Reasoning About a Machine with Local Capabilities

Provably Safe Stack and Return Pointer Management

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

[1]Aarhus University

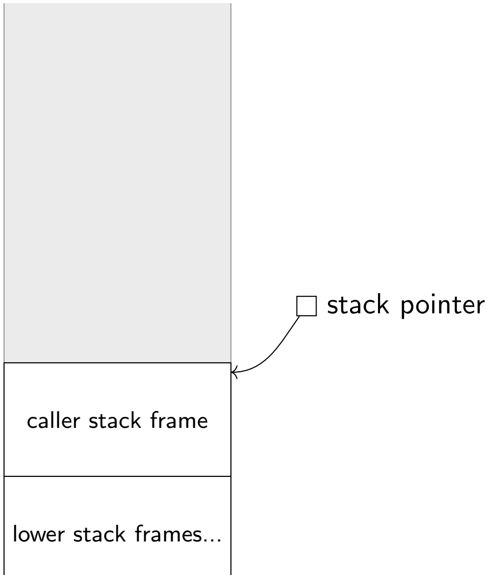
[2]imec-DistriNet, KU Leuven

ESOP, April 17, 2018

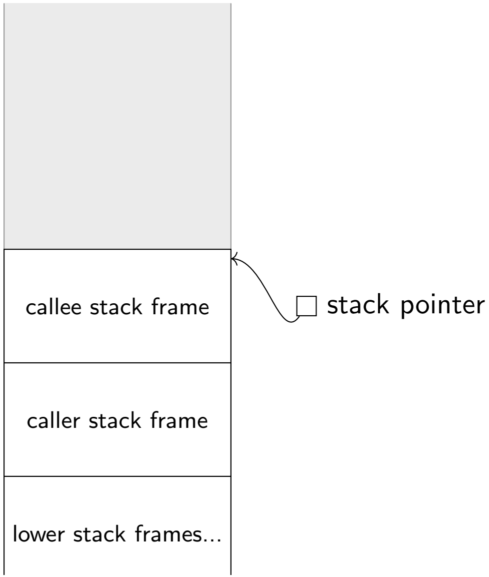
How Do We Reason About Programs Informally

```
let x = ref 0 in
  λf.(x := 0;
      f();
      x := 1;
      f();
      assert(x == 1))
```

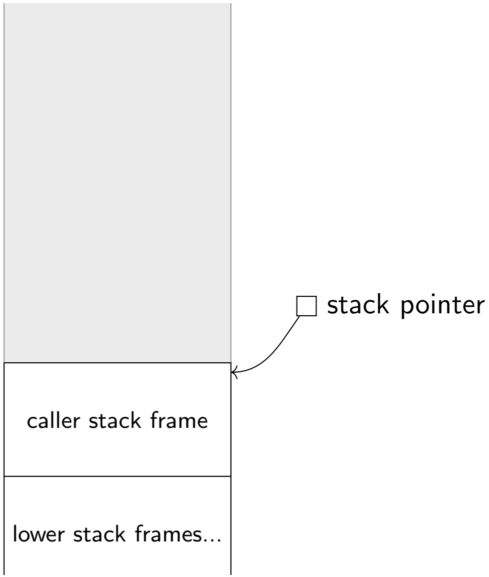
Traditional Stack Pointers



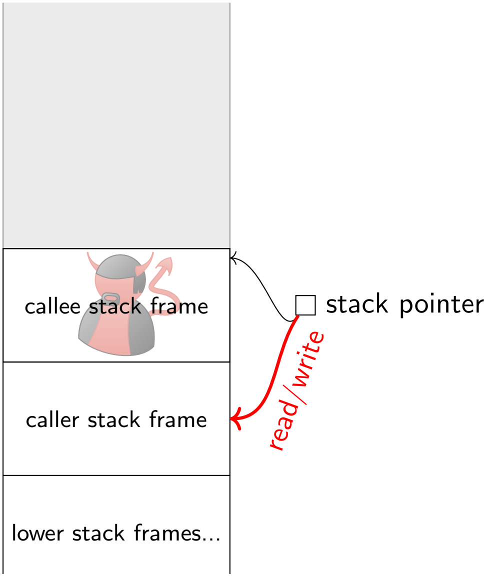
Traditional Stack Pointers



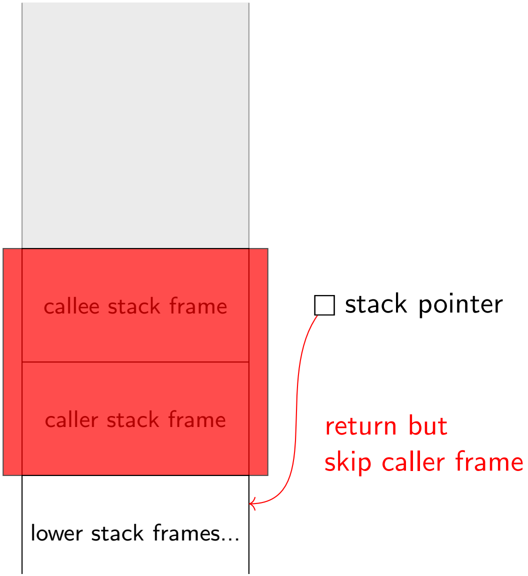
Traditional Stack Pointers



Traditional Stack Pointers



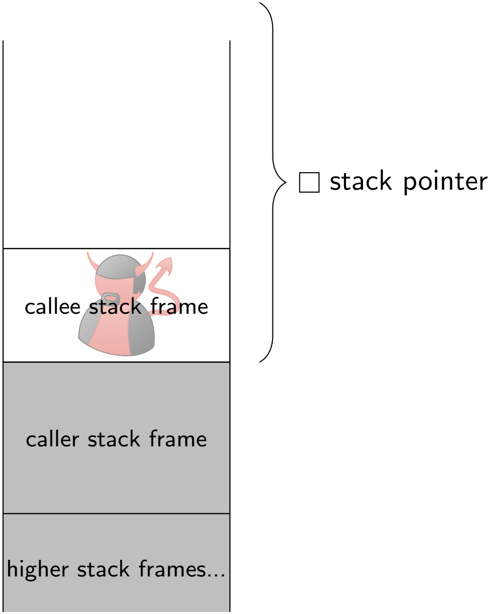
Traditional Stack Pointers



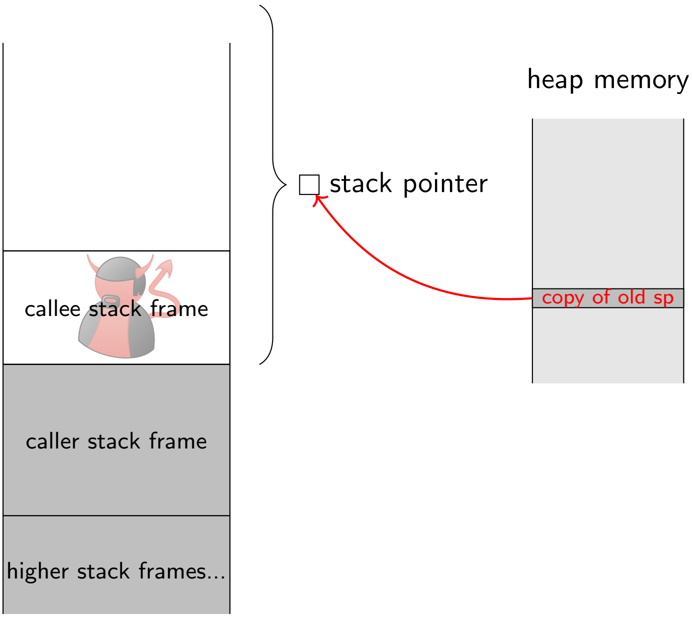
Capability Machine

- Low-level machine
- Capabilities replace pointers
  - Pointer
  - Range of authority
  - Kind of authority
    - read/write/execute
    - enter
- Authority checked dynamically

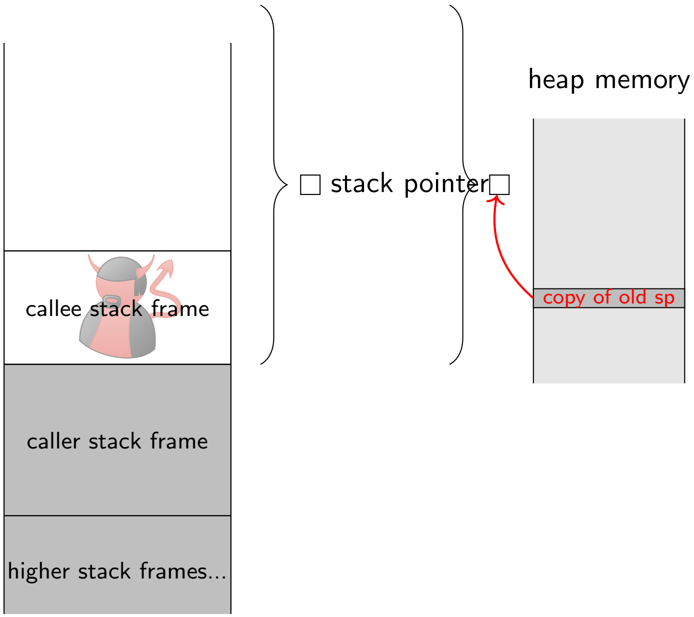
Stack and Return Capabilities: Attack 1



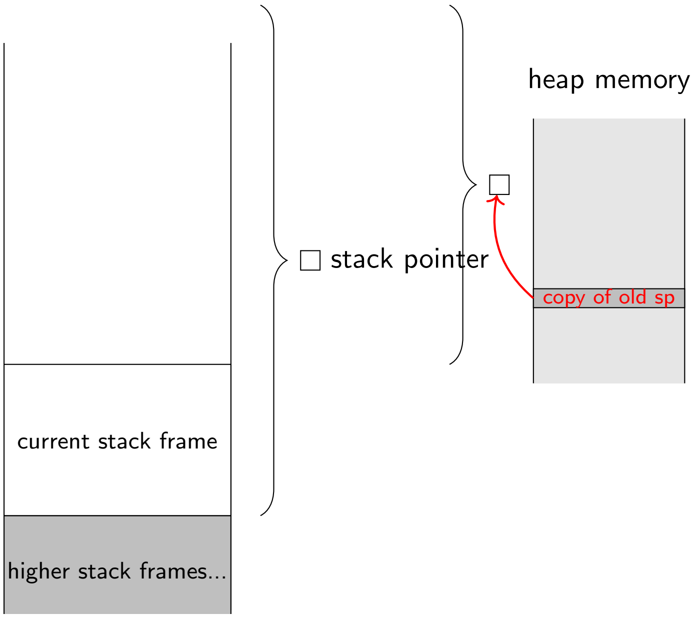
Stack and Return Capabilities: Attack 1



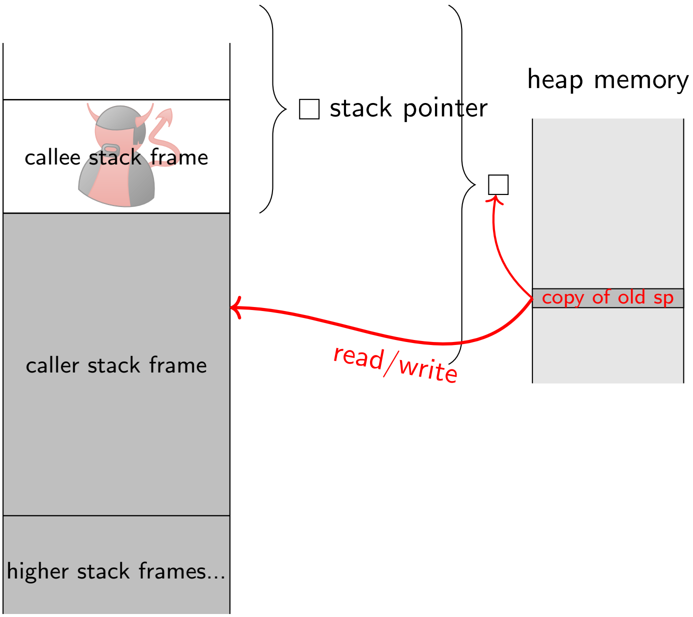
Stack and Return Capabilities: Attack 1



Stack and Return Capabilities: Attack 1



Stack and Return Capabilities: Attack 1



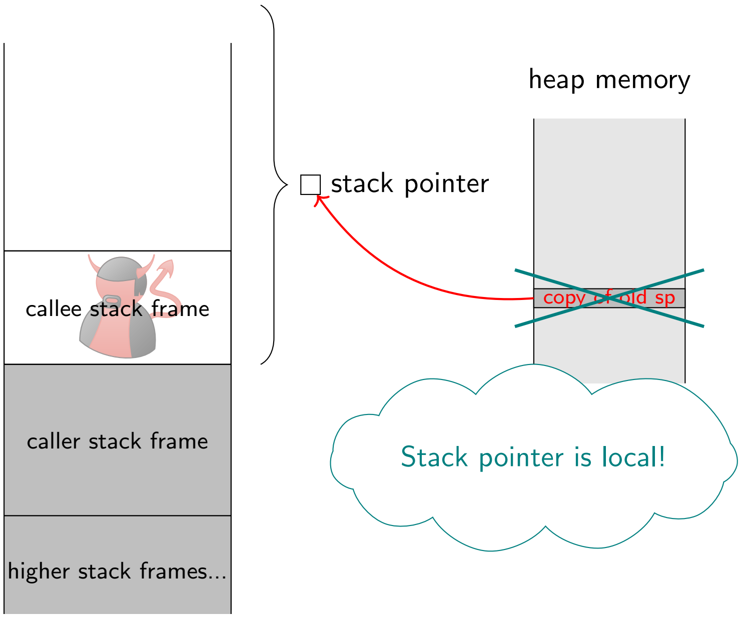
Local Capabilities

- Capabilities tagged with locality (local or global)
- New <u>write-local</u> permission.
- Local capabilities can only be stored by capabilities with <u>write-local</u> permission

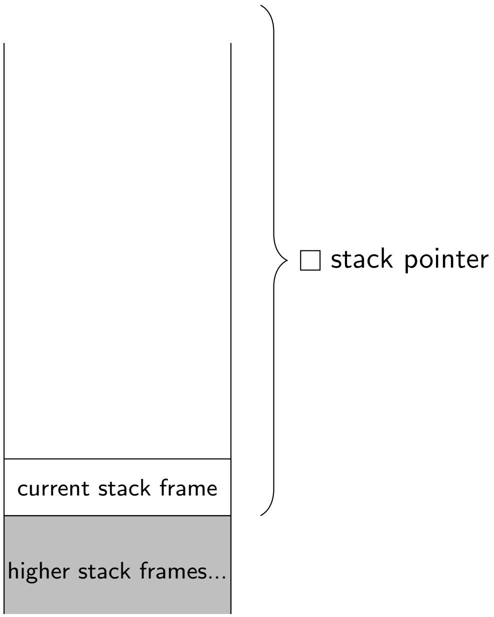
Calling convention highlights

- Stack capability is local with permission read, write-local, and execute.
- Clear stack before passing stack capability to untrusted code.

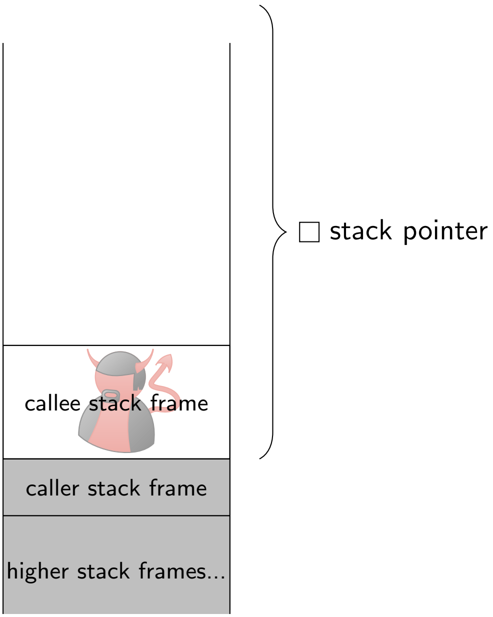
Local Stack Capabilities Prevent Attack 1



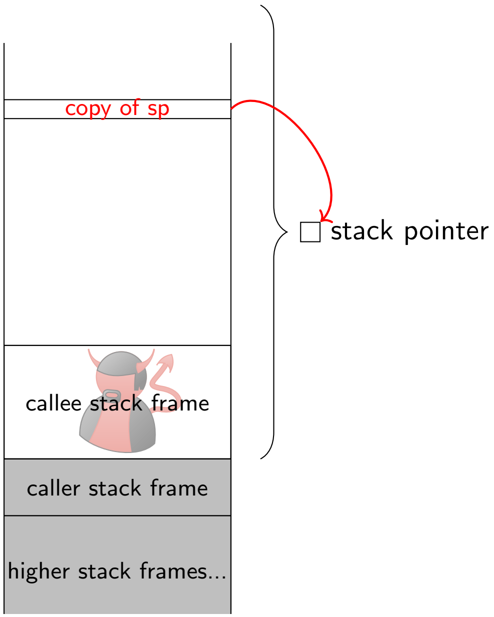
Stack and Return Capabilities: Attack 2



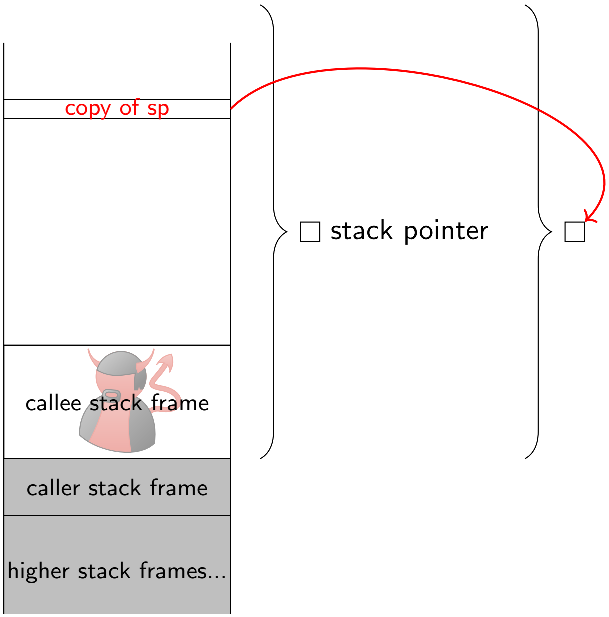
Stack and Return Capabilities: Attack 2



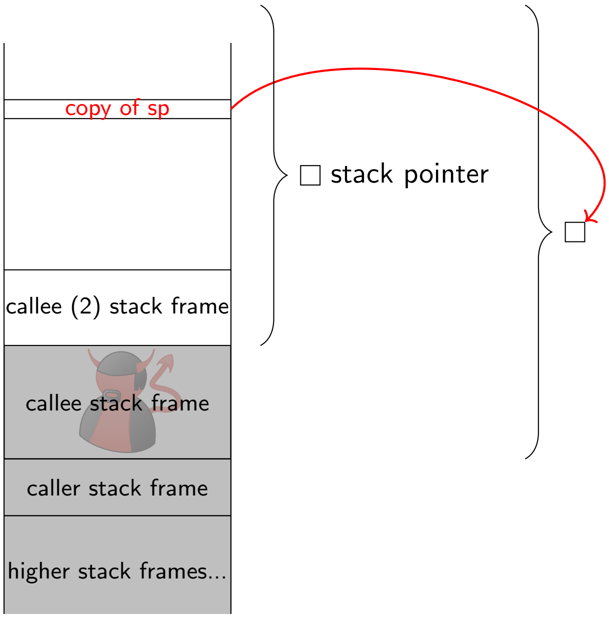
Stack and Return Capabilities: Attack 2



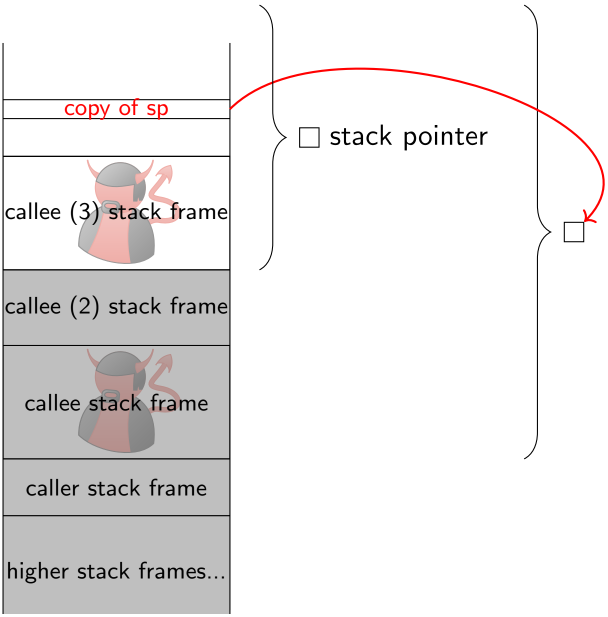
Stack and Return Capabilities: Attack 2



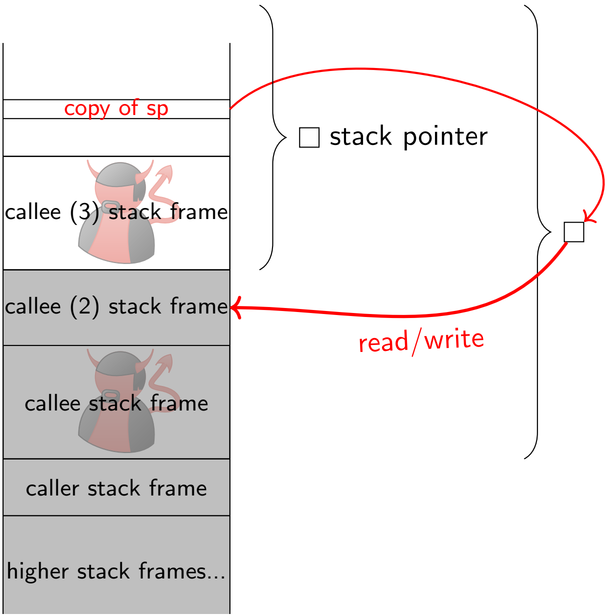
Stack and Return Capabilities: Attack 2



Stack and Return Capabilities: Attack 2



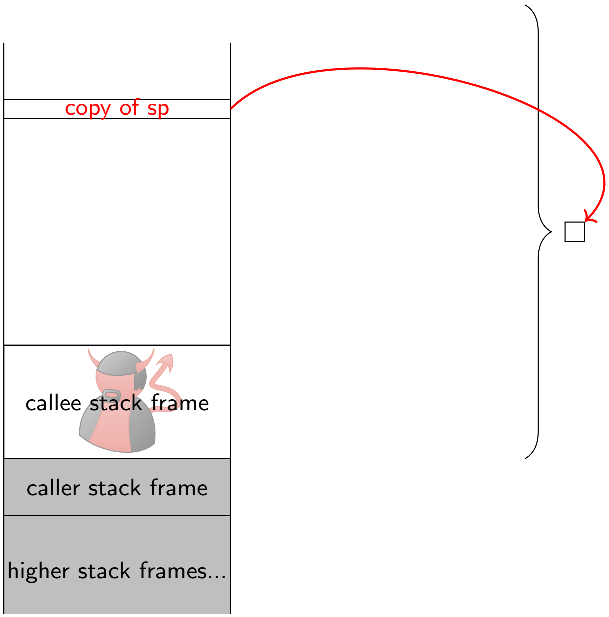
Stack and Return Capabilities: Attack 2



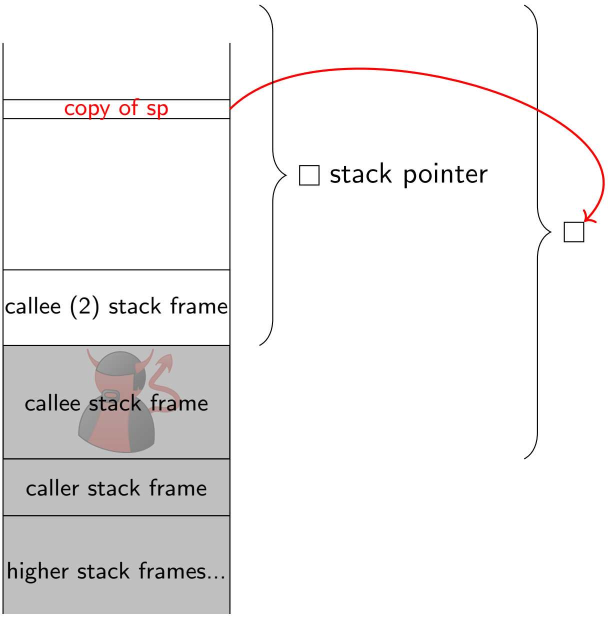
Calling Convention (Continued)

- ...
- Clear stack and non-argument registers before invoking untrusted code.

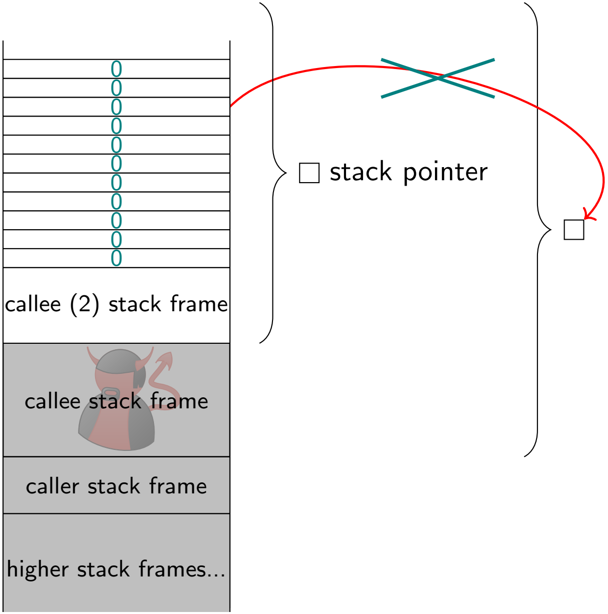
Stack Clearing Prevents Attack 2



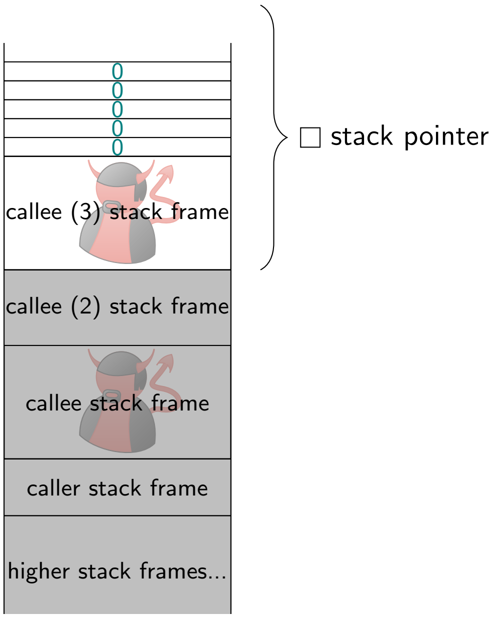
Stack Clearing Prevents Attack 2



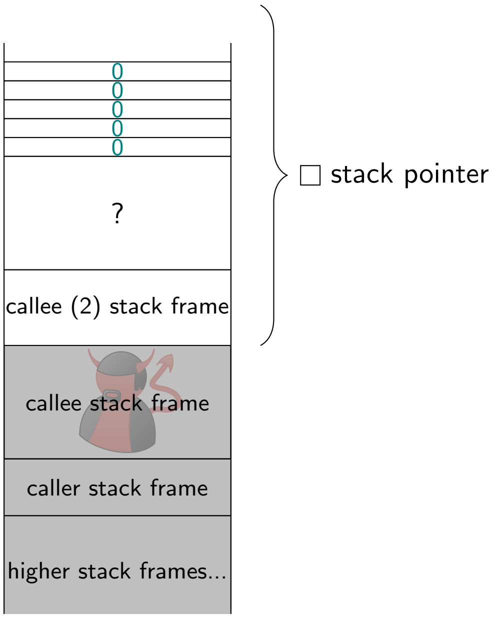
Stack Clearing Prevents Attack 2



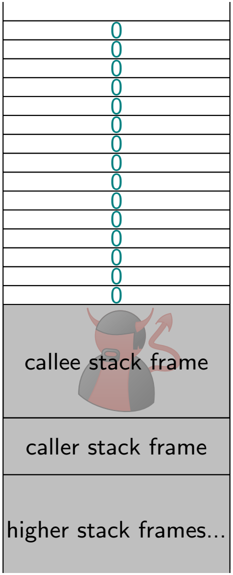
Stack Clearing Prevents Attack 2



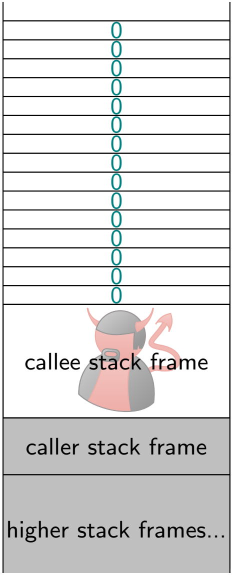
Stack Clearing Prevents Attack 2



Stack Clearing Prevents Attack 2



Stack Clearing Prevents Attack 2



(Full) Calling Convention

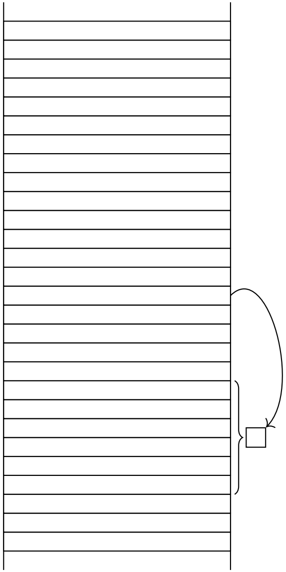
- Initially:
  - Stack capabilitiy local capability with read, write-local, and execute authority.
  - No global write-local capabilities on the machine.
- Prior to returning to untrusted code:
  - Clear the stack.
  - Clear non-return registers.
- Prior to calls to untrusted code:
  - Push activation record to the stack and create enter-capability.
  - Restrict the stack pointer to the unused part and clear that part.
  - Clear non-argument registers.
- Only invoke global call-backs.
- When invoked by untrusted code
  - Make sure the stack pointer has read, write-local and execute authority.

Formalizing the Guarantees of a Capability Machine

- How do we know the calling convention works?
- Unary step-indexed Kripke logical relation over recursive worlds
  - Statement of guarantees probided by the capability machine

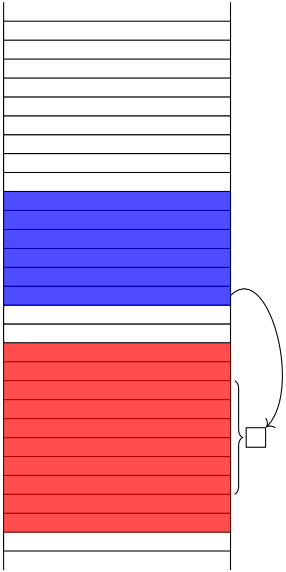
Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code



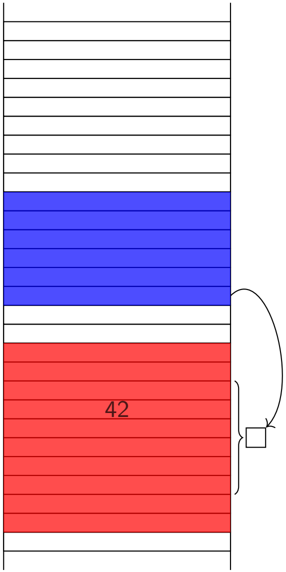
Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code
- World, W
  - Collection of invariants



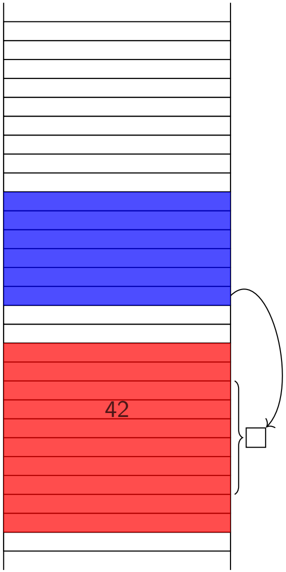
Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code
- World, W
  - Collection of invariants



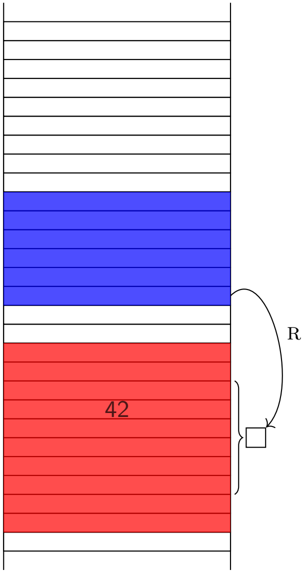
Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$



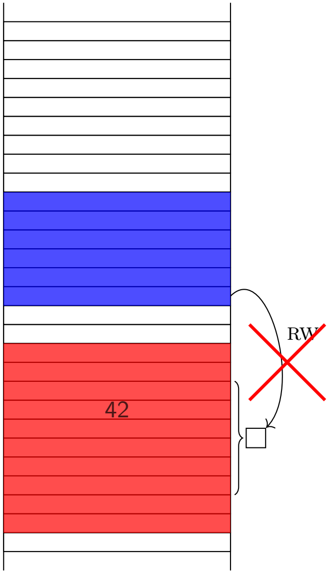
Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$



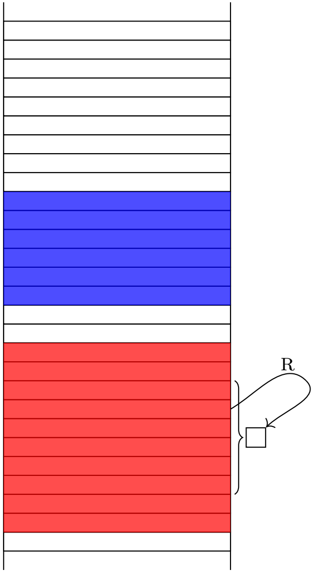
Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code
- World, W
    - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$



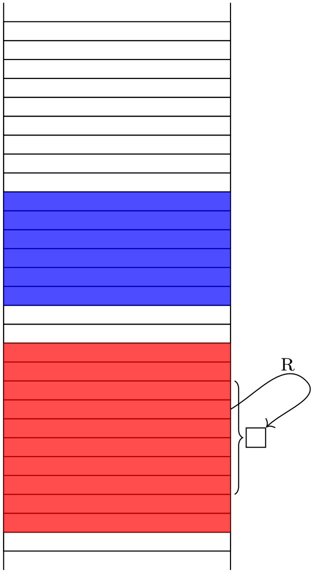
Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$

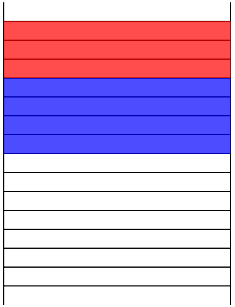


Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$
  - Recursively defined

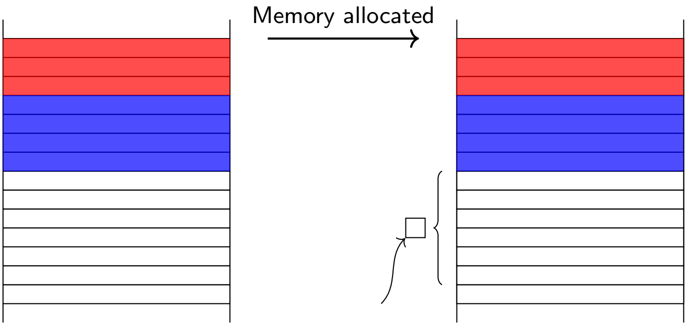


Future Worlds and Invariants, and Recursive Worlds



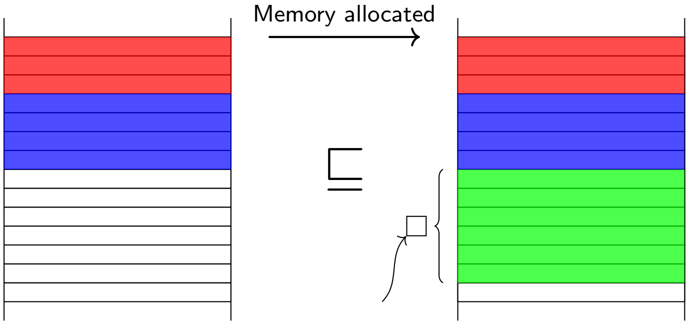
- Memory evolves over time

Future Worlds and Invariants, and Recursive Worlds



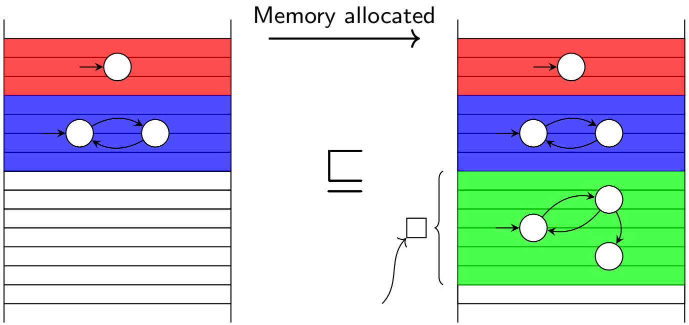
- Memory evolves over time

Future Worlds and Invariants, and Recursive Worlds



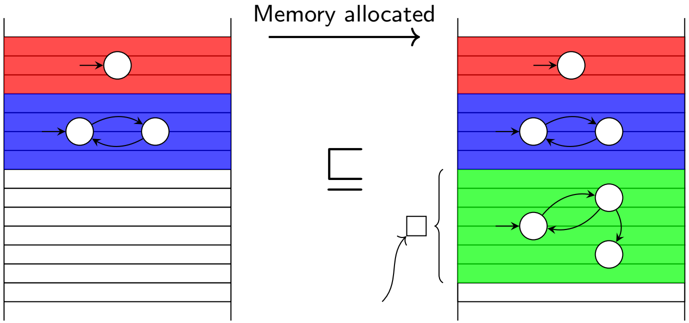
- Memory evolves over time
- Add invariants in future worlds

Future Worlds and Invariants, and Recursive Worlds



- Memory evolves over time
- Add invariants in future worlds
- Invariants as state machines

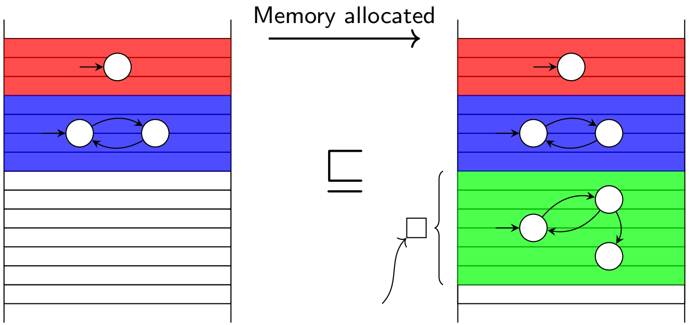
Future Worlds and Invariants, and Recursive Worlds



- Each state contains a predicate of accepted memory segments

$$H \quad : \qquad \mathrm{Pred}(\mathrm{MemSeg})$$

Future Worlds and Invariants, and Recursive Worlds



- Each state contains a predicate of accepted memory segments
- World indexed

$$H : \mathrm{World} \to \mathrm{Pred}(\mathrm{MemSeg})$$

Local Capabilities

`f` is unknown code and `c` is a capability.

```
f(c);
f(1)
```

Local Capabilities

`f` is unknown code and `c` is a capability.

```
f(c);
f(1)
```

- `c` global $\Rightarrow$ available in second invocation of `f`

Local Capabilities

`f` is unknown code and `c` is a capability.

```
f(c);
f(1)
```

- `c` global ⇒ available in second invocation of `f`
- `c` local ⇒ not available in second invocation of `f`

Local Capabilities

`f` is unknown code and `c` is a capability.

```
f(c);
f(1)
```

- `c` global ⇒ available in second invocation of `f`
- `c` local ⇒ not available in second invocation of `f`

Lemma (Double monotonicity of value relation)

- *If* $(n, w) \in \mathcal{V}(W)$ *and* $W' \sqsupseteq^{pub} W$ *then* $(n, w) \in \mathcal{V}(W')$.
- *If* $(n, w) \in \mathcal{V}(W)$ *and* $W' \sqsupseteq^{priv} W$ *and* w *is not a local capability, then* $(n, w) \in \mathcal{V}(W')$.

Fundamental Theorem of Logical Relations

- General statement about the guarantees provided by the capability machine.
- Intuitively: any program is safe as long as it only has access to safe values.

Theorem (Fundamental theorem (simplified))

If

$$(n, (b, e)) \in \mathit{readCond}(g)(W)$$

then

$$(n, ((\mathrm{RX}, g), b, e, a)) \in \mathcal{E}(W)$$

“Awkward Example”

```
let x = ref 0 in
  λf.(x := 0;
      f();
      x := 1;
      f();
      assert(x == 1))
```

Conclusion

- Capability machines can guarantee properties of high-level languages
- Calling convention for well-bracketedness and local-state encapsulation
- Unary step-indexed Kripke logical relation over recursive worlds
    - Formal statement about guarantees provided by capability machine
    - Reasoning about programs in general
- Applied on the “awkward example”

Thank you!