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

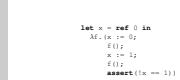
ESOP, April 17, 2018

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

What Does This Program Do?

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


What Does This Program Do?



1. Consider program. Assuming a standard ML semantics we can say what it does.
2. Bind x to freshly allocated reference in a closure that...
3. takes callback f, sets x to 0, calls f, sets x to 1, calls f and finally asserts x points to 1.
4. Note the assumption that when we call f, then we return to a specific program point. This is what we call well-bracketedness and we assume we have this in many programming languages.
5. However, in order to execute this code, we need to compile it to assembly.
6. How is well-bracketedness guaranteed? In particular, how is it guaranteed if f is a piece of code we do not trust (maybe handwritten assembly).

What Does This Program Do?

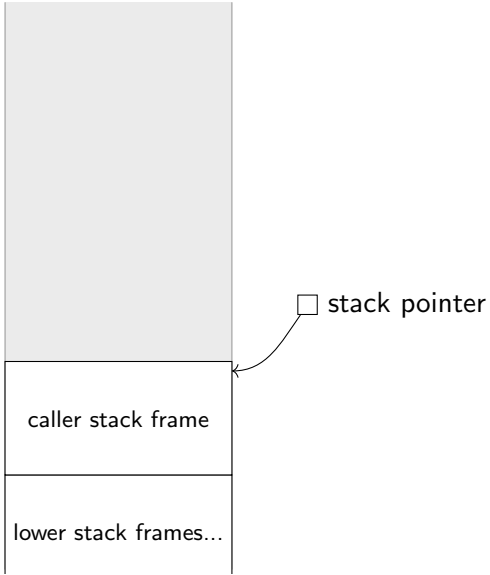
```
let x = ref 0 in
  λf.(x := 0;
      f();
      x := 1;
      f();
      assert(!x == 1))
```

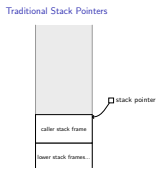

What Does This Program Do?

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

1. We present a calling convention for capability machines that provide well-bracketedness and local state encapsulation as well as a logical relation that allows us to reason about such programs.
2. Let's first consider how stack pointers traditionally are handled.

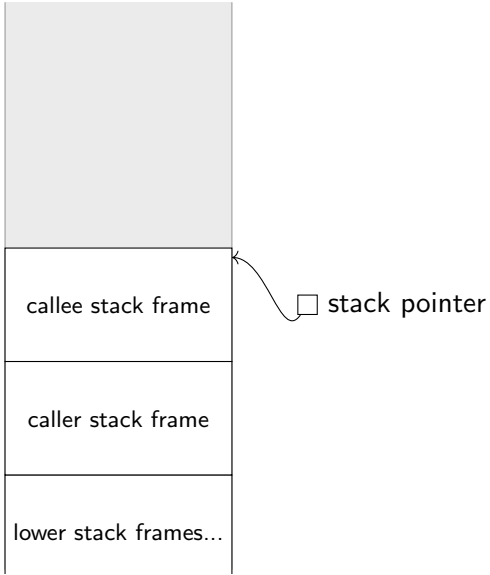
Traditional Stack Pointers

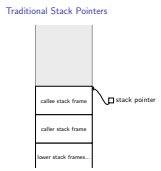




1. Simply put, a caller calls a function which
2. pushes a new stack frame on the stack the callee uses for its execution.
3. When the callee is done, then it returns to the caller by popping its stack frams.

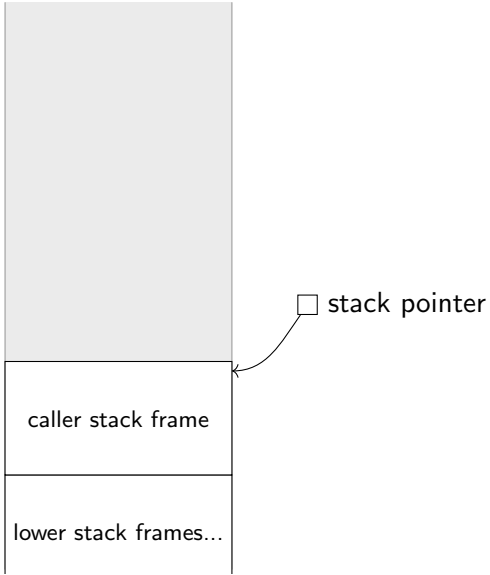
Traditional Stack Pointers

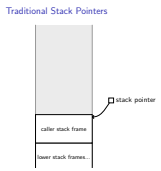




1. Simply put, a caller calls a function which
2. pushes a new stack frame on the stack the callee uses for its execution.
3. When the callee is done, then it returns to the caller by popping its stack frams.

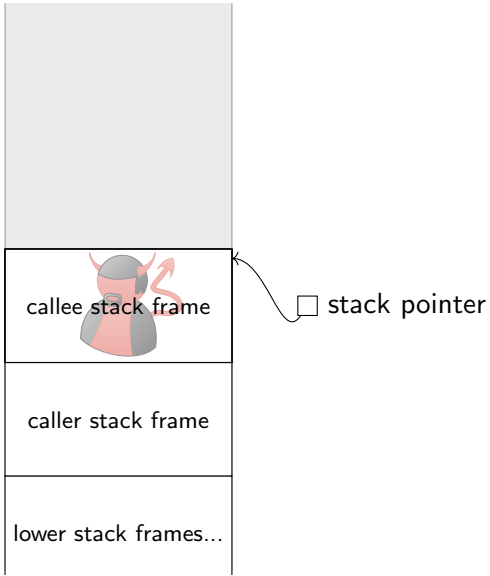
Traditional Stack Pointers

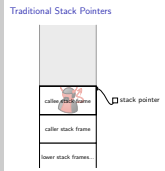




1. Simply put, a caller calls a function which
2. pushes a new stack frame on the stack the callee uses for its execution.
3. When the callee is done, then it returns to the caller by popping its stack frams.

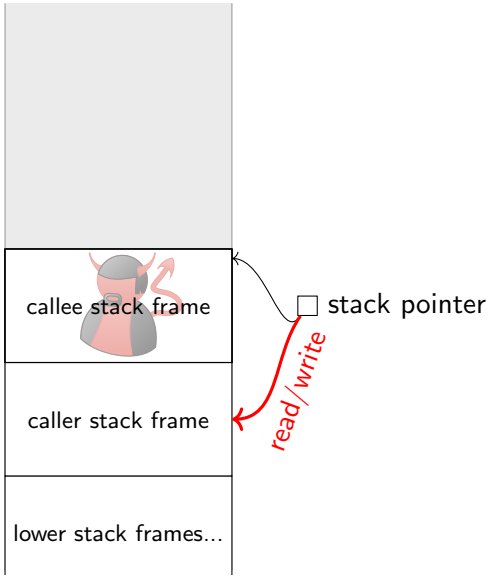
Traditional Stack Pointers

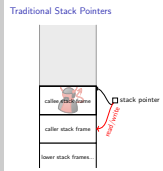




1. If callee (evil) assembly code with no intention to follow the CC, then there are multiple ways for them to break things:

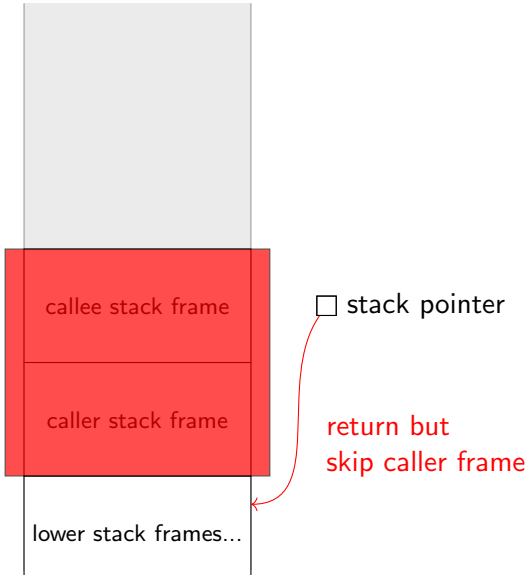
Traditional Stack Pointers

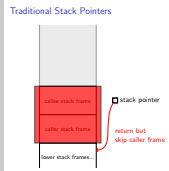




1. If callee (evil) assembly code with no intention to follow the CC, then there are multiple ways for them to break things:
2. Read or write directly from or to the caller's stack frame, breaking local-state encapsulation

Traditional Stack Pointers



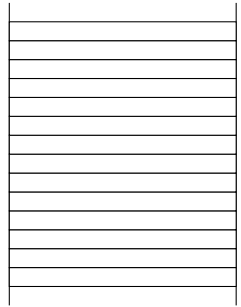


1. If callee (evil) assembly code with no intention to follow the CC, then there are multiple ways for them to break things:
2. Read or write directly from or to the caller's stack frame, breaking local-state encapsulation
3. Skip the caller's stack frame and return to one further down breaking well-bracketedness.
4. Clearly we need some kind of low-level enforcement mechanism.

Capability Machine

- Low-level machine

Memory



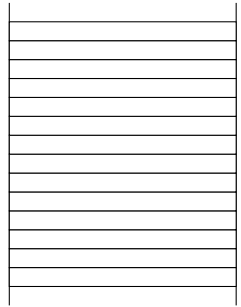
Capability Machine
Low-level machine
Memory

1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.

Capability Machine

- Low-level machine
- Capabilities replace pointers

Memory

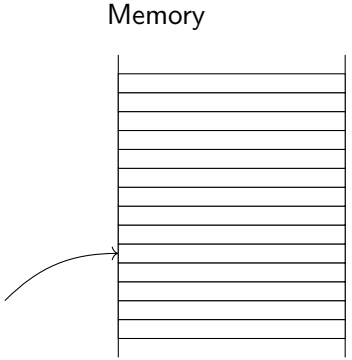


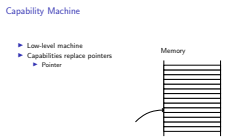
Capability Machine
- Low-level machine
- Capabilities replace pointers
Memory

1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.

Capability Machine

- Low-level machine
- Capabilities replace pointers
  - Pointer

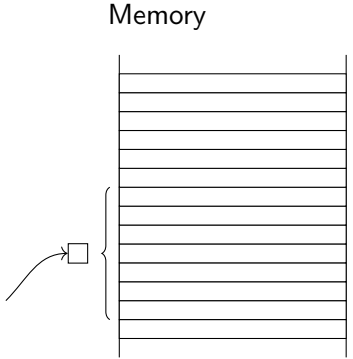


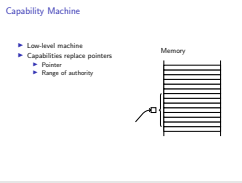


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.

Capability Machine

- Low-level machine
- Capabilities replace pointers
  - Pointer
  - Range of authority

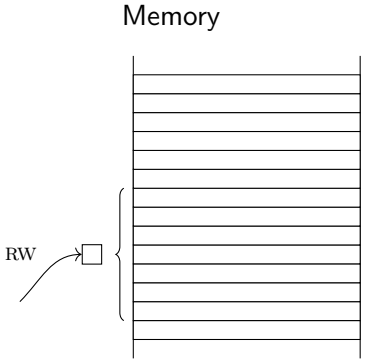


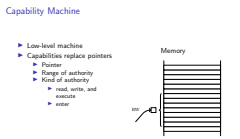


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.

Capability Machine

- Low-level machine
- Capabilities replace pointers
  - Pointer
  - Range of authority
  - Kind of authority
    - read, write, and execute
    - enter

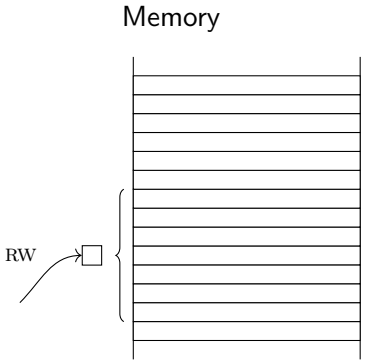


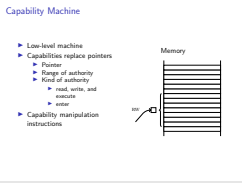


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.
4. Roughly two kinds of capabilities:
5. Memory capabilities, allows you to do all the standard memory operations.
6. Provides encapsulation mechanism which allows seperation of security domains.
7. Can not be used for anything but jump, when jumped to becomes read/execute.

Capability Machine

- Low-level machine
- Capabilities replace pointers
  - Pointer
  - Range of authority
  - Kind of authority
    - read, write, and execute
    - enter
- Capability manipulation instructions

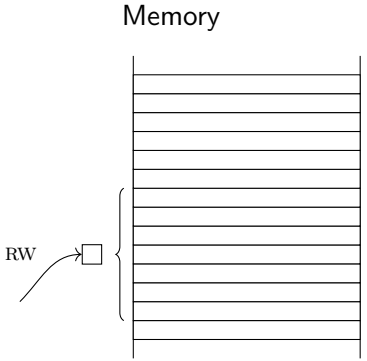


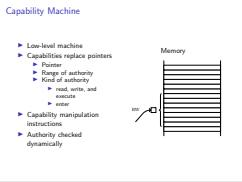


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.
4. Roughly two kinds of capabilities:
5. Memory capabilities, allows you to do all the standard memory operations.
6. Provides encapsulation mechanism which allows seperation of security domains.
7. Can not be used for anything but jump, when jumped to becomes read/execute.

Capability Machine

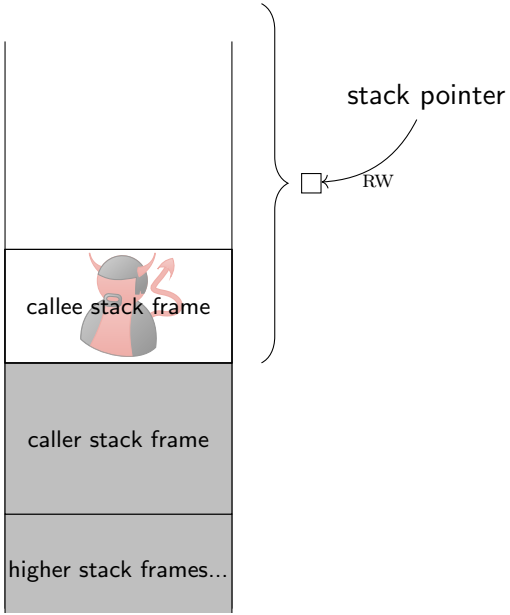
- Low-level machine
- Capabilities replace pointers
  - Pointer
  - Range of authority
  - Kind of authority
    - read, write, and execute
    - enter
- Capability manipulation instructions
- Authority checked dynamically

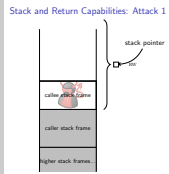




1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.
4. Roughly two kinds of capabilities:
5. Memory capabilities, allows you to do all the standard memory operations.
6. Provides encapsulation mechanism which allows seperation of security domains.
7. Can not be used for anything but jump, when jumped to becomes read/execute.

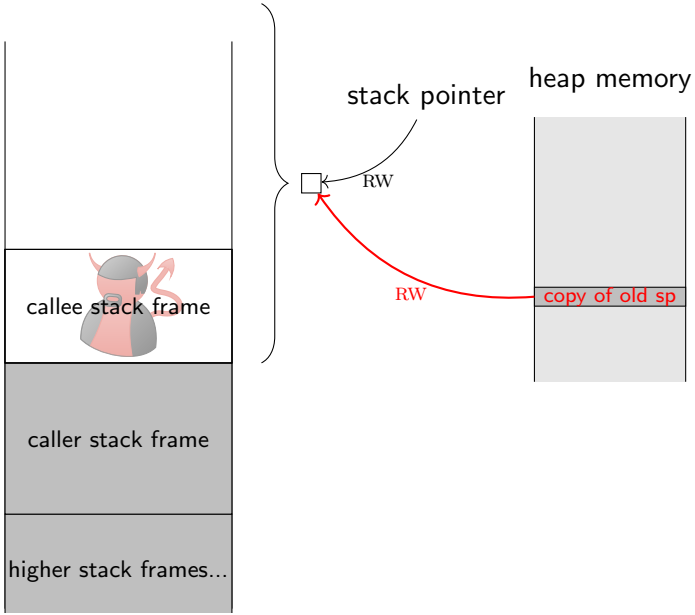
Stack and Return Capabilities: Attack 1

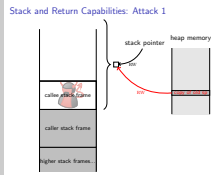




1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.

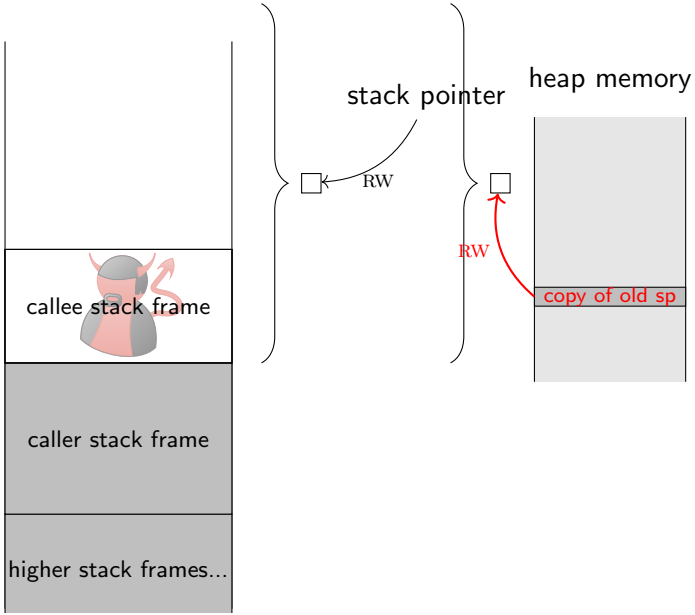
Stack and Return Capabilities: Attack 1

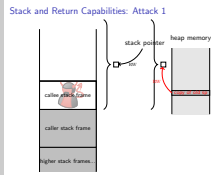




1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.

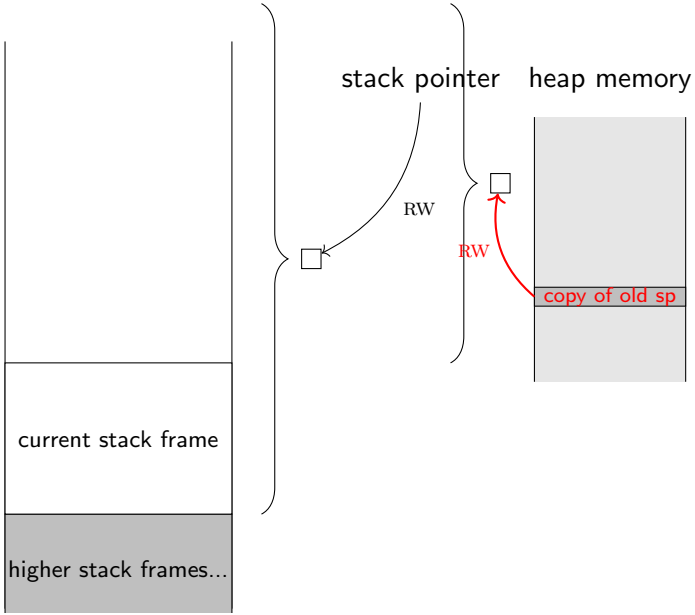
Stack and Return Capabilities: Attack 1

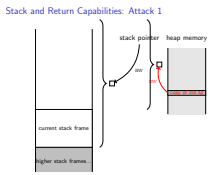




1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.
3. Upon return the callee regains its stack capability which has authority over the callee stack frame and everything above.
4. The caller pushes some important things on the stack and calls the untrusted code again. With a smaller stack pointer.

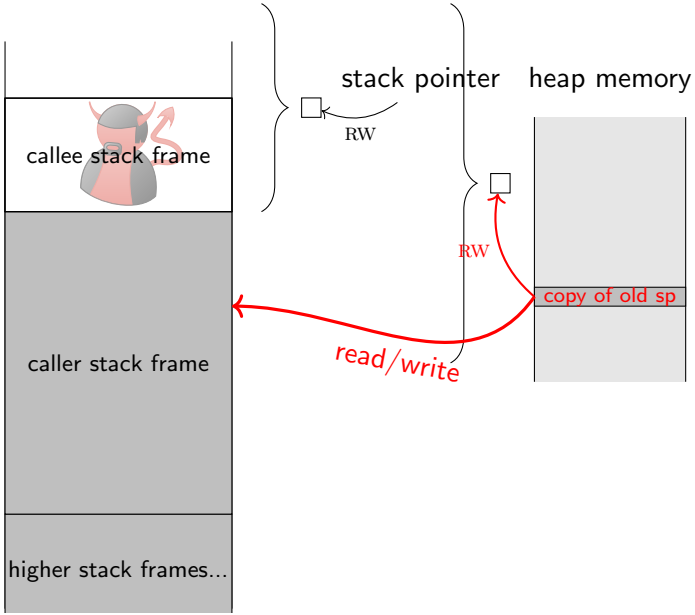
Stack and Return Capabilities: Attack 1

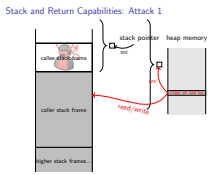




1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.
3. Upon return the callee regains its stack capability which has authority over the callee stack frame and everything above.
4. The caller pushes some important things on the stack and calls the untrusted code again. With a smaller stack pointer.
5. The stack pointer the caller gives the untrusted code cannot be used to access the callee stack frame, but because the untrusted code stored the old stack pointer, it now has access to part of the callee's stack frame.
6. Again breaking local state encapsulation.
7. Need a way to make sure stack pointer is not stored for later use.

Stack and Return Capabilities: Attack 1





1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.
3. Upon return the callee regains its stack capability which has authority over the callee stack frame and everything above.
4. The caller pushes some important things on the stack and calls the untrusted code again. With a smaller stack pointer.
5. The stack pointer the caller gives the untrusted code cannot be used to access the callee stack frame, but because the untrusted code stored the old stack pointer, it now has access to part of the callee's stack frame.
6. Again breaking local state encapsulation.
7. Need a way to make sure stack pointer is not stored for later use.

Local Capabilities

CHERI inspired

- Capabilities tagged with locality (local or global)
- New *write-local* permission
- Local capabilities can only be stored by capabilities with *write-local* permission

Local Capabilities
CHERI inspired
- Capabilities tagged with locality (local or global)
- New *write-local* permission
- Local capabilities can only be stored by capabilities with *write-local* permission

1. To revoke a capability, we need to find it in memory which means we need access + need to search the entire memory.
2. Restricted where local capabilities can be stored. restricts where we need to look for a capability.
3. We define a calling convention. In order to prevent attack 1, we do the follwoing.

Local Capabilities

CHERI inspired

- Capabilities tagged with locality (local or global)
- New *write-local* permission
- Local capabilities can only be stored by capabilities with *write-local* permission

Calling convention

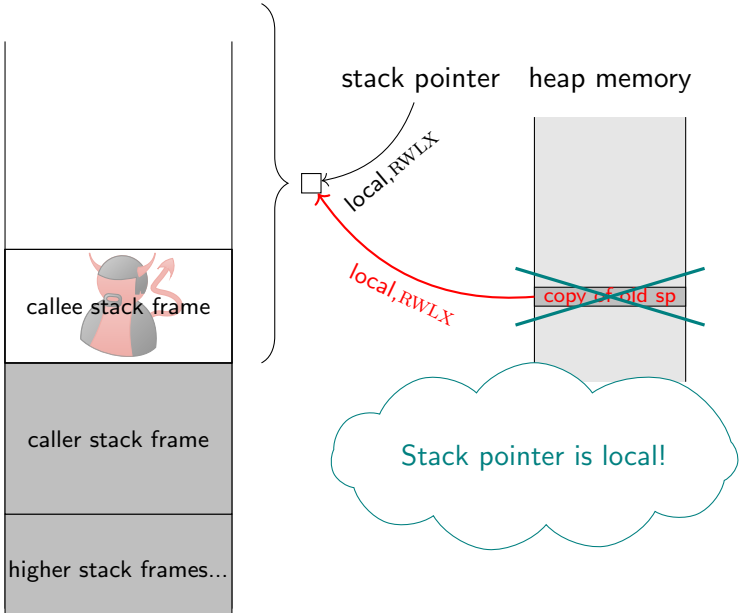
- Stack capability is local with permission read, write-local, and execute.
- No global write-local capabilities.

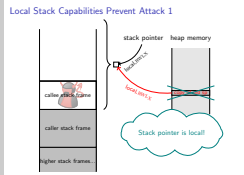
Local Capabilities
CHERI inspired
- Capabilities tagged with locality (local or global)
- New *write-local* permission
- Local capabilities can only be stored by capabilities with *write-local* permission

Calling convention
- Stack capability is local with permission read, write-local, and execute.
- No global write-local capabilities.

1. To revoke a capability, we need to find it in memory which means we need access + need to search the entire memory.
2. Restricted where local capabilities can be stored. restricts where we need to look for a capability.
3. We define a calling convention. In order to prevent attack 1, we do the follwoing.
4. Local stack capability cannot be stored on the heap. We need to be able to store old stack pointers somewhere, traditionally stack.
5. Global write-local capabilities would undermine the entire idea as it would allow local capabilities to be stored indirectly.

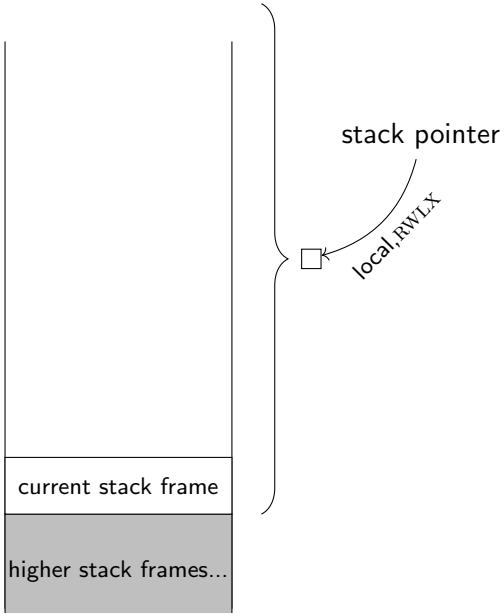
Local Stack Capabilities Prevent Attack 1

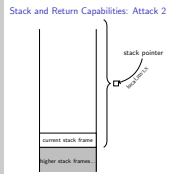




In the attack from before, when the attacker attempts to store the stack capability on the heap, then the machine checks that we have the correct authority to perform the operation. Assuming we only have global capabilities for the heap, it cannot have write-local authority, due to the assumption on the previous slide, so we try to store the stack capability through a capability that does not have write-local authority, so it fails.

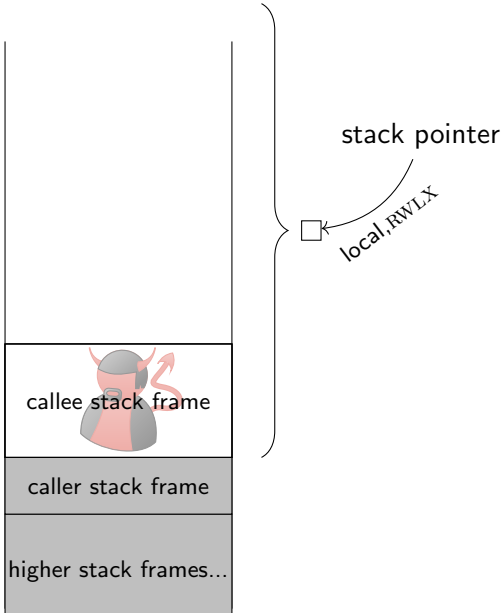
Stack and Return Capabilities: Attack 2

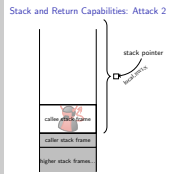




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.

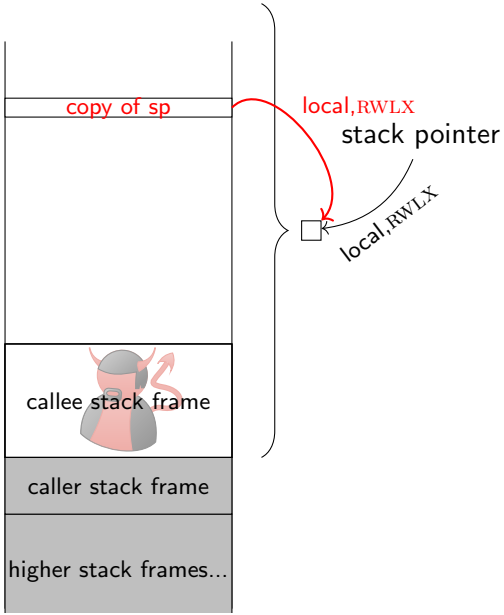
Stack and Return Capabilities: Attack 2

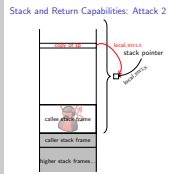




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.

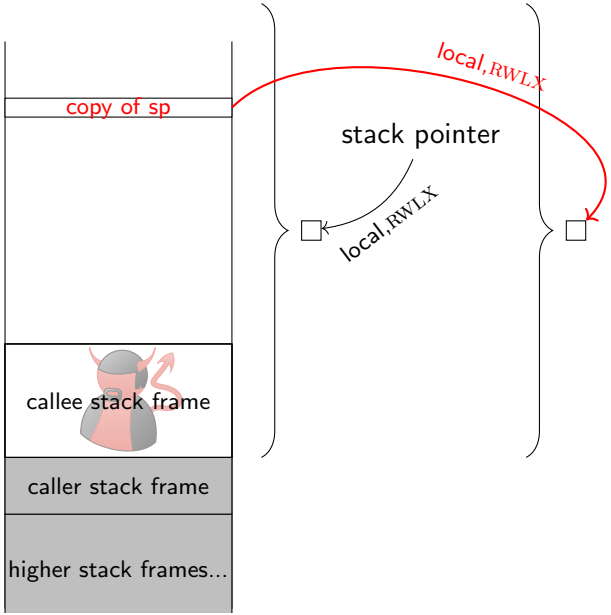
Stack and Return Capabilities: Attack 2

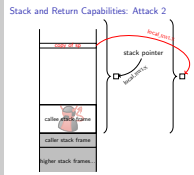




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.

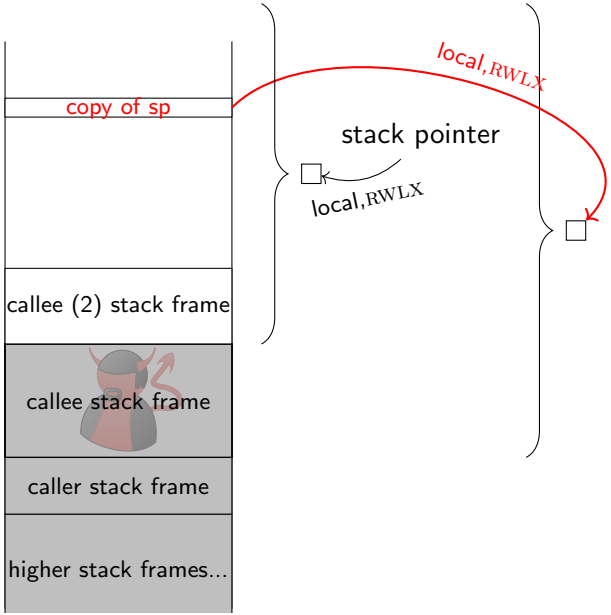
Stack and Return Capabilities: Attack 2

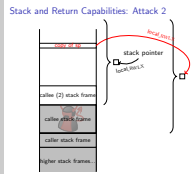




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.

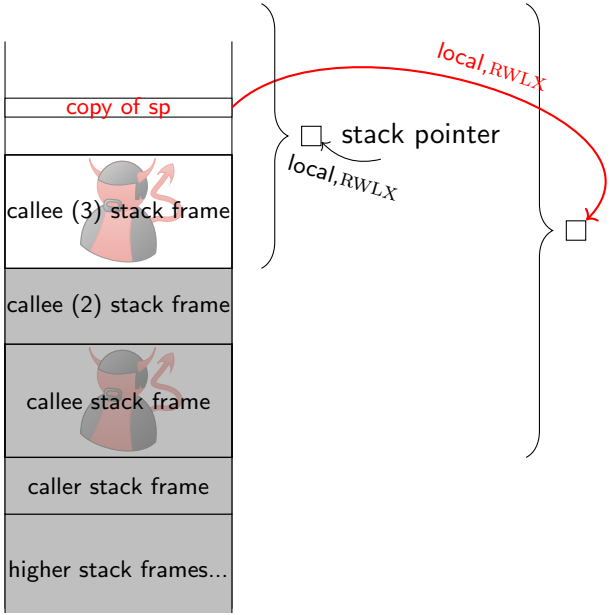
Stack and Return Capabilities: Attack 2

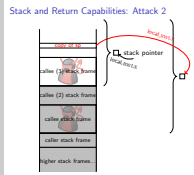




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.
6. trusted code runs for a bit pushes some local data to the stack and calls the callback.

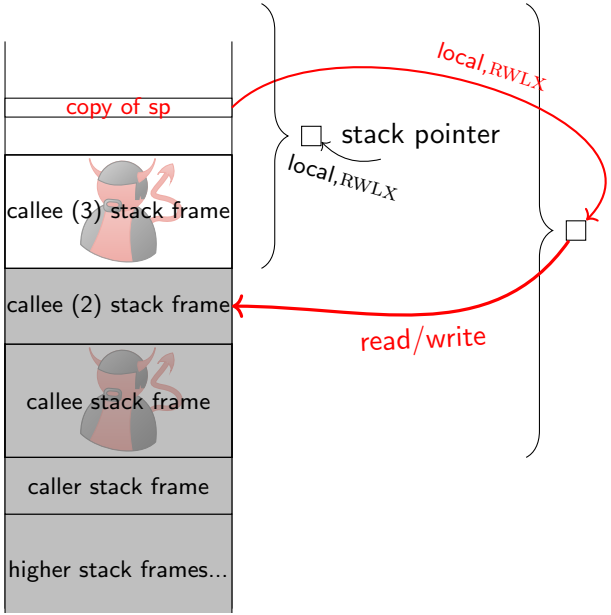
Stack and Return Capabilities: Attack 2

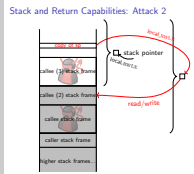




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.
6. trusted code runs for a bit pushes some local data to the stack and calls the callback.
7. The stack pointer is still on the stack allowing the untrusted code to read write to the stack frame of the trusted code.

Stack and Return Capabilities: Attack 2





1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.
6. trusted code runs for a bit pushes some local data to the stack and calls the callback.
7. The stack pointer is still on the stack allowing the untrusted code to read write to the stack frame of the trusted code.

Calling Convention (Continued)

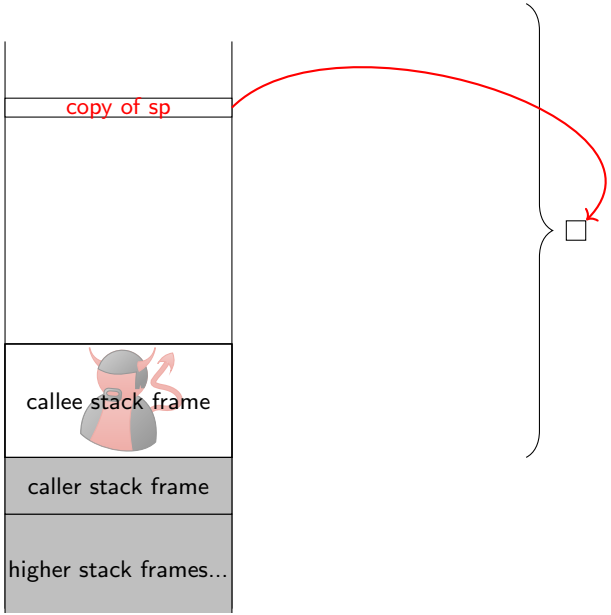
...

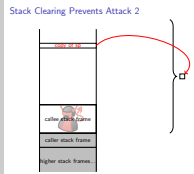
- Clear stack and non-argument registers before invoking untrusted code.

Calling Convention (Continued)
...
- Clear stack and non-argument registers before invoking untrusted code.

1. Stack is basically the only place we can store local capabiliites.
2. Make sure that untrusted code don't "sneak" capabilities between calls on the stack
3. Clear stack and argument registers

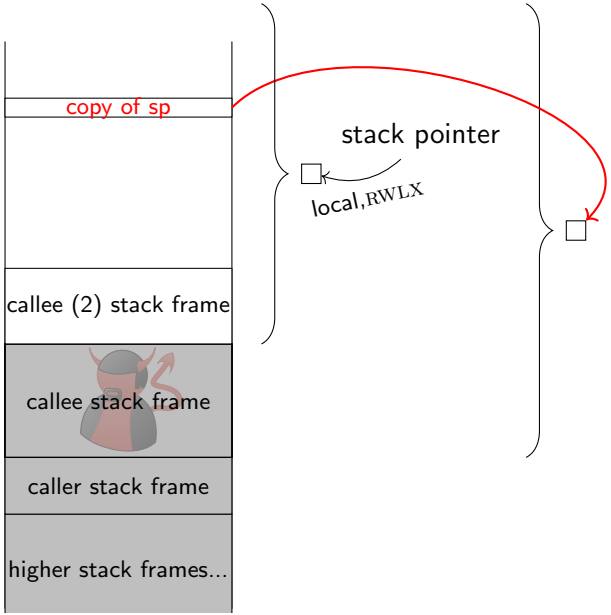
Stack Clearing Prevents Attack 2

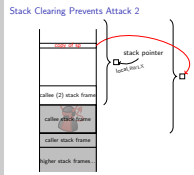




1. Let's see that the addition to the CC prevents attack 2.
2. The untrusted code has been called. It calls the well-behaved code.
3. The well-behaved code does its thing, but this time it clears the stack overwritting the old stack pointer the untrusted code had saved for later.
4. The untrusted code starts running, but it does not have an old stack pointer available only the one given to them by the well-behaved code

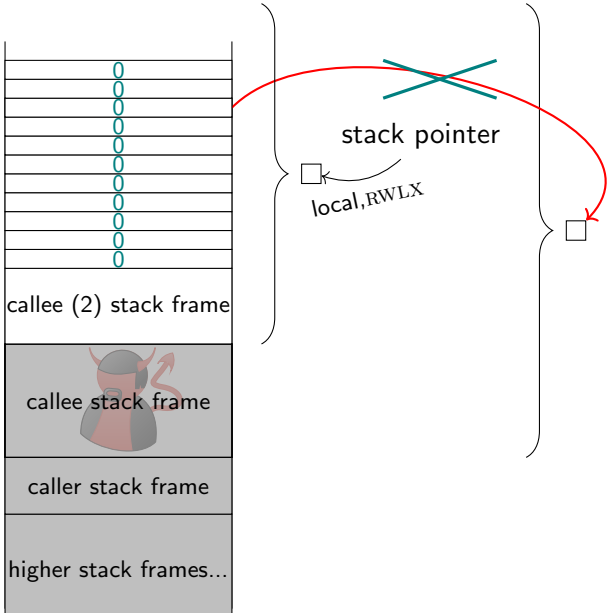
Stack Clearing Prevents Attack 2

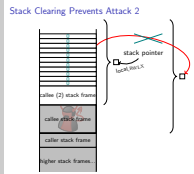




1. Let's see that the addition to the CC prevents attack 2.
2. The untrusted code has been called. It calls the well-behaved code.
3. The well-behaved code does its thing, but this time it clears the stack overwritting the old stack pointer the untrusted code had saved for later.
4. The untrusted code starts running, but it does not have an old stack pointer available only the one given to them by the well-behaved code

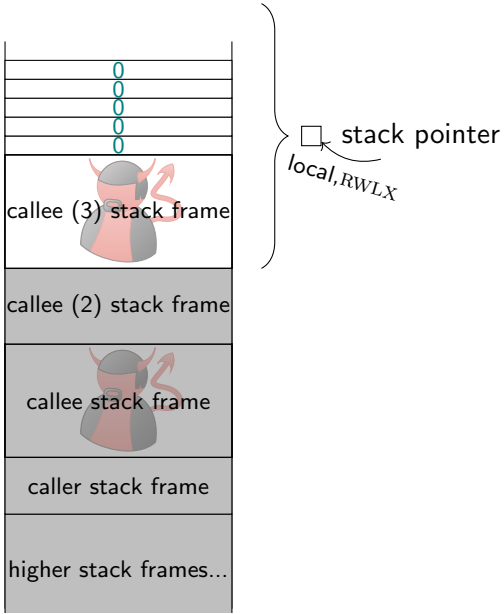
Stack Clearing Prevents Attack 2

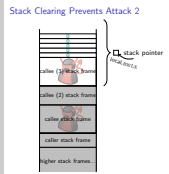




1. Let's see that the addition to the CC prevents attack 2.
2. The untrusted code has been called. It calls the well-behaved code.
3. The well-behaved code does its thing, but this time it clears the stack overwritting the old stack pointer the untrusted code had saved for later.
4. The untrusted code starts running, but it does not have an old stack pointer available only the one given to them by the well-behaved code

Stack Clearing Prevents Attack 2





1. Let's see that the addition to the CC prevents attack 2.
2. The untrusted code has been called. It calls the well-behaved code.
3. The well-behaved code does its thing, but this time it clears the stack overwritting the old stack pointer the untrusted code had saved for later.
4. The untrusted code starts running, but it does not have an old stack pointer available only the one given to them by the well-behaved code

(Full) Calling Convention

- Initially:
  - *Stack capability 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.

(Full) Calling Convention
- Initially:
  - *Stack capability 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.

1. The calling convention contains a bit more, but all of it is motivated by some attack.
2. I won't motivate the rest here, but I wanted to show you that it does not take many more precautions.

Formalizing the Guarantees of a Capability Machine

- How can we be sure the calling convention works?

Formalizing the Guarantees of a Capability Machine
- How can we be sure the calling convention works?

1. How can we be sure the calling convention works?
2. Specifically, if we have a program that interacts with untrusted code using the calling convention, how do we formally show correctness of the program.
3. We need a formal statement about the guarantees provided by the capabilities including the specific guarantees for local capabilities.
4. Traditionally syntactic very syntactic (e.g. reference graph), does not take into account what the program does with its capabilities.
5. We have defined a logical relation which also give us a statement about the guarantees provided by the capability machine.

Formalizing the Guarantees of a Capability Machine

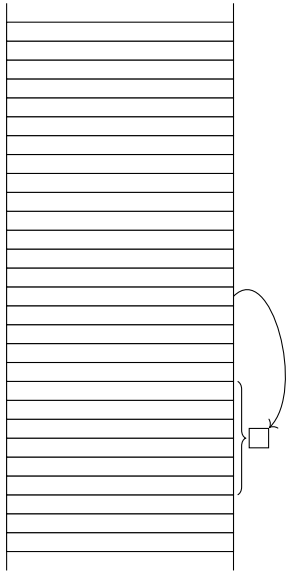
- How can we be sure the calling convention works?
- Unary step-indexed Kripke logical relation over recursive worlds
  - Statement of guarantees provided by the capability machine

Formalizing the Guarantees of a Capability Machine
- How can we be sure the calling convention works?
- Unary step-indexed Kripke logical relation over recursive worlds
  - Statement of guarantees provided by the capability machine

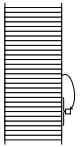
1. How can we be sure the calling convention works?
2. Specifically, if we have a program that interacts with untrusted code using the calling convention, how do we formally show correctness of the program.
3. We need a formal statement about the guarantees provided by the capabilities including the specific guarantees for local capabilities.
4. Traditionally syntactic very syntactic (e.g. reference graph), does not take into account what the program does with its capabilities.
5. We have defined a logical relation which also give us a statement about the guarantees provided by the capability machine.
6. Calling convention main application, but it is general
7. In the following: give some intuition about what a LR looks like for a capability machine

Worlds, Safe Values, and Step-Indexing

- Capabilities represent bounds on executing code

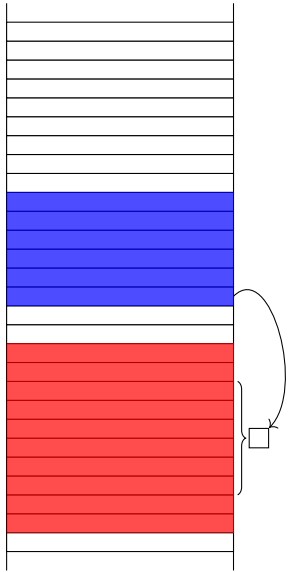


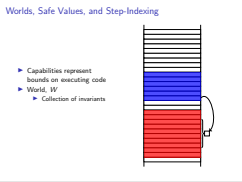
Worlds, Safe Values, and Step-Indexing
- Capabilities represent bounds on executing code



Worlds, Safe Values, and Step-Indexing

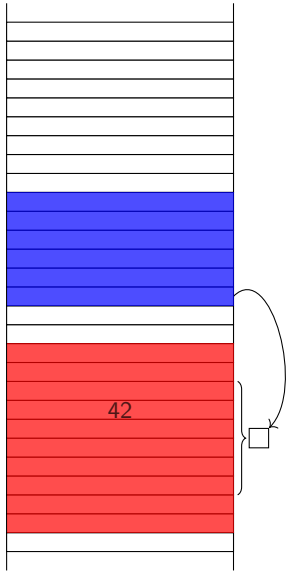
- Capabilities represent bounds on executing code
- World, W
  - Collection of invariants

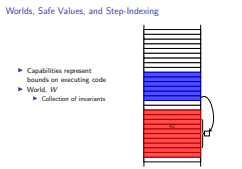




Worlds, Safe Values, and Step-Indexing

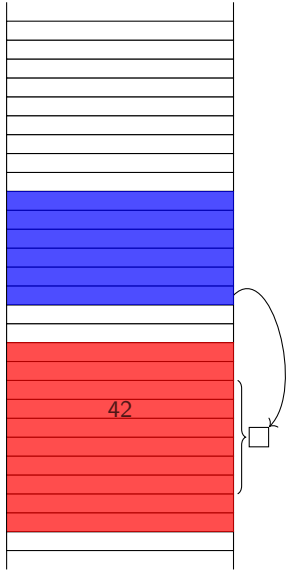
- Capabilities represent bounds on executing code
- World, W
  - Collection of invariants

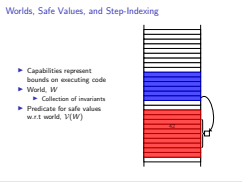




Worlds, Safe Values, and Step-Indexing

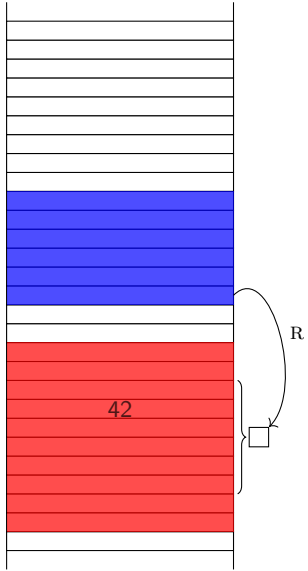
- Capabilities represent bounds on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$

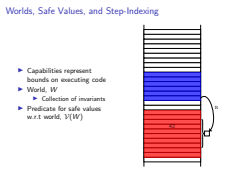




Worlds, Safe Values, and Step-Indexing

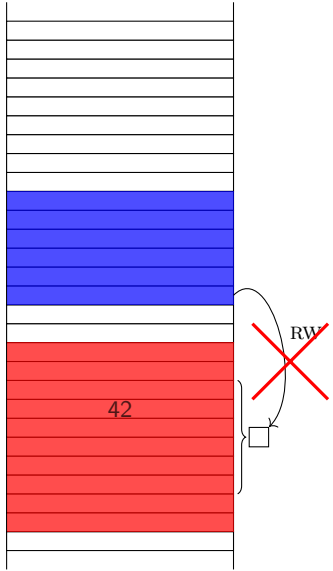
- Capabilities represent bounds on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$

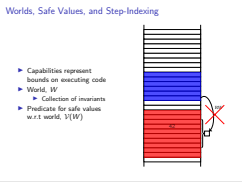




Worlds, Safe Values, and Step-Indexing

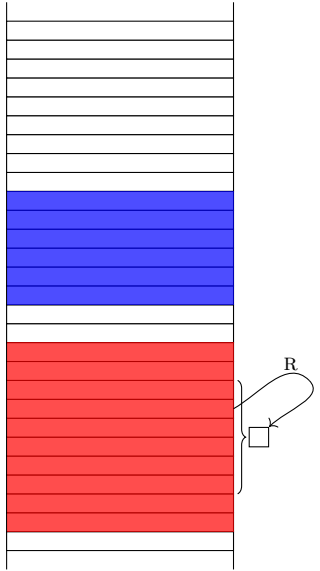
- Capabilities represent bounds on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$

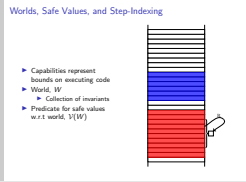




Worlds, Safe Values, and Step-Indexing

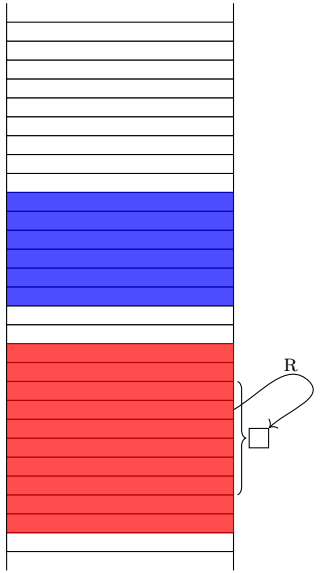
- Capabilities represent bounds on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$

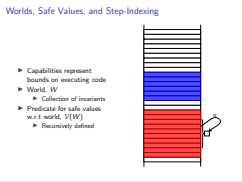




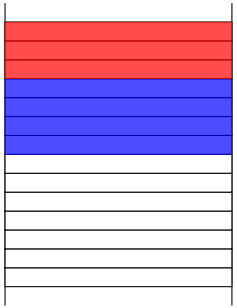
Worlds, Safe Values, and Step-Indexing

- Capabilities represent bounds 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

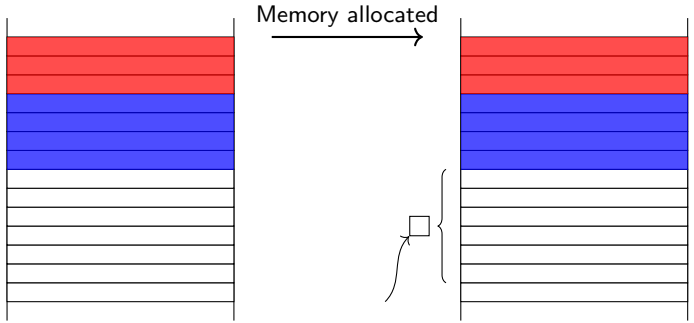


- Memory evolves over time

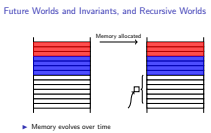
Future Worlds and Invariants, and Recursive Worlds
Memory evolves over time

Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

Future Worlds and Invariants, and Recursive Worlds

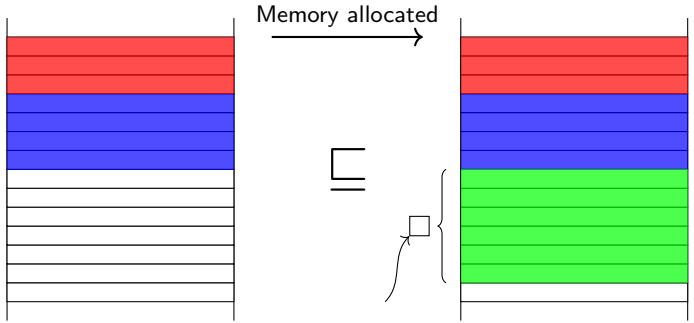


- Memory evolves over time

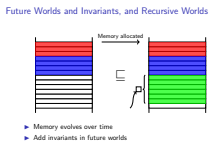


Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

Future Worlds and Invariants, and Recursive Worlds

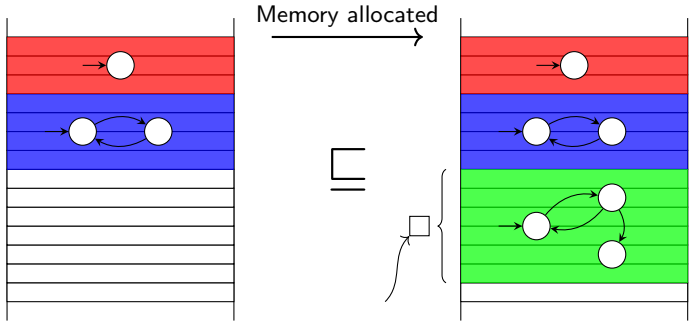


- Memory evolves over time
- Add invariants in future worlds

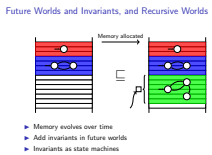


Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

Future Worlds and Invariants, and Recursive Worlds

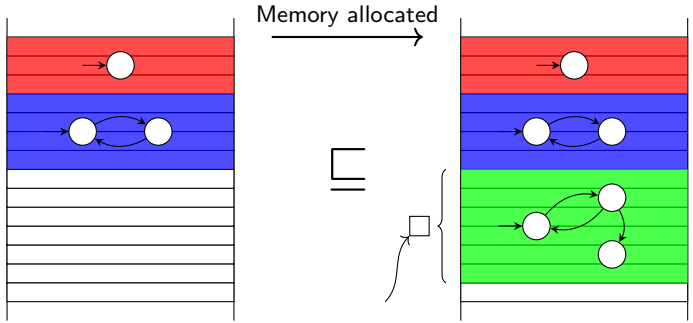


- Memory evolves over time
- Add invariants in future worlds
- Invariants as state machines



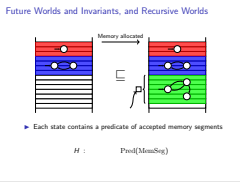
Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

Future Worlds and Invariants, and Recursive Worlds



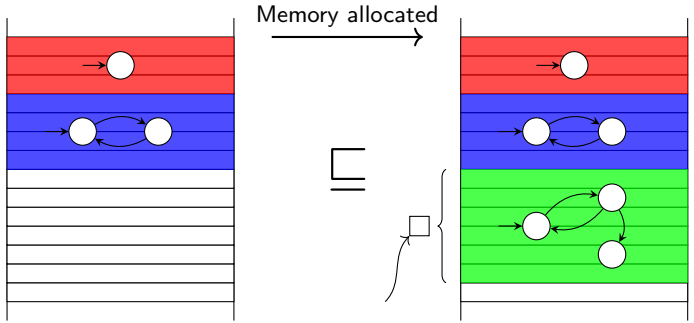
- Each state contains a predicate of accepted memory segments

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



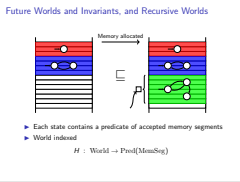
Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

Future Worlds and Invariants, and Recursive Worlds



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

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



Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

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)

when c global, second invocation in a world where c is safewhen c local, second invocation in a world where c is not (necessarily) safec essentially revoked, so the invariants it relies on need not hold.Two future world relations, one for all capabilities and one for non-local capabilitiesIf c global, then second invocation must happen in public future world, so c valid.If c local, then second invocation may happen in a private future world.How does local/global capabilities affect all this.If we hand a global capability to untrusted code, then it may be stored in memory, so we will only be able to reinvoke that code if we can guarantee that those values are still valid. Formally, the worlds contain the invariants that the global capability depend on and reinvocation is only possible in future worlds where these invariants are respected.Local capabilities on the other provides a means to revoke capabilities. If we invoke untrusted code and give them a local capability, then they have no way to store it aside from the register file (and the stack), so when they return we can be sure that the local capability

Local Capabilities

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

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

- `c` global ⇒ 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

when c global, second invocation in a world where c is safewhen c local, second invocation in a world where c is not (necessarily) safec essentially revoked, so the invariants it relies on need not hold.Two future world relations, one for all capabilities and one for non-local capabilitiesIf c global, then second invocation must happen in public future world, so c valid.If c local, then second invocation may happen in a private future world.How does local/global capabilities affect all this.If we hand a global capability to untrusted code, then it may be stored in memory, so we will only be able to reinvoke that code if we can guarantee that those values are still valid. Formally, the worlds contain the invariants that the global capability depend on and reinvocation is only possible in future worlds where these invariants are respected.Local capabilities on the other provides a means to revoke capabilities. If we invoke untrusted code and give them a local capability, then they have no way to store it aside from the register file (and the stack), so when they return we can be sure that the local capability

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

when c global, second invocation in a world where c is safewhen c local, second invocation in a world where c is not (necessarily) safec essentially revoked, so the invariants it relies on need not hold.Two future world relations, one for all capabilities and one for non-local capabilitiesIf c global, then second invocation must happen in public future world, so c valid.If c local, then second invocation may happen in a private future world.How does local/global capabilities affect all this.If we hand a global capability to untrusted code, then it may be stored in memory, so we will only be able to reinvoke that code if we can guarantee that those values are still valid. Formally, the worlds contain the invariants that the global capability depend on and reinvocation is only possible in future worlds where these invariants are respected.Local capabilities on the other provides a means to revoke capabilities. If we invoke untrusted code and give them a local capability, then they have no way to store it aside from the register file (and the stack), so when they return we can be sure that the local capability

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* $(n, w) \in \mathcal{V}(W')$.

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) ∈ V(W) and W' ⊒^pub W then (n, w) ∈ V(W').
▶ If (n, w) ∈ V(W) and W' ⊒^priv W and w is not a local capability, then (n, w) ∈ V(W').

when c global, second invocation in a world where c is safewhen c local, second invocation in a world where c is not (necessarily) safec essentially revoked, so the invariants it relies on need not hold.Two future world relations, one for all capabilities and one for non-local capabilitiesIf c global, then second invocation must happen in public future world, so c valid.If c local, then second invocation may happen in a private future world.How does local/global capabilities affect all this.If we hand a global capability to untrusted code, then it may be stored in memory, so we will only be able to reinvoke that code if we can guarantee that those values are still valid. Formally, the worlds contain the invariants that the global capability depend on and reinvocation is only possible in future worlds where these invariants are respected.Local capabilities on the other provides a means to revoke capabilities. If we invoke untrusted code and give them a local capability, then they have no way to store it aside from the register file (and the stack), so when they return we can be sure that the local capability

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

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

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

readCond is the assumption that every thing in the interval [a,e] is safe to read.$\mathcal{E}$ is safe to execute relation. That is, it will respect all the memory invariants.That is take an arbitrary capability. If it only has access to safe capabilities then it will preserve the invariants of the world.Remeber, dynamic checks = failing is concidered secure

"Awkward Example"

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


"Awkward Example"
let x = ref 0 in
λf.(x := 0;
f();
x := 1;
f();
assert(!x == 1))

example known from the literatureeven just in ML difficult as f can be the closure.the assert can fail if the calls are not well-bracketed!the local state is difficult to handle as the closure and the context needs to be able to update the invariant for x in different ways. (closure can switch between 0 and 1 as it pleases, but context can transition only from 0 to 1.)relies heavily on well-bracketednesswe have made a faithful translation and proved correctness (i.e., the assertion never fails).more semantic statement of guarantees allows us to do this.

Conclusion

- Capability machines can guarantee properties of high-level languages.
- We developed a calling convention that ensures well-bracketedness and local-state encapsulation.
- We define a unary step-indexed Kripke logical relation over recursive worlds.
  - Formal statement about guarantees provided by capability machine.
  - Allows reasoning about programs on capability machine.
- We apply it on the “awkward example”.

Conclusion
- Capability machines can guarantee properties of high-level languages.
- We developed a calling convention that ensures well-bracketedness and local-state encapsulation.
- We define a unary step-indexed Kripke logical relation over recursive worlds.
  - Formal statement about guarantees provided by capability machine.
  - Allows reasoning about programs on capability machine.
- We apply it on the “awkward example”.

Thank you!

Thank you!