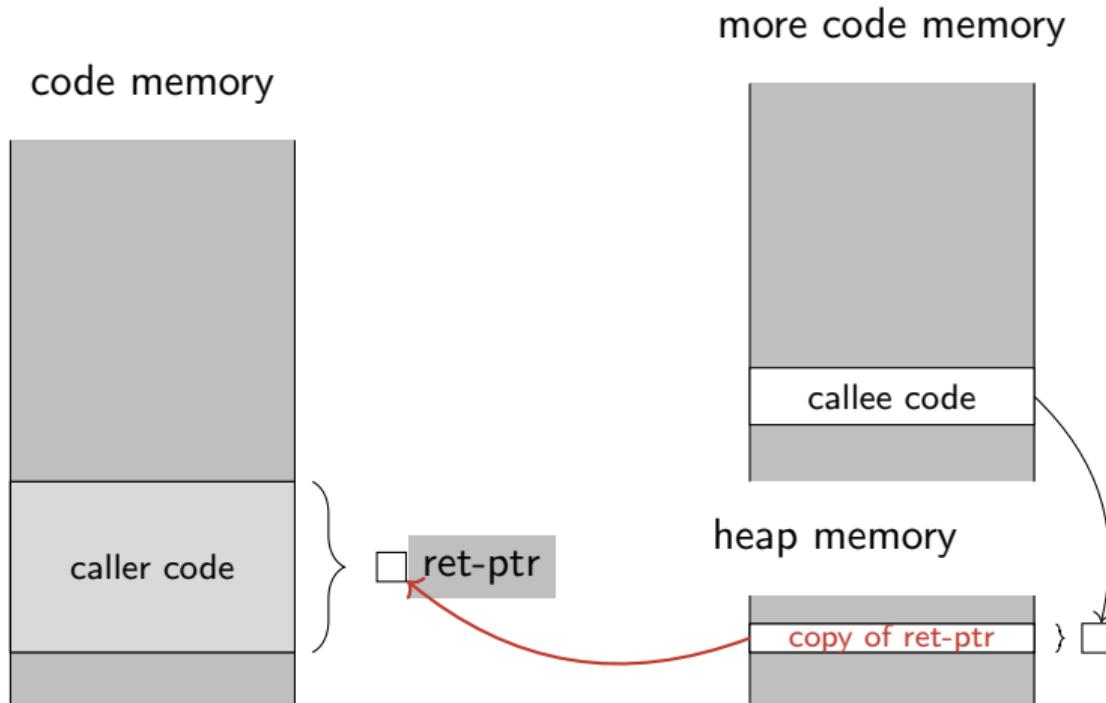
Attack 3 on naive stack and return capabilities



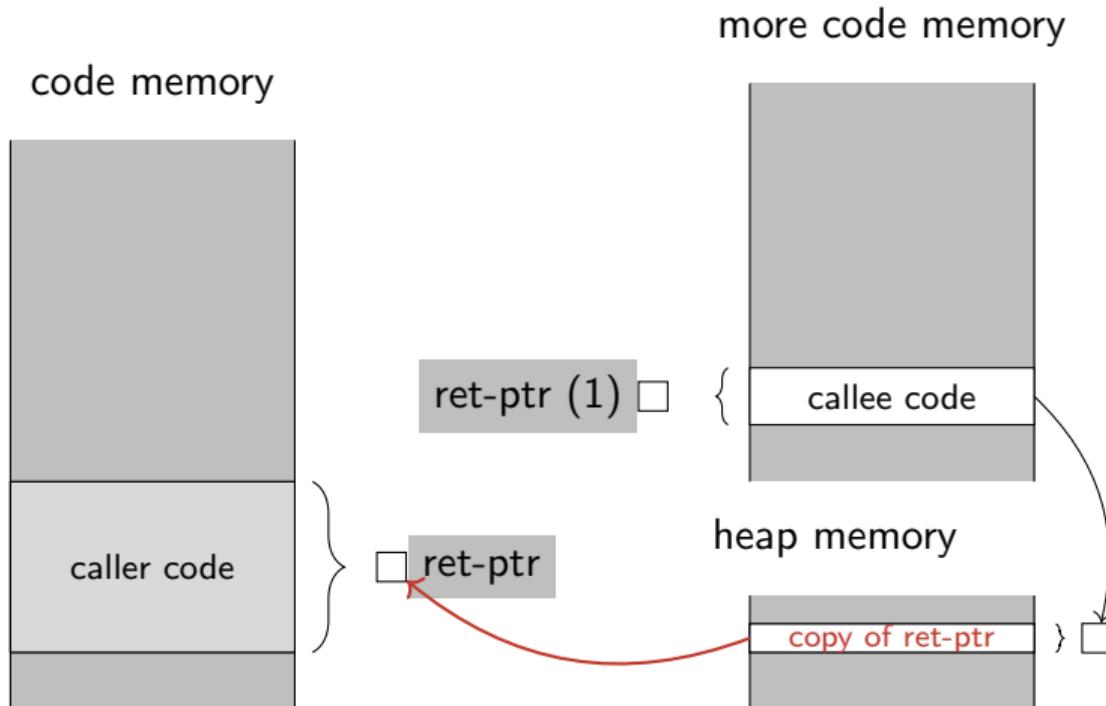
Attack 3 on naive stack and return capabilities



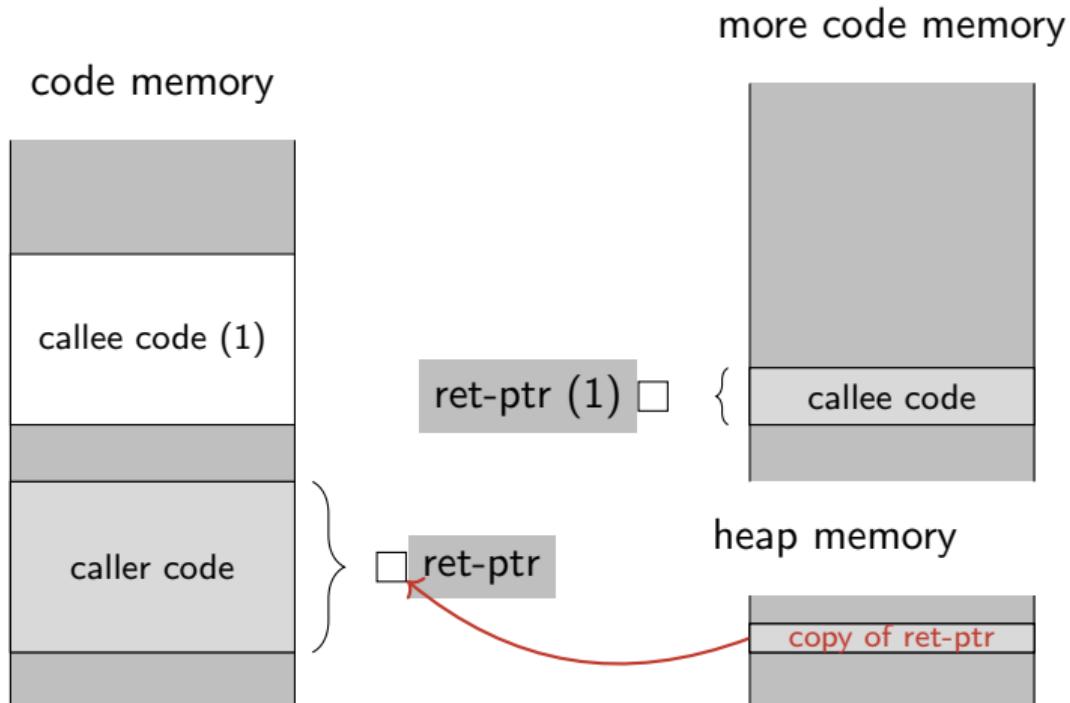
Attack 3 on naive stack and return capabilities



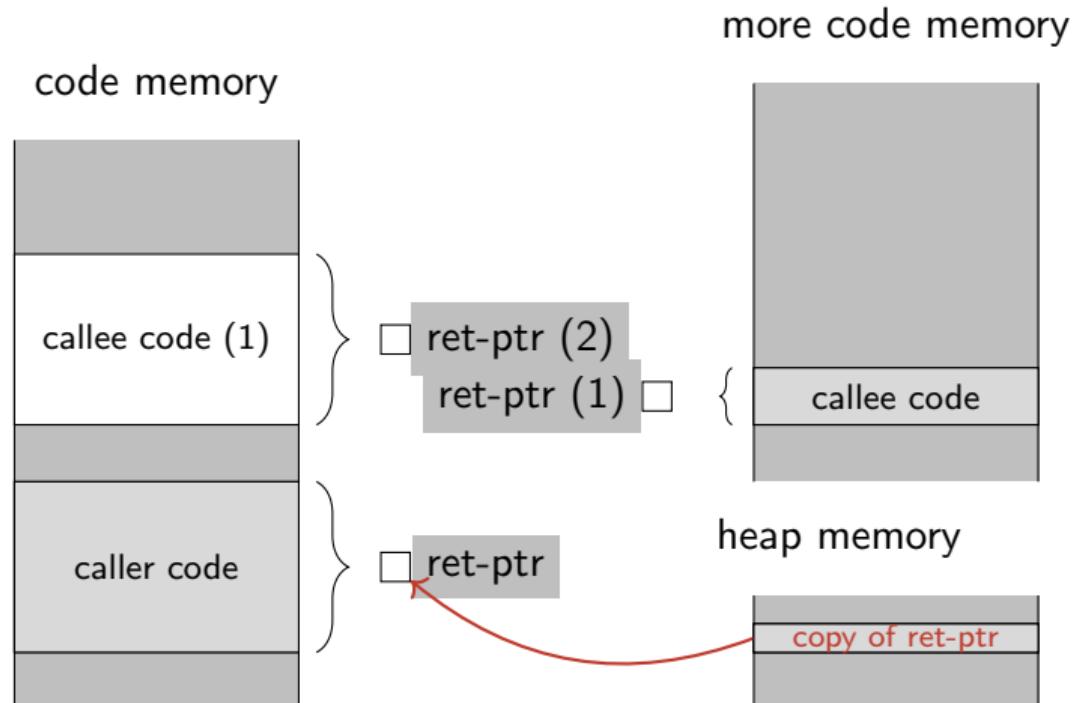
Attack 3 on naive stack and return capabilities



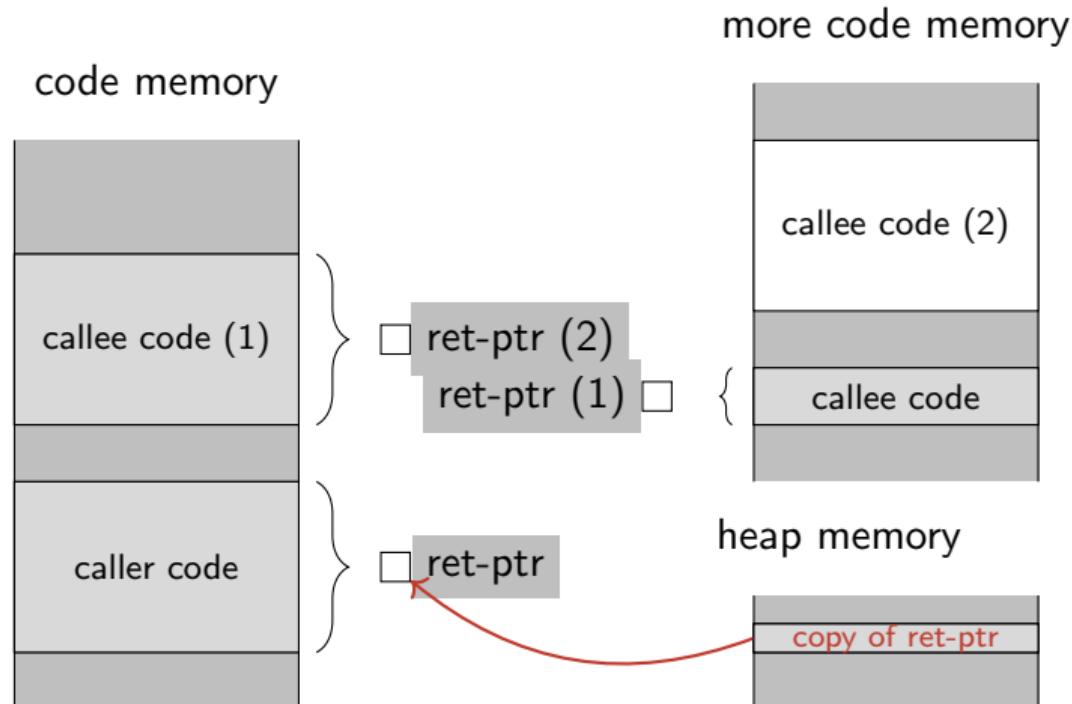
Attack 3 on naive stack and return capabilities



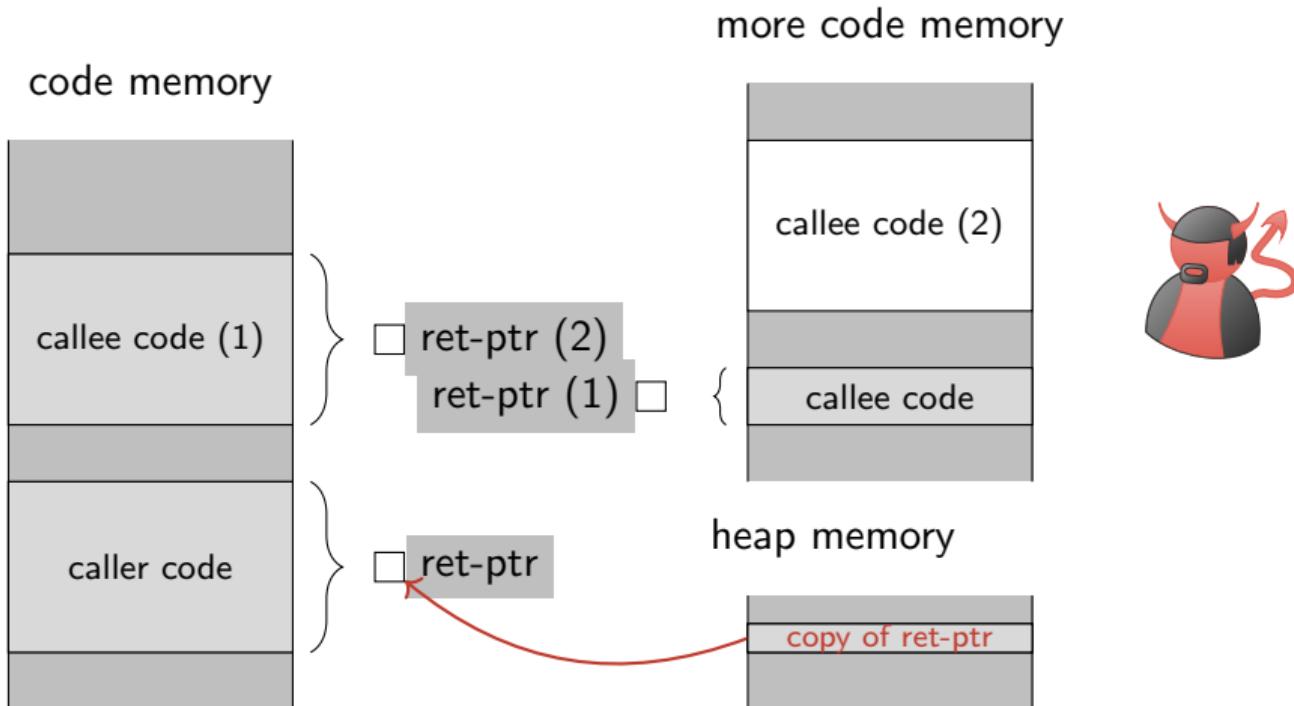
Attack 3 on naive stack and return capabilities



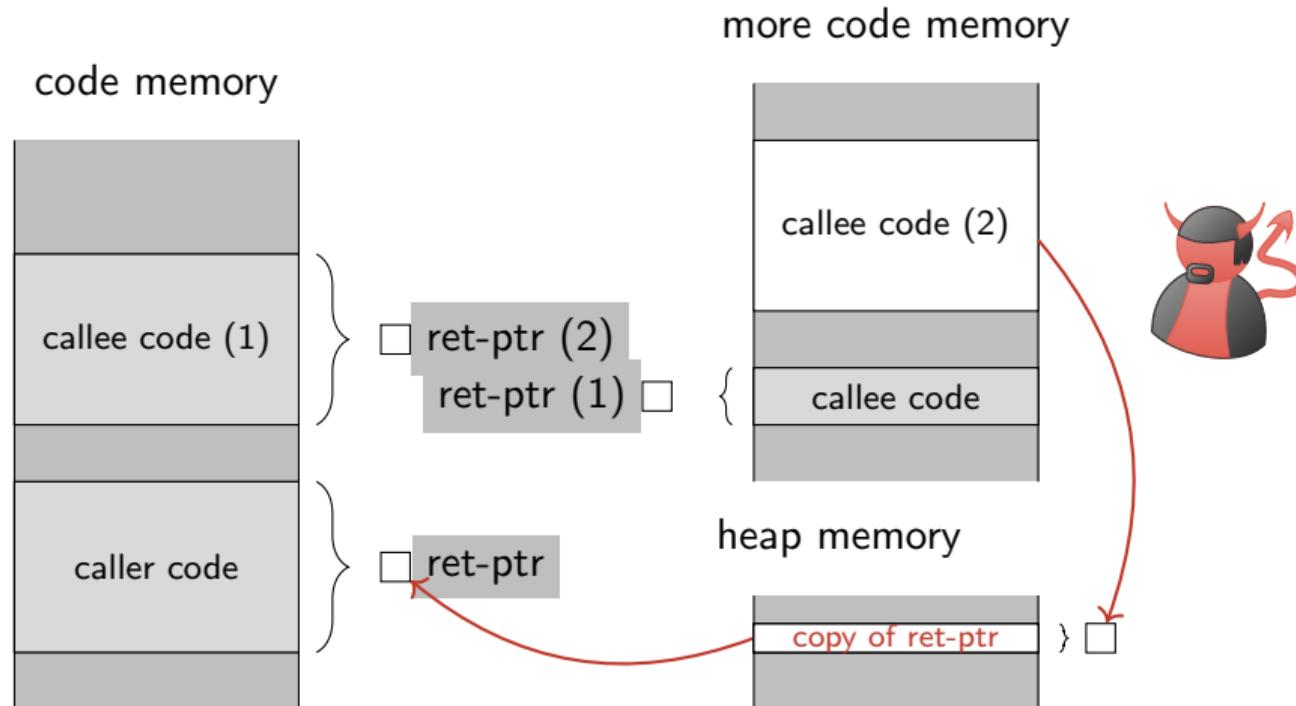
Attack 3 on naive stack and return capabilities



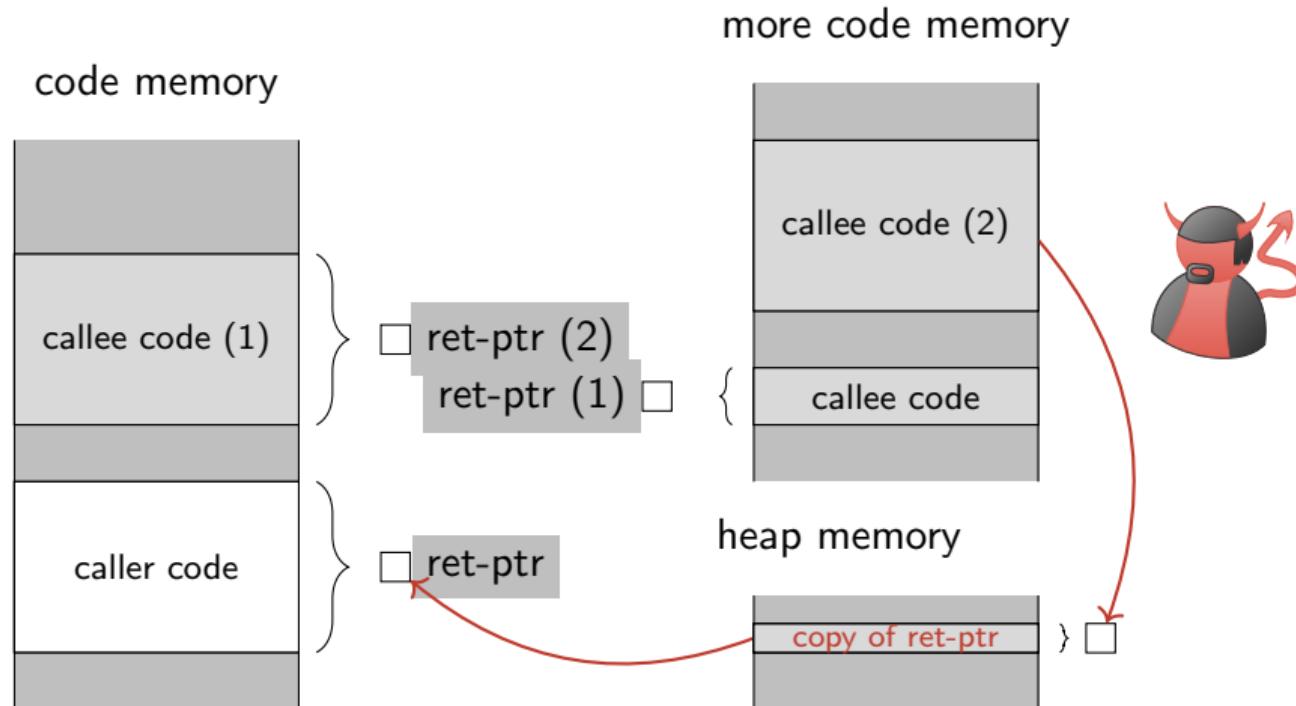
Attack 3 on naive stack and return capabilities



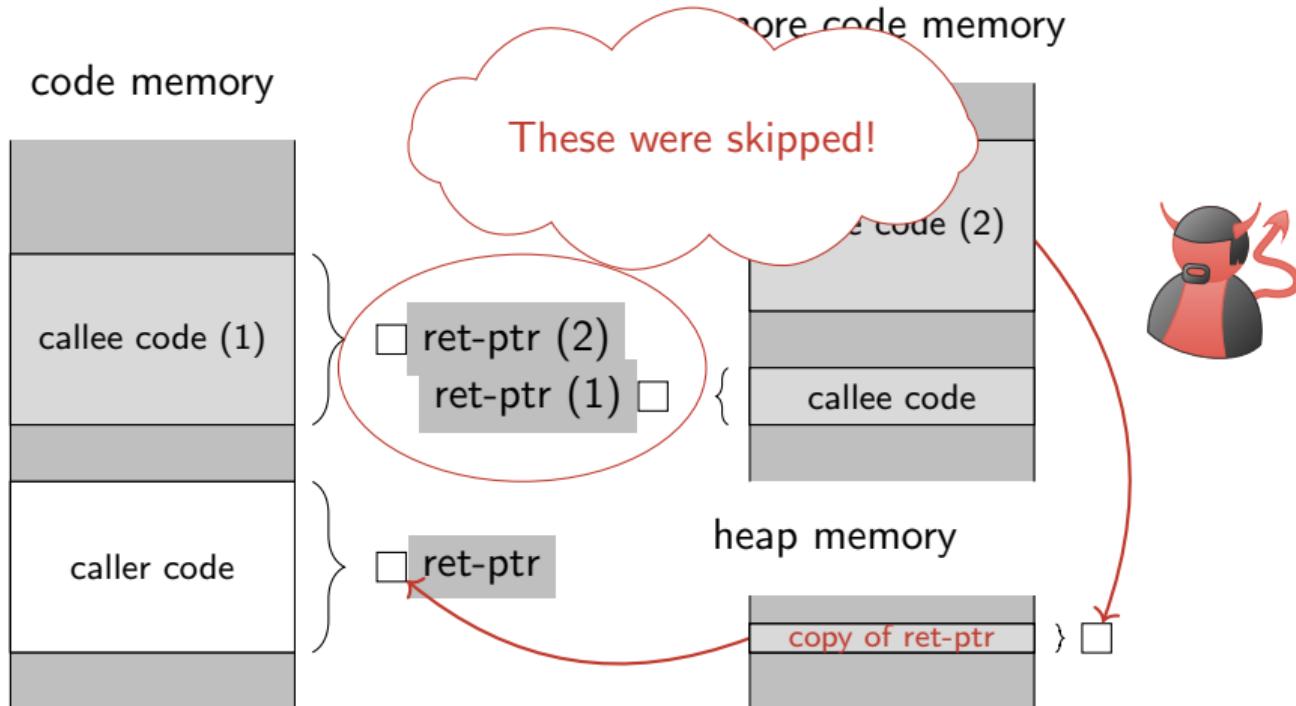
Attack 3 on naive stack and return capabilities



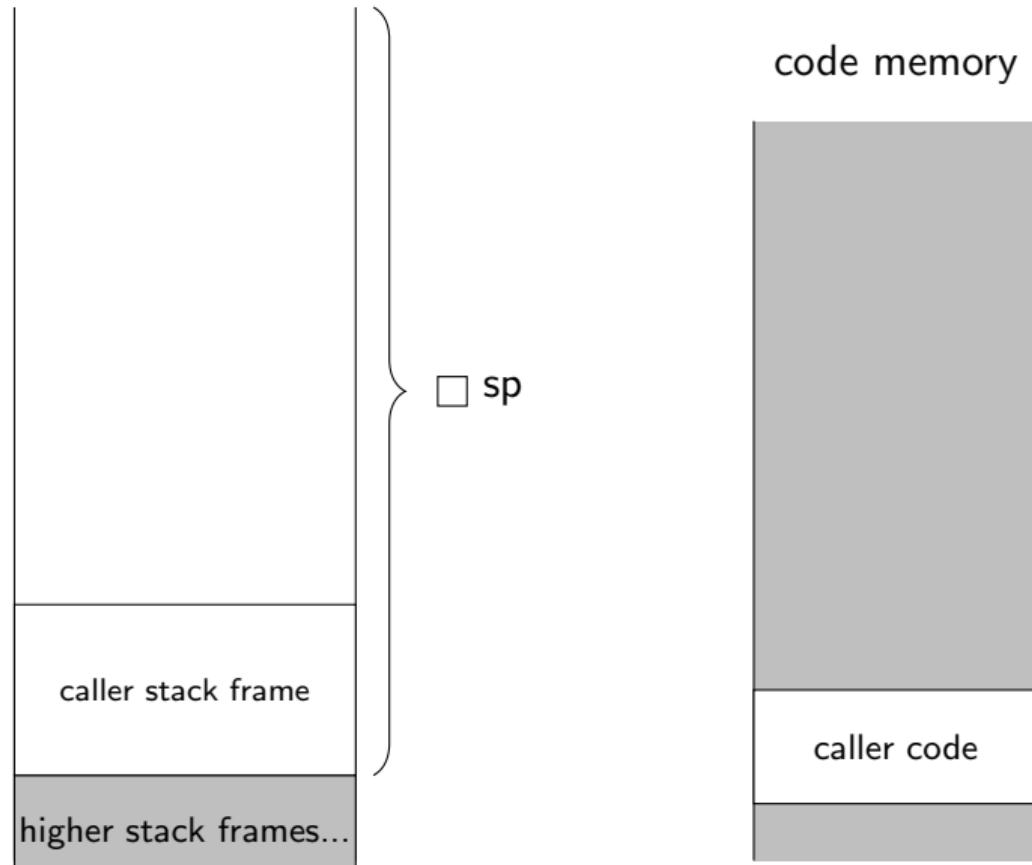
Attack 3 on naive stack and return capabilities



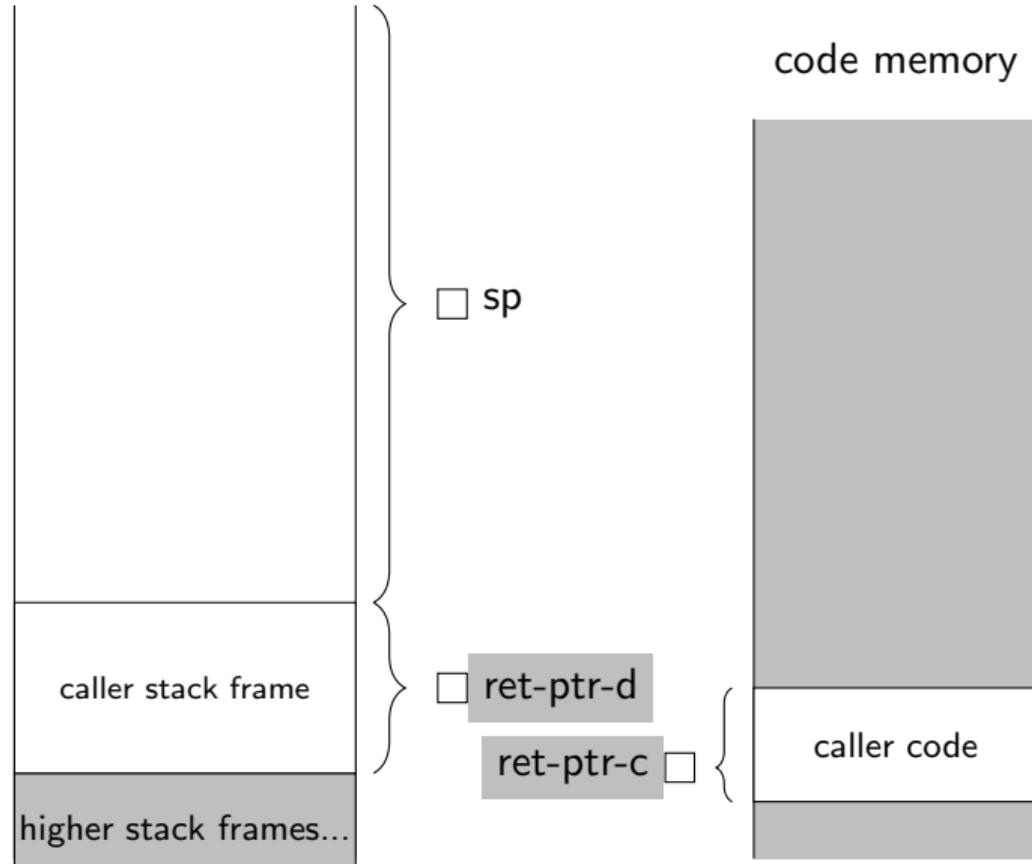
Attack 3 on naive stack and return capabilities



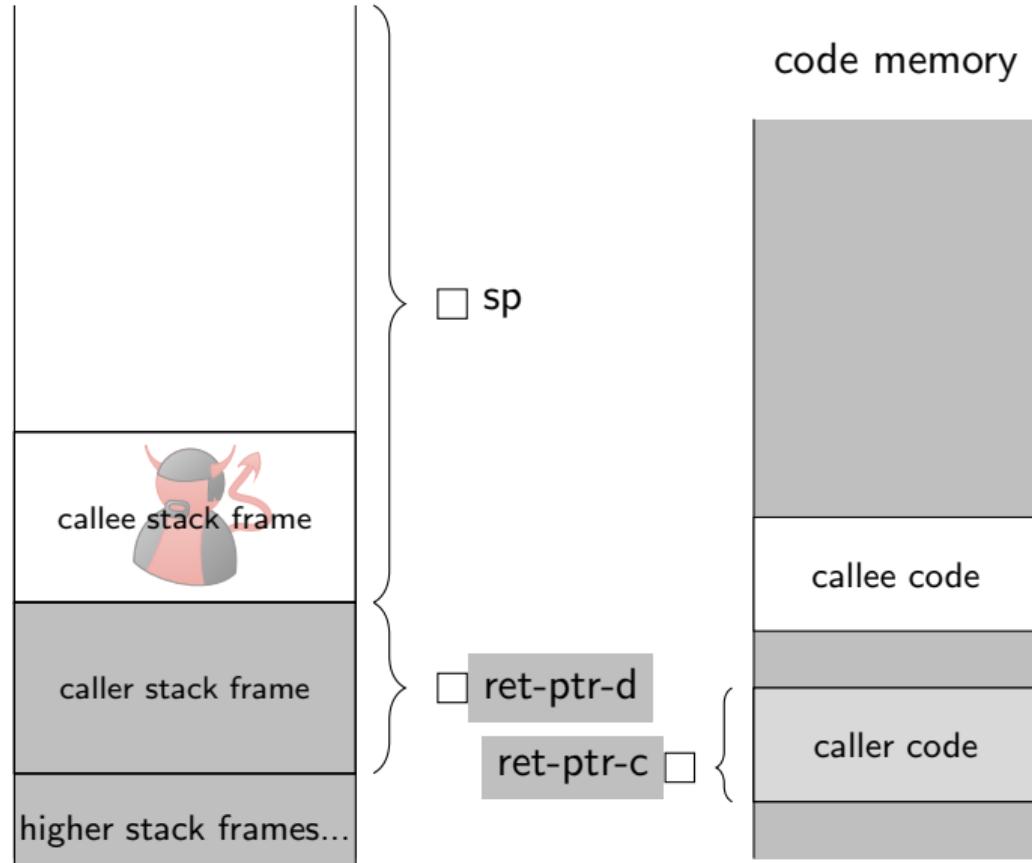
STKTOKENS prevent attack 3



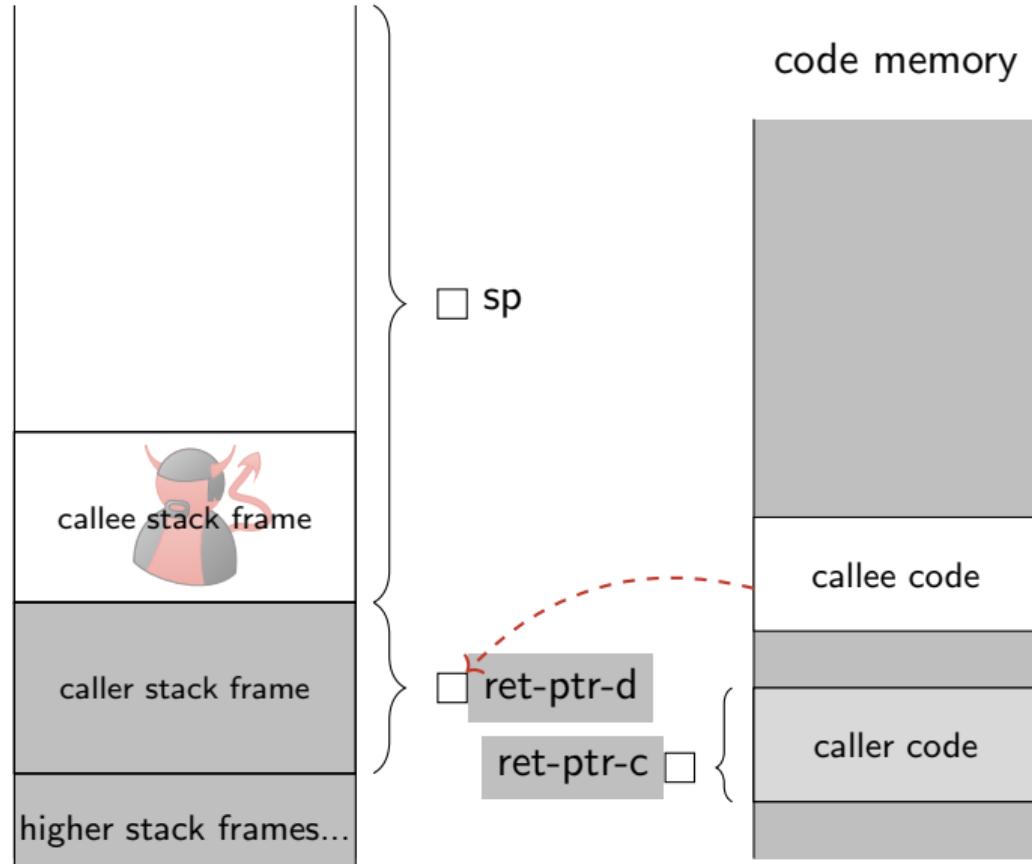
STKTOKENS prevent attack 3



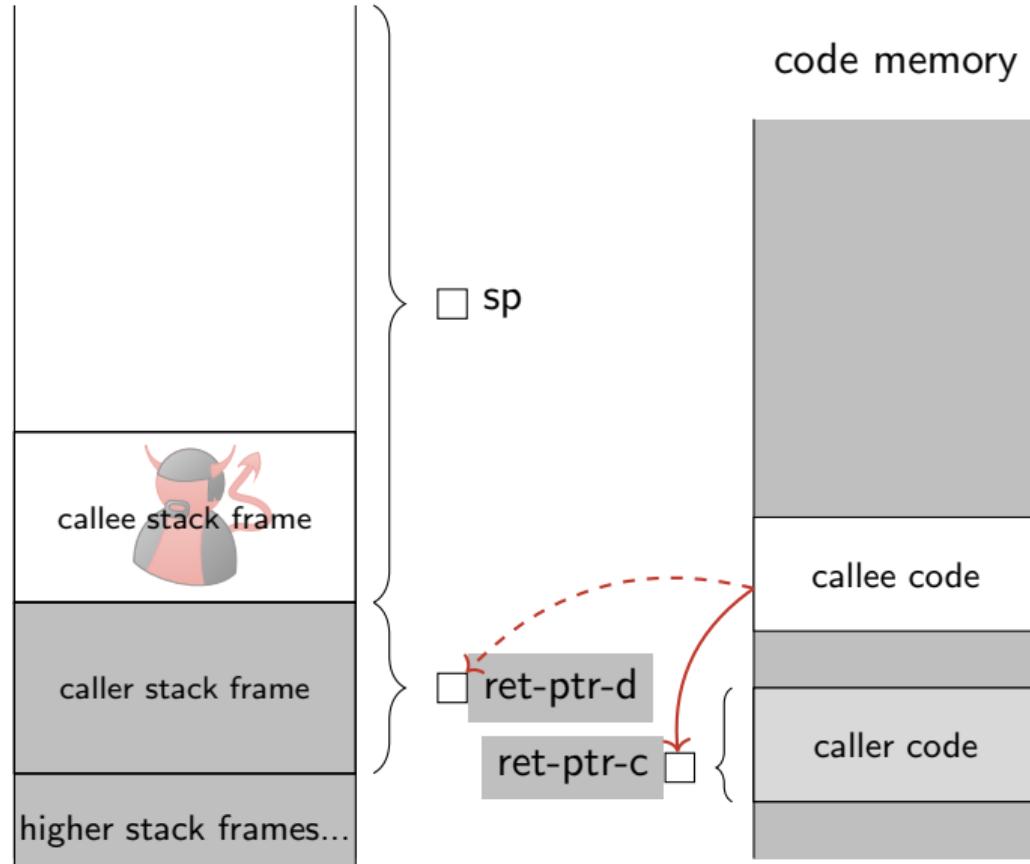
STKTOKENS prevent attack 3



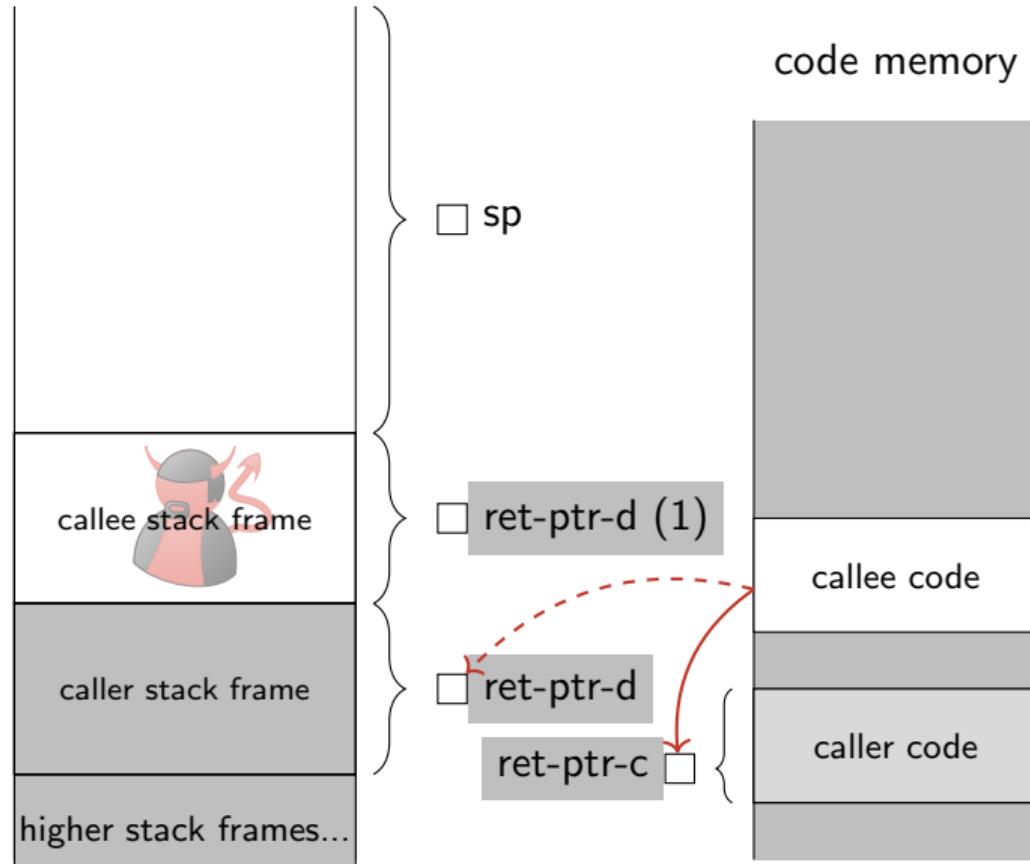
STKTOKENS prevent attack 3



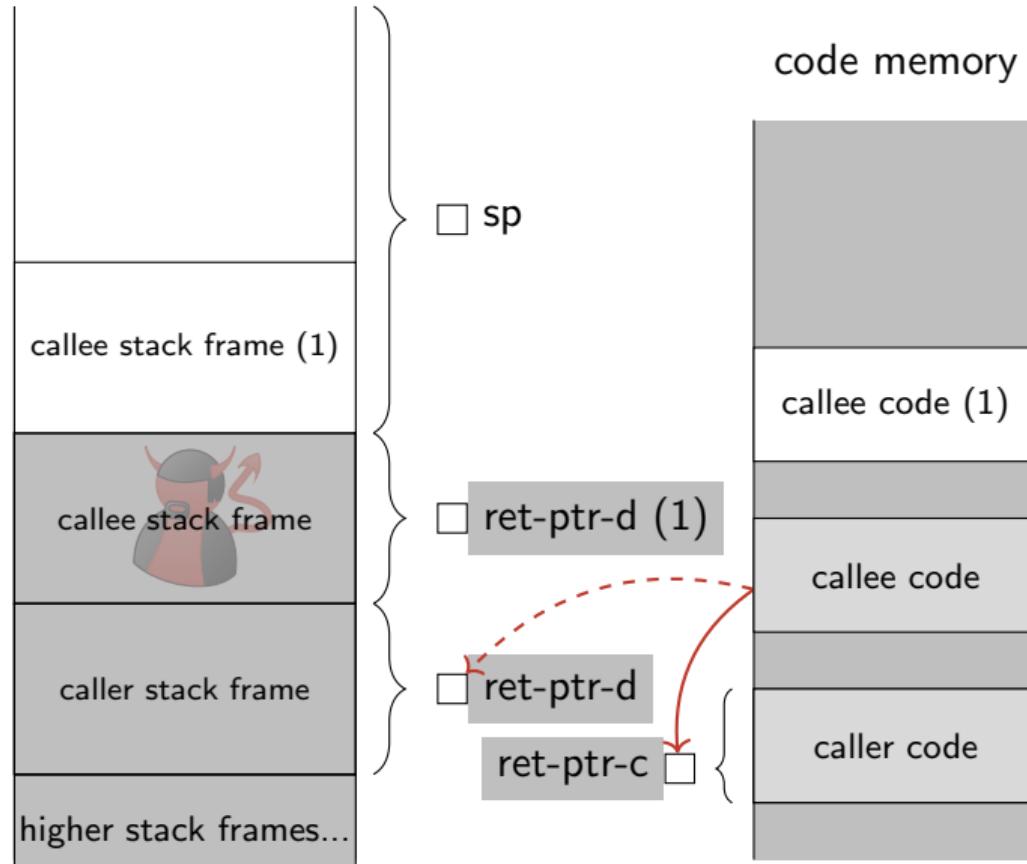
STKTOKENS prevent attack 3



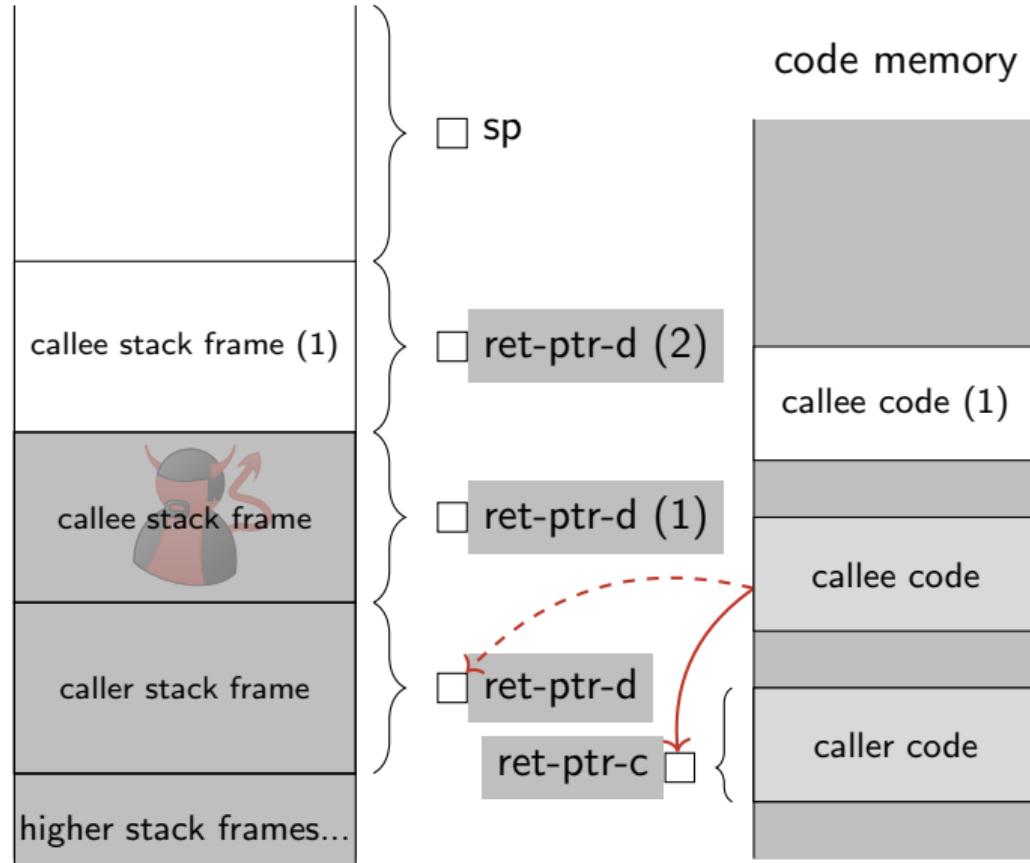
STKTOKENS prevent attack 3



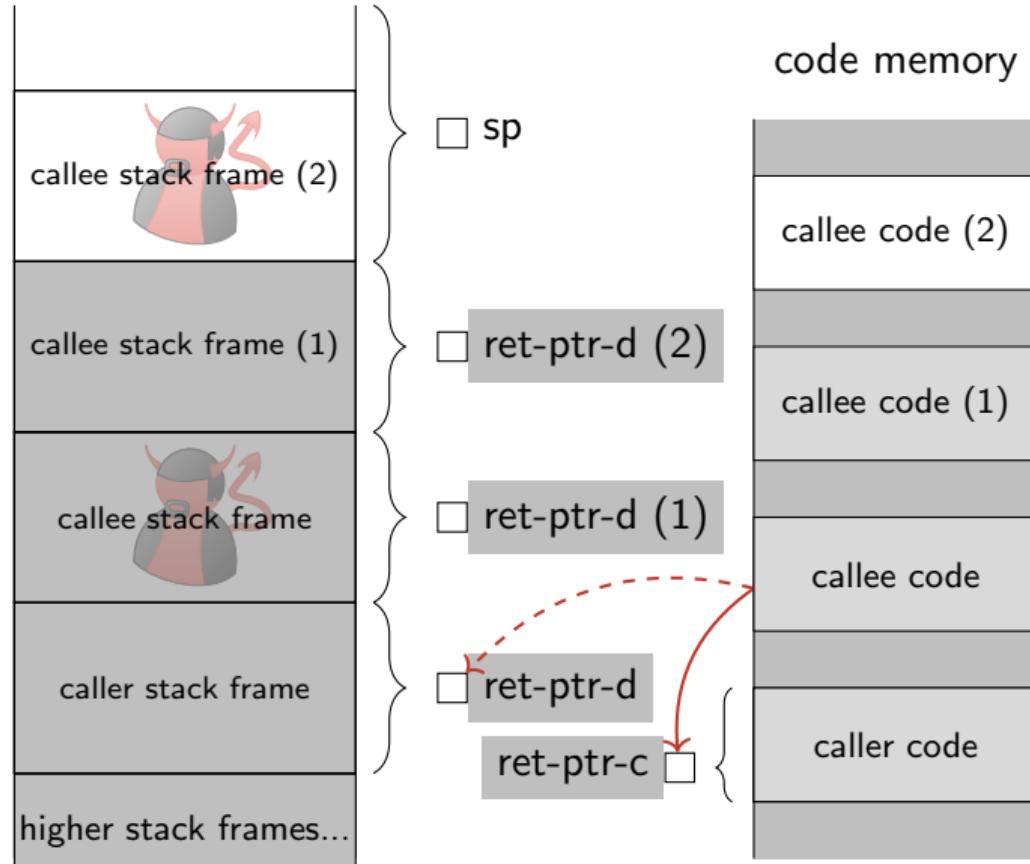
STKTOKENS prevent attack 3



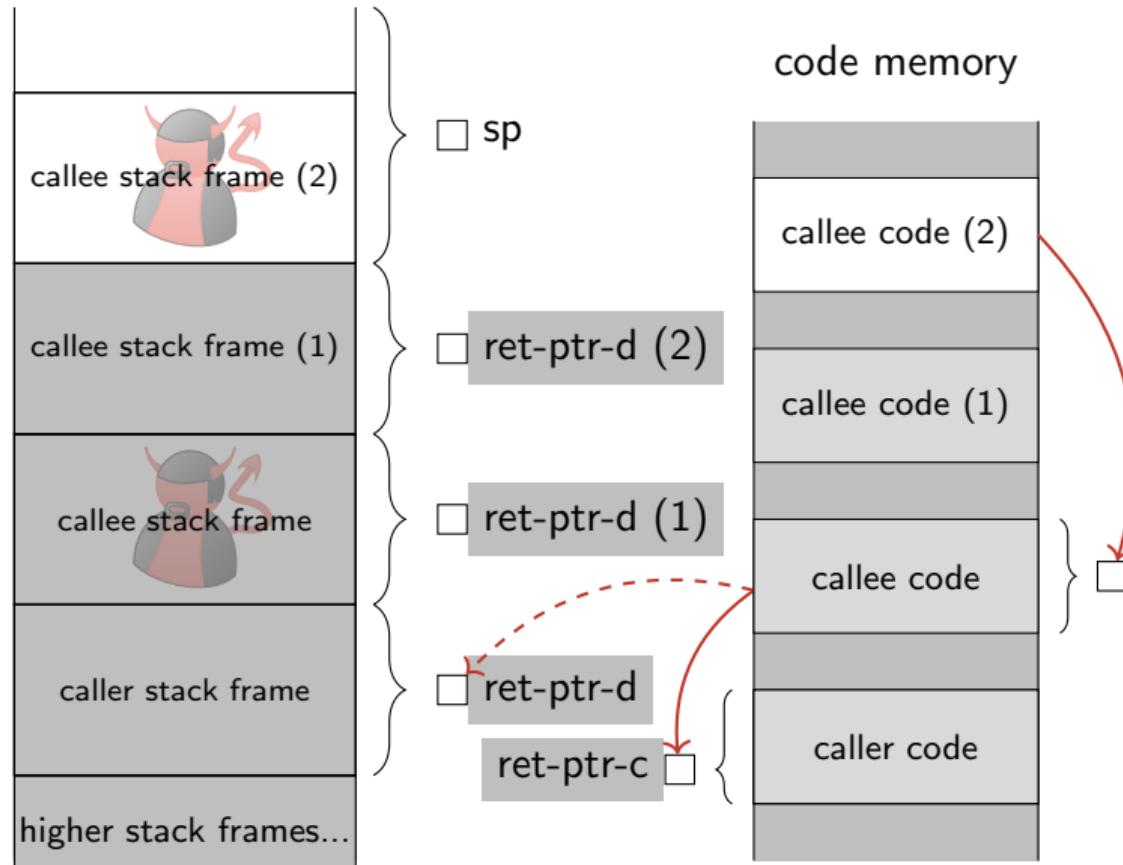
STKTOKENS prevent attack 3



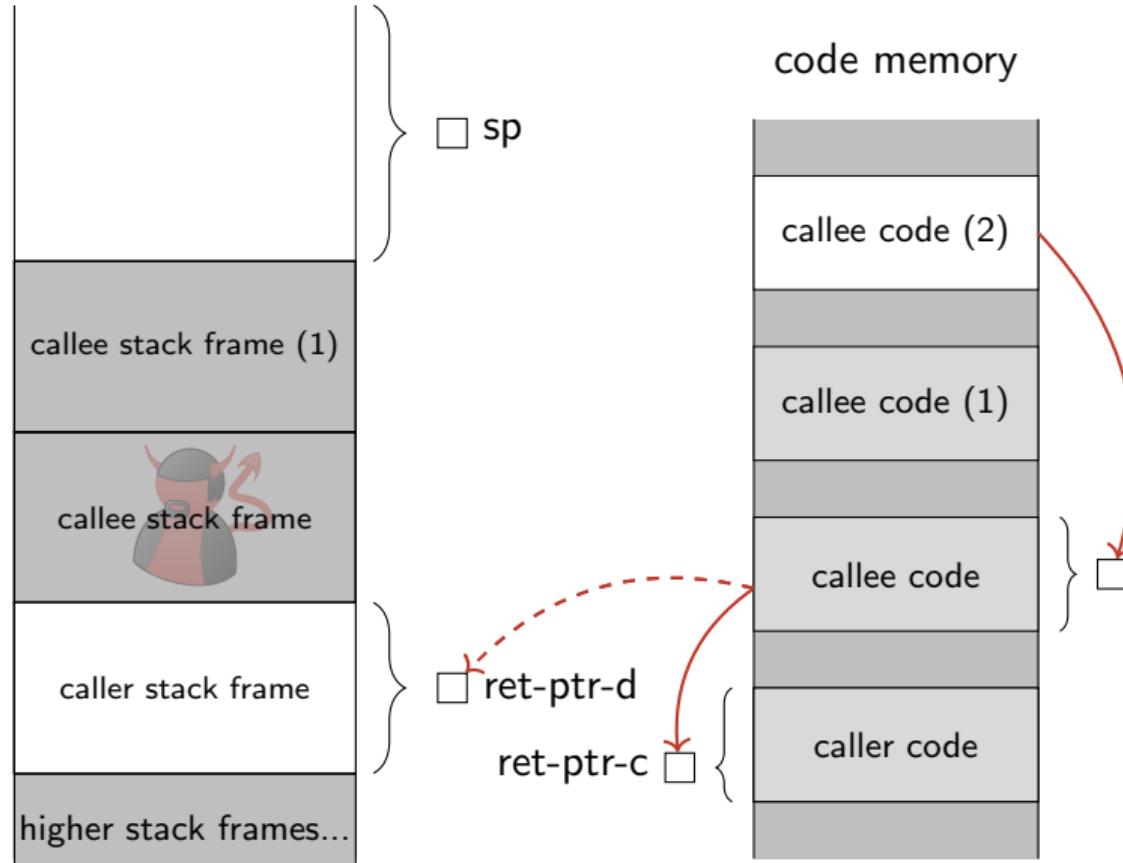
STKTOKENS prevent attack 3



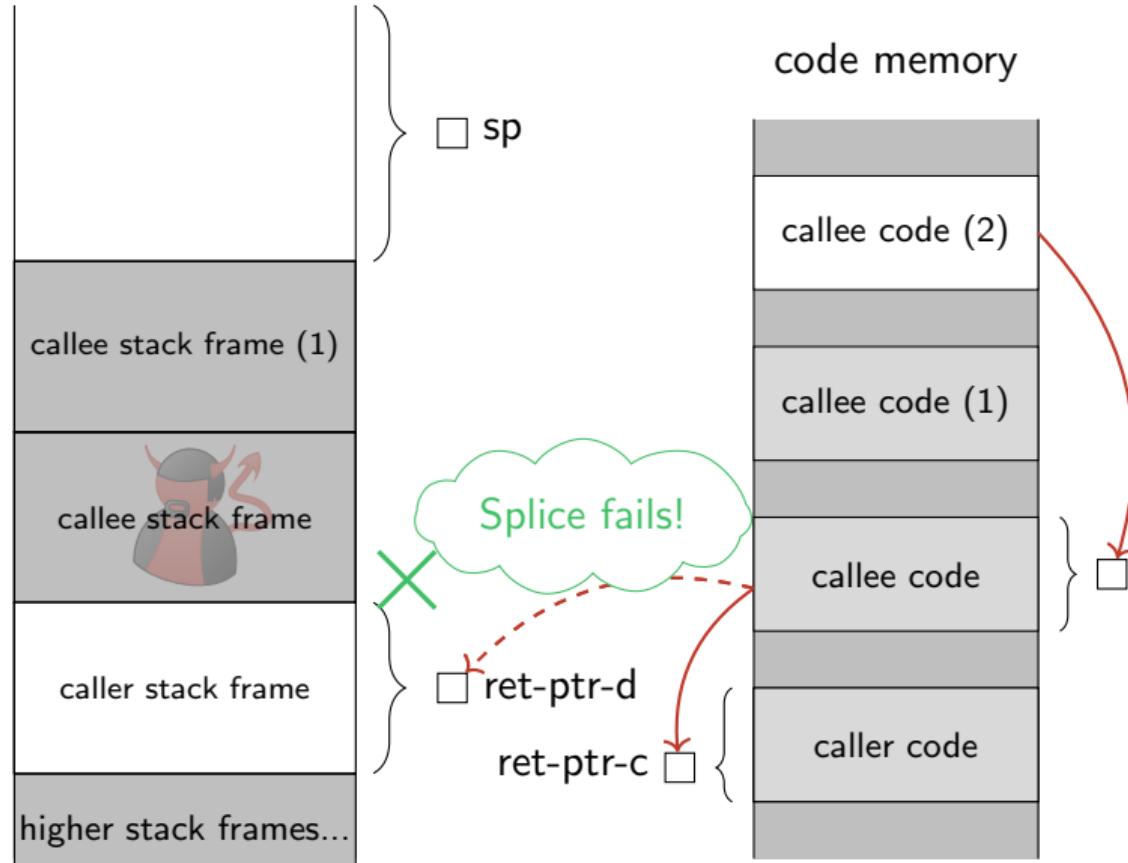
STKTOKENS prevent attack 3



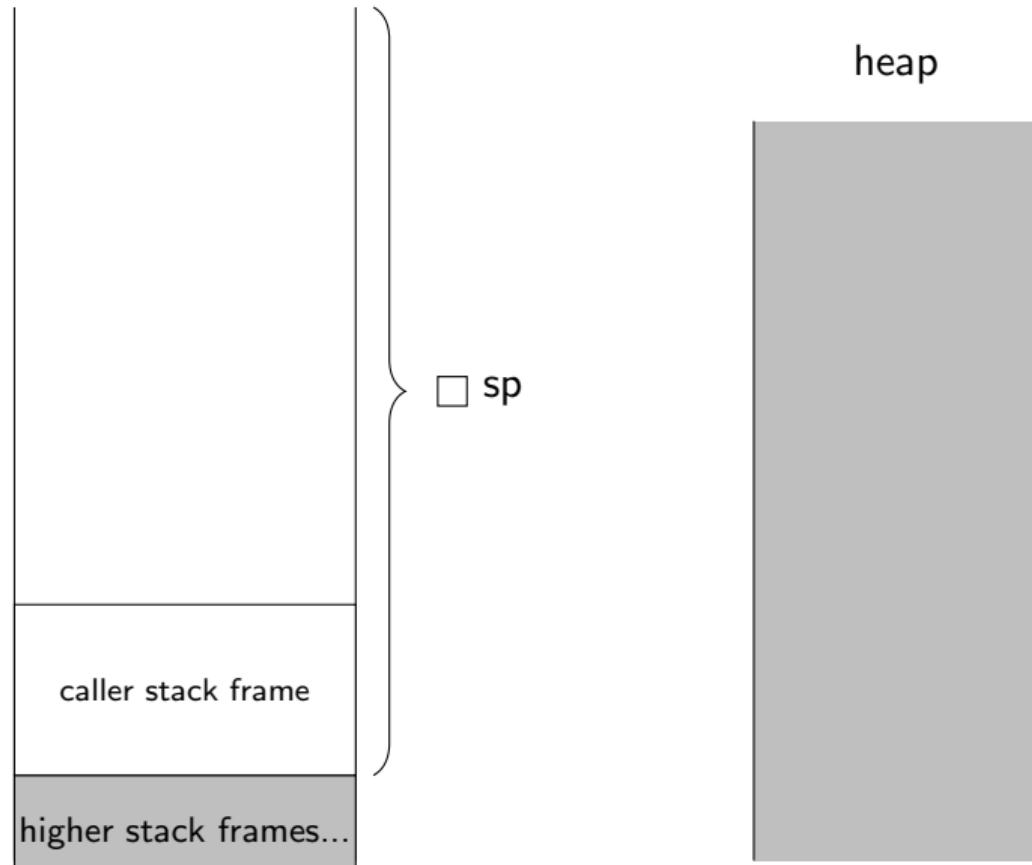
STKTOKENS prevent attack 3



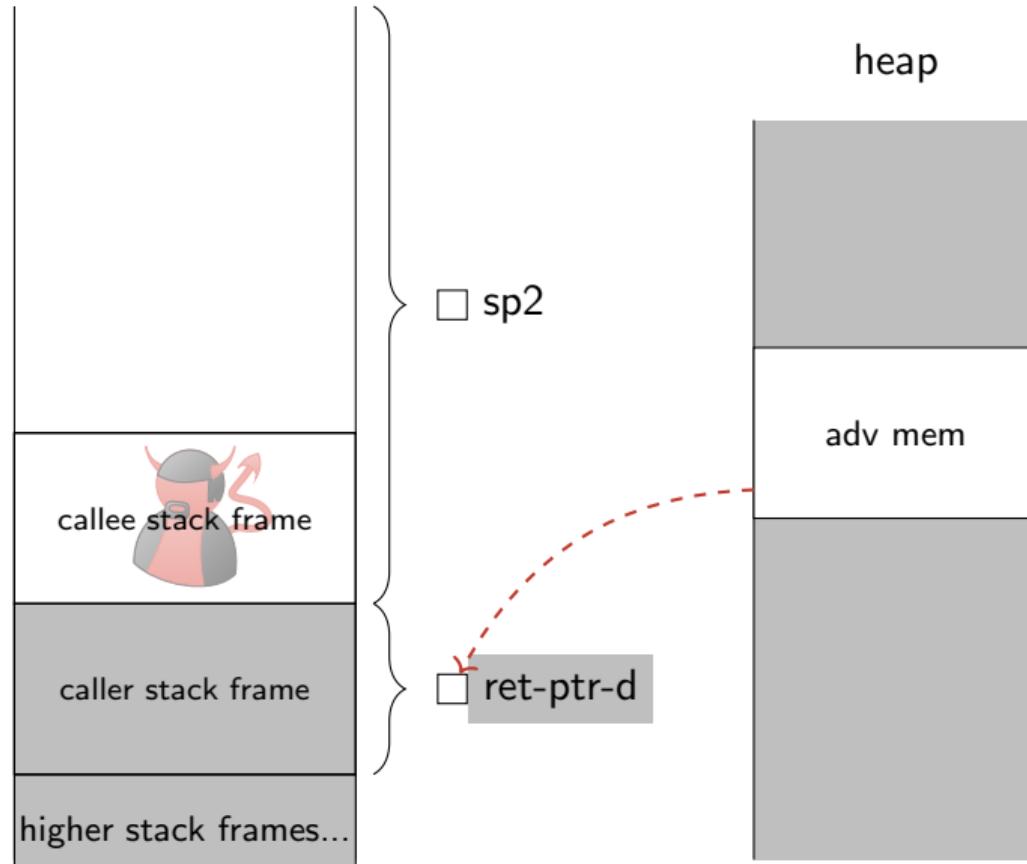
STKTOKENS prevent attack 3



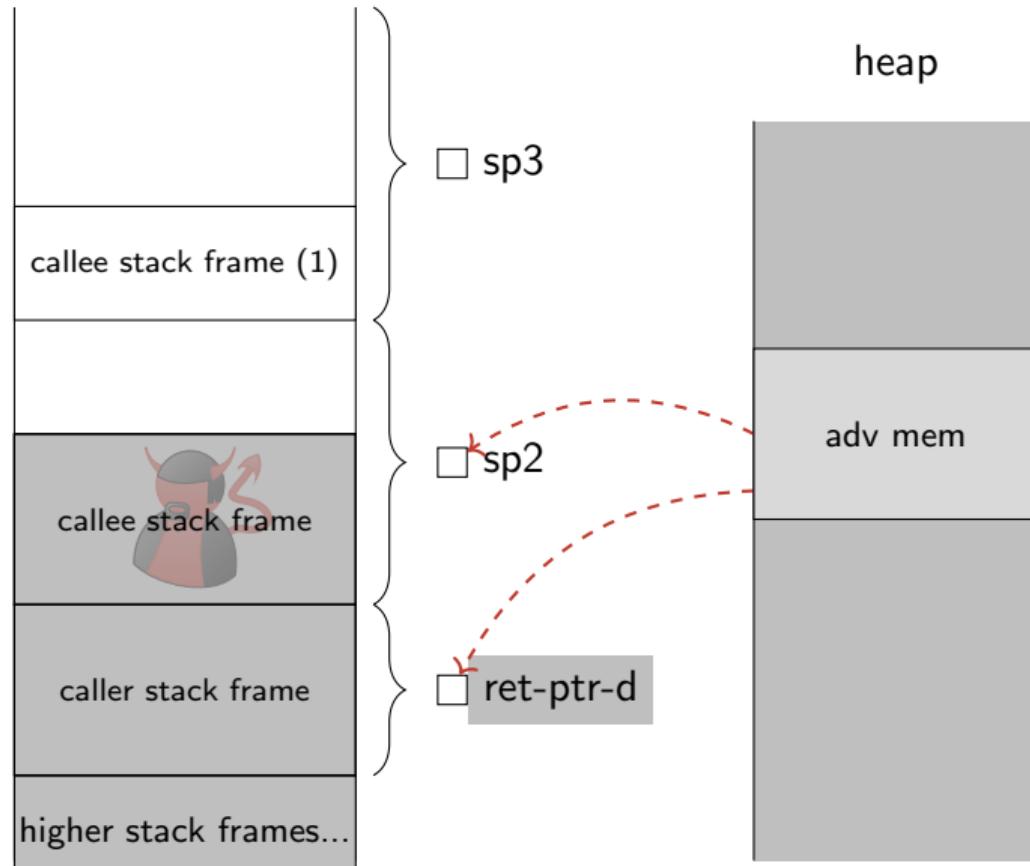
STKTOKENS enforces single stack



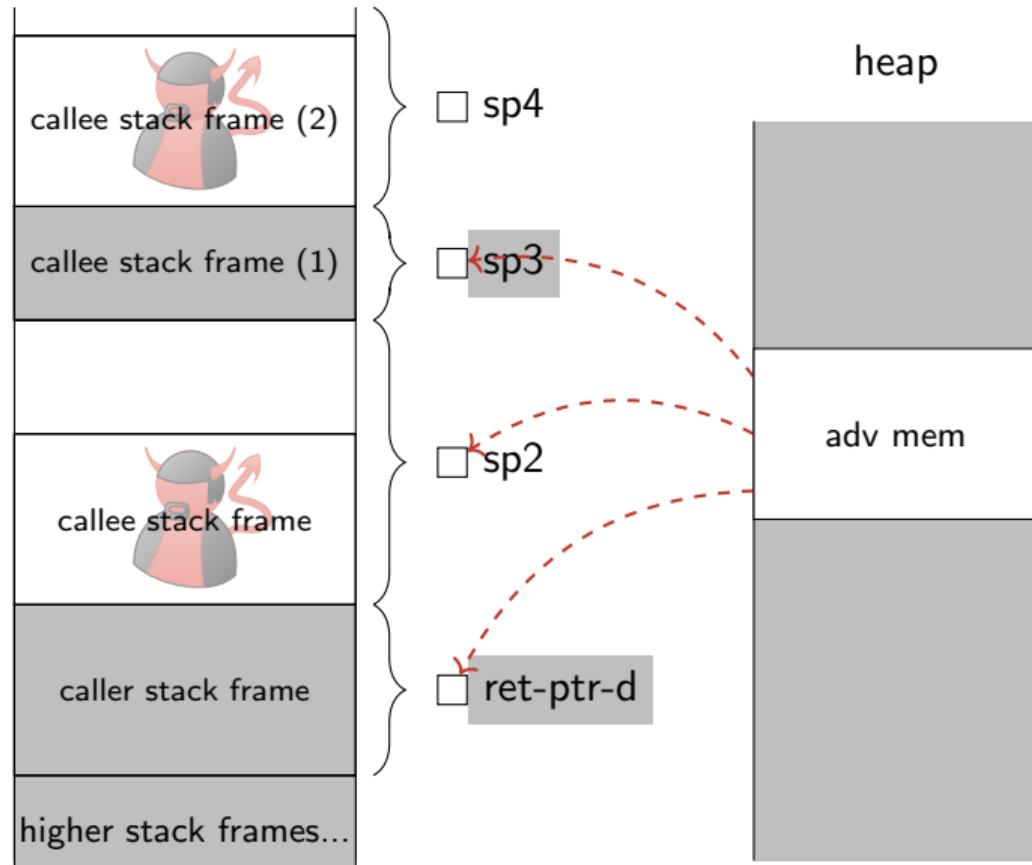
STKTOKENS enforces single stack



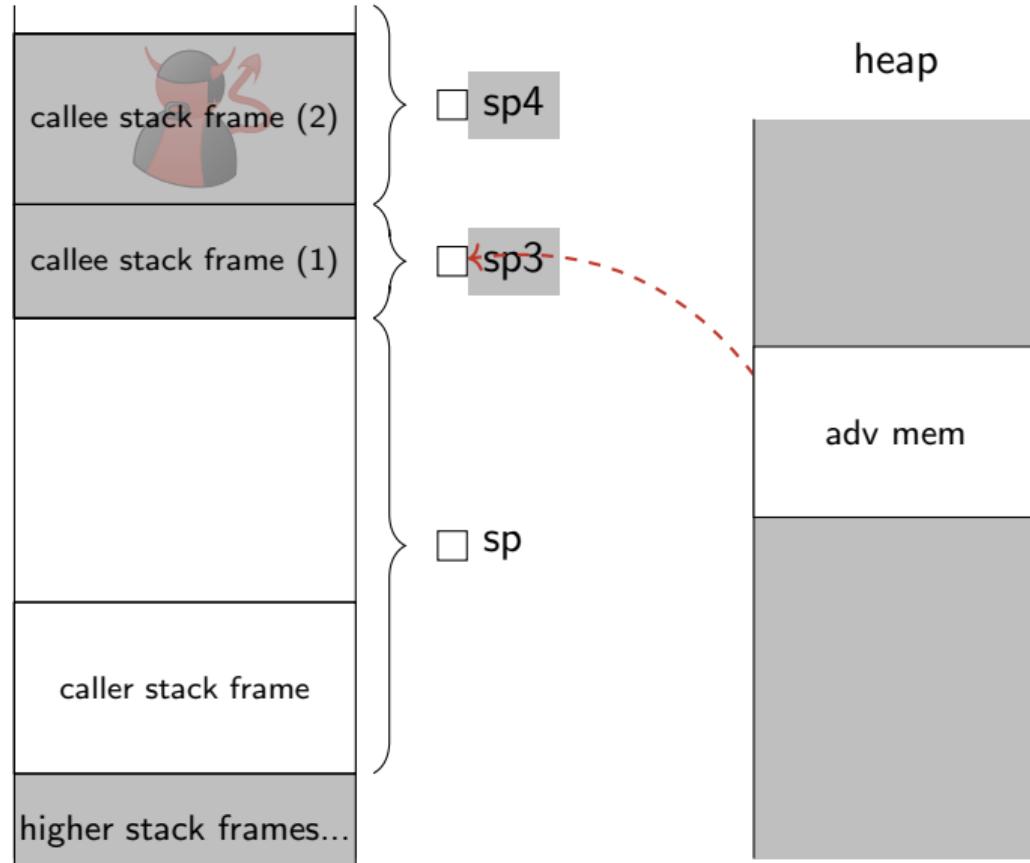
STKTOKENS enforces single stack



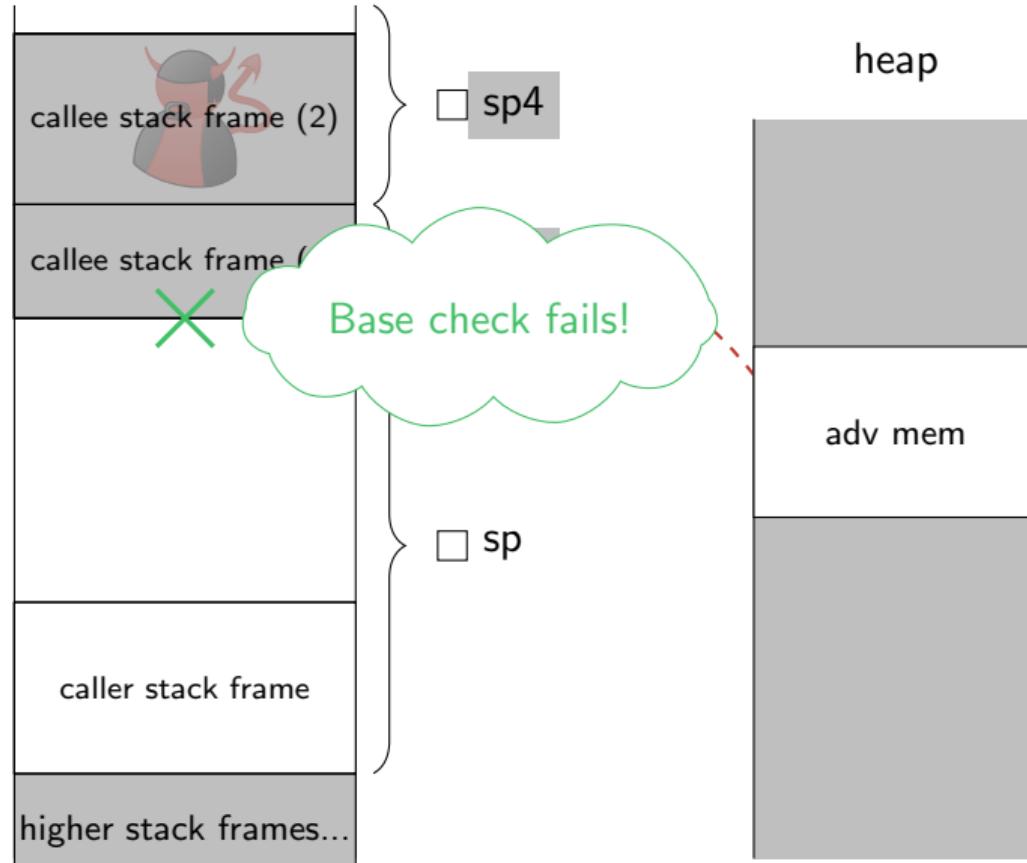
STKTOKENS enforces single stack



STKTOKENS enforces single stack



STKTOKENS enforces single stack



STKTOKENS summary

- ▶ Check the base address of the stack capability before and after calls.
- ▶ **Make sure that local stack frames are non-empty.**
- ▶ Create token and data return capability on call: split the stack capability in two to get a stack capability for your local stack frame and a stack capability for the unused part of the stack. The former is sealed and used for the data part of the return pair.
- ▶ Create code return capability on call: Seal the old program pointer.
- ▶ Reasonable use of seals: Return seals are only used to seal old program pointers, every return seal is only used for one call site, and they are not leaked.

STKTOKENS summary

- ▶ Check the base address of the stack capability before and after calls.
- ▶ Make sure that local stack frames are non-empty.
- ▶ Create token and data return capability on call: split the stack capability in two to get a stack capability for your local stack frame and a stack capability for the unused part of the stack. The former is sealed and used for the data part of the return pair.
- ▶ Create code return capability on call: Seal the old program pointer.
- ▶ Reasonable use of seals: Return seals are only used to seal old program pointers, every return seal is only used for one call site, and they are not leaked.

STKTOKENS FAQ

- ▶ *Do you support tail calls?*
 - ▶ Yes.
- ▶ *Do you support higher-order functions?*
 - ▶ Yes.

Fully-abstract overlay semantics

```
move  rtmp1 42          load  rtmp1 rtmp1
store rstk rtmp1        cca   rtmp1 -21
ccs   rstk -1          cseal rretd rtmp1
geta  rtmp1 rstk        move   rretc pc
ccs   rretc 5           xjmp  r1 r2
move  rtmp1 pc          cseal rretc rtmp1
ccs   rtmp1 -20         move   rtmp1 0
```

Linear Capability
Machine

Fully-abstract overlay semantics

```
move  rtmp1 42          load  rtmp1 rtmp1
store rstk rtmp1        cca   rtmp1 -21
cca   rstk -1          cseal rretd rtmp1
geta  rtmp1 rstk        move   rretc pc
cca   rretc 5           xjmp   r1 r2
move  rtmp1 pc          cseal rretc rtmp1
cca   rtmp1 -20         move   rtmp1 0
```

Overlay Semantics

```
move  rtmp1 42          load  rtmp1 rtmp1
store rstk rtmp1        cca   rtmp1 -21
cca   rstk -1          cseal rretd rtmp1
geta  rtmp1 rstk        move   rretc pc
cca   rretc 5           xjmp   r1 r2
move  rtmp1 pc          cseal rretc rtmp1
cca   rtmp1 -20         move   rtmp1 0
```

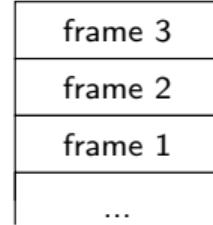
Linear Capability
Machine

Fully-abstract overlay semantics

```
move rtmp1 42
store rstk rtmp1
cca rstk -1
geta rtmp1 rstk
cca rretc 5
move rtmp1 pc
cca rtmp1 -20
```

```
load rtmp1 rtmp1
cca rtmp1 -21
cseal rretd rtmp1
move rretc pc
xjmp r1 r2
cseal rretc rtmp1
move rtmp1 0
```

Builtin call stack



Overlay Semantics

```
move rtmp1 42
store rstk rtmp1
cca rstk -1
geta rtmp1 rstk
cca rretc 5
move rtmp1 pc
cca rtmp1 -20
```

```
load rtmp1 rtmp1
cca rtmp1 -21
cseal rretd rtmp1
move rretc pc
xjmp r1 r2
cseal rretc rtmp1
move rtmp1 0
```

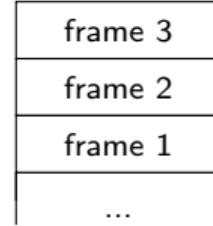
Linear Capability
Machine

Fully-abstract overlay semantics

```
move rtmp1 42
store rstk rtmp1
cca rstk -1
call
geta rtmp1 rstk
cca rretc 5
move rtmp1 pc
cca rtmp1 -20
```

```
load rtmp1 rtmp1
cca rtmp1 -21
cseal rretd rtmp1
move rretc pc
return
cseal rretc rtmp1
move rtmp1 0
```

Builtin call stack



Overlay Semantics

```
move rtmp1 42
store rstk rtmp1
cca rstk -1
geta rtmp1 rstk
cca rretc 5
move rtmp1 pc
cca rtmp1 -20
```

```
load rtmp1 rtmp1
cca rtmp1 -21
cseal rretd rtmp1
move rretc pc
xjmp r1 r2
cseal rretc rtmp1
move rtmp1 0
```

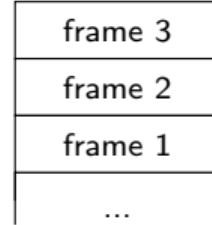
Linear Capability
Machine

Fully-abstract overlay semantics

```
move rtmp1 42
store rstk rtmp1
cca rstk 1
geta rtmp1 rstk
cca rretc 5
move rtmp1 pc
cca rtmp1 -20
```

```
load rtmp1 rtmp1
cca rtmp1 -21
cseal rretd rtmp1
move rretc pc
x
return
cseal rretc rtmp1
move rtmp1 0
```

Builtin call stack



Overlay Semantics

```
move rtmp1 42
store rstk rtmp1
cca rstk -1
geta rtmp1 rstk
cca rretc 5
move rtmp1 pc
cca rtmp1 -20
```

```
load rtmp1 rtmp1
cca rtmp1 -21
cseal rretd rtmp1
move rretc pc
xjmp r1 r2
cseal rretc rtmp1
move rtmp1 0
```

Linear Capability
Machine

Fully-abstract overlay semantics

```
move rtmp1 42  
store rstk rtmp1  
cca rstk 1  
geta rtmp1 rstk  
cca rretc 5  
move rtmp1 pc  
cca rtmp1 -20
```

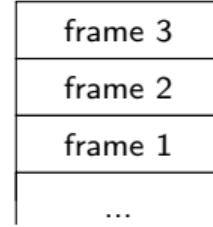
call

```
move rtmp1 42  
store rstk rtmp1  
cca rstk -1  
geta rtmp1 rstk  
cca rretc 5  
move rtmp1 pc  
cca rtmp1 -20
```

return

id

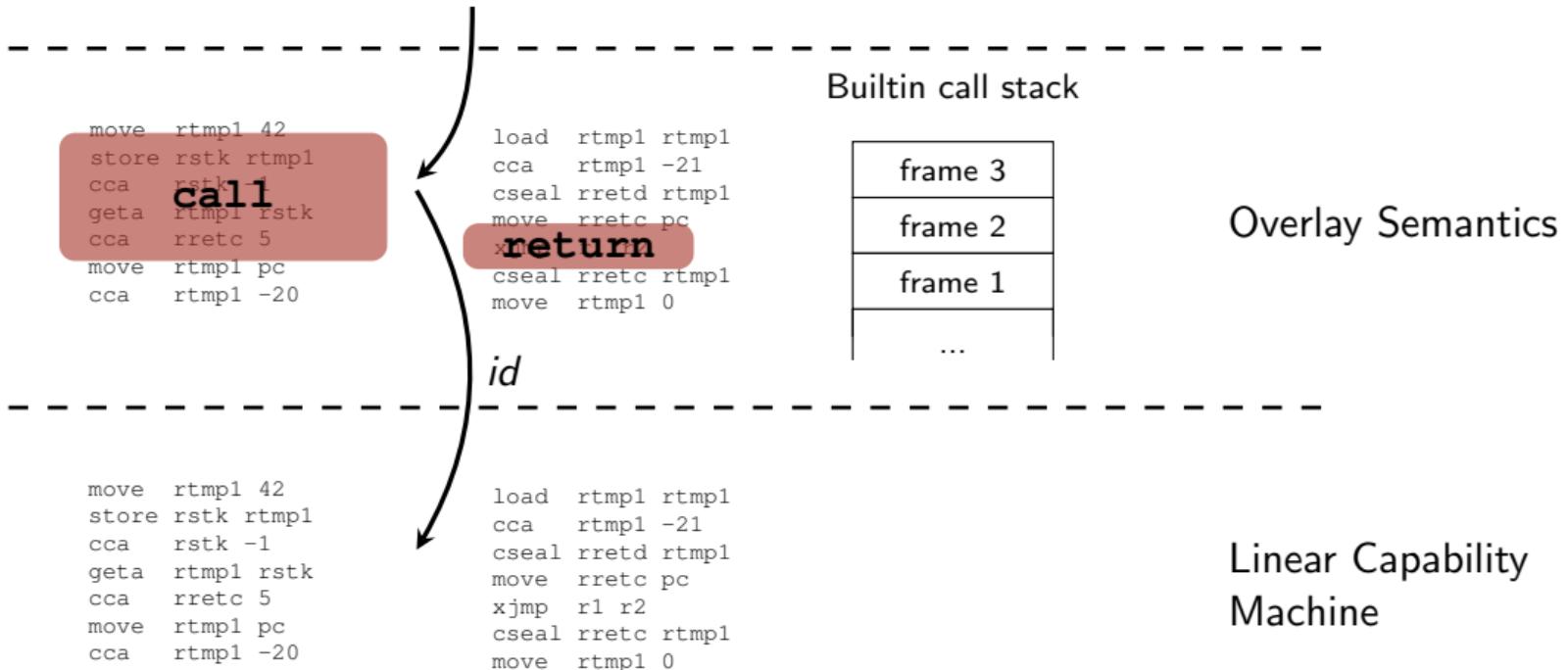
Builtin call stack



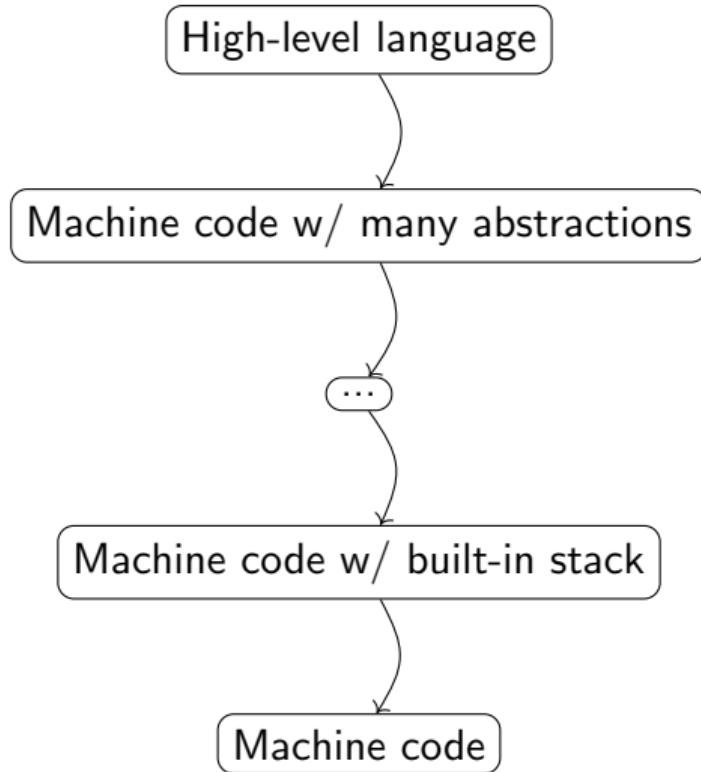
Overlay Semantics

Linear Capability Machine

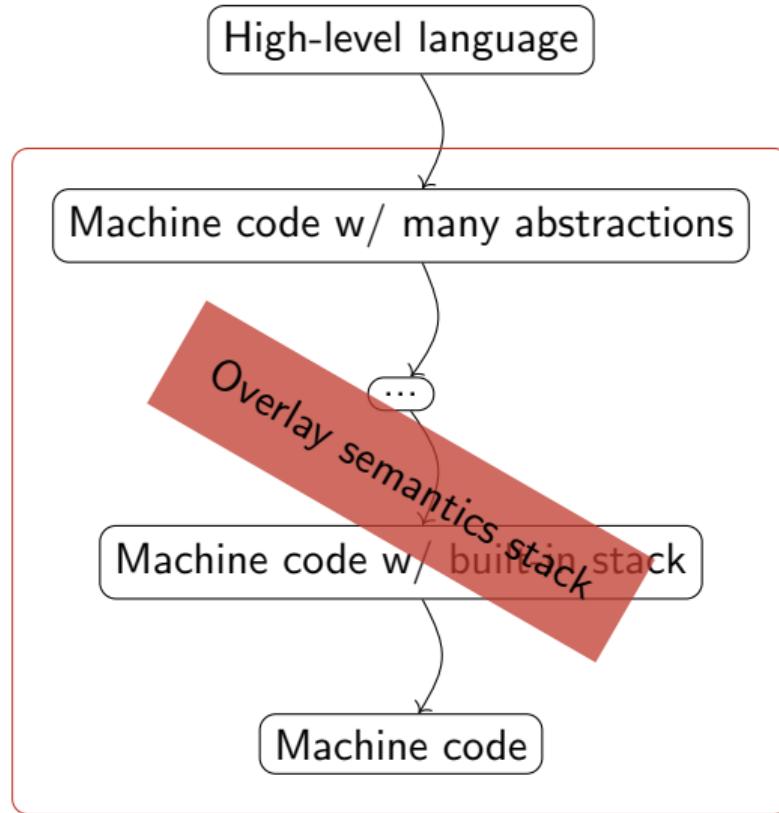
Fully-abstract overlay semantics



Proof sketch for a realistic secure compiler



Proof sketch for a realistic secure compiler



Full-abstraction proof sketch

Contextual equivalence preservation



$$\begin{array}{c} comp_1 \approx_{\text{ctx}} comp_2 \\ \mathcal{C}[comp_1] \Downarrow^{gc} \Rightarrow \mathcal{C}[comp_2] \Downarrow^{gc} \\ \mathcal{C} \sqsupseteq \mathcal{C} \quad \uparrow \qquad \downarrow \quad \mathcal{C} \sqsupseteq \mathcal{C} \\ comp_1 \approx comp_1 \qquad \qquad \qquad comp_2 \approx comp_2 \\ \mathcal{C}[comp_1] \Downarrow \stackrel{?}{\Rightarrow} \mathcal{C}[comp_2] \Downarrow \\ comp_1 \approx_{\text{ctx}} comp_2 \end{array}$$

Desired properties of LSE and WBCF definition

1. *intuitive*
2. *useful for reasoning*: we should be able to use WBCF and LSE when reasoning about correctness and security of programs using STKTOKENS.
3. *reusable in secure compiler chains*: for compilers using STKTOKENS, one should be able to rely on WBCF and LSE when proving correctness and security of other compiler passes and then compose such results with ours to obtain results about the full compiler.
4. *arguably "complete"*: the formalization should arguably capture the entire meaning of WBCF and LSE and should arguably be applicable to any reasonable program.
5. *potentially scalable*: although dynamic code generation and multi-threading are currently out of scope, the formalization should, at least potentially, extend to such settings.

Paper overview

- ▶ LCM: A formalization of a simple CHERI-like capability machine with linear capabilities
- ▶ STKTOKENS, a new calling convention that provably guarantees LSE and WBCF on LCM
- ▶ A better way to formalize these guarantees based on a novel technique called *fully-abstract overlay semantics*
- ▶ Proof of LSE and WBCF which includes
 - ▶ oLCM: an overlay semantics for LCM with built-in LSE and WBCF
 - ▶ proving full-abstraction for the embedding of oLCM into LCM by
 - ▶ defining and using a cross-language, step-indexed, Kripke logical relation with recursive worlds

Thank you!