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

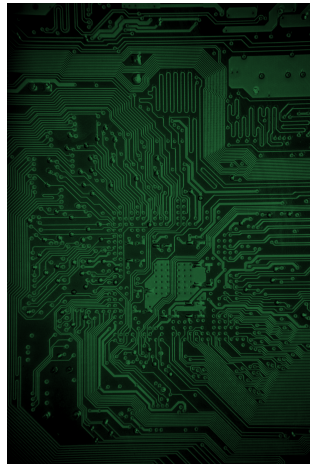
Lau Skorstengaard[1] Dominique Devriese[2] Lars Birkedal[1]

[1]Aarhus University

[2]Vrije Universiteit Brussel

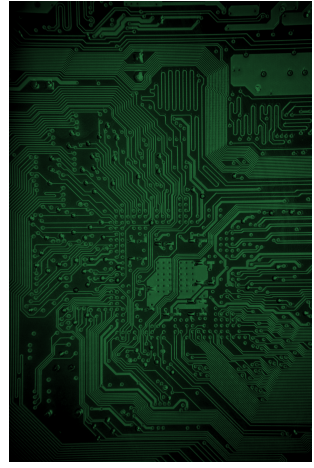
POPL, January 16, 2019

Abstractions all the way down



Abstractions all the way down

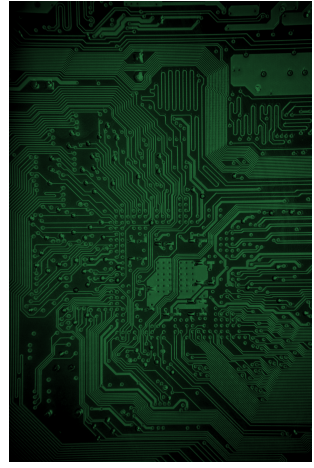
```
main:
  .cfi_startproc
# BB#0:
  pushq %rbp
.Ltmp0:
  .cfi\_def\_cfa\_offset 16
.Ltmp1:
  .cfi\_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp2:
  .cfi\_def\_cfa\_register %rbp
  subq  $16, %rsp
  movabsq $.L.str, %rdi
  movl  $0, -4(%rbp)
  movb  $0, %al
  callq printf
  xorl  %ecx, %ecx
  movl  %eax,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

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

```
main:
  .cfi_startproc
# BB#0:
  pushq %rbp
.Ltmp0:
  .cfi\_def\_cfa\_offset 16
.Ltmp1:
  .cfi\_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp2:
  .cfi\_def\_cfa\_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
```

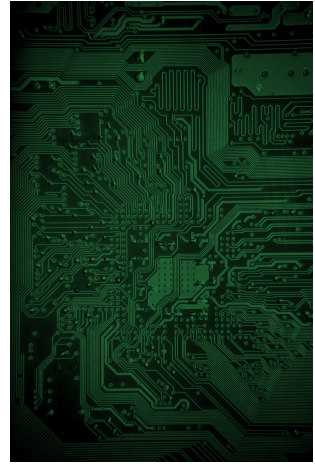


Abstractions all the way down

compilation

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

```
main:
  .cfi_startproc
# BB#0:
  pushq %rbp
.Ltmp0:
  .cfi\_def\_cfa\_offset 16
.Ltmp1:
  .cfi\_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp2:
  .cfi\_def\_cfa\_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
```

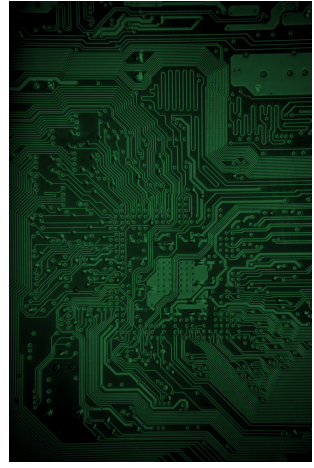


Abstractions all the way down

secure compilation

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

```
main:
  .cfi_startproc
# BB#0:
  pushq %rbp
.Ltmp0:
  .cfi\_def\_cfa\_offset 16
.Ltmp1:
  .cfi\_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp2:
  .cfi\_def\_cfa\_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
```

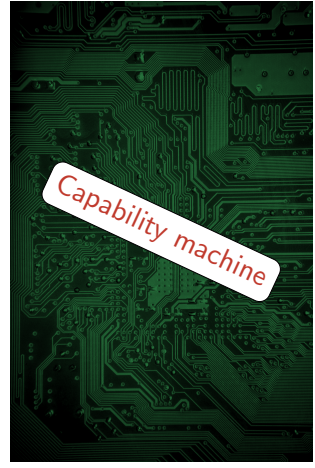


Abstractions all the way down

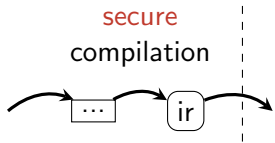
secure compilation

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

```
main:
  .cfi_startproc
# BB#0:
  pushq %rbp
.Ltmp0:
  .cfi\_def\_cfa\_offset 16
.Ltmp1:
  .cfi\_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp2:
  .cfi\_def\_cfa\_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
```

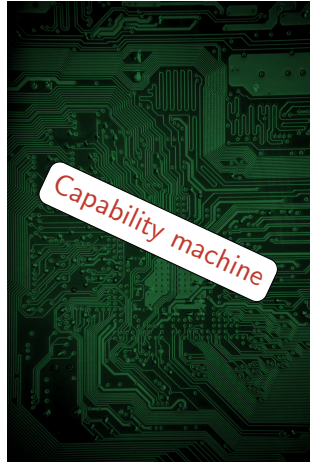


Abstractions all the way down

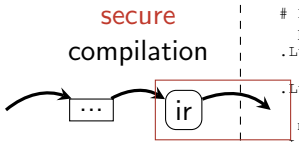



```c
#include <stdio.h>
int main()
{
  printf("Hello, World!");
  return 0;
}
```

```
main:
  .cfi_startproc
# BB#0:
  pushq %rbp
.Ltmp0:
  .cfi\_def\_cfa\_offset 16
.Ltmp1:
  .cfi\_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp2:
  .cfi\_def\_cfa\_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
```

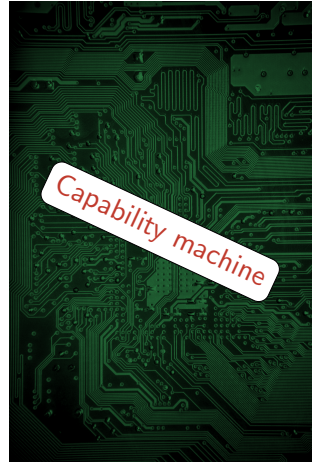


Abstractions all the way down

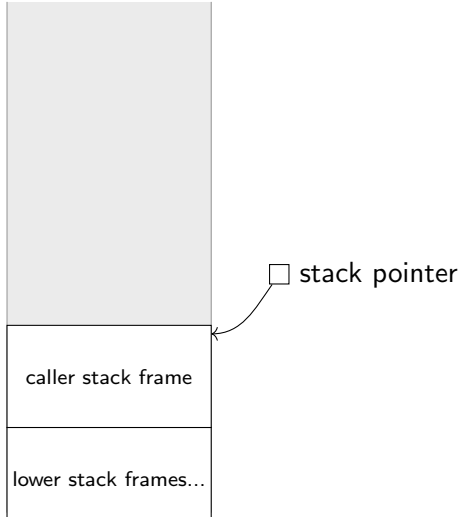



```c
#include <stdio.h>
int main()
{
  printf("Hello, World!");
  return 0;
}
```

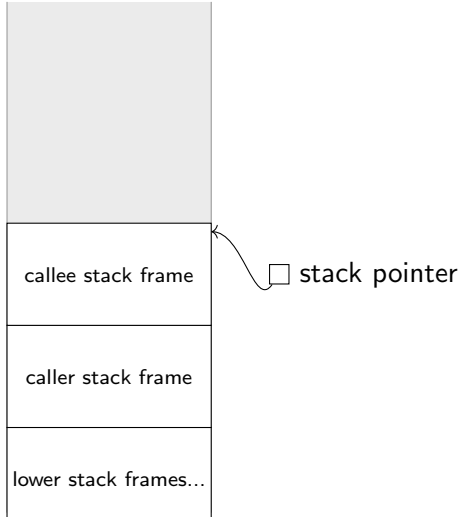
```
main:
  .cfi_startproc
# BB#0:
  pushq %rbp
.Ltmp0:
  .cfi\_def\_cfa\_offset 16
.Ltmp1:
  .cfi\_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp2:
  .cfi\_def\_cfa\_register %rbp
  subq  $16, %rsp
  movabsq $.L.str, %rdi
  movl  $0, -4(%rbp)
  movb  $0, %al
  callq printf
  xorl  %ecx, %ecx
  movl  %eax, -8(%rbp)
  movl  %ecx, %eax
  addq  $16, %rsp
  popq  %rbp
  retq
.Lfunc\_end0:
  .size main, .Lfunc\_end0-main
  .cfi\_endproc
```



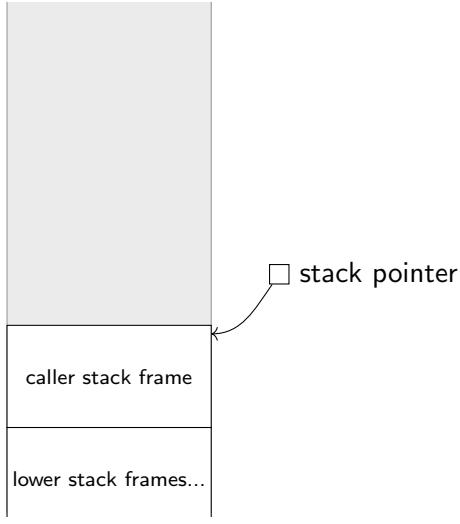
Traditional stack pointers



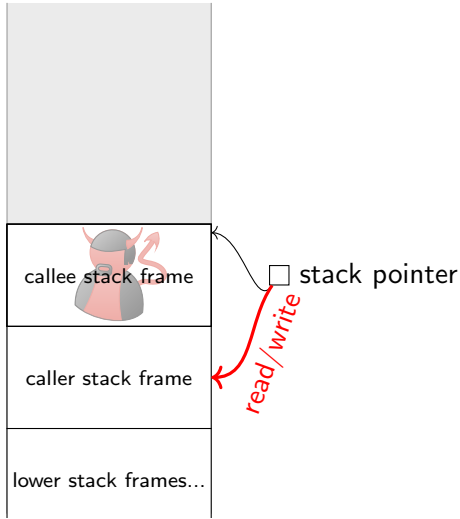
Traditional stack pointers



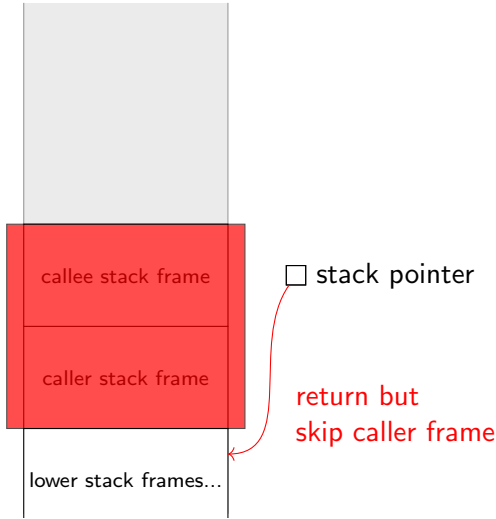
Traditional stack pointers



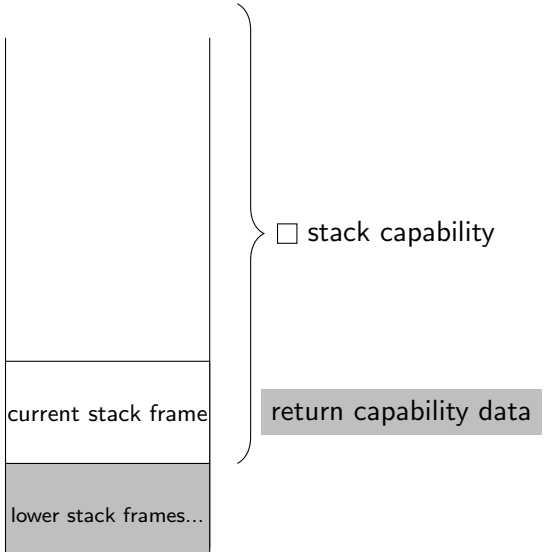
Traditional stack pointers



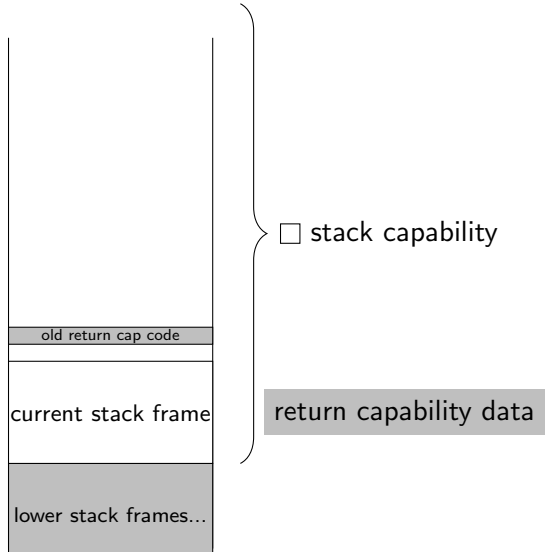
Traditional stack pointers



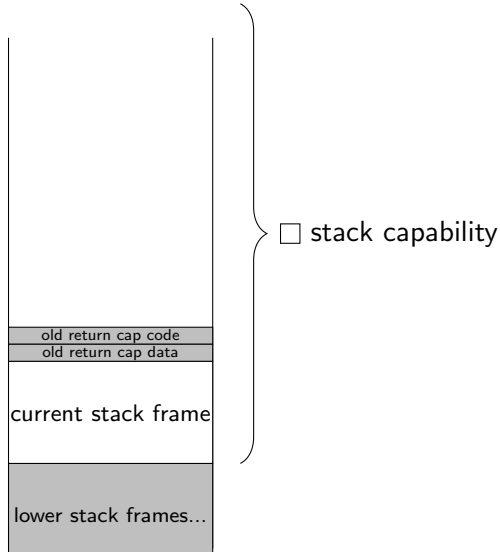
Naive stack and return capabilities



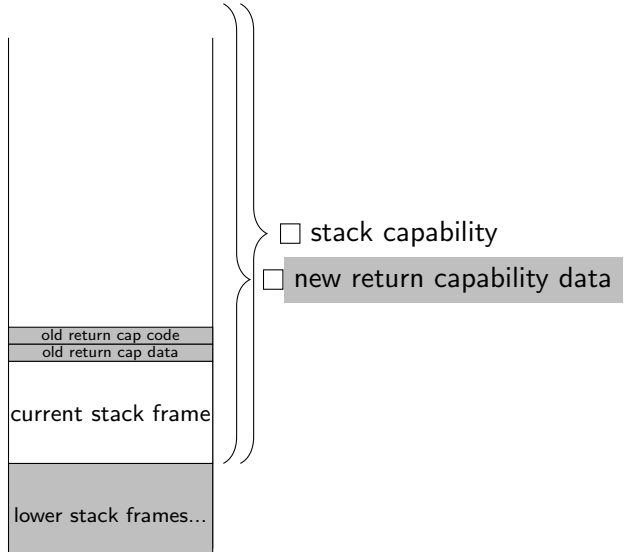
Naive stack and return capabilities



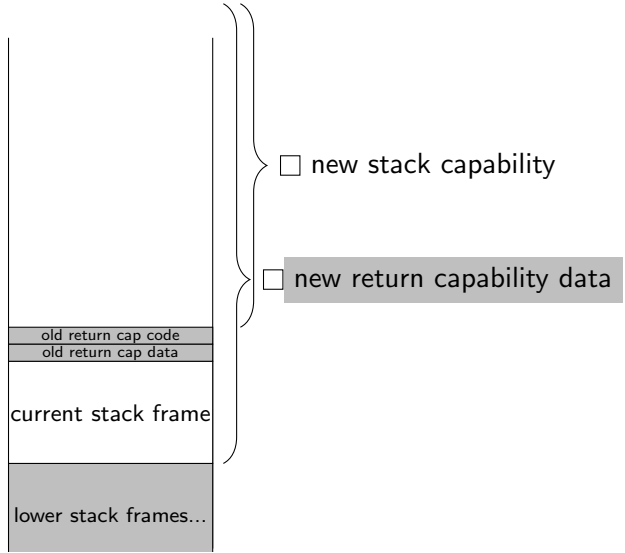
Naive stack and return capabilities



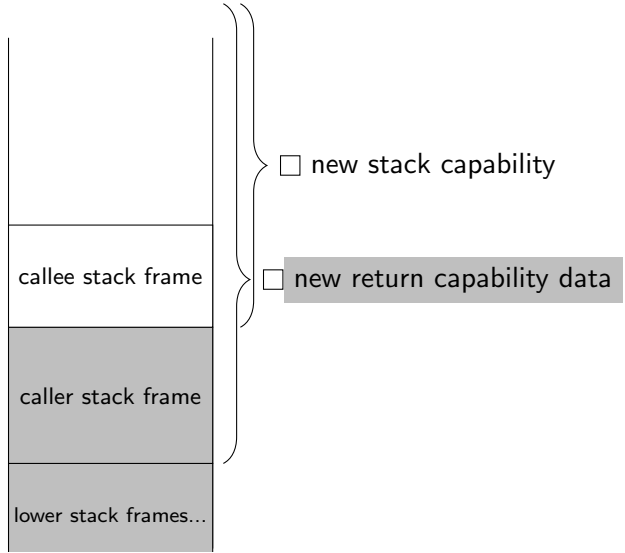
Naive stack and return capabilities



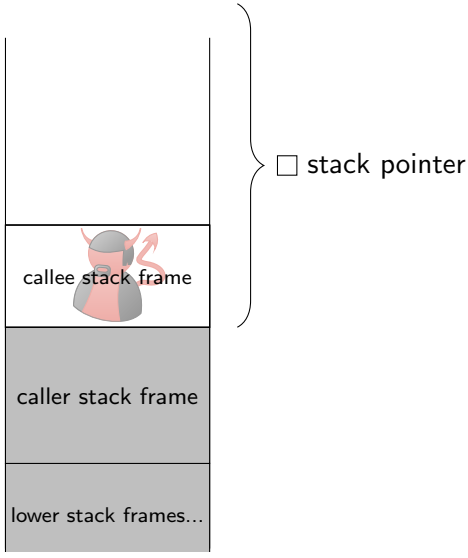
Naive stack and return capabilities



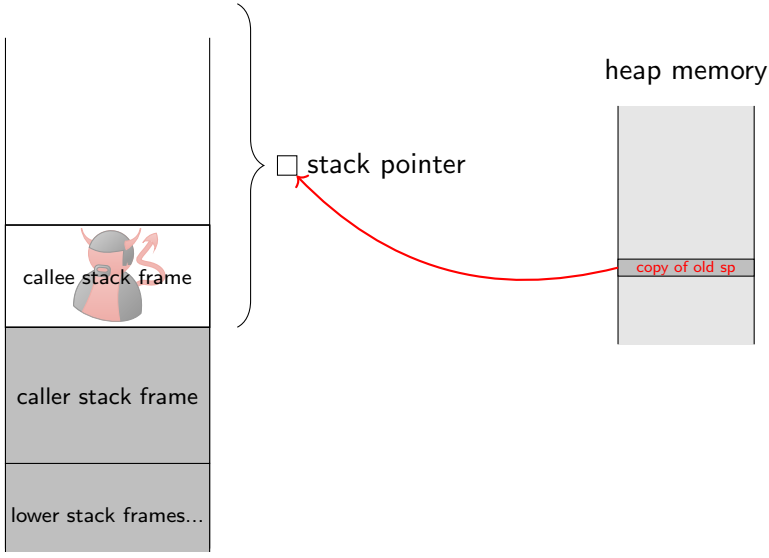
Naive stack and return capabilities



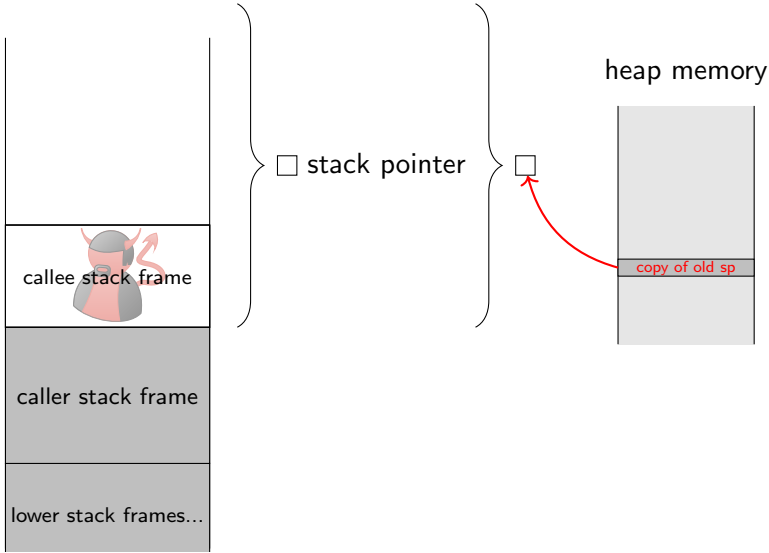
Attack on naive stack and return capabilities



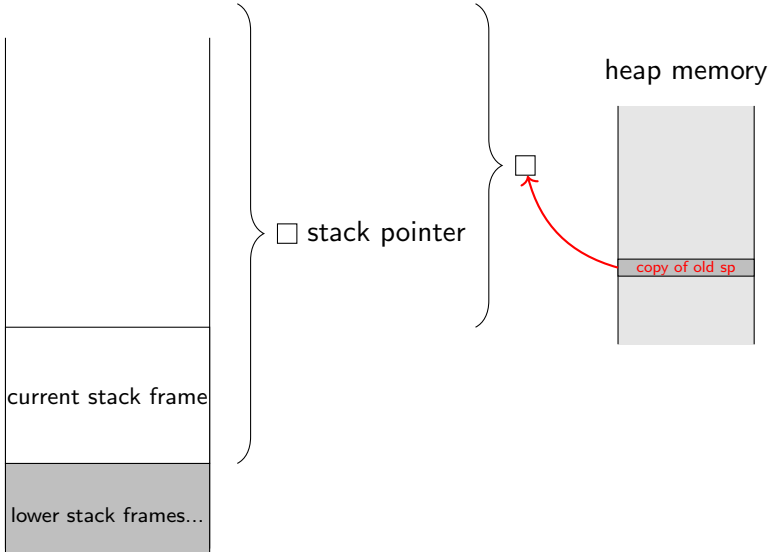
Attack on naive stack and return capabilities



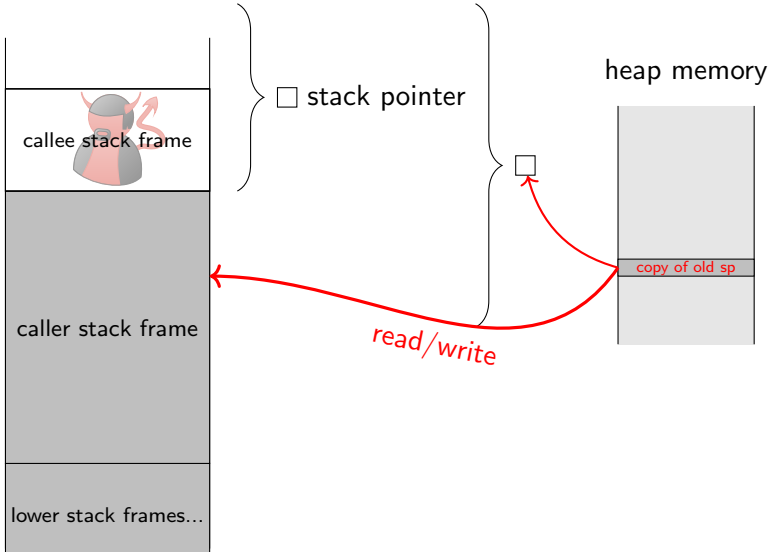
Attack on naive stack and return capabilities



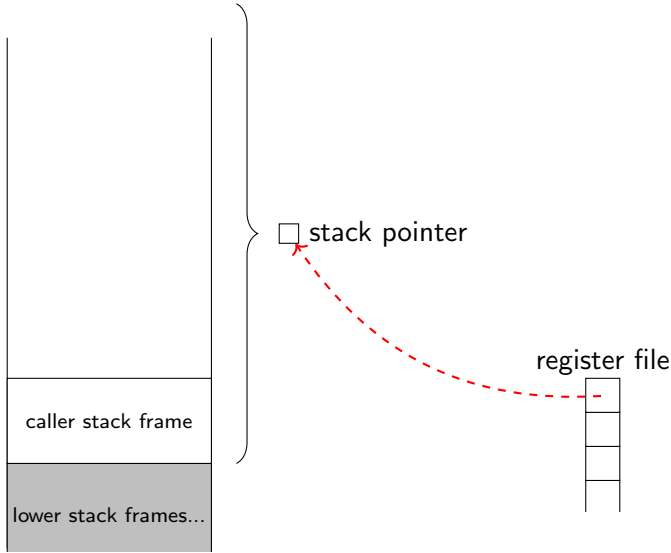
Attack on naive stack and return capabilities



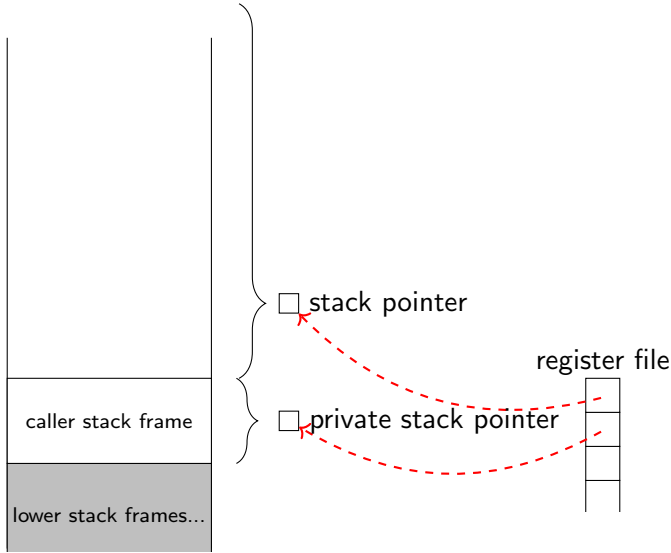
Attack on naive stack and return capabilities



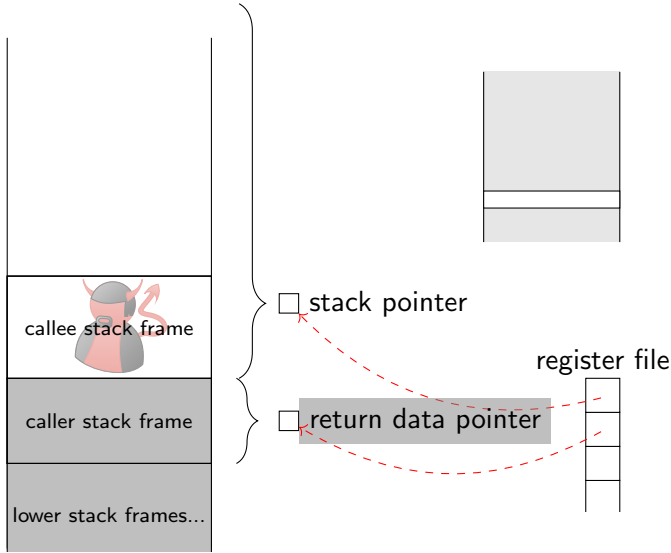
StkTokens prevent the attack



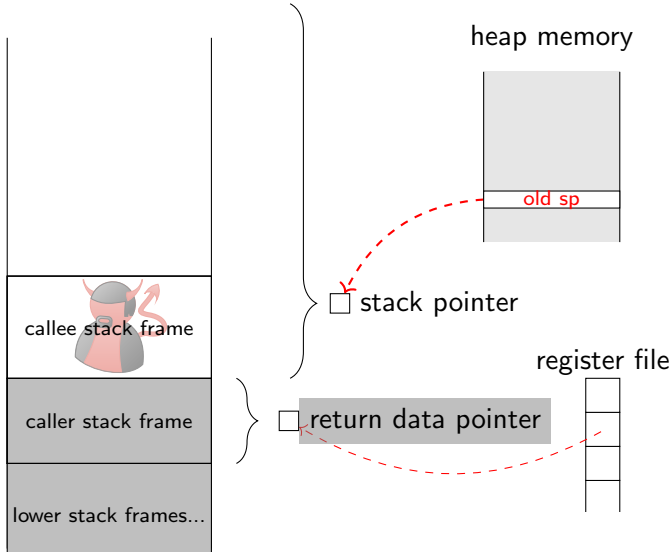
StkTokens prevent the attack



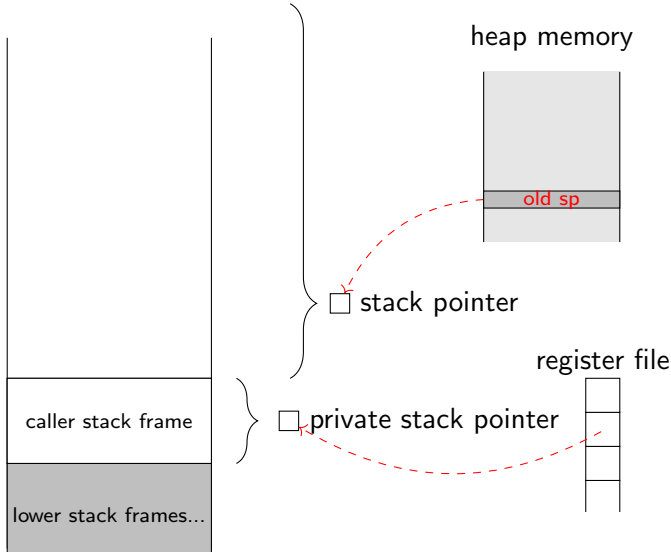
StkTokens prevent the attack



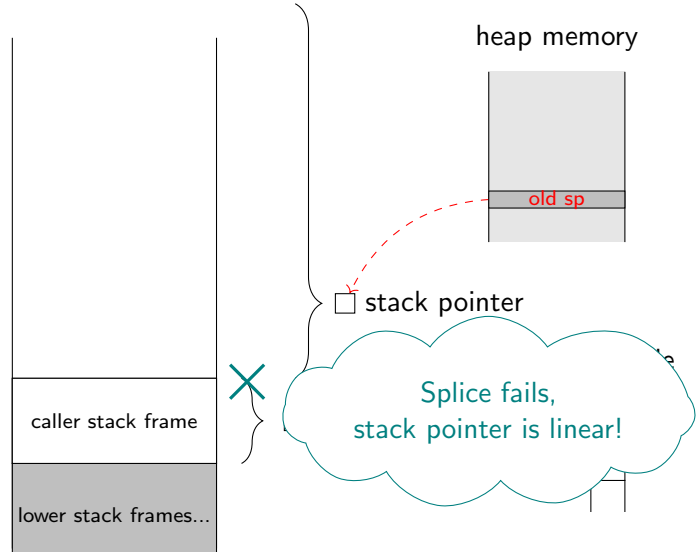
StkTokens prevent the attack



StkTokens prevent the attack



StkTokens prevent the attack



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)

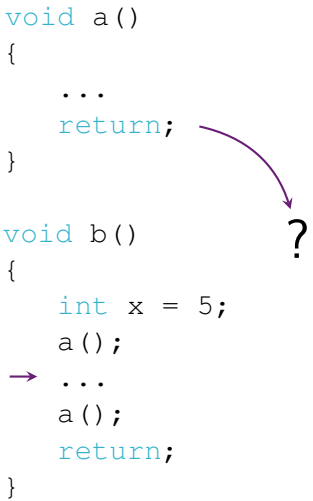
Well-bracketed control flow and local state encapsulation

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

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

Well-bracketed control flow (WBCF)

Well-bracketed control flow and local state encapsulation



Well-bracketed control flow (WBCF)

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
```

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
```

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
```

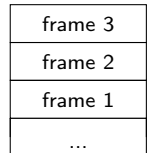
Linear Capability Machine

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
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

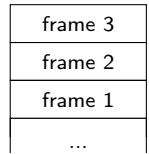
```
move  rtmp1 42
store rstk rtmp1
cca   rstk 1
geta  rtmp1 rstk
cca   rretc 5
move  rtmp1 pc
cca   rtmp1 -20
```

call

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

return

Builtin call stack



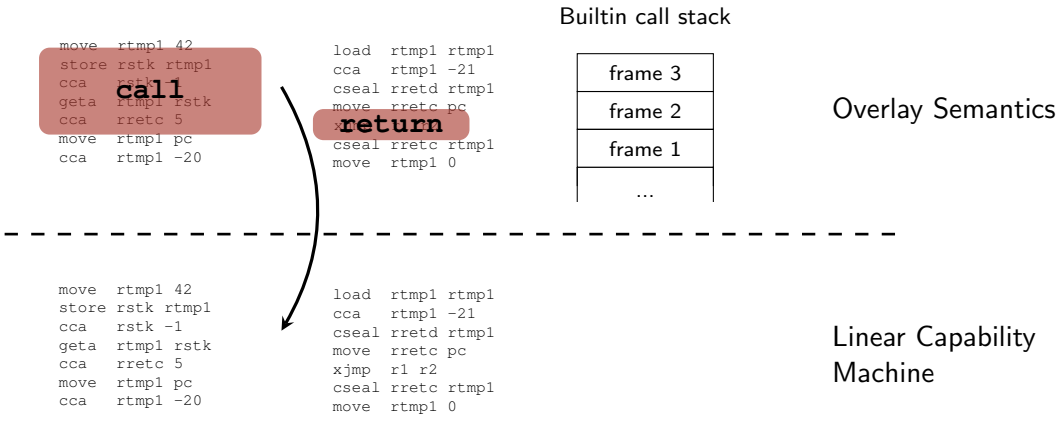
Overlay Semantics

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

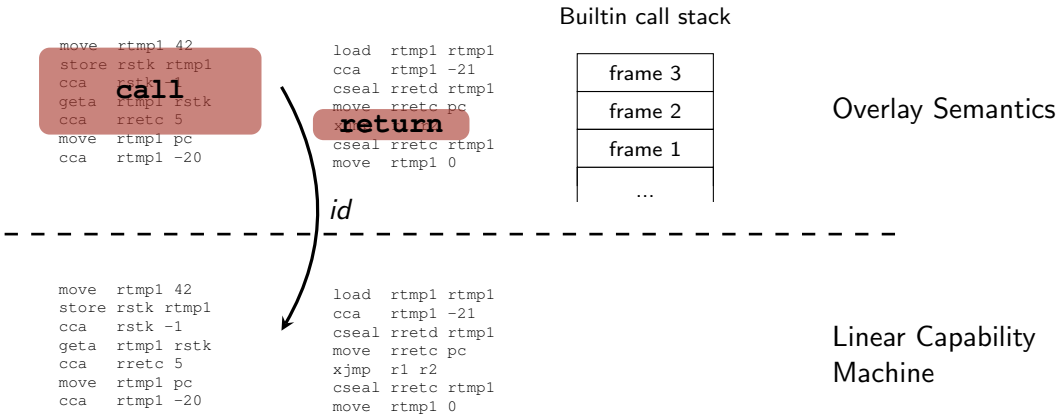
```
load  rtmp1 rtmp1
cca   rtmp1 -21
cseal rretd rtmp1
move  rretc pc
xjmp  r1 r2
cseal rretc rtmp1
move  rtmp1 0
```

Linear Capability Machine

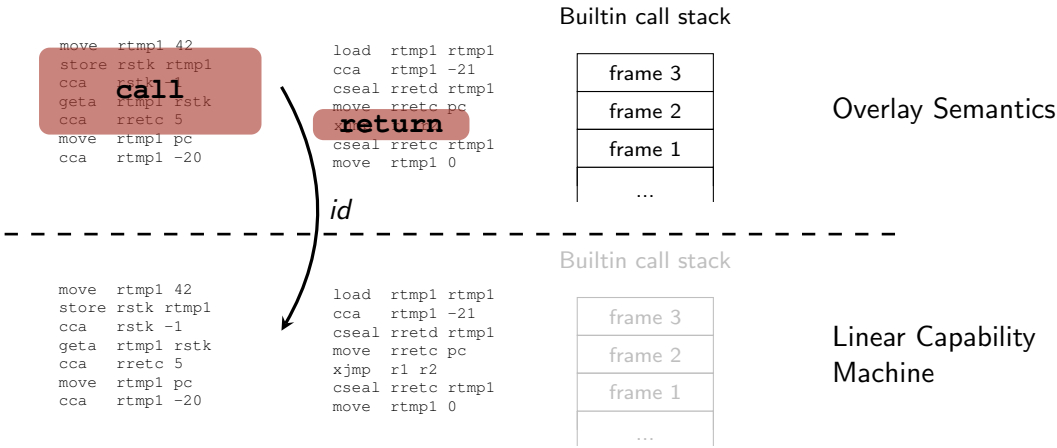
Fully-abstract overlay Semantics



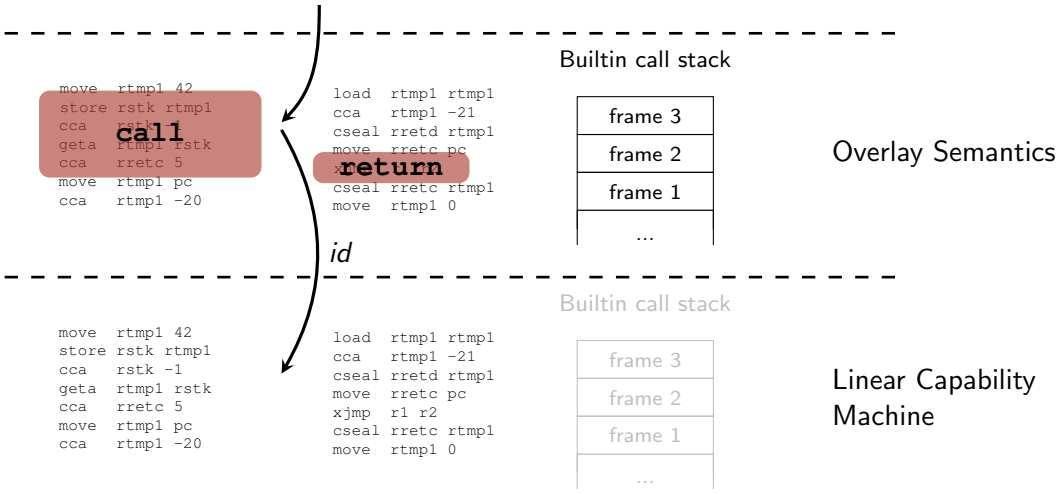
Fully-abstract overlay Semantics



Fully-abstract overlay Semantics



Fully-abstract overlay Semantics

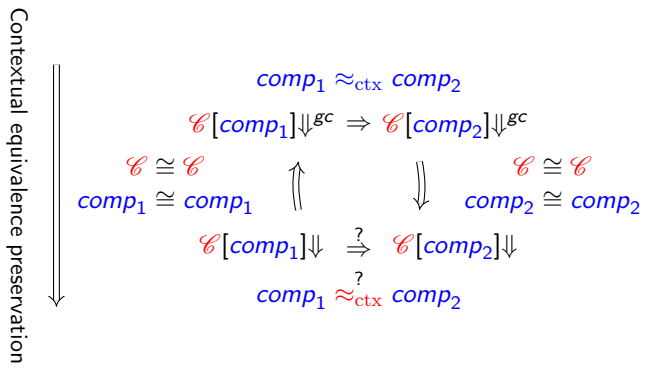


Paper overview

- 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 new 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!

Full-abstraction proof sketch



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.

FAQ

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