

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

Lau Skorstengaard<sup>1</sup>    Dominique Devriese<sup>2</sup>    Lars Birkedal<sup>1</sup>

<sup>1</sup>Aarhus University

<sup>2</sup>Vrije Universiteit Brussel

Cambridge, March, 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
```

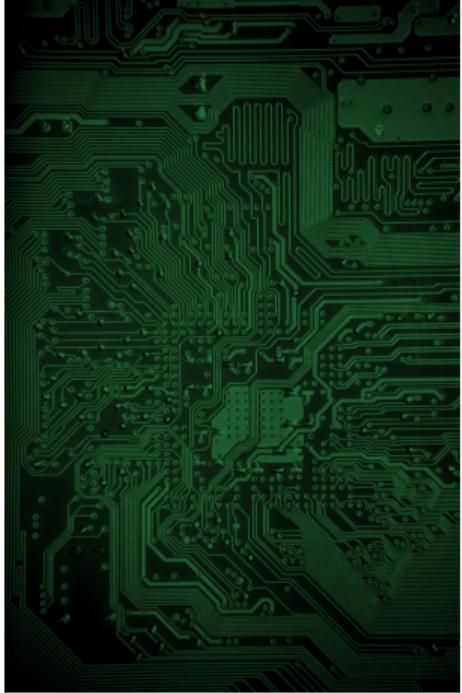


# Abstractions all the way down

```
#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_register %rbp
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
.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
```

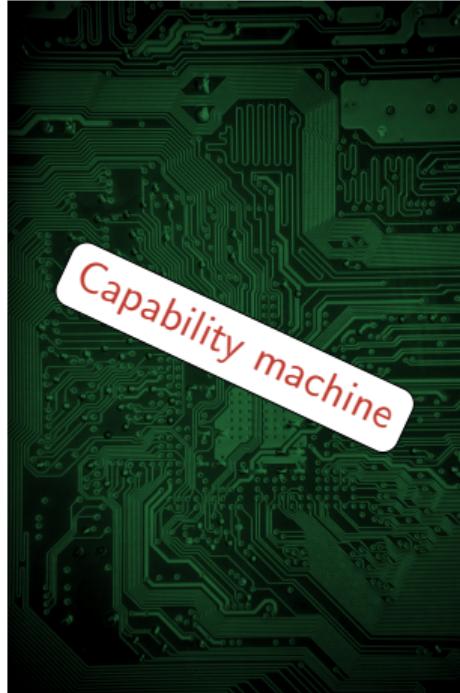


# Abstractions all the way down

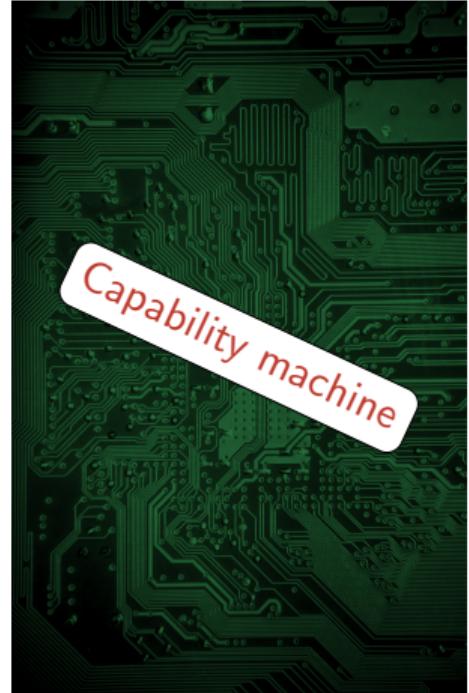
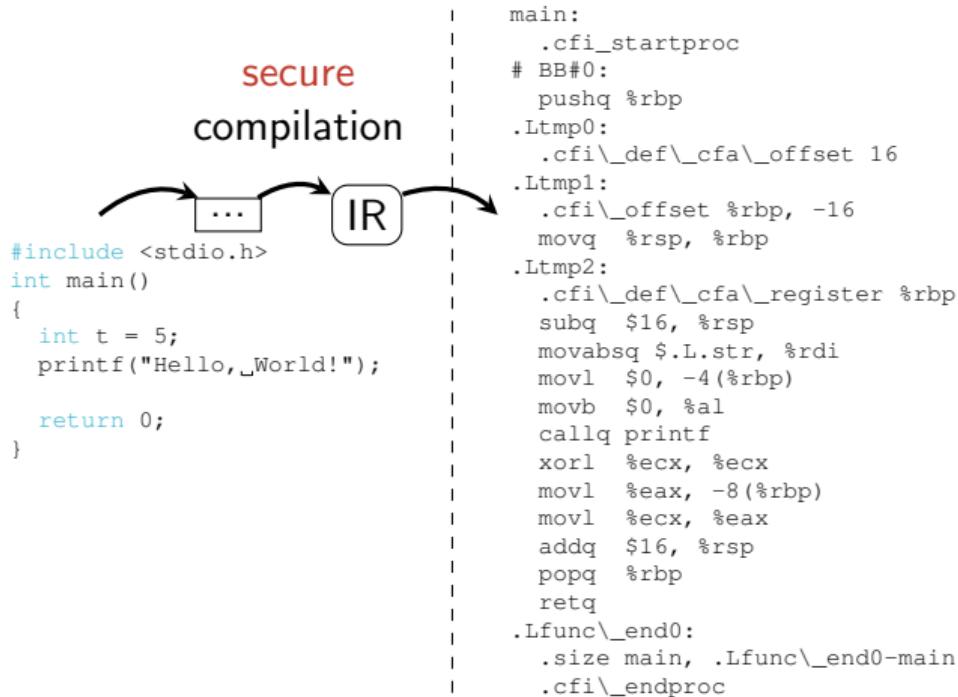
secure  
compilation

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

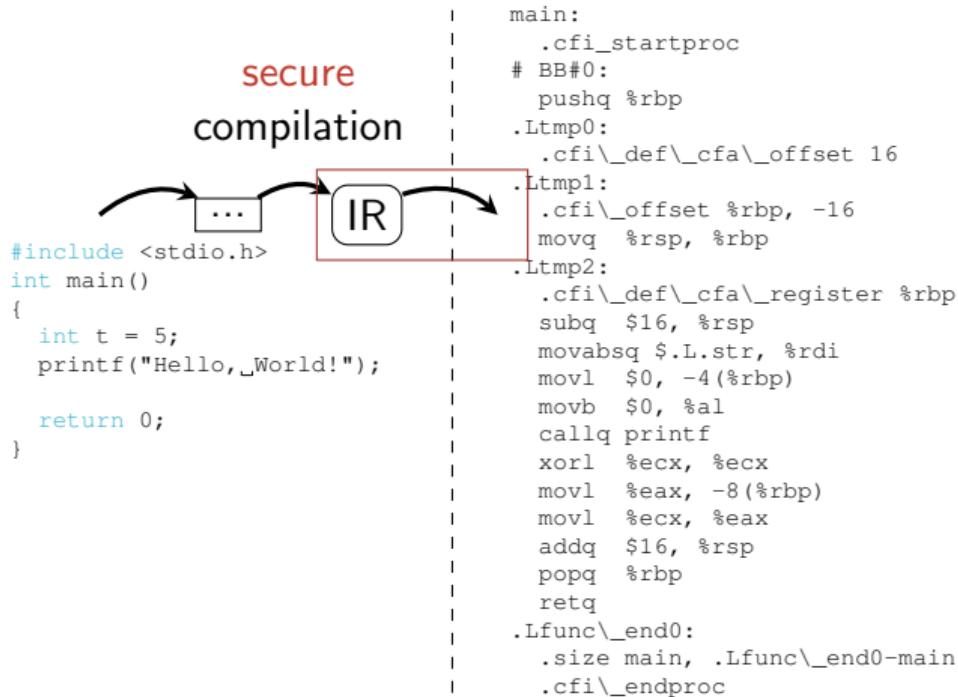
```
main:
    .cfi_startproc
# BB#0:
    pushq %rbp
.Ltmp0:
    .cfi_offset %rbp, -16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
.Ltmp2:
    .cfi_offset %rbp, -16
    subq $16, %rsp
    movabsq $.L.str, %rdi
    movl $0, -4(%rbp)
    movb $0, %al
    callq printf
    xorl %ecx, %ecx
    movl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

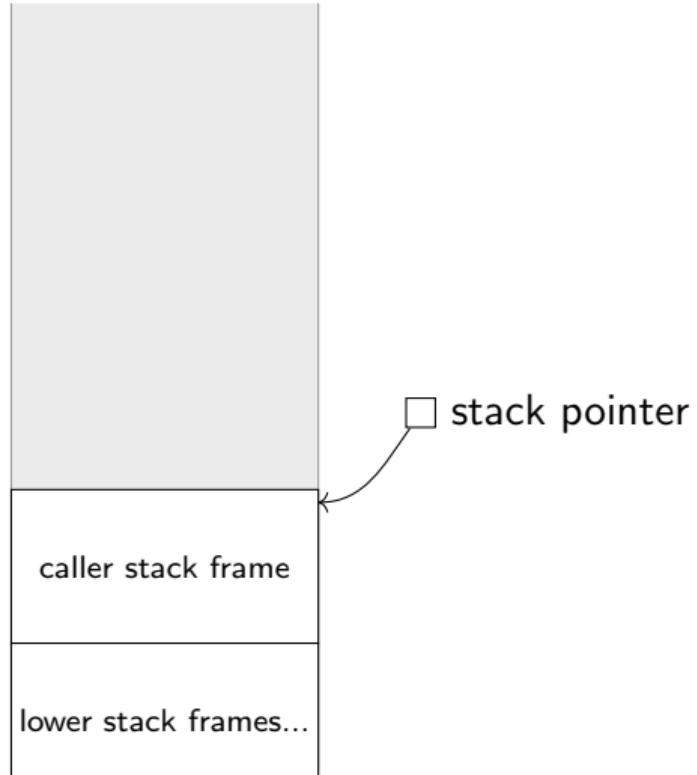
## Well-bracketed control flow and local state encapsulation

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

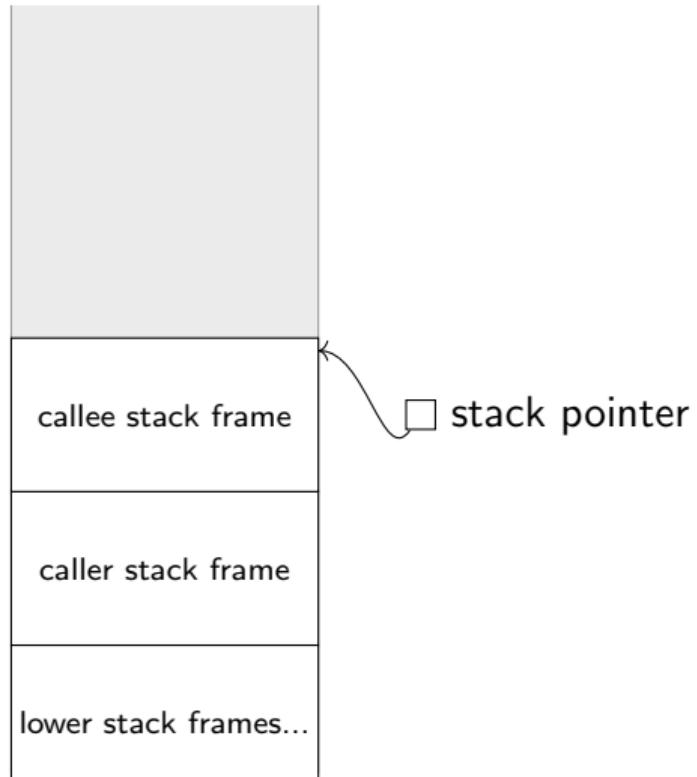


Well-bracketed control flow (WBCF)

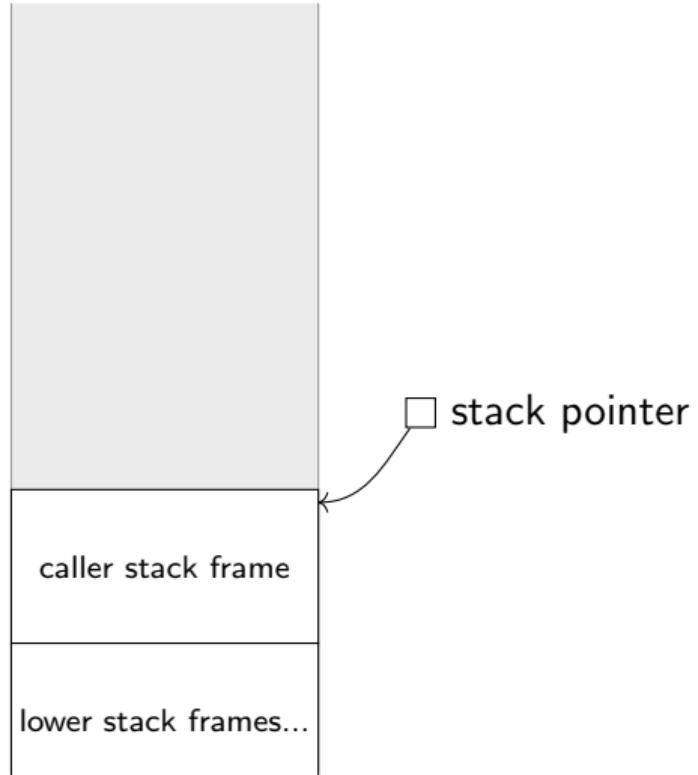
## Traditional stack pointers



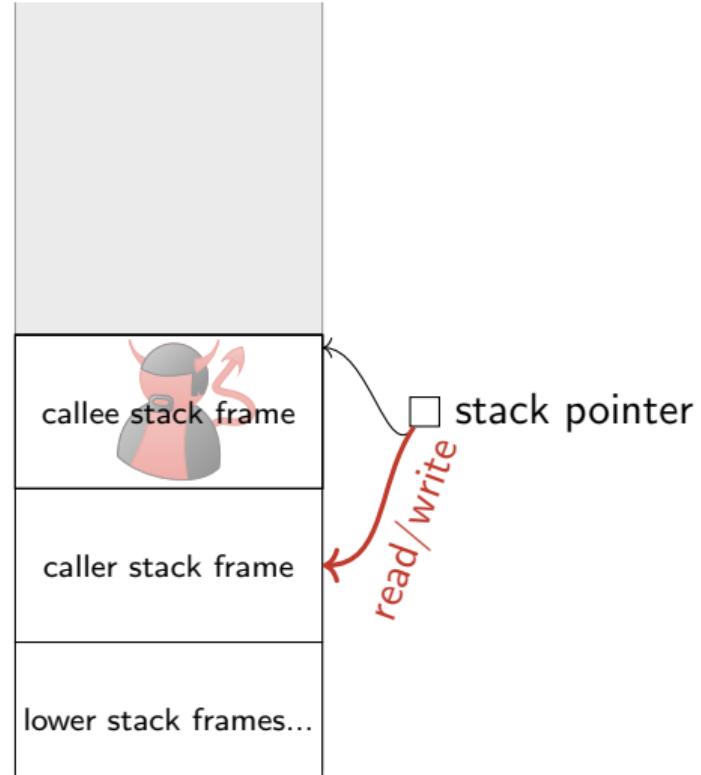
## Traditional stack pointers



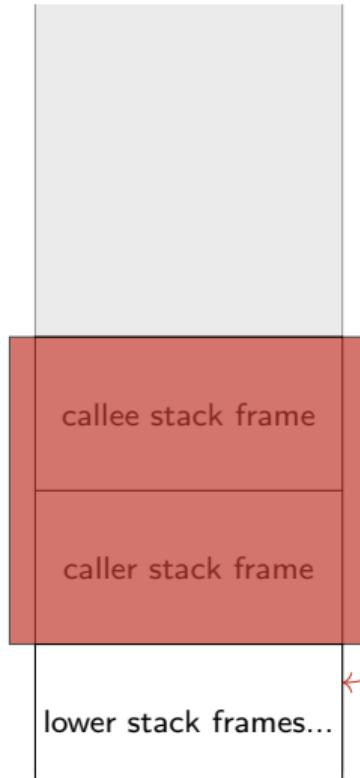
## Traditional stack pointers



## Traditional stack pointers



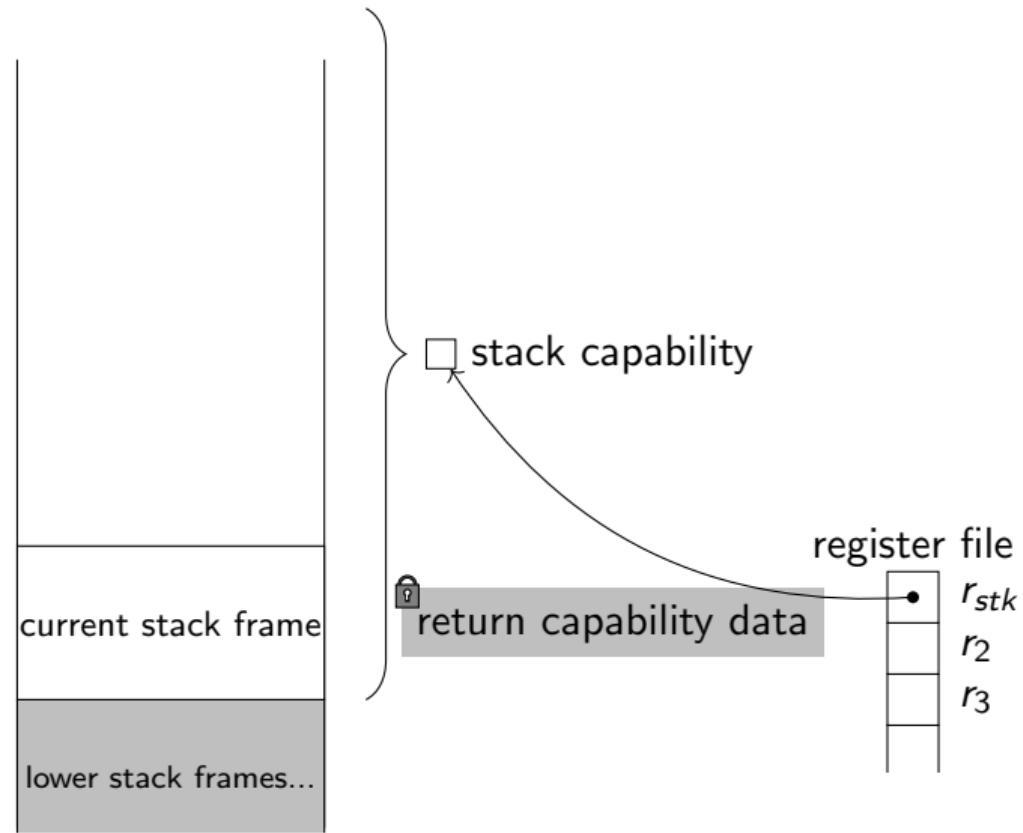
## Traditional stack pointers



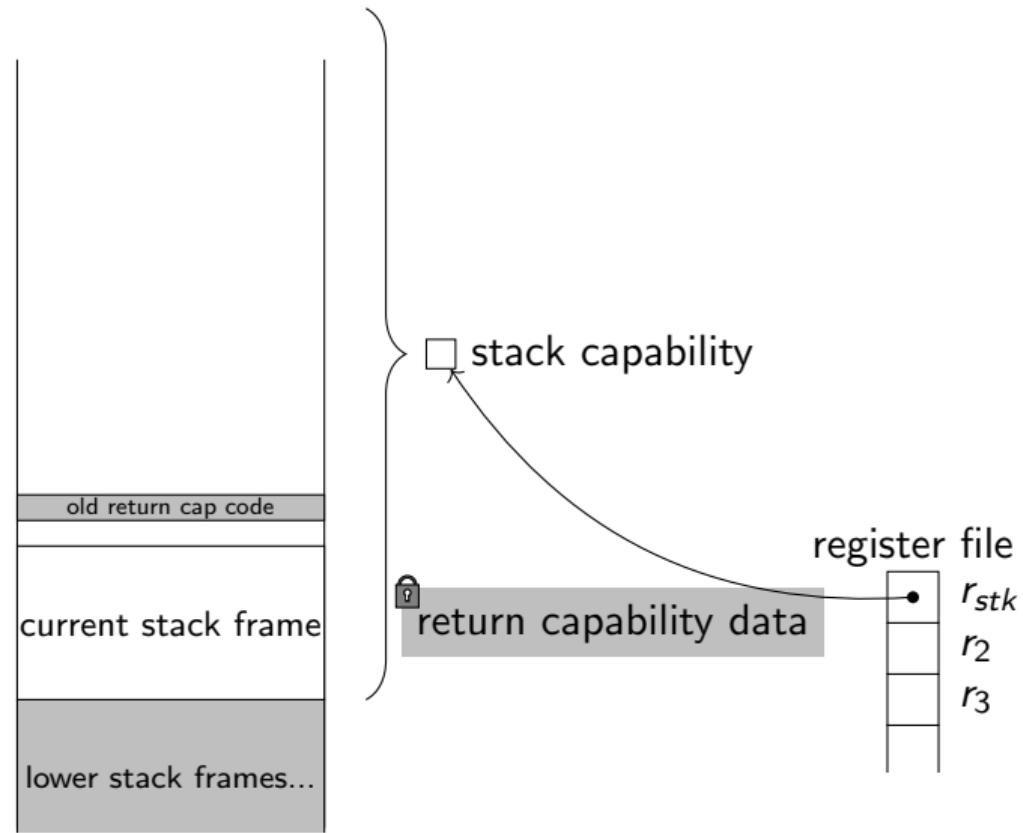
stack pointer

return but  
skip caller frame

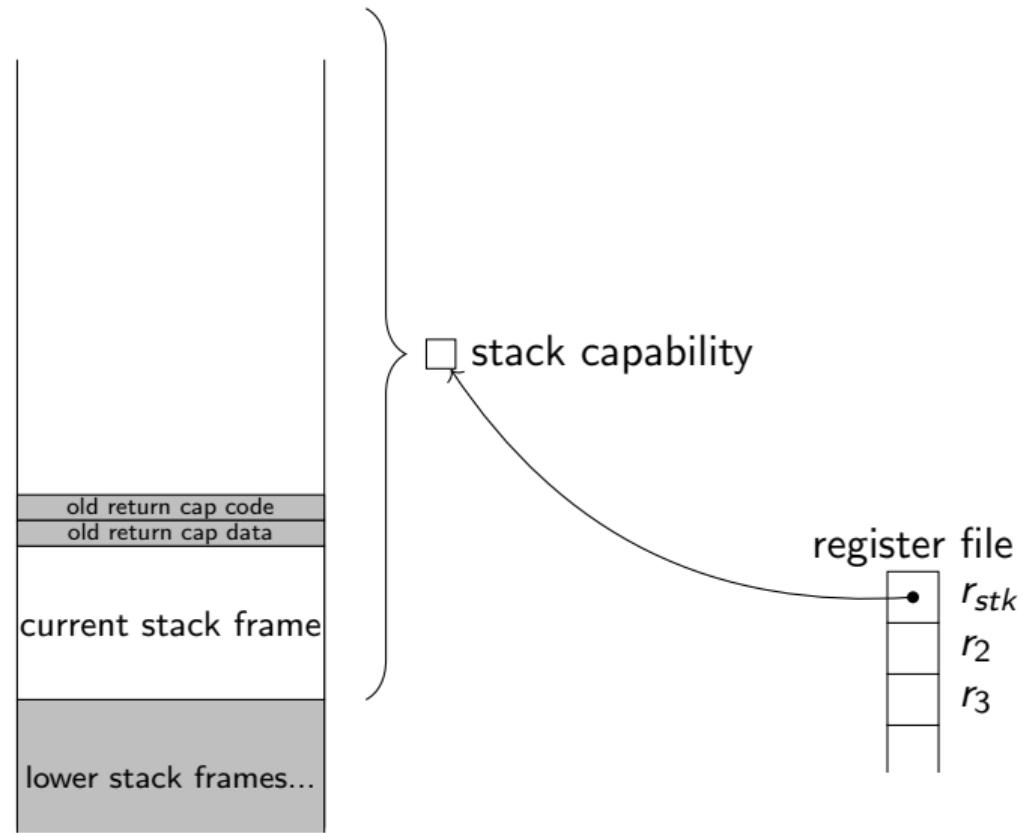
## Naive stack and return capabilities



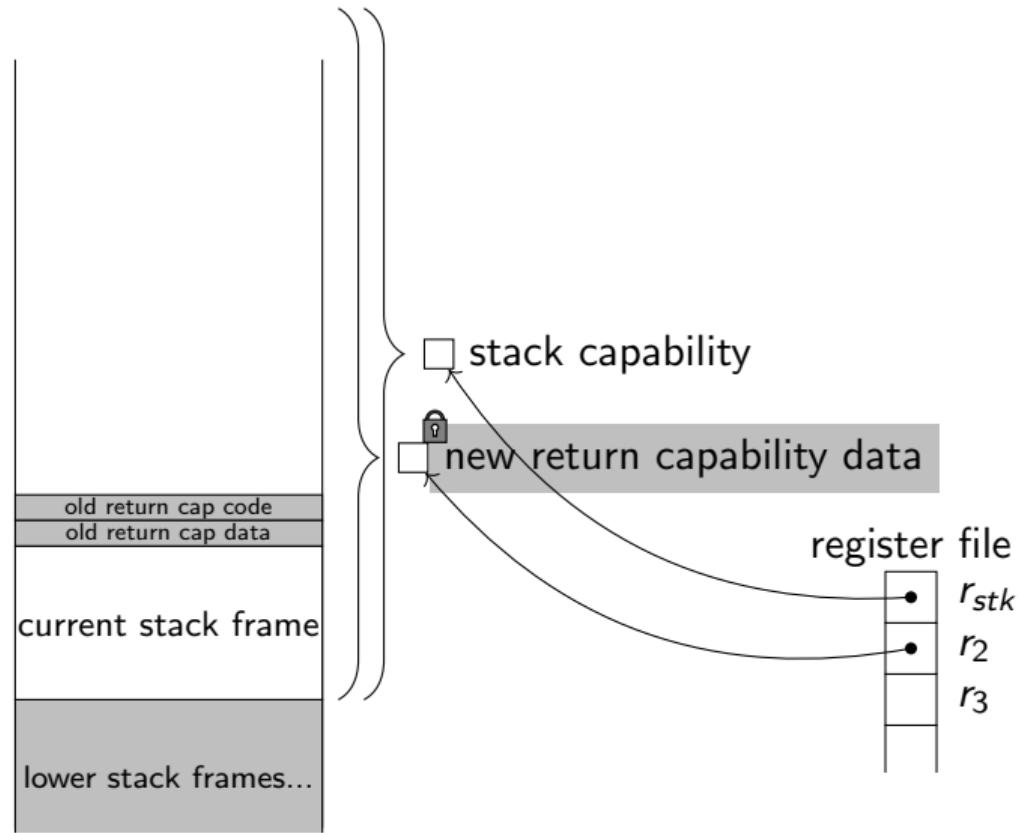
## Naive stack and return capabilities



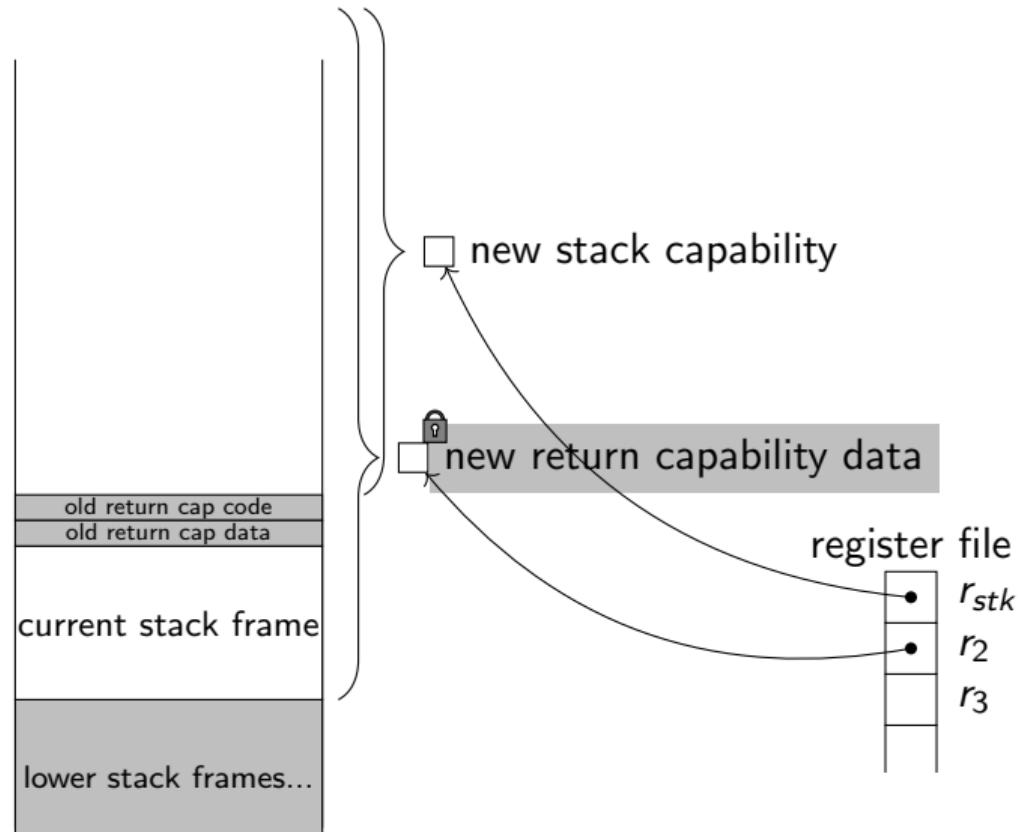
## Naive stack and return capabilities



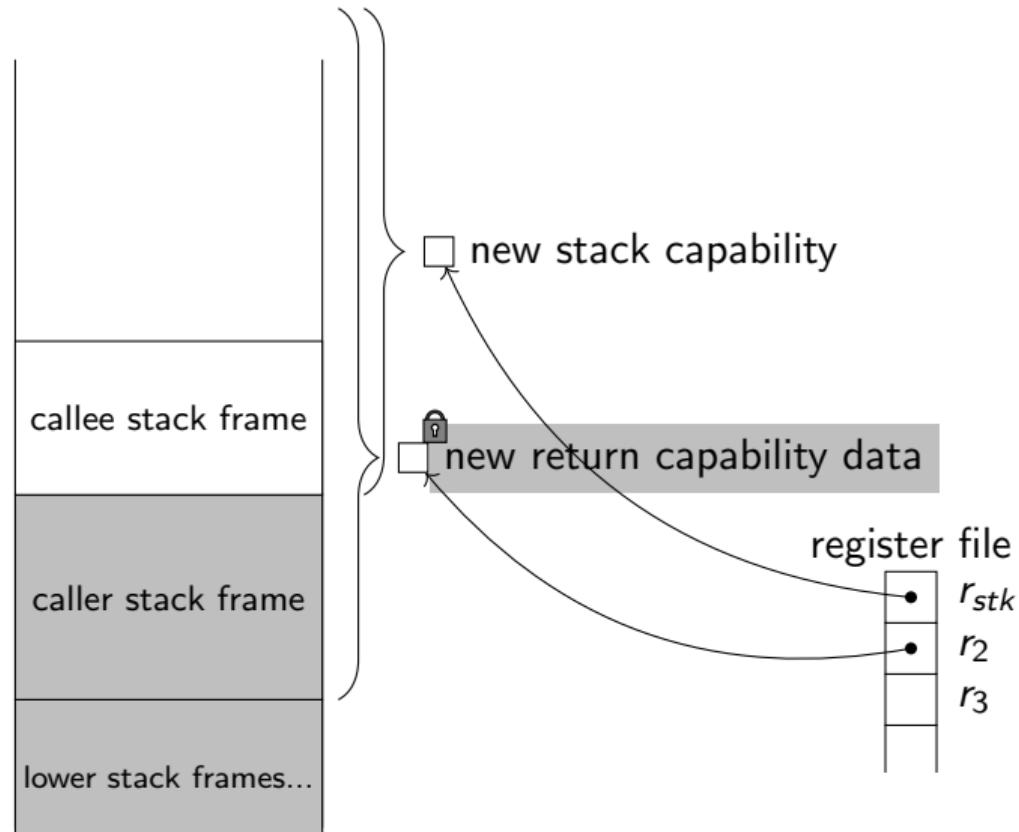
## Naive stack and return capabilities



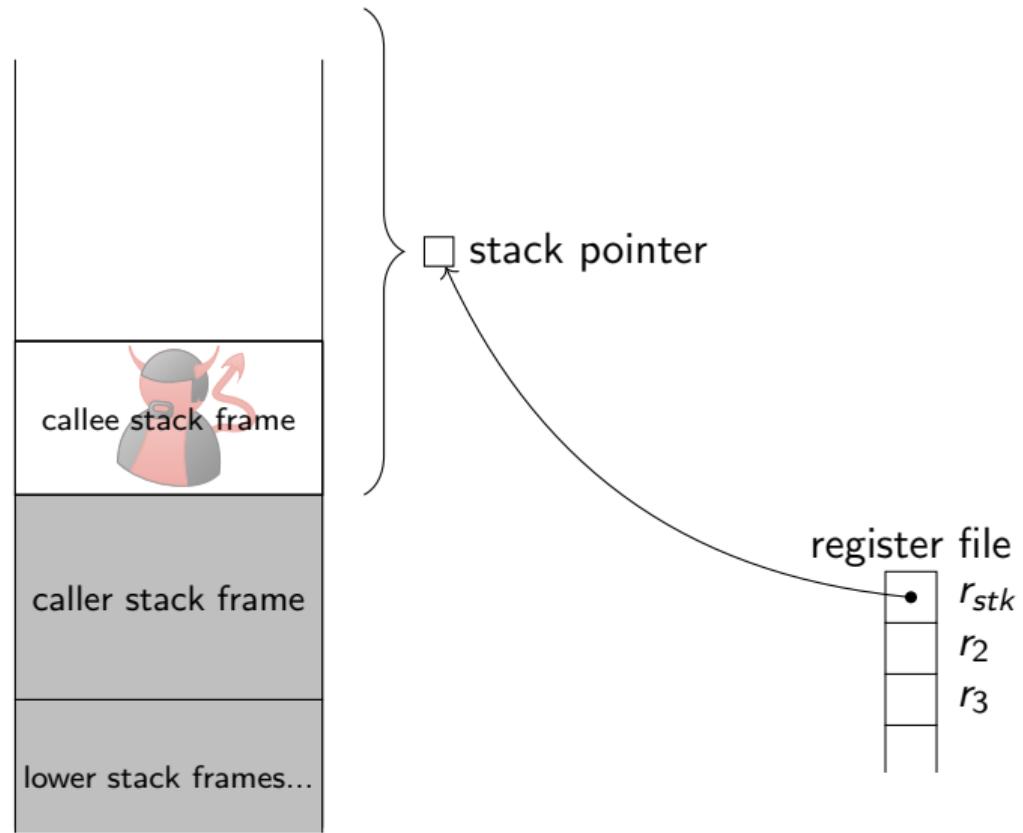
## Naive stack and return capabilities



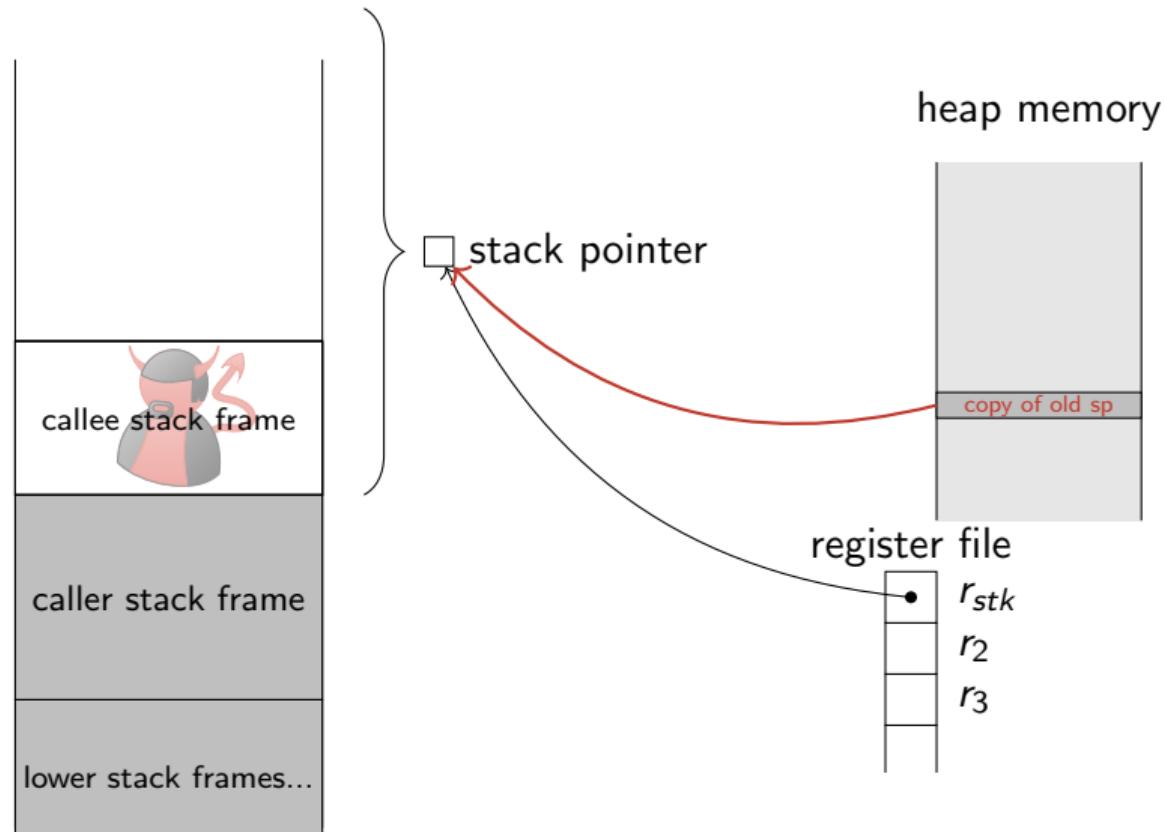
## Naive stack and return capabilities



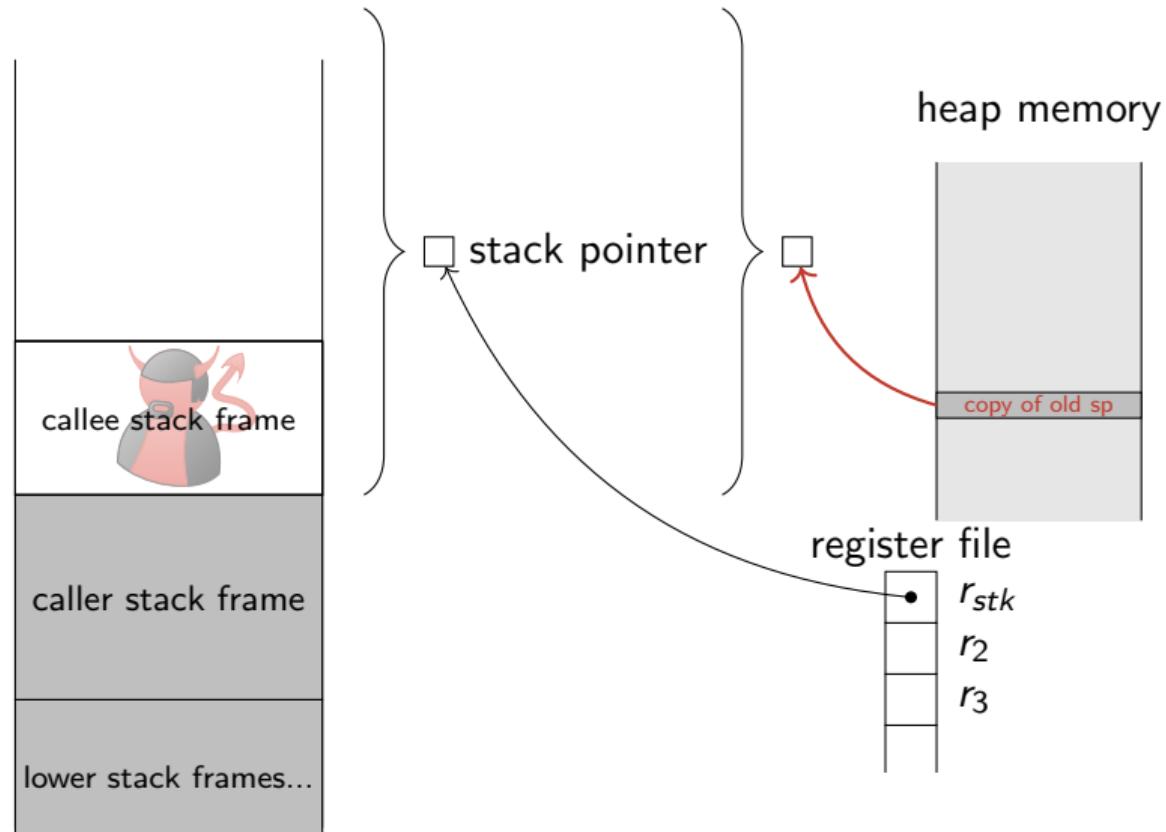
## Attack on naive stack and return capabilities



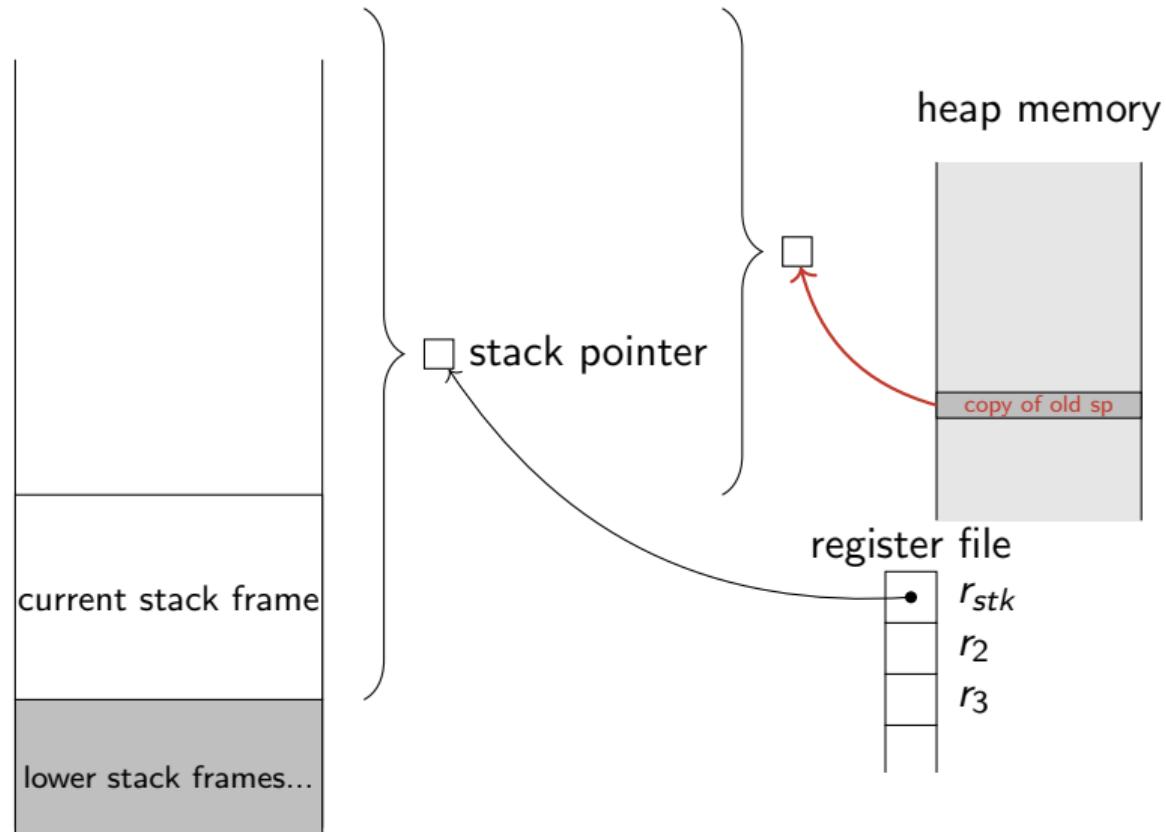
## Attack on naive stack and return capabilities



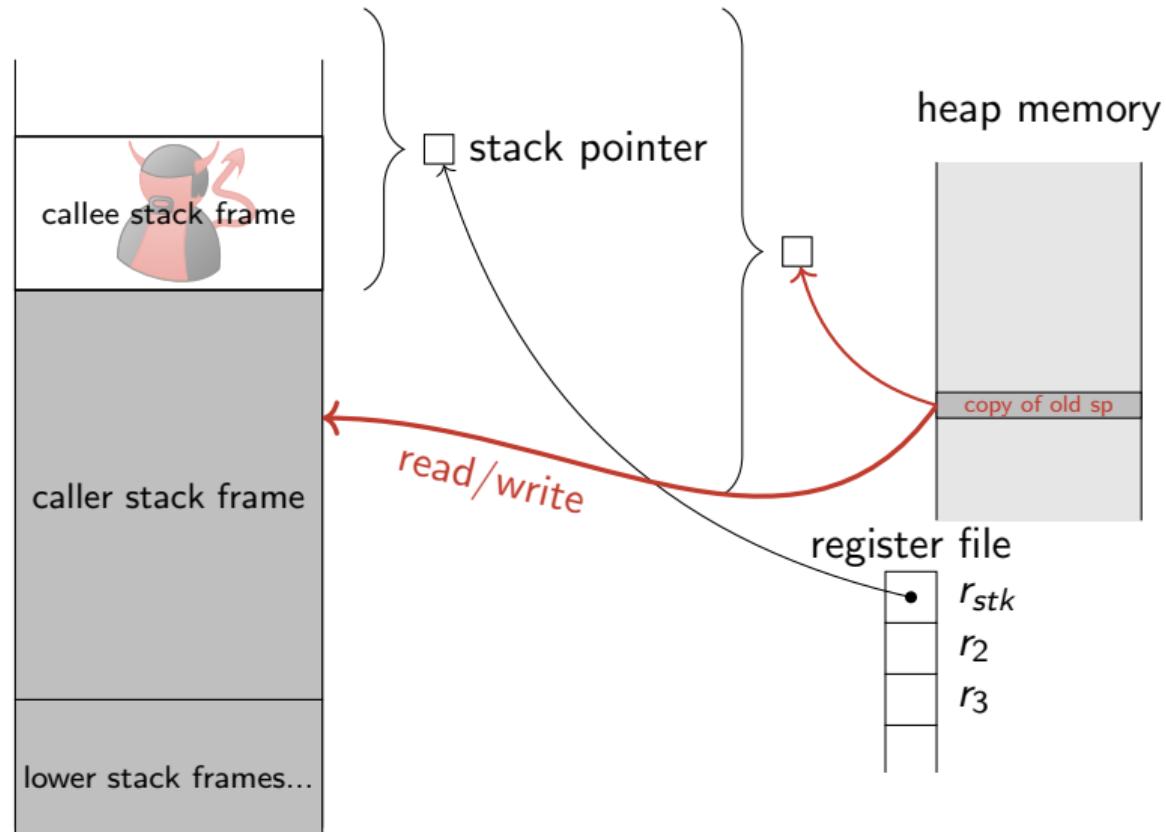
## Attack on naive stack and return capabilities



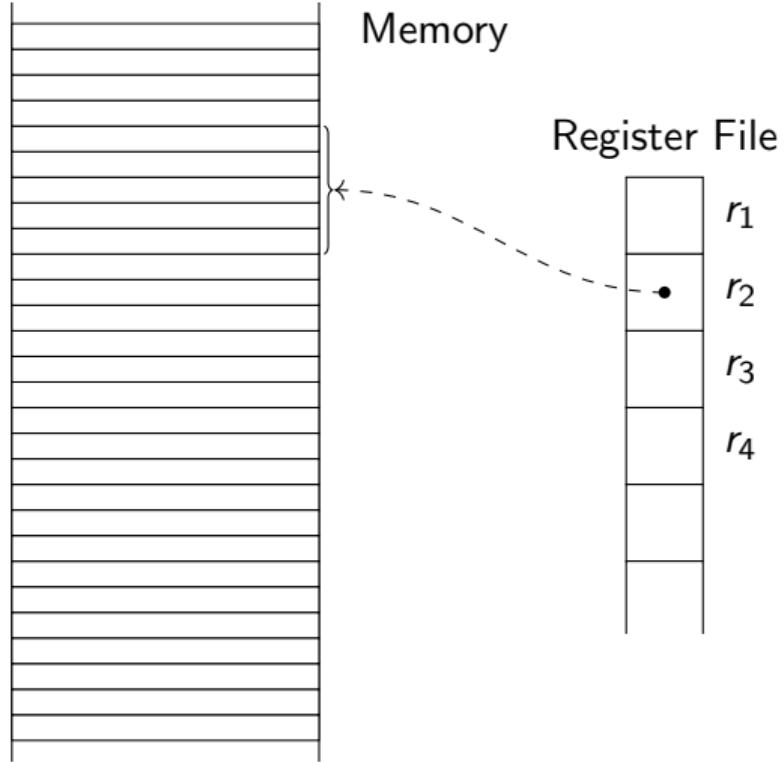
## Attack on naive stack and return capabilities



# Attack on naive stack and return capabilities



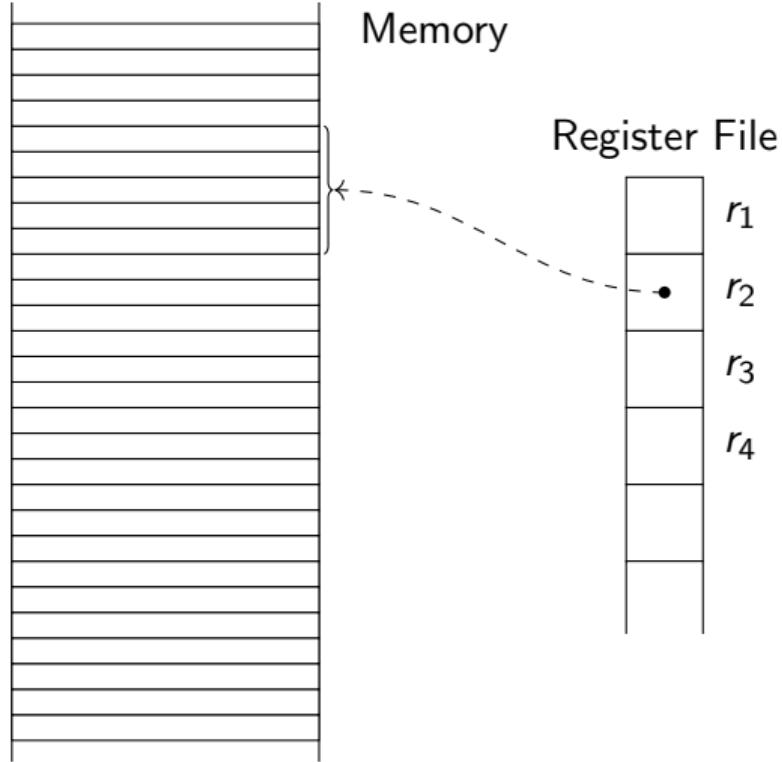
## Linear capabilities



Linear capabilities

- ▶ Non-duplicable

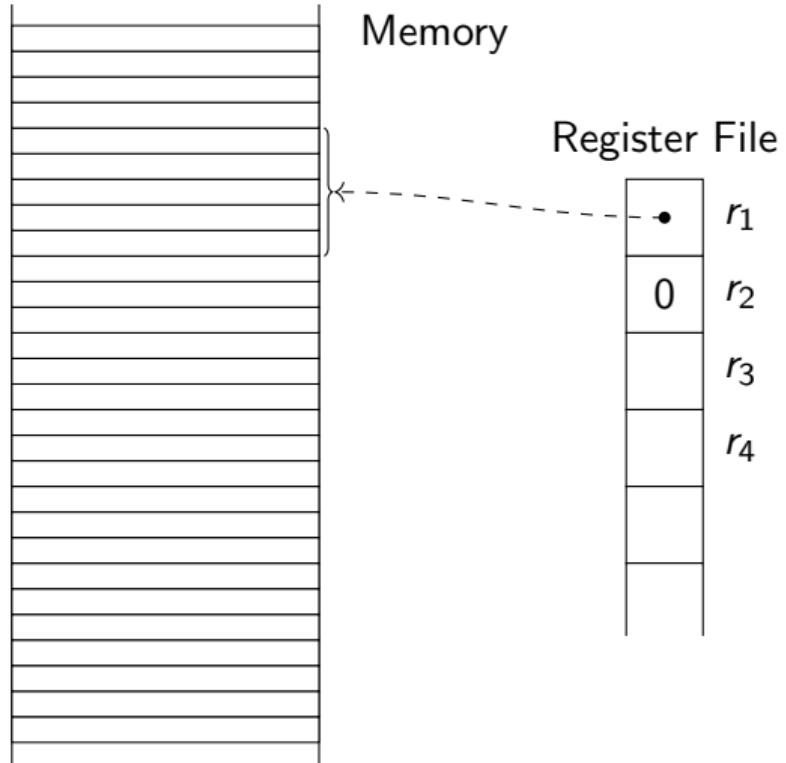
# Linear capabilities



Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$

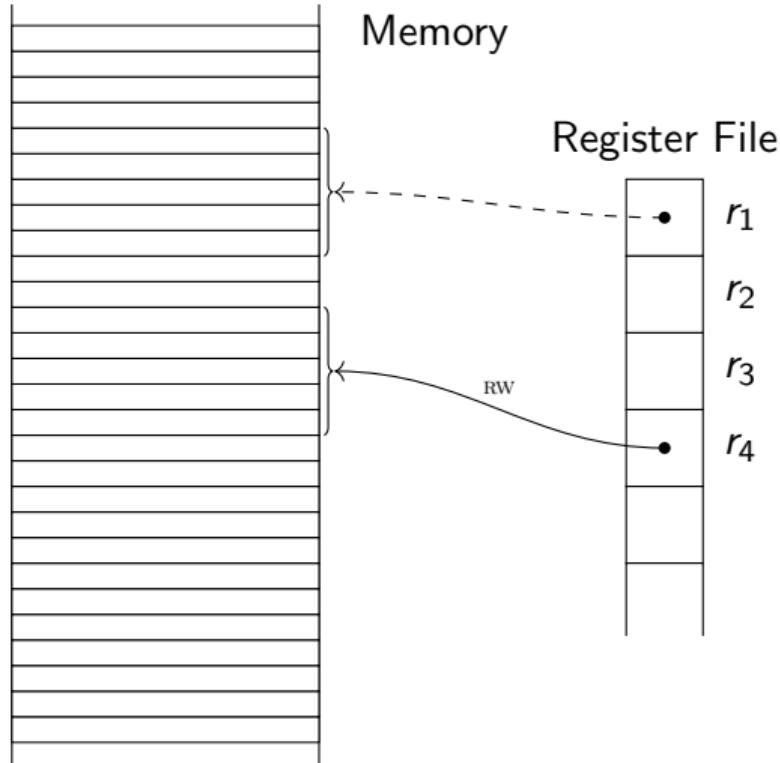
# Linear capabilities



Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$

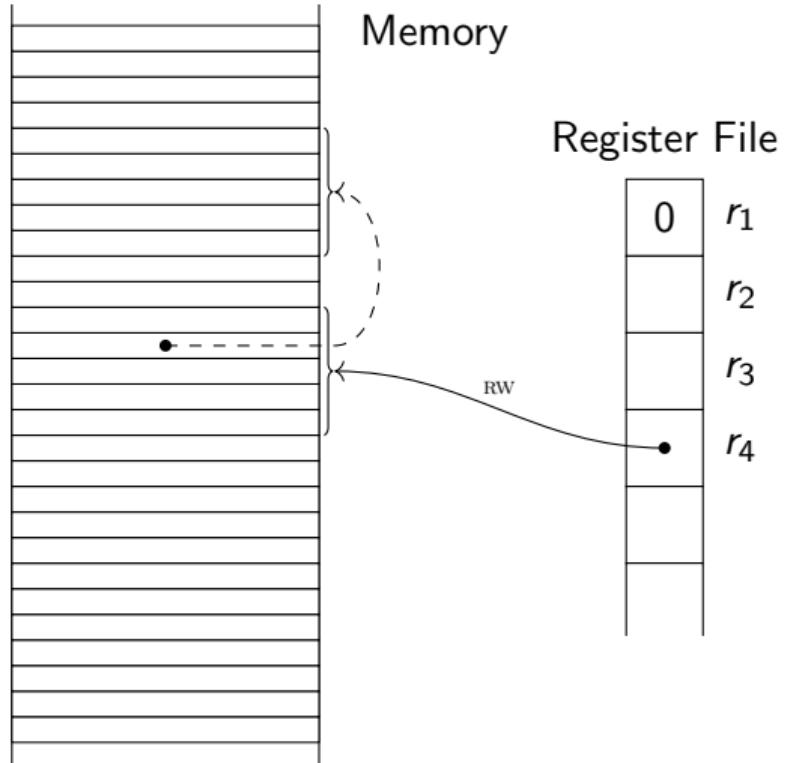
# Linear capabilities



## Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$
- ▶ store  $r_4 \ r_1$

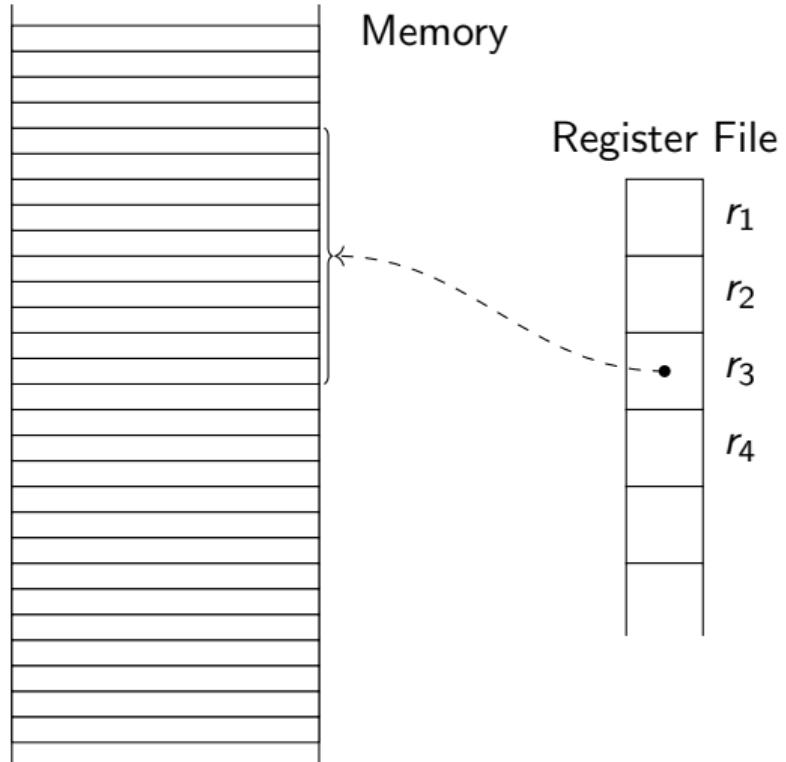
# Linear capabilities



## Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$
- ▶ store  $r_4 \ r_1$

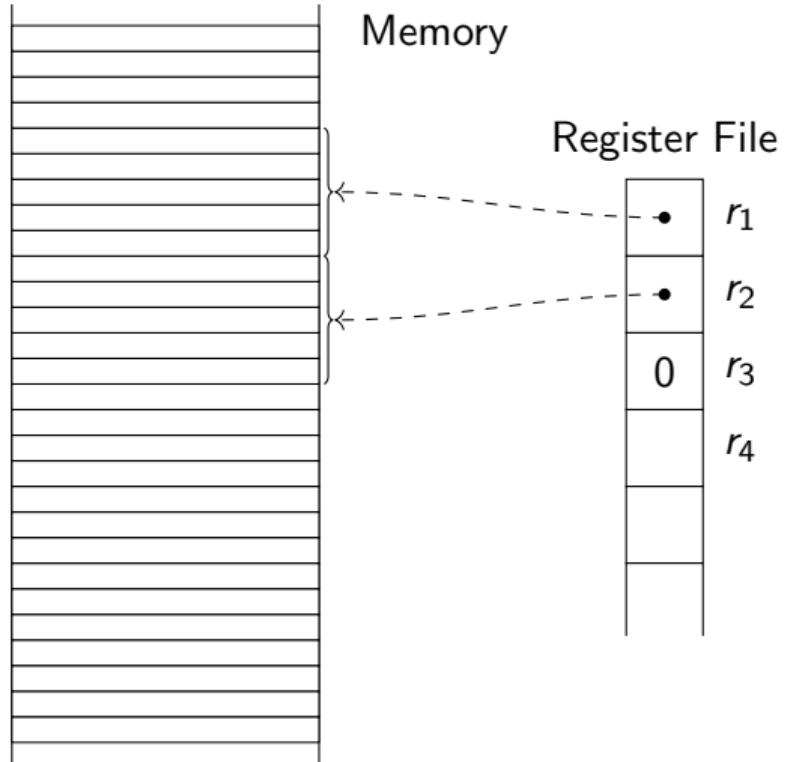
# Linear capabilities



## Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$
- ▶ store  $r_4 \ r_1$
- ▶ split  $r_1 \ r_2 \ r_3 \ n$

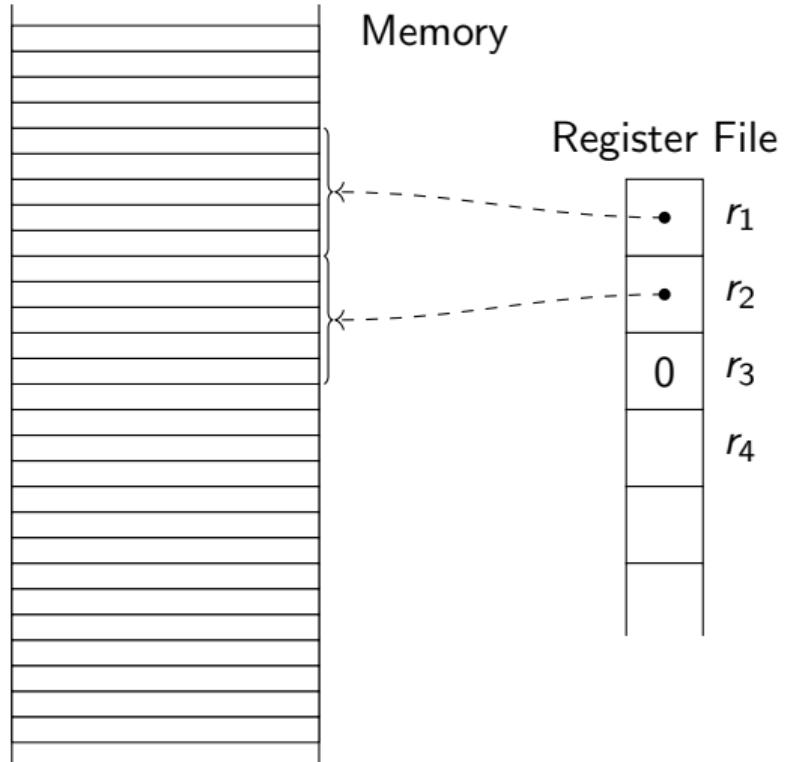
# Linear capabilities



## Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$
- ▶ store  $r_4 \ r_1$
- ▶ split  $r_1 \ r_2 \ r_3 \ n$

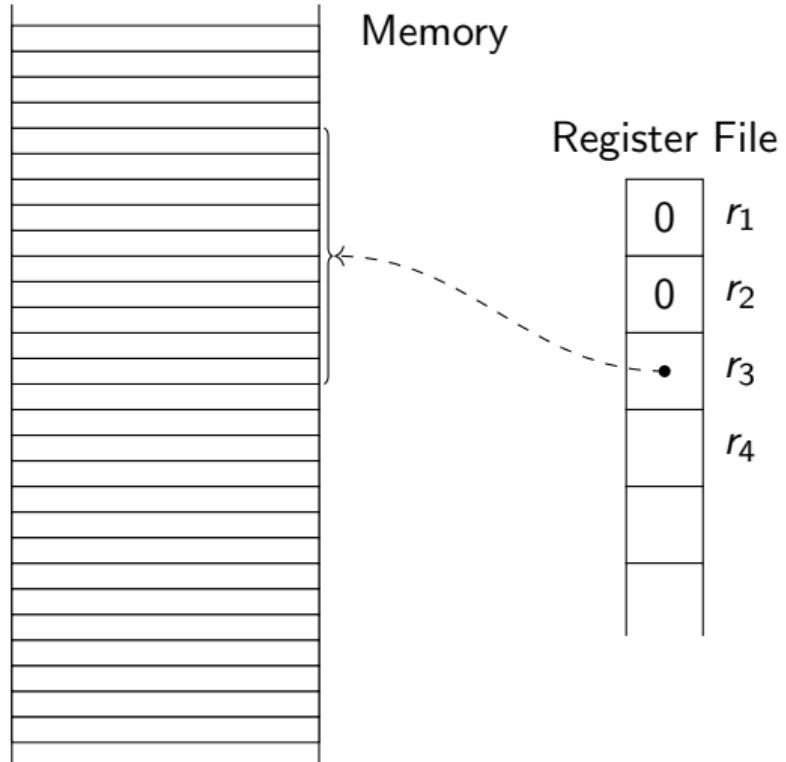
# Linear capabilities



## Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$
- ▶ store  $r_4 \ r_1$
- ▶ split  $r_1 \ r_2 \ r_3 \ n$
- ▶ splice  $r_3 \ r_1 \ r_2$

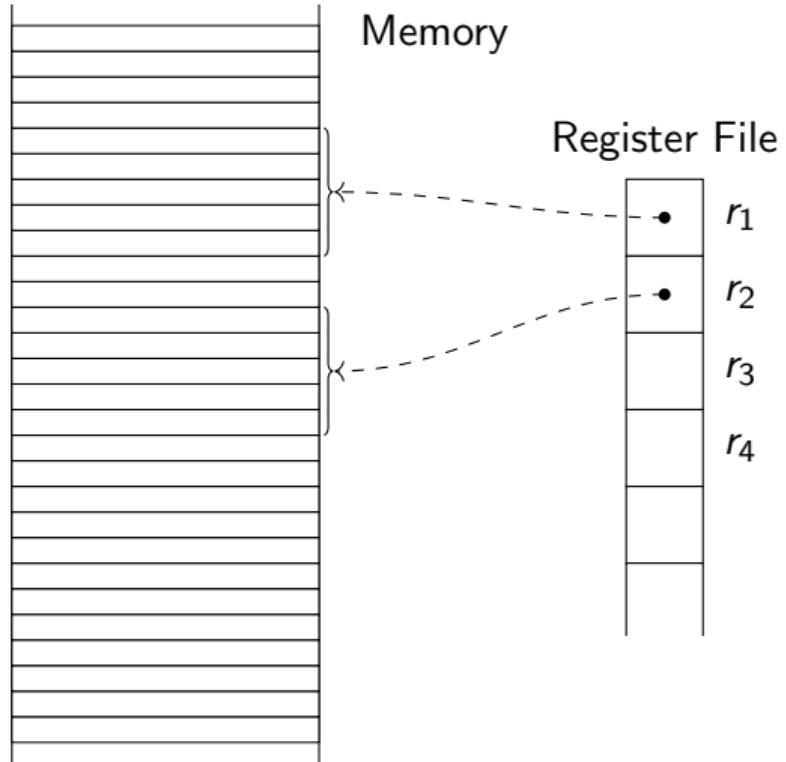
# Linear capabilities



## Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$
- ▶ store  $r_4 \ r_1$
- ▶ split  $r_1 \ r_2 \ r_3 \ n$
- ▶ splice  $r_3 \ r_1 \ r_2$

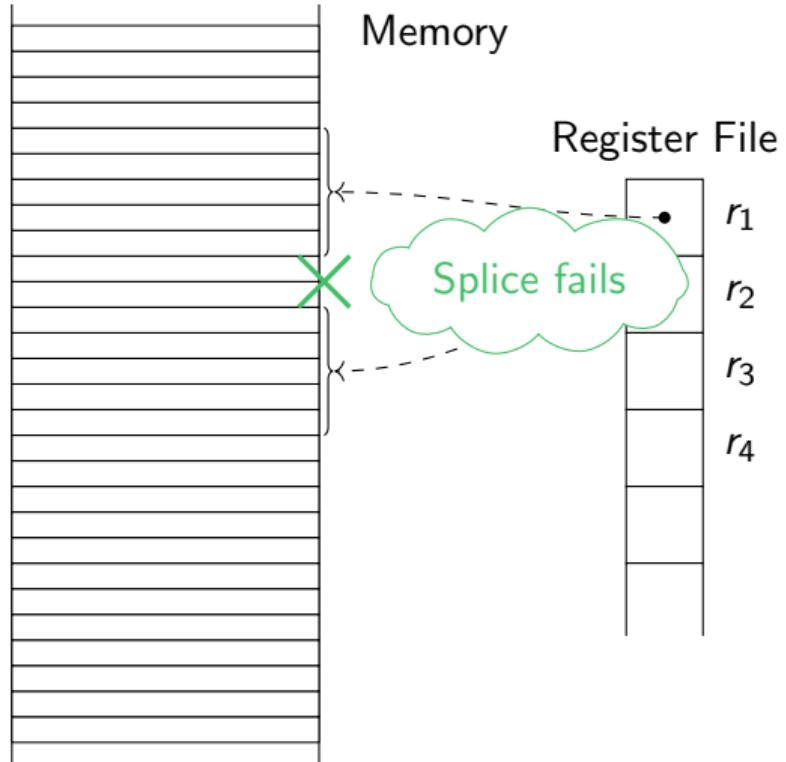
# Linear capabilities



## Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$
- ▶ store  $r_4 \ r_1$
- ▶ split  $r_1 \ r_2 \ r_3 \ n$
- ▶ splice  $r_3 \ r_1 \ r_2$

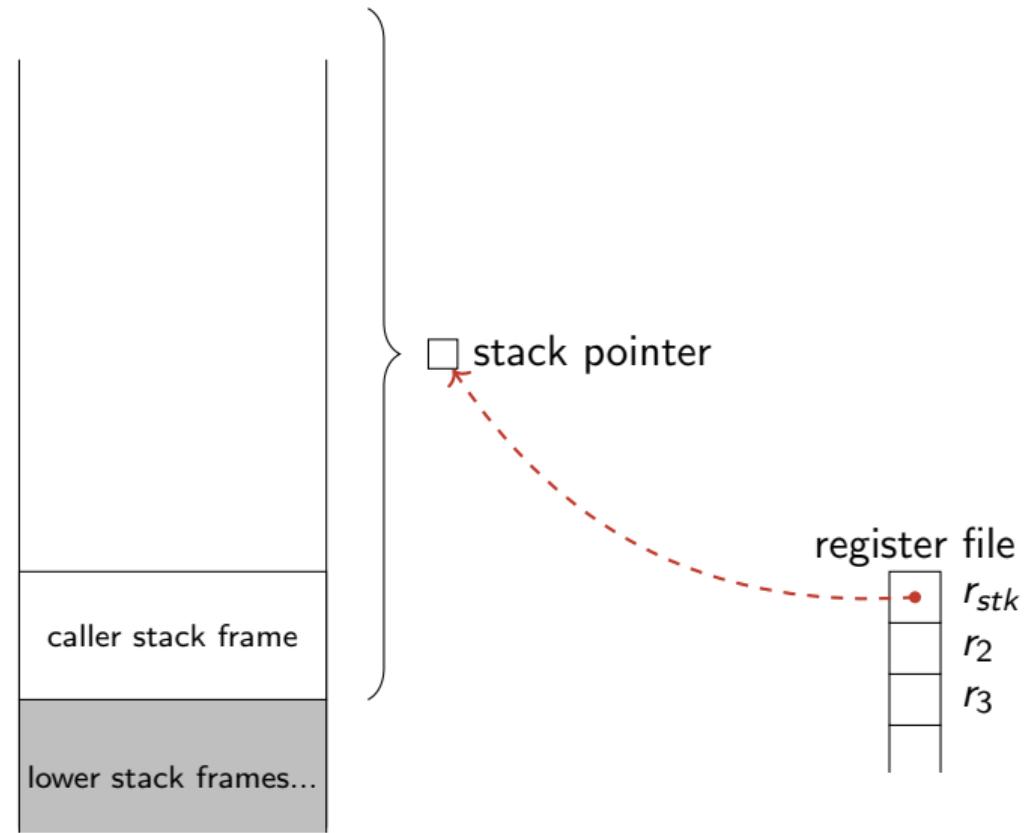
# Linear capabilities



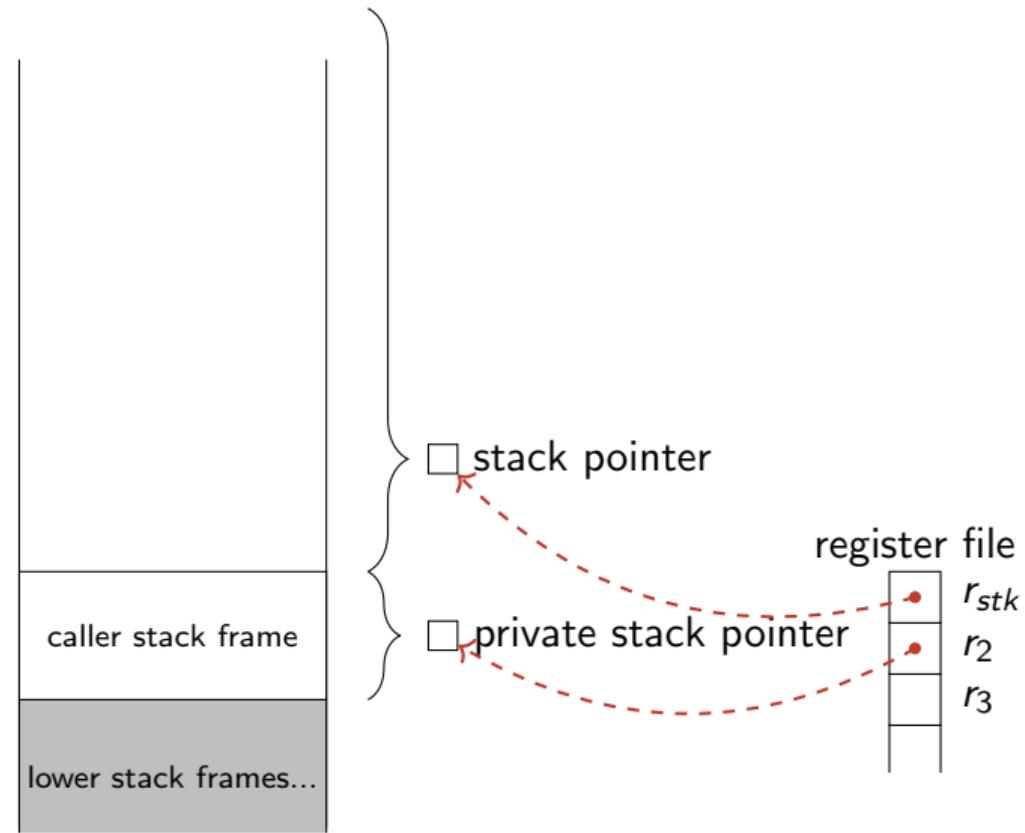
## Linear capabilities

- ▶ Non-duplicable
- ▶ move  $r_1 \ r_2$
- ▶ store  $r_4 \ r_1$
- ▶ split  $r_1 \ r_2 \ r_3 \ n$
- ▶ splice  $r_3 \ r_1 \ r_2$

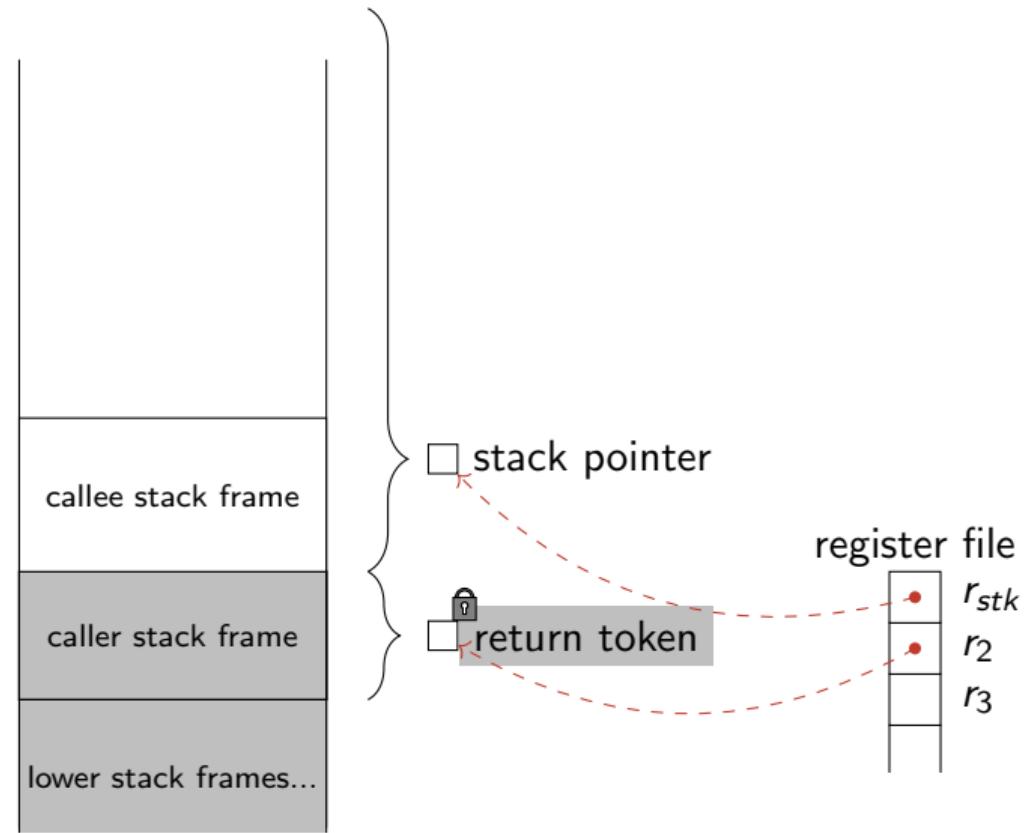
## Call with STKTOKENS



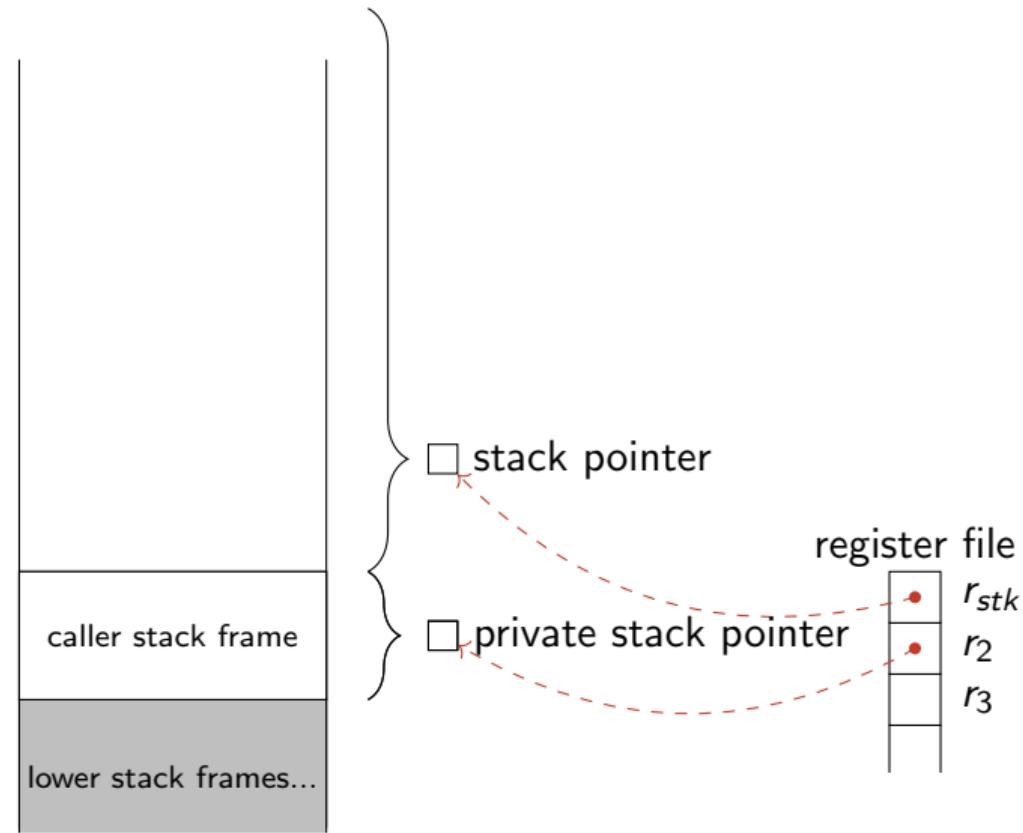
## Call with STKTOKENS



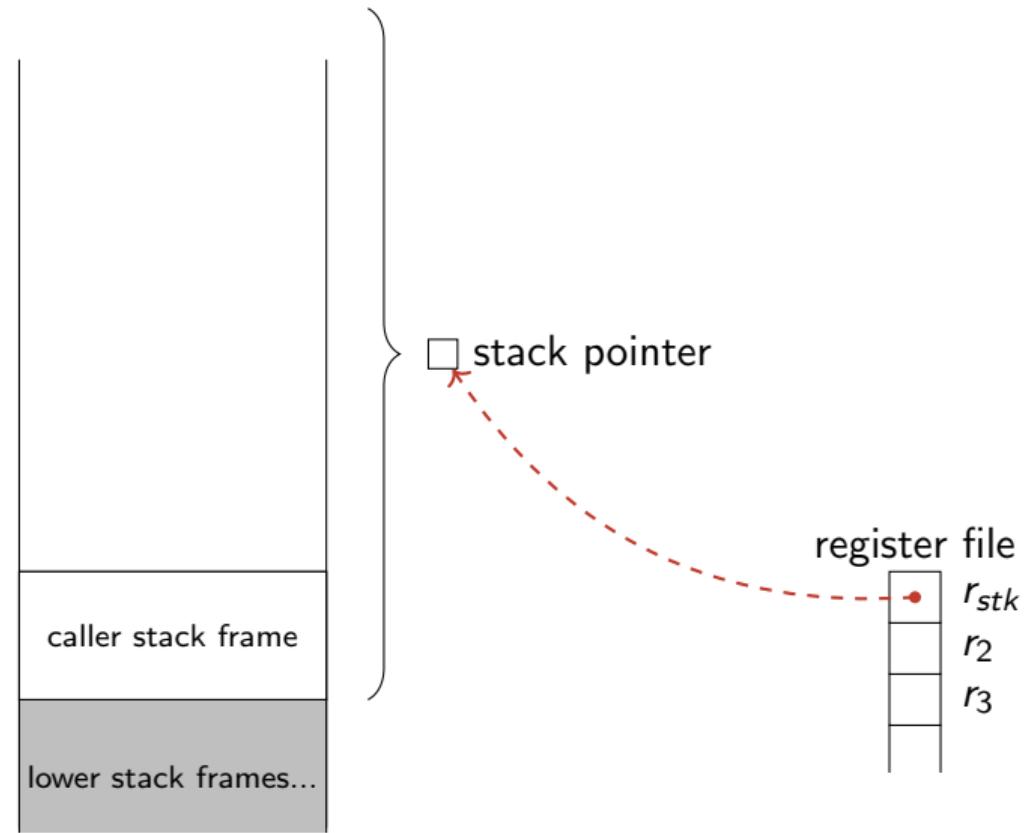
## Call with STKTOKENS



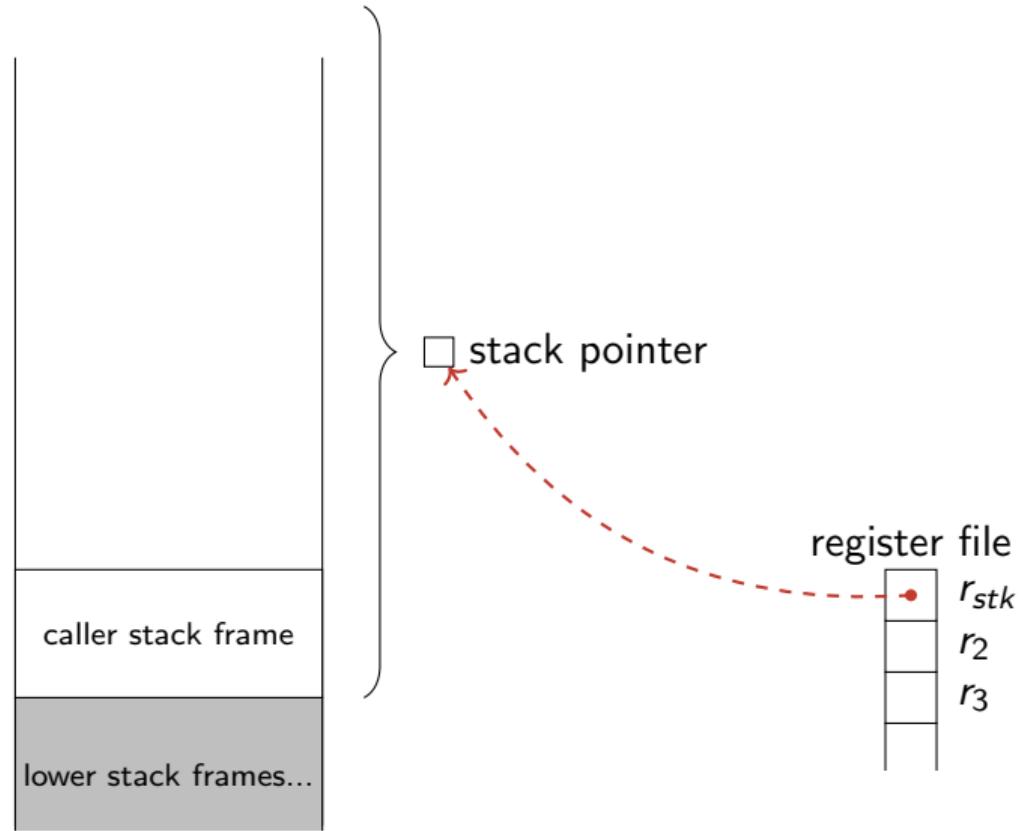
## Call with STKTOKENS



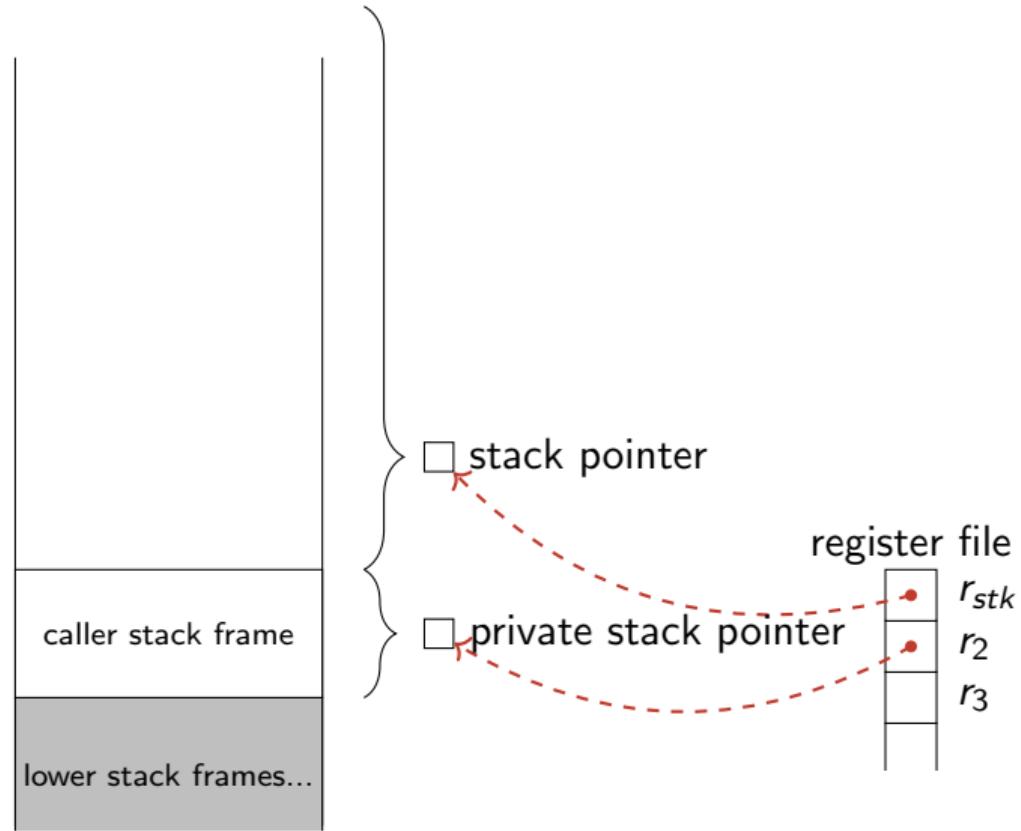
## Call with STKTOKENS



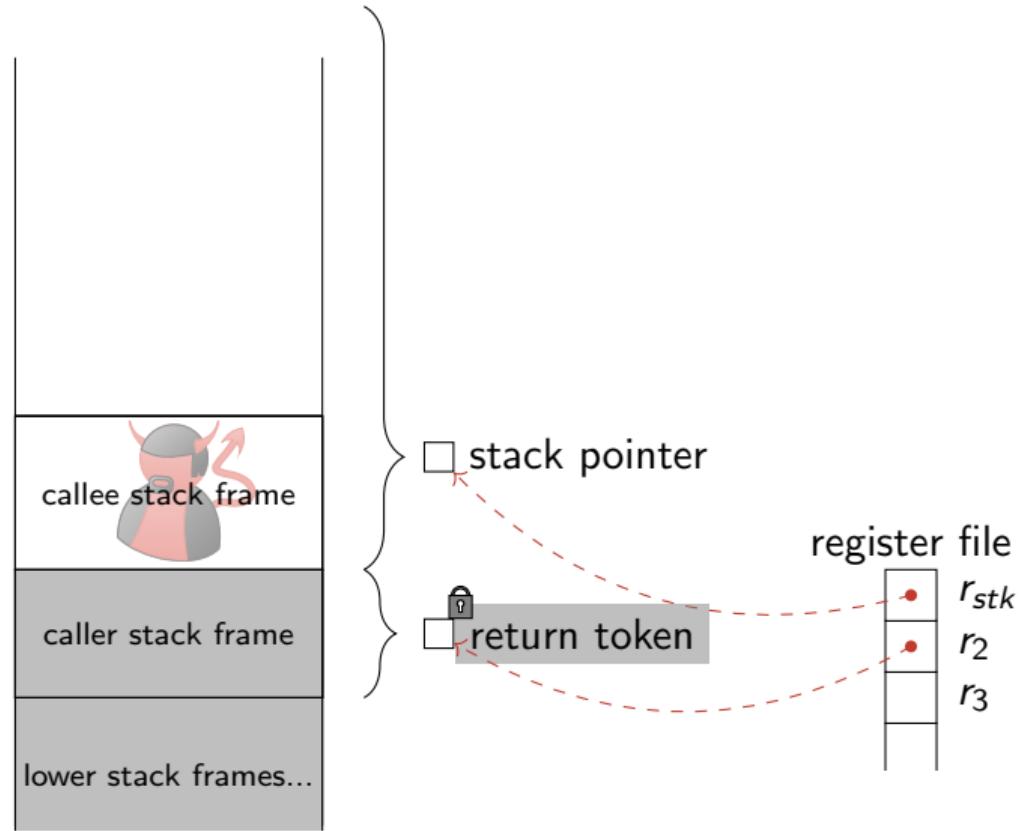
## STKTOKENS prevents the attack



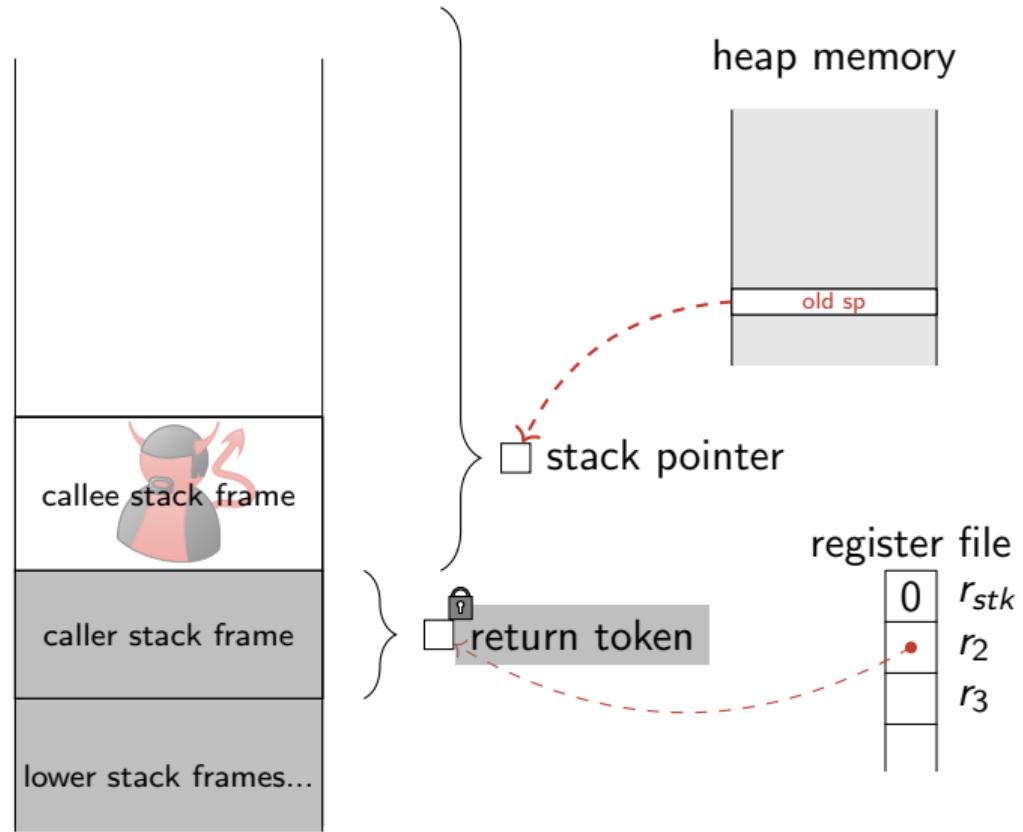
## STKTOKENS prevents the attack



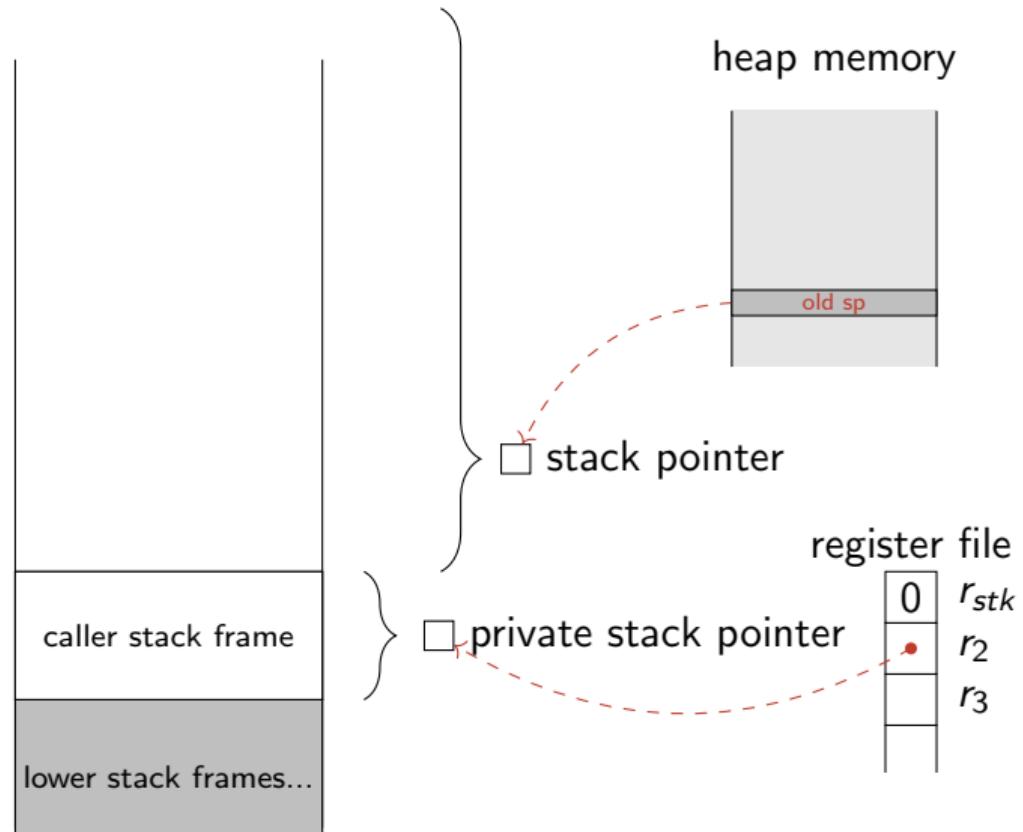
# STKTOKENS prevents the attack



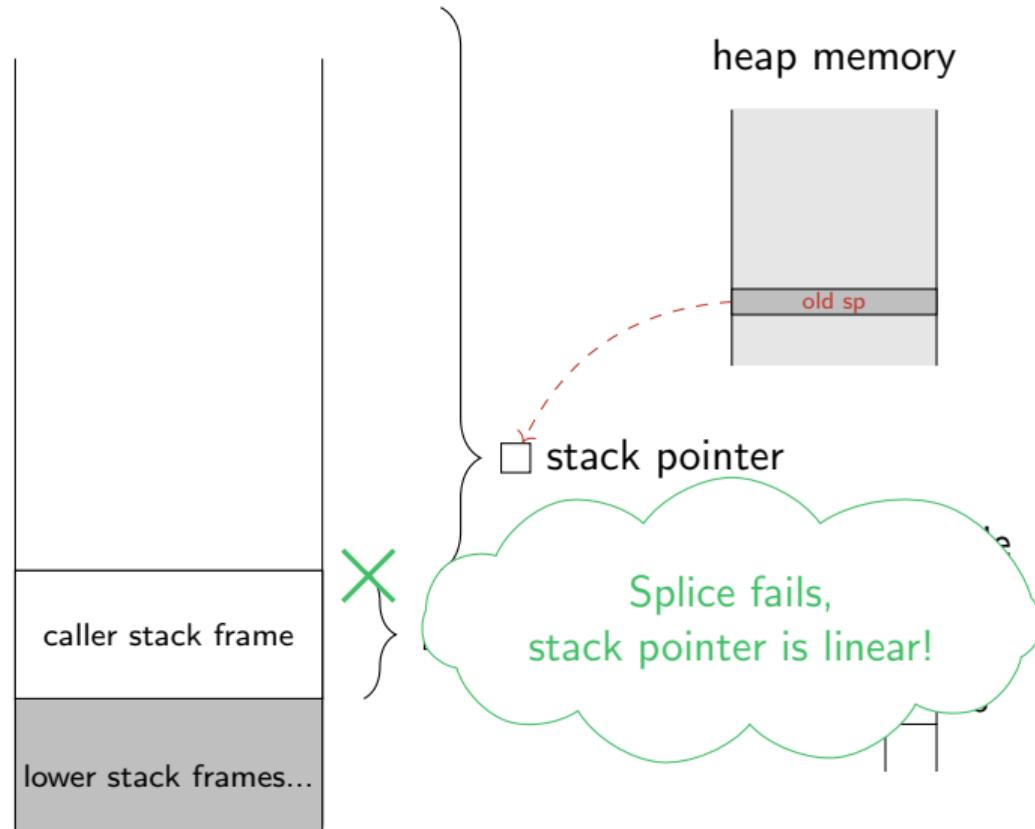
# STKTOKENS prevents the attack



## STKTOKENS prevents the attack



## STKTOKENS prevents the attack



# 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
```

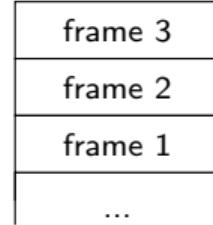
Linear Capability  
Machine

# 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
```

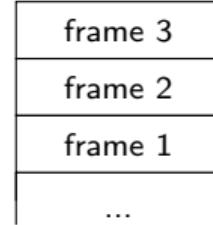
Linear Capability  
Machine

# Fully-abstract overlay semantics

```
move rtmp1 42
store rstk rtmp1
cca rstk -1
call1
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
```

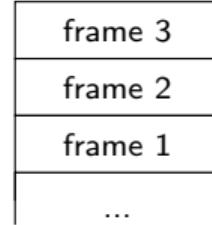
Linear Capability  
Machine

# 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**

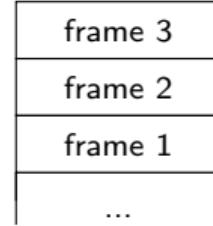
```
move rtmp1 42  
store rstk rtmp1  
cca rstk -1  
geta rtmp1 rstk  
cca rretc 5  
move rtmp1 pc  
cca rtmp1 -20
```

**return**

*id*

---

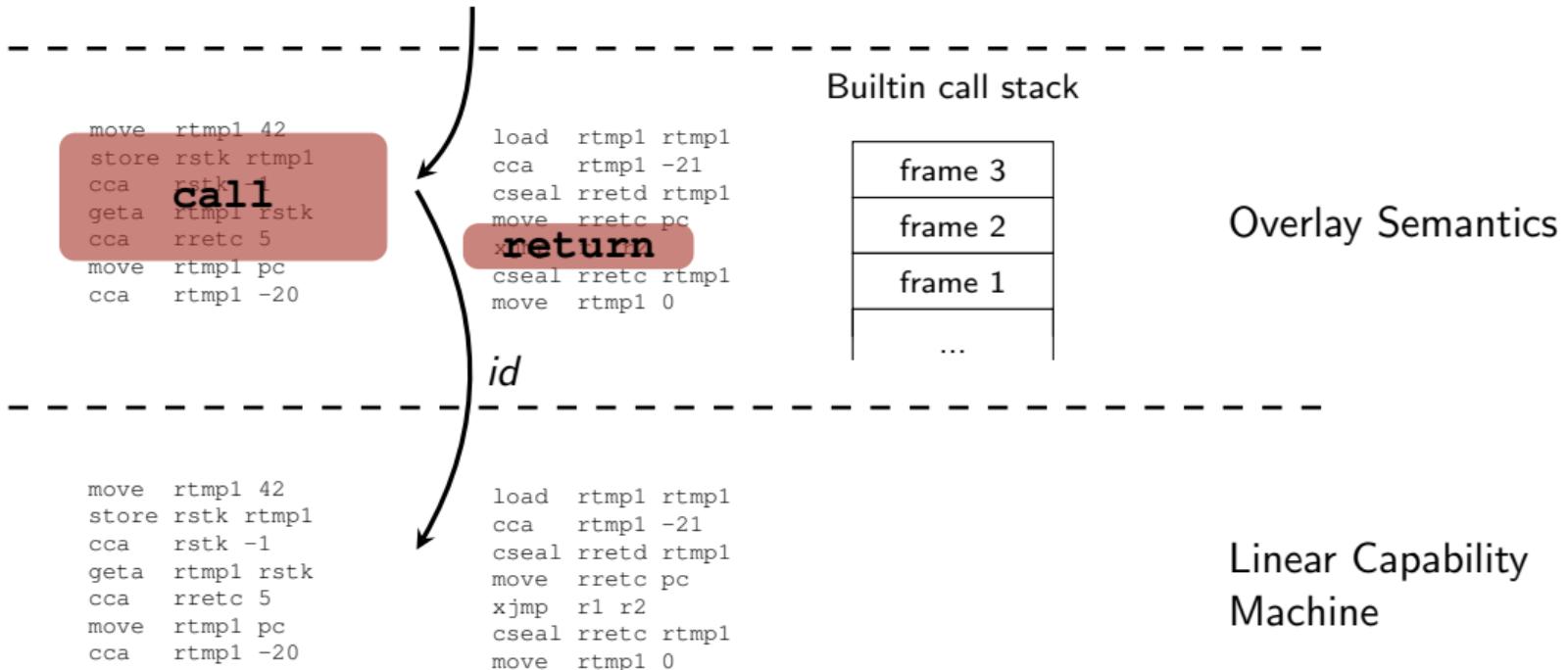
Builtin call stack



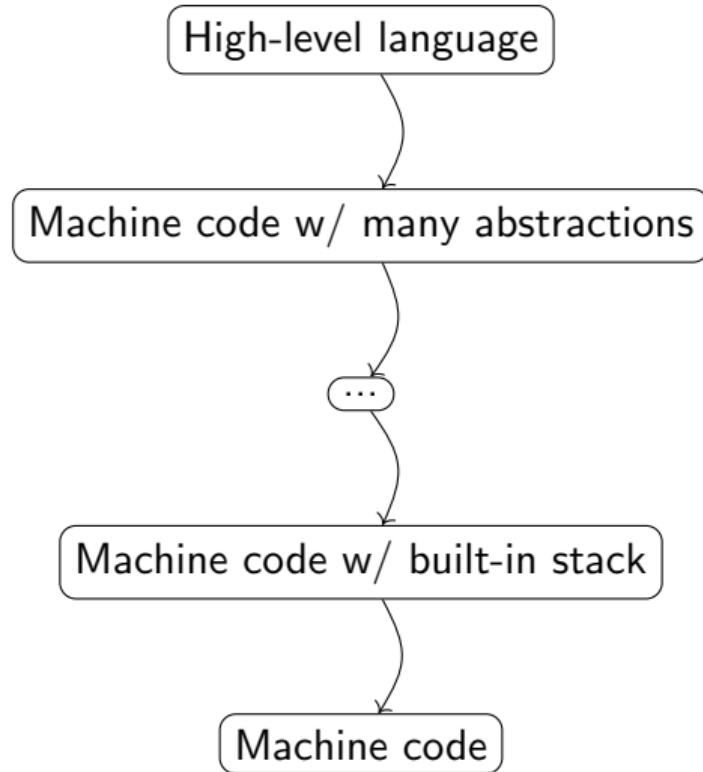
Overlay Semantics

Linear Capability  
Machine

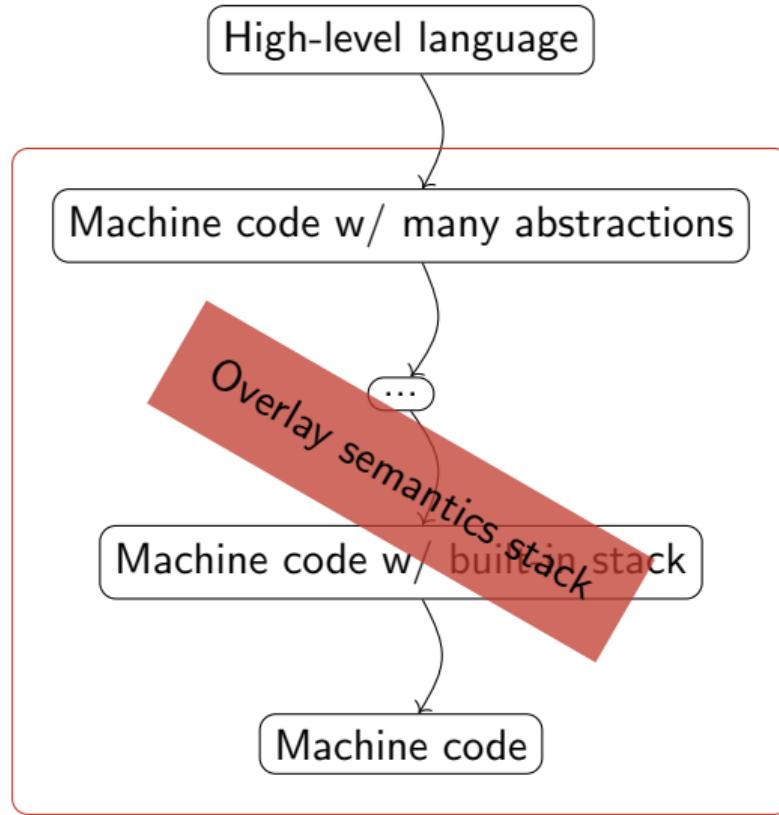
# Fully-abstract overlay semantics



# Proof sketch for a realistic secure compiler



# Proof sketch for a realistic secure compiler



## Full abstraction

Theorem (Full abstraction)

*For reasonable, well-formed  $comp_1$  and  $comp_2$ , we have*

$$comp_1 \approx_{\text{ctx}} comp_2 \quad \Leftrightarrow \quad comp_1 \approx_{\text{ctx}}^{\text{red}} comp_2$$

---

# Full abstraction

Theorem (Full abstraction)

For *reasonable*, well-formed  $comp_1$  and  $comp_2$ , we have

$$comp_1 \approx_{\text{ctx}} comp_2 \quad \Leftrightarrow \quad comp_1 \approx_{\text{ctx}}^{\text{red}} comp_2$$

---

# Full abstraction

Theorem (Full abstraction)

For *reasonable*, well-formed  $comp_1$  and  $comp_2$ , we have

$$comp_1 \approx_{ctx} comp_2 \Leftrightarrow comp_1 \approx_{ctx}^{\textcolor{red}{\text{ctx}}} comp_2$$

- 
- ▶ STKTOKENS reasonable use of return seals
    - ▶ Return seals are only used to seal old program pointers.
    - ▶ Every return seal is only used for one call site.
    - ▶ Return seals are not leaked.

# Full abstraction

Theorem (Full abstraction)

For *reasonable*, well-formed  $comp_1$  and  $comp_2$ , we have

$$comp_1 \approx_{\text{ctx}} comp_2 \quad \Leftrightarrow \quad comp_1 \approx_{\text{ctx}}^{\text{red}} comp_2$$

- 
- ▶ STKTOKENS reasonable use of return seals
    - ▶ Return seals are only used to seal old program pointers.
    - ▶ Every return seal is only used for one call site.
    - ▶ Return seals are not leaked.
  - ▶ No measures against us

# Full abstraction

Theorem (Full abstraction)

For reasonable, well-formed  $comp_1$  and  $comp_2$ , we have

$$comp_1 \approx_{ctx} comp_2 \Leftrightarrow comp_1 \approx_{ctx}^{\text{red}} comp_2$$

- 
- ▶ STKTOKENS reasonable use of return seals
    - ▶ Return seals are only used to seal old program pointers.
    - ▶ Every return seal is only used for one call site.
    - ▶ Return seals are not leaked.
  - ▶ No measures against us

# Full abstraction

Theorem (Full abstraction)

For reasonable, well-formed  $comp_1$  and  $comp_2$ , we have

$$comp_1 \approx_{ctx} comp_2 \Leftrightarrow comp_1 \approx_{ctx}^{\text{red}} comp_2$$

- 
- ▶ STKTOKENS reasonable use of return seals
    - ▶ Return seals are only used to seal old program pointers.
    - ▶ Every return seal is only used for one call site.
    - ▶ Return seals are not leaked.
    - ▶ Component structure
    - ▶ Sane linker behaviour
  - ▶ No measures against us

# Full abstraction

Theorem (Full abstraction)

For reasonable, well-formed  $comp_1$  and  $comp_2$ , we have

$$comp_1 \approx_{ctx} comp_2 \Leftrightarrow comp_1 \approx_{ctx}^{\text{red}} comp_2$$

- 
- ▶ STKTOKENS reasonable use of return seals
    - ▶ Return seals are only used to seal old program pointers.
    - ▶ Every return seal is only used for one call site.
    - ▶ Return seals are not leaked.
    - ▶ Component structure
    - ▶ Sane linker behaviour
  - ▶ No measures against us

Should be handled in earlier compiler phase

## Conclusion

- ▶ STKTOKENS ensures well-bracketed control flow and local-state encapsulation.
- ▶ Overlay semantics used to define well-bracketed calls and local state encapsulation.
- ▶ Capability machines may be a good target for secure compilation.

Thank you!

## 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.

# STKTOKENS vs. CHERI stack

## StkTokens

- ▶ No built-in stack
- ▶ Fine granularity of security domains
- ▶ No context switch on call
- ▶ Caller has the full responsibility for securing call

## CHERI stack

- ▶ Central, built-in stack
- ▶ Call and return instructions interact with stack
- ▶ Context switch on call/return
- ▶ Local capabilities cannot cross security boundary