Reasoning About a Machine with Local Capabilities

Provably Safe Stack and Return Pointer Management

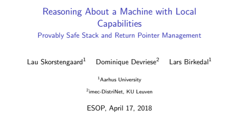
Lau Skorstengaard[1] Dominique Devriese[2] Lars Birkedal[1]

[1]Aarhus University

[2]imec-DistriNet, KU Leuven

ESOP, April 17, 2018

2018-04-13



Reasoning About a Machine with Local Capabilities
Provably Safe Stack and Return Pointer Management

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

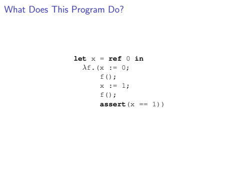
[1]Aarhus University

[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))
```

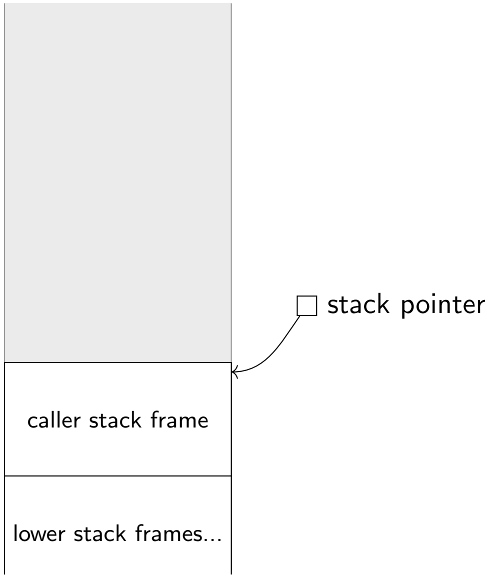


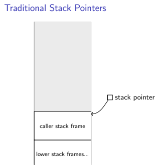
What Does This Program Do?

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

Informally, what does this program do?It allocates a reference to 0 and saves a closure to x. The closure takes a callback, assigns 0 to x, calls the callback, assigns 1 to x, calls the callback another time and finally asserts x to be 1.As a programmer that is how I would think. But what happens if I compile this closure to assembly and let some untrusted piece of code interact with it?My reasoning depends on the assumption that when I call f, then if f returns it returns to a certain point. The low-level machine needs to enforce this if I want to be able to do have this closure interact with arbitrary machine code.

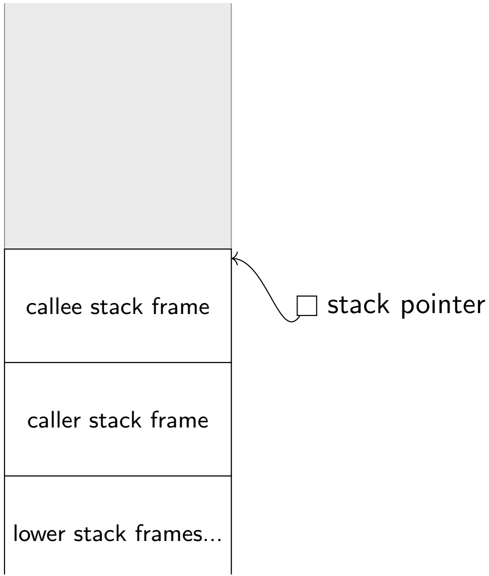
Traditional Stack Pointers

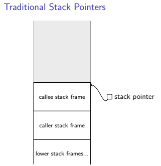




Let's first consider what happens with the stack during a call on a traditional low-level machine and what can go wrong.We need something to enforce security properties on a low-level machine.

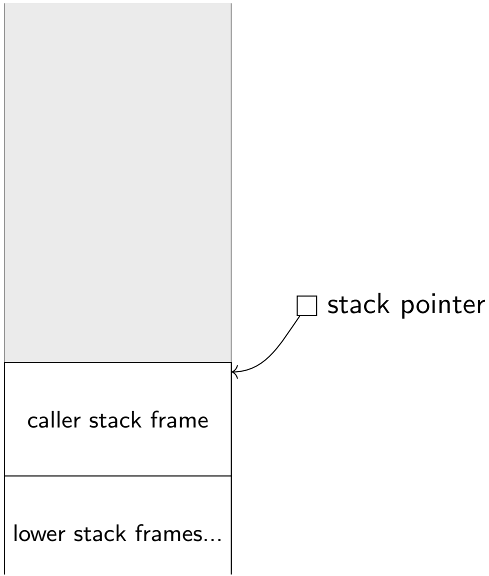
Traditional Stack Pointers

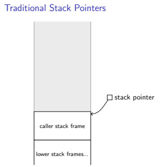




Let's first consider what happens with the stack during a call on a traditional low-level machine and what can go wrong.We need something to enforce security properties on a low-level machine.

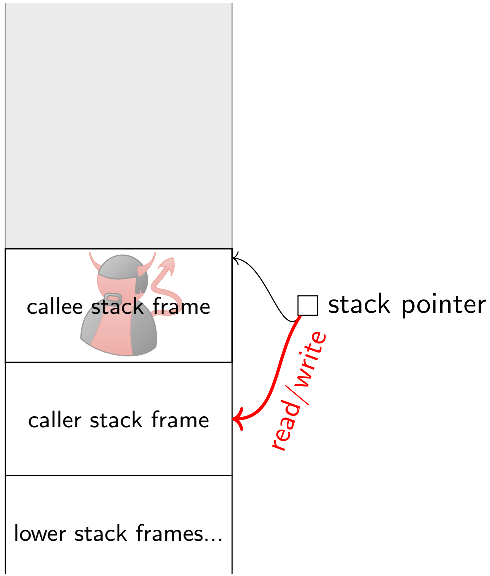
Traditional Stack Pointers

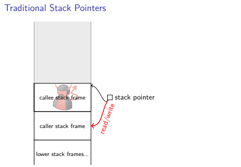




Let’s first consider what happens with the stack during a call on a traditional low-level machine and what can go wrong.We need something to enforce security properties on a low-level machine.

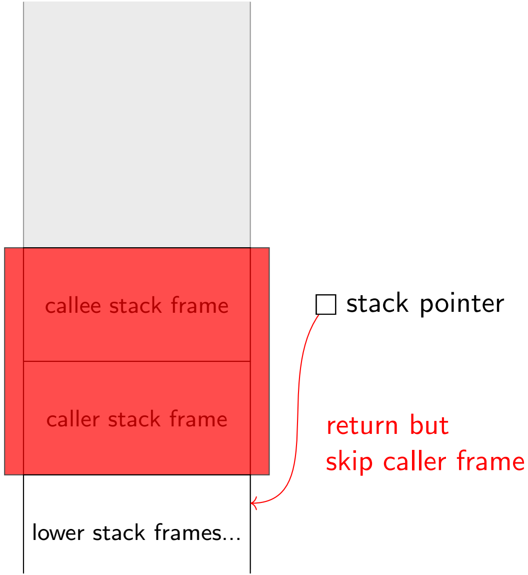
Traditional Stack Pointers

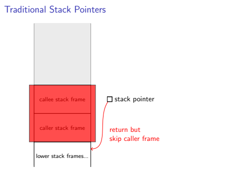



Traditional Stack Pointers
callee stack frame
caller stack frame
lower stack frames...
stack pointer
read/write

Let’s first consider what happens with the stack during a call on a traditional low-level machine and what can go wrong.We need something to enforce security properties on a low-level machine.

Traditional Stack Pointers

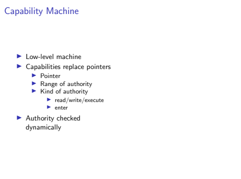




Let's first consider what happens with the stack during a call on a traditional low-level machine and what can go wrong.We need something to enforce security properties on a low-level machine.

Capability Machine

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

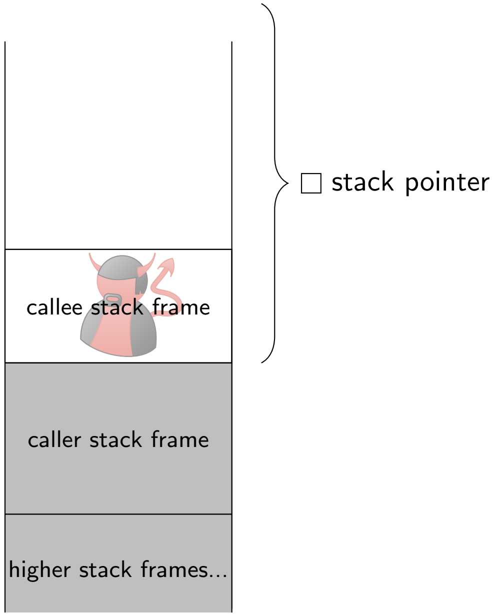


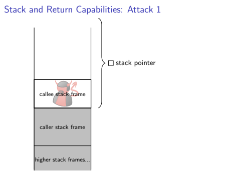
Capability Machine

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

Has instructions as you would expect, load, store, jmp, etc.Has instructions for manipulating capabilities.This particular setup gives a very fine-grained memory control.

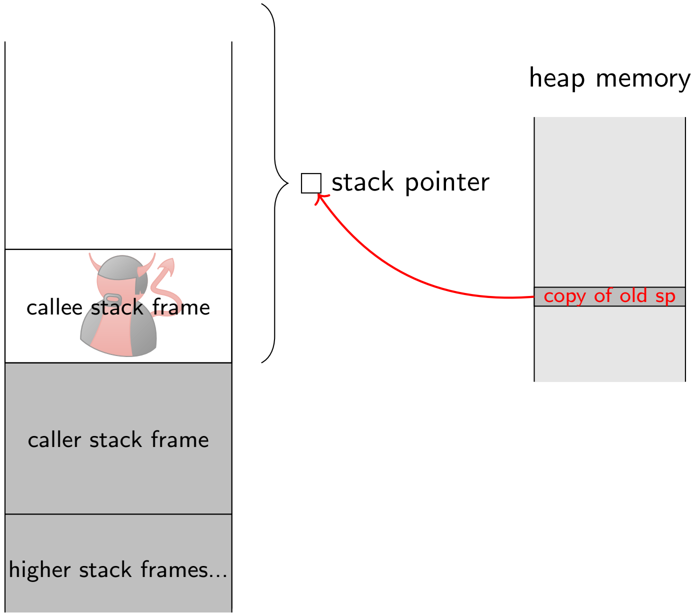
Stack and Return Capabilities: Attack 1

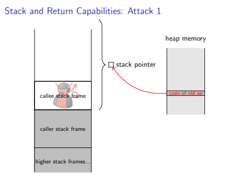



Stack and Return Capabilities: Attack 1

In the following, part of memory is used for the stack and we use capabilities to govern them. Callee not able to read from the caller stack frame, so that is already an improvement.Evil callee stores stack pointer on the heap.Evil callee returns to the caller.caller uses a bit more of the stack and calls the evil callee again.the evil callee can now load the old stack capability from memory and access part of the callers private stack!

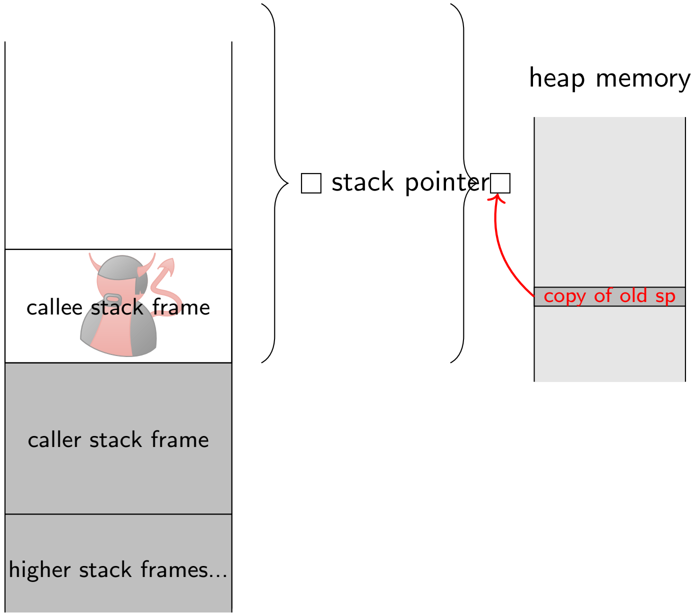
Stack and Return Capabilities: Attack 1

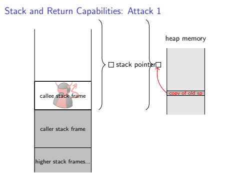




In the following, part of memory is used for the stack and we use capabilities to govern them. Callee not able to read from the caller stack frame, so that is already an improvement.Evil callee stores stack pointer on the heap.Evil callee returns to the caller.caller uses a bit more of the stack and calls the evil callee again.the evil callee can now load the old stack capability from memory and access part of the callers private stack!

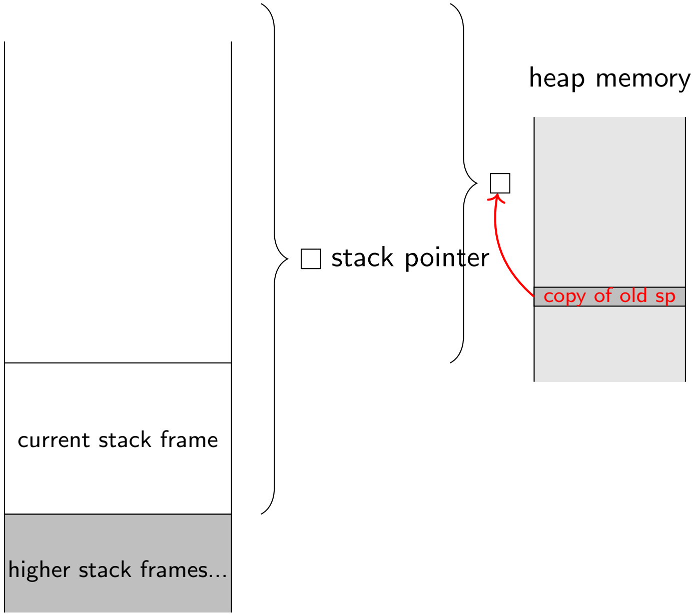
Stack and Return Capabilities: Attack 1

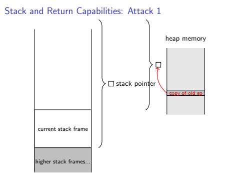




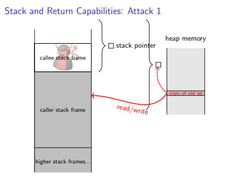
In the following, part of memory is used for the stack and we use capabilities to govern them. Callee not able to read from the caller stack frame, so that is already an improvement.Evil callee stores stack pointer on the heap.Evil callee returns to the caller.caller uses a bit more of the stack and calls the evil callee again.the evil callee can now load the old stack capability from memory and access part of the callers private stack!

Stack and Return Capabilities: Attack 1

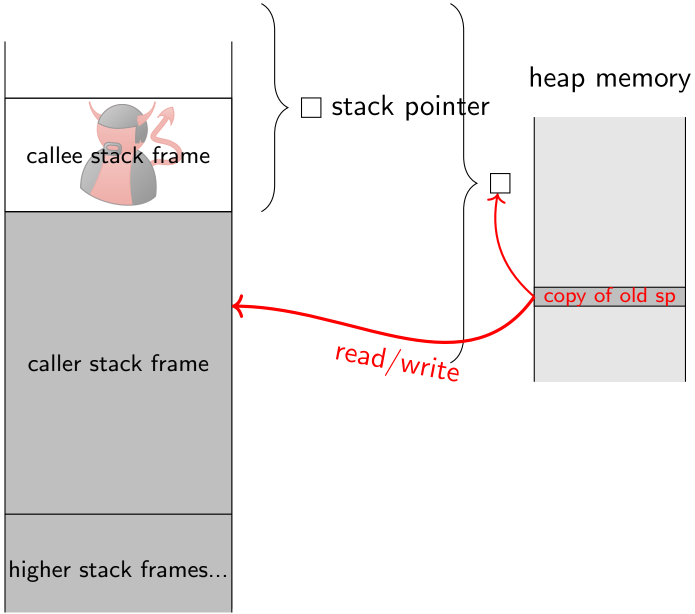




In the following, part of memory is used for the stack and we use capabilities to govern them. Callee not able to read from the caller stack frame, so that is already an improvement.Evil callee stores stack pointer on the heap.Evil callee returns to the caller.caller uses a bit more of the stack and calls the evil callee again.the evil callee can now load the old stack capability from memory and access part of the callers private stack!



Stack and Return Capabilities: Attack 1



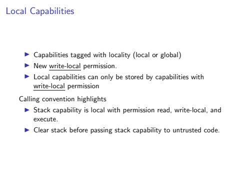
In the following, part of memory is used for the stack and we use capabilities to govern them. Callee not able to read from the caller stack frame, so that is already an improvement.Evil callee stores stack pointer on the heap.Evil callee returns to the caller.caller uses a bit more of the stack and calls the evil callee again.the evil callee can now load the old stack capability from memory and access part of the callers private stack!

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 Capabilities

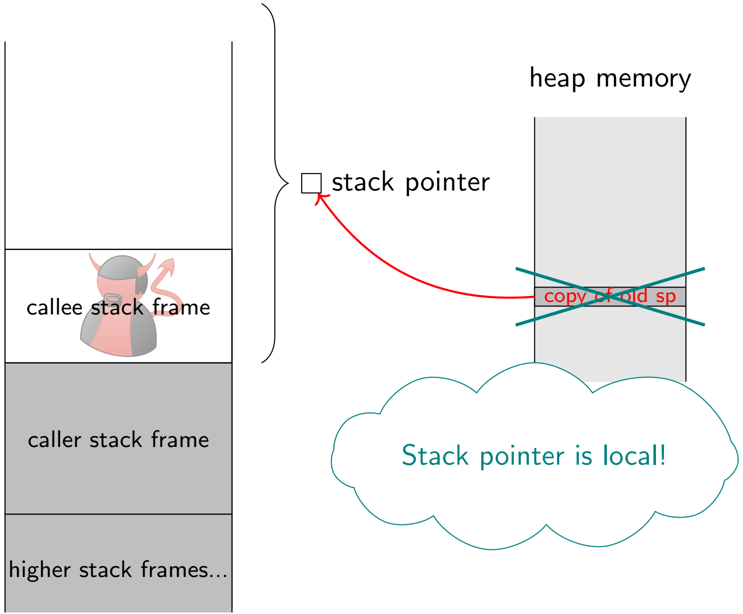
- Capabilities tagged with locality (local or global)
- New write-local permission.
- Local capabilities can only be stored by capabilities with write-local permission

Calling convention highlights

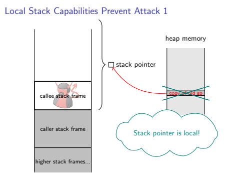
- Stack capability is local with permission read, write-local, and execute.
- Clear stack before passing stack capability to untrusted code.

We call non-local capabilities global

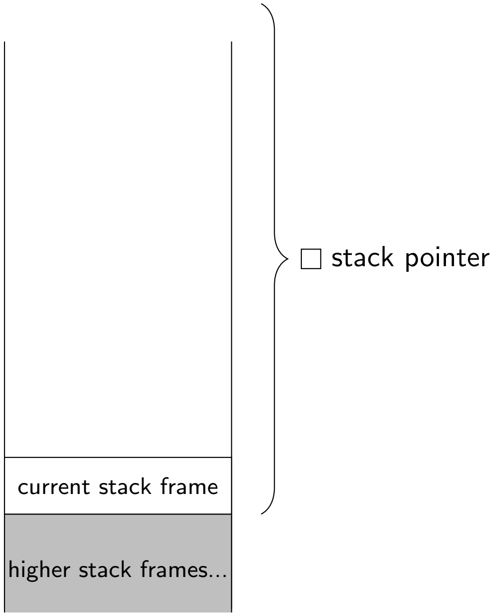
Local Stack Capabilities Prevent Attack 1

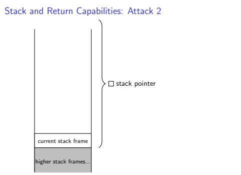


Stack pointer is local!



Stack and Return Capabilities: Attack 2




Stack and Return Capabilities: Attack 2

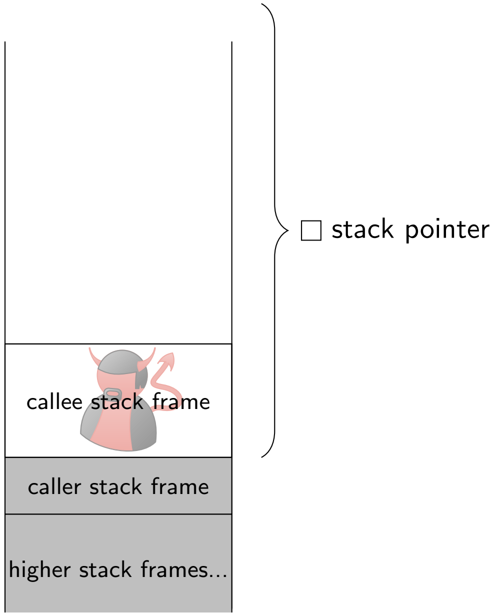
□ stack pointer

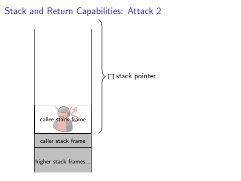
current stack frame

higher stack frames...

The stack was the only place to store the local capability, so the adversary hid it there.

Stack and Return Capabilities: Attack 2

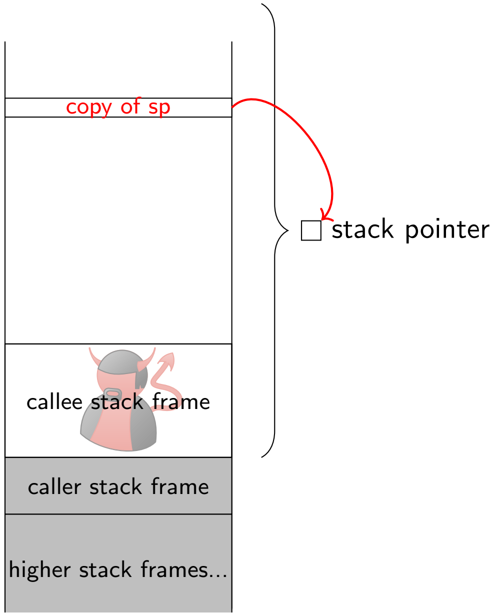


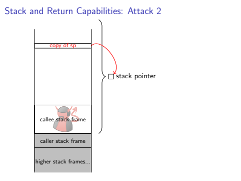

Stack and Return Capabilities: Attack 2

□ stack pointer
callee stack frame
caller stack frame
higher stack frames...

The stack was the only place to store the local capability, so the adversary hid it there.

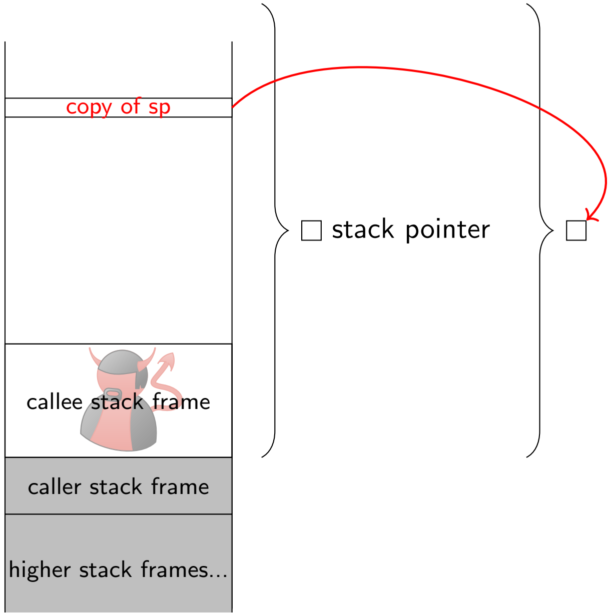
Stack and Return Capabilities: Attack 2

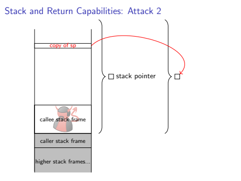




The stack was the only place to store the local capability, so the adversary hid it there.

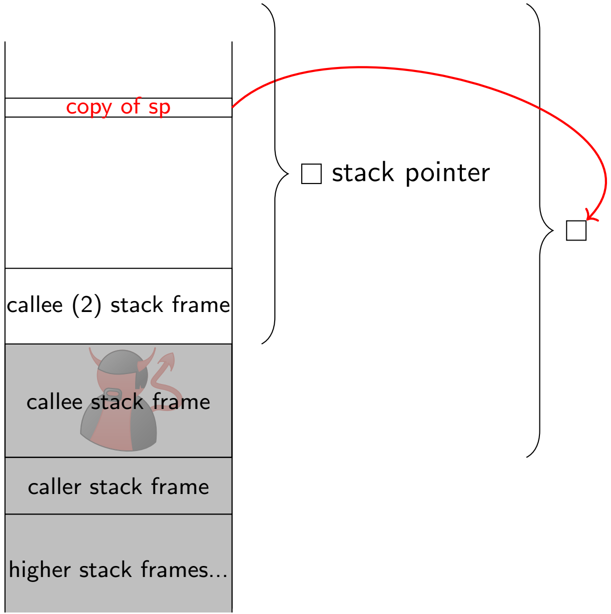
Stack and Return Capabilities: Attack 2





The stack was the only place to store the local capability, so the adversary hid it there.

Stack and Return Capabilities: Attack 2



copy of sp

□ stack pointer

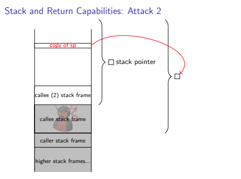
□

callee (2) stack frame

callee stack frame

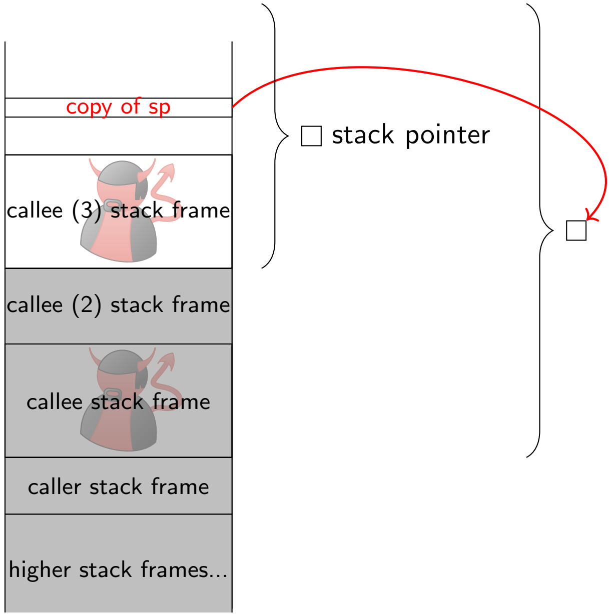
caller stack frame

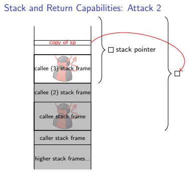
higher stack frames...



The stack was the only place to store the local capability, so the adversary hid it there.

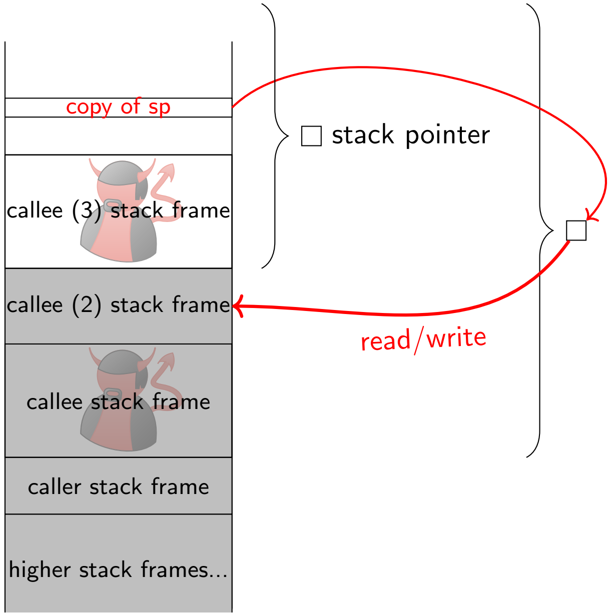
Stack and Return Capabilities: Attack 2

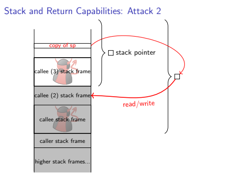




The stack was the only place to store the local capability, so the adversary hid it there.

Stack and Return Capabilities: Attack 2

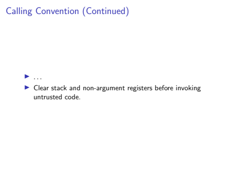




The stack was the only place to store the local capability, so the adversary hid it there.

Calling Convention (Continued)

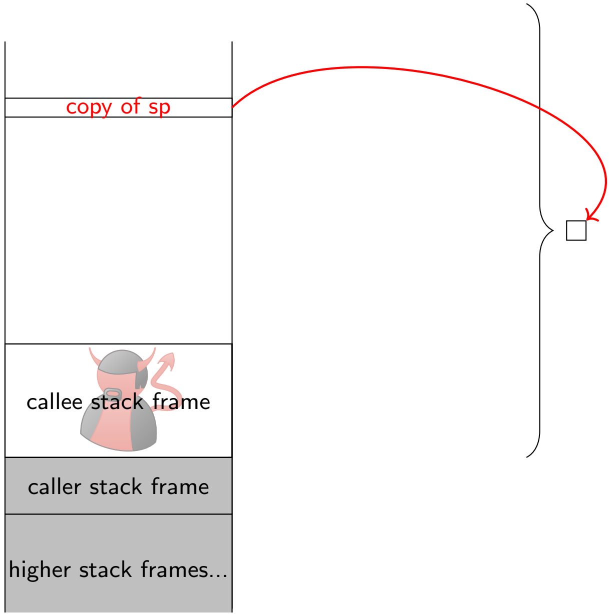
- . . .
- Clear stack and non-argument registers before invoking untrusted code.



Calling Convention (Continued)

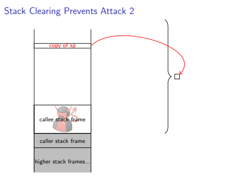
- . . .
- Clear stack and non-argument registers before invoking untrusted code.

Stack Clearing Prevents Attack 2

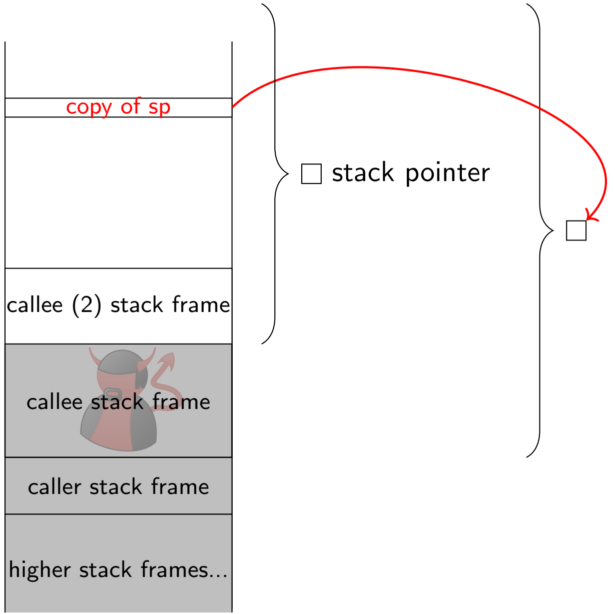


Reasoning About a Machine with Local Capabilities

└─Stack Clearing Prevents Attack 2



Stack Clearing Prevents Attack 2



copy of sp

□ stack pointer

□

callee (2) stack frame

callee stack frame

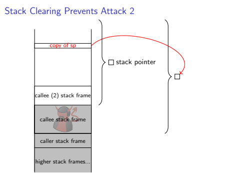
caller stack frame

higher stack frames...

2018-04-13

Reasoning About a Machine with Local Capabilities

└Stack Clearing Prevents Attack 2



Stack Clearing Prevents Attack 2

copy of sp

□ stack pointer

□

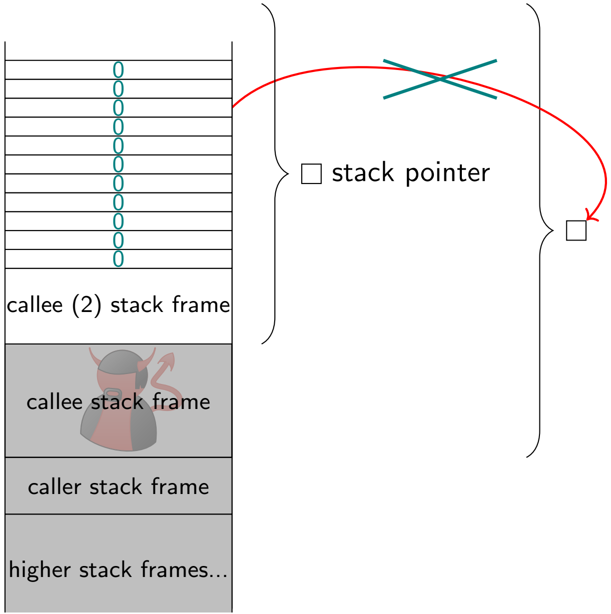
callee (2) stack frame

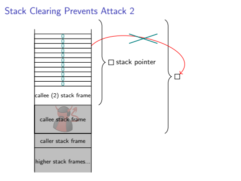
callee stack frame

caller stack frame

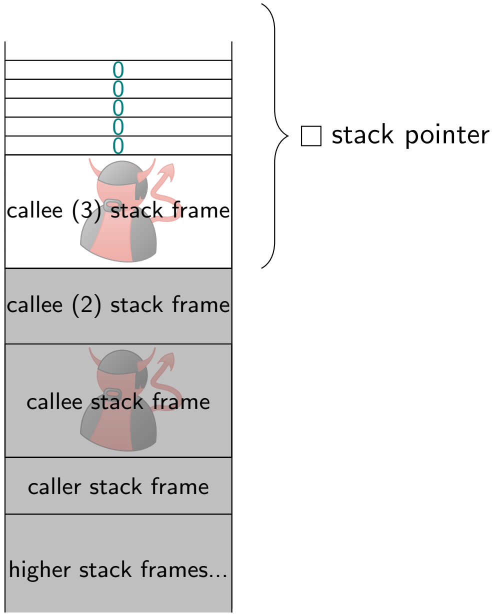
higher stack frames...

Stack Clearing Prevents Attack 2





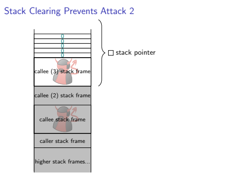
Stack Clearing Prevents Attack 2



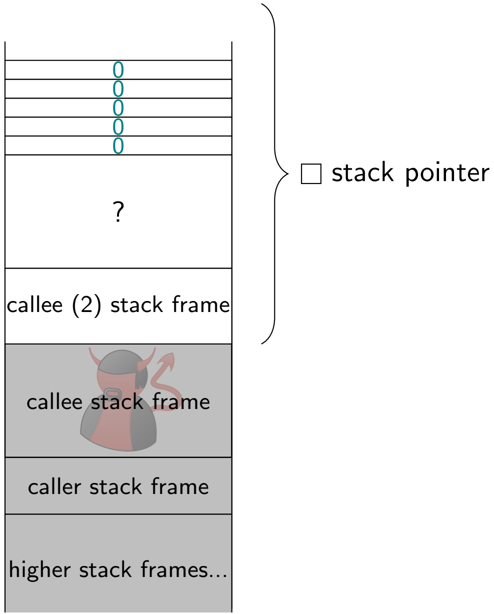
2018-04-13

Reasoning About a Machine with Local Capabilities

└Stack Clearing Prevents Attack 2


Stack Clearing Prevents Attack 2

Stack Clearing Prevents Attack 2



0
0
0
0
0

?

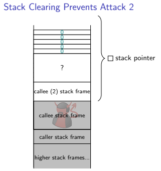
callee (2) stack frame

callee stack frame

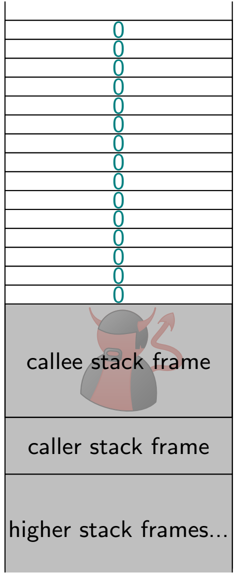
caller stack frame

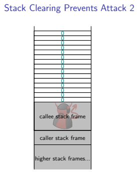
higher stack frames...

□ stack pointer

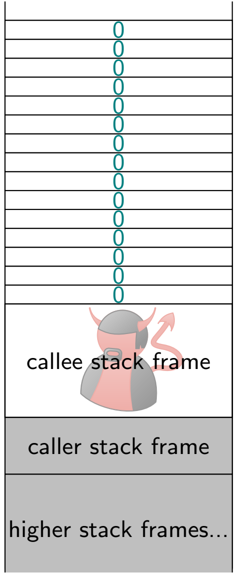


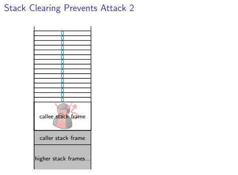
Stack Clearing Prevents Attack 2





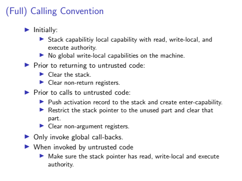
Stack Clearing Prevents Attack 2





(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.



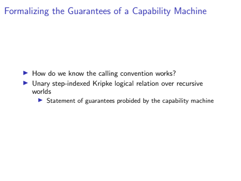
(Full) Calling Convention

- 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.

We need to do a few more things, but this is the rest (just to illustrate that it is fairly simple).Every point is motivated by some attack.

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

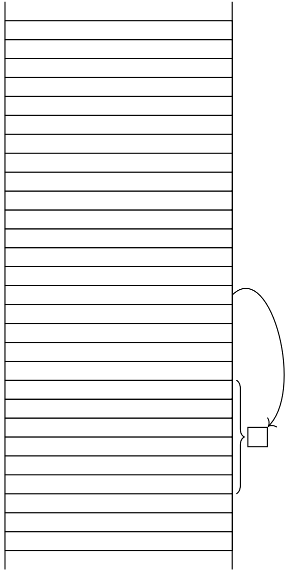

Formalizing the Guarantees of a Capability Machine

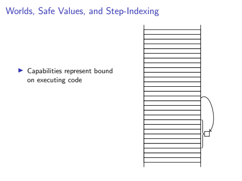
- How do we know the calling convention works?
- Unary step-indexed Kripke logical relation over recursive worlds
    - Statement of guarantees probided by the capability machine

How can we be sure calling convention works. Specifically, if a program interacts with intrusted code using the CC, can we formally show the correctness of the program if it relies on well-bracketedness or local-state encapsulation.Need formal statement of the guarantees provided by the capability machine including the specific guarantees for local capabilities.We state this formal statement in terms of a unary step-indexed Kripke logical relation over recursive worldsCalling convention main application, but it is very general - can be used to reason about other programs.In the following: give some intuition for different parts of LRcorrectness here could be assert not violatedmention better than previous. Define the guarantees

Worlds, Safe Values, and Step-Indexing

- Capabilities represent bound on executing code





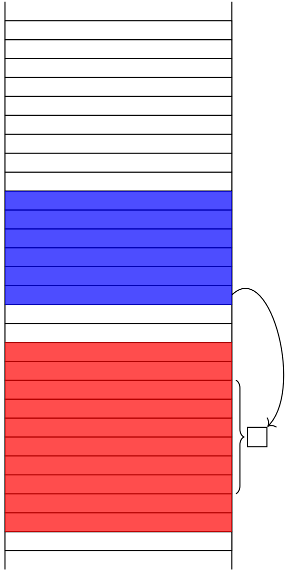
Worlds, Safe Values, and Step-Indexing

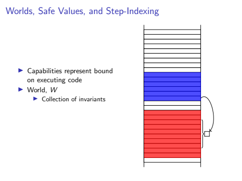
- Capabilities represent bound on executing code

compared to normal assembly, capabilities represent bounds on executing block of code.we have no observable I/O, so the authority bounds we consider are related to memory. However, more fine-grained/detailed than read/write-authority.a piece of code can be bound by arbitrary memory invariants which we define in a world.essentially, a world is a collection of invariants and safety of words defined with respect to a world.Define a set of words that are safe w.r.t. world W V(W) in P(Word)Whether a capability is safe depends on the authority it carriesExample, w.r.t. a world with the invariant that an address contains constant. Safe for read capability, not write as write capability can break this invariant.safety for a read capability is the case if the read capability only gives access to safe capabilities.What if that part of memory contains a read capability for the same part of memory? Cyclic definition. Solved by step-indexing.related to similar issue with languages with recursive types or higher-order ML-list references.Solved by step-indexing - safety up to a certain number of

Worlds, Safe Values, and Step-Indexing

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





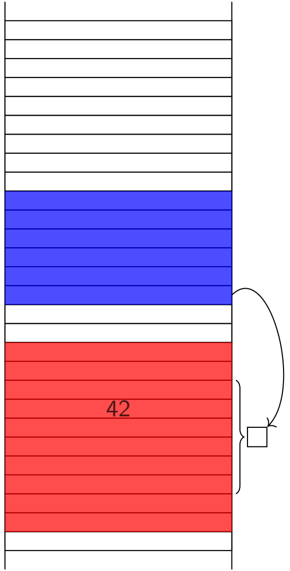
Worlds, Safe Values, and Step-Indexing

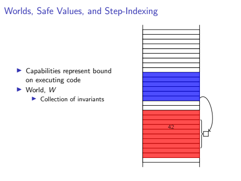
- Capabilities represent bound on executing code
- World, W
  - Collection of invariants

compared to normal assembly, capabilities represent bounds on executing block of code.we have no observable I/O, so the authority bounds we consider are related to memory. However, more fine-grained/detailed than read/write-authority.a piece of code can be bound by arbitrary memory invariants which we define in a world.essentially, a world is a collection of invariants and safety of words defined with respect to a world.Define a set of words that are safe w.r.t. world W V(W) in P(Word)Whether a capability is safe depends on the authority it carriesExample, w.r.t. a world with the invariant that an address contains constant. Safe for read capability, not write as write capability can break this invariant.safety for a read capability is the case if the read capability only gives access to safe capabilities.What if that part of memory contains a read capability for the same part of memory? Cyclic definition. Solved by step-indexing.related to similar issue with languages with recursive types or higher-order ML-list references.Solved by step-indexing - safety up to a certain number of

Worlds, Safe Values, and Step-Indexing

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



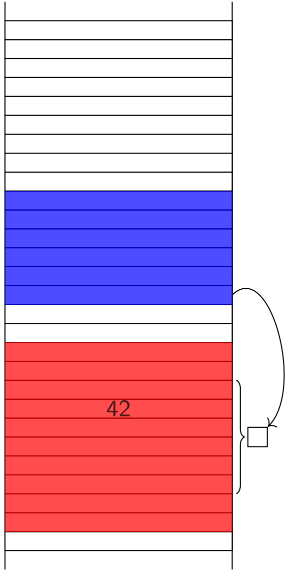

Worlds, Safe Values, and Step-Indexing

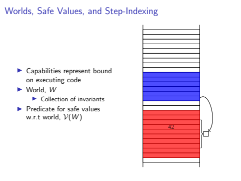
- Capabilities represent bound on executing code
- World, *W*
  - Collection of invariants

compared to normal assembly, capabilities represent bounds on executing block of code.we have no observable I/O, so the authority bounds we consider are related to memory. However, more fine-grained/detailed than read/write-authority.a piece of code can be bound by arbitrary memory invariants which we define in a world.essentially, a world is a collection of invariants and safety of words defined with respect to a world.Define a set of words that are safe w.r.t. world W V(W) in P(Word)Whether a capability is safe depends on the authority it carriesExample, w.r.t. a world with the invariant that an address contains constant. Safe for read capability, not write as write capability can break this invariant.safety for a read capability is the case if the read capability only gives access to safe capabilities.What if that part of memory contains a read capability for the same part of memory? Cyclic definition. Solved by step-indexing.related to similar issue with languages with recursive types or higher-order ML-list references.Solved by step-indexing - safety up to a certain number of

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, $\mathcal{V}(W)$

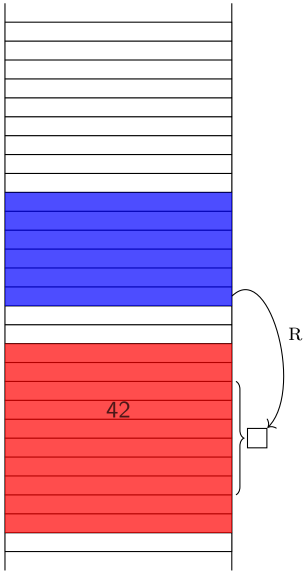


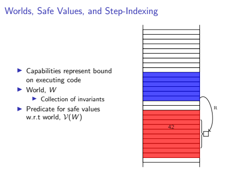

Worlds, Safe Values, and Step-Indexing
- Capabilities represent bound on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$
42

compared to normal assembly, capabilities represent bounds on executing block of code.we have no observable I/O, so the authority bounds we consider are related to memory. However, more fine-grained/detailed than read/write-authority.a piece of code can be bound by arbitrary memory invariants which we define in a world.essentially, a world is a collection of invariants and safety of words defined with respect to a world.Define a set of words that are safe w.r.t. world W V(W) in P(Word)Whether a capability is safe depends on the authority it carriesExample, w.r.t. a world with the invariant that an address contains constant. Safe for read capability, not write as write capability can break this invariant.safety for a read capability is the case if the read capability only gives access to safe capabilities.What if that part of memory contains a read capability for the same part of memory? Cyclic definition. Solved by step-indexing.related to similar issue with languages with recursive types or higher-order ML-list references.Solved by step-indexing - safety up to a certain number of

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, $\mathcal{V}(W)$

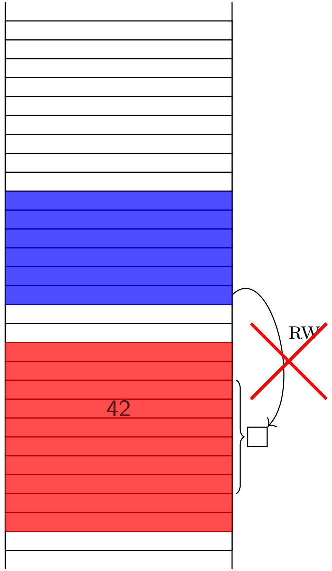


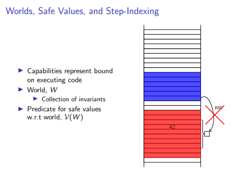

Worlds, Safe Values, and Step-Indexing
- Capabilities represent bound on executing code
- World, W
    - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$
R
42

compared to normal assembly, capabilities represent bounds on executing block of code.we have no observable I/O, so the authority bounds we consider are related to memory. However, more fine-grained/detailed than read/write-authority.a piece of code can be bound by arbitrary memory invariants which we define in a world.essentially, a world is a collection of invariants and safety of words defined with respect to a world.Define a set of words that are safe w.r.t. world W V(W) in P(Word)Whether a capability is safe depends on the authority it carriesExample, w.r.t. a world with the invariant that an address contains constant. Safe for read capability, not write as write capability can break this invariant.safety for a read capability is the case if the read capability only gives access to safe capabilities.What if that part of memory contains a read capability for the same part of memory? Cyclic definition. Solved by step-indexing.related to similar issue with languages with recursive types or higher-order ML-list references.Solved by step-indexing - safety up to a certain number of

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, $\mathcal{V}(W)$



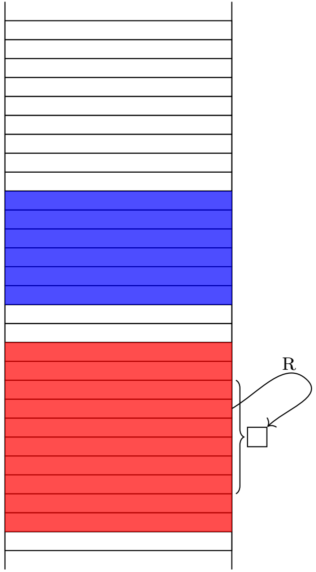


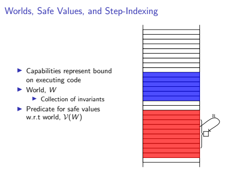
Worlds, Safe Values, and Step-Indexing
Capabilities represent bound on executing code
World, W
Collection of invariants
Predicate for safe values w.r.t world, $\mathcal{V}(W)$
42
RW

compared to normal assembly, capabilities represent bounds on executing block of code.we have no observable I/O, so the authority bounds we consider are related to memory. However, more fine-grained/detailed than read/write-authority.a piece of code can be bound by arbitrary memory invariants which we define in a world.essentially, a world is a collection of invariants and safety of words defined with respect to a world.Define a set of words that are safe w.r.t. world W V(W) in P(Word)Whether a capability is safe depends on the authority it carriesExample, w.r.t. a world with the invariant that an address contains constant. Safe for read capability, not write as write capability can break this invariant.safety for a read capability is the case if the read capability only gives access to safe capabilities.What if that part of memory contains a read capability for the same part of memory? Cyclic definition. Solved by step-indexing.related to similar issue with languages with recursive types or higher-order ML-list references.Solved by step-indexing - safety up to a certain number of

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, $\mathcal{V}(W)$



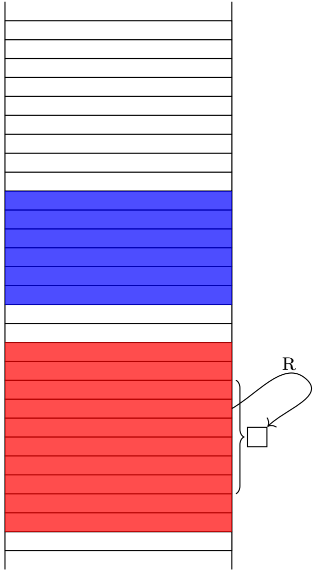

Worlds, Safe Values, and Step-Indexing

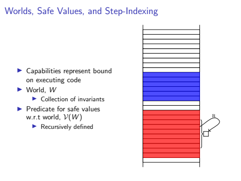
- Capabilities represent bound on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$

compared to normal assembly, capabilities represent bounds on executing block of code.we have no observable I/O, so the authority bounds we consider are related to memory. However, more fine-grained/detailed than read/write-authority.a piece of code can be bound by arbitrary memory invariants which we define in a world.essentially, a world is a collection of invariants and safety of words defined with respect to a world.Define a set of words that are safe w.r.t. world W V(W) in P(Word)Whether a capability is safe depends on the authority it carriesExample, w.r.t. a world with the invariant that an address contains constant. Safe for read capability, not write as write capability can break this invariant.safety for a read capability is the case if the read capability only gives access to safe capabilities.What if that part of memory contains a read capability for the same part of memory? Cyclic definition. Solved by step-indexing.related to similar issue with languages with recursive types or higher-order ML-list references.Solved by step-indexing - safety up to a certain number of

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

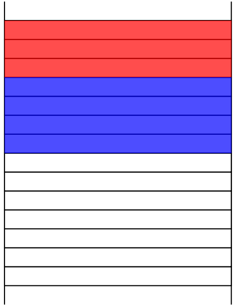



Worlds, Safe Values, and Step-Indexing

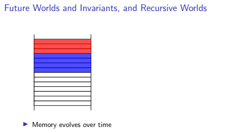
- Capabilities represent bound on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$
  - Recursively defined

compared to normal assembly, capabilities represent bounds on executing block of code.we have no observable I/O, so the authority bounds we consider are related to memory. However, more fine-grained/detailed than read/write-authority.a piece of code can be bound by arbitrary memory invariants which we define in a world.essentially, a world is a collection of invariants and safety of words defined with respect to a world.Define a set of words that are safe w.r.t. world W V(W) in P(Word)Whether a capability is safe depends on the authority it carriesExample, w.r.t. a world with the invariant that an address contains constant. Safe for read capability, not write as write capability can break this invariant.safety for a read capability is the case if the read capability only gives access to safe capabilities.What if that part of memory contains a read capability for the same part of memory? Cyclic definition. Solved by step-indexing.related to similar issue with languages with recursive types or higher-order ML-list references.Solved by step-indexing - safety up to a certain number of

Future Worlds and Invariants, and Recursive Worlds



- Memory evolves over time

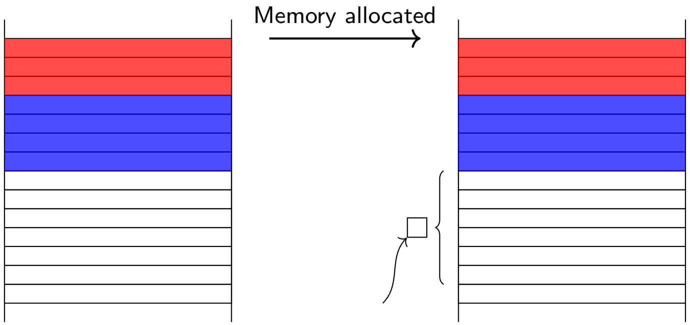


Future Worlds and Invariants, and Recursive Worlds

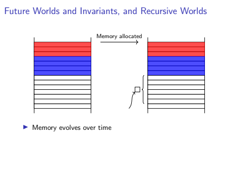
- Memory evolves over time

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

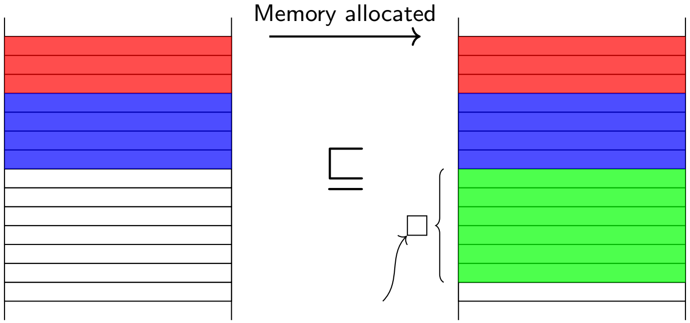


- Memory evolves over time

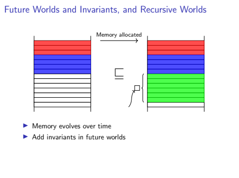


Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

Future Worlds and Invariants, and Recursive Worlds

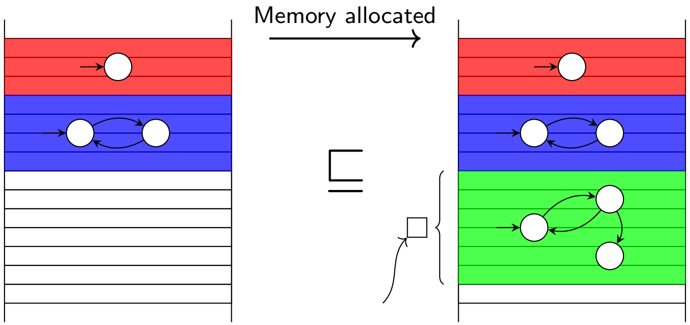


- Memory evolves over time
- Add invariants in future worlds

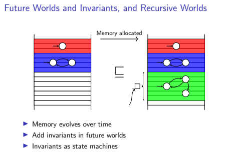

Future Worlds and Invariants, and Recursive Worlds
Memory allocated
⊑
- Memory evolves over time
- Add invariants in future worlds

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

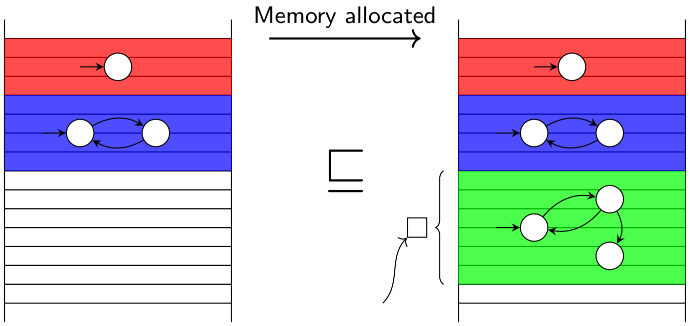


- Memory evolves over time
- Add invariants in future worlds
- Invariants as state machines


Future Worlds and Invariants, and Recursive Worlds
Memory allocated
⊑
- Memory evolves over time
- Add invariants in future worlds
- Invariants as state machines

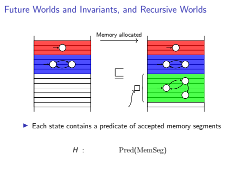
Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

Future Worlds and Invariants, and Recursive Worlds



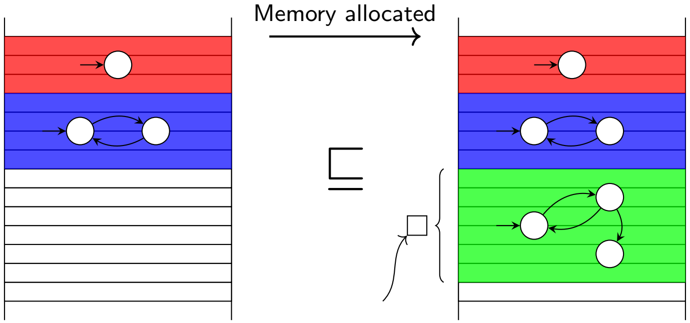
- Each state contains a predicate of accepted memory segments

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


Future Worlds and Invariants, and Recursive Worlds
Memory allocated
- Each state contains a predicate of accepted memory segments
$H :$ Pred(MemSeg)

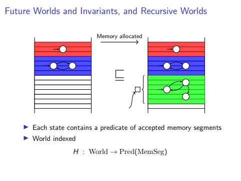
Memory changes over time, for instance new memory may be allocated.Allow worlds to evolve. New invariants can be added to handle freshly allocated memory.Safety of words monotone w.r.t. worlds (which makes it into a Kripke Logical relation).

Future Worlds and Invariants, and Recursive Worlds



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

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

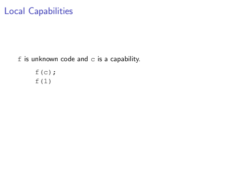

Future Worlds and Invariants, and Recursive Worlds
Memory allocated
⊑
- Each state contains a predicate of accepted memory segments
- World indexed
H : World → Pred(MemSeg)

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)

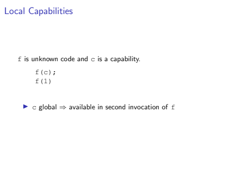
when c global, second invocation in a world where c is safewhen c local, second invocation in a world where c is not (necessarily) safec essentially revoked, so the invariants it relies on need not hold.Two future world relations, one for all capabilities and one for non-local capabilitiesIf c global, 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

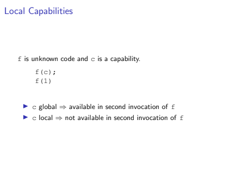
when c global, second invocation in a world where c is safewhen c local, second invocation in a world where c is not (necessarily) safec essentially revoked, so the invariants it relies on need not hold.Two future world relations, one for all capabilities and one for non-local capabilitiesIf c global, 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

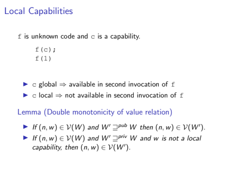
f is unknown code and c is a capability.

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

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.

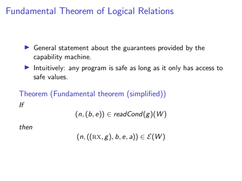
Theorem (Fundamental theorem (simplified))

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

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

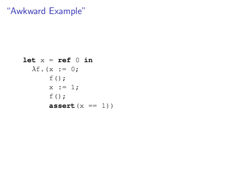
then

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

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))
```



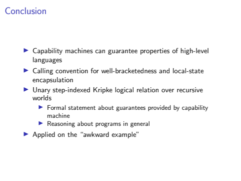
“Awkward Example”

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

example known from the litteratureeven 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 contex 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 faithfull 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
- 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”



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!

Reasoning About a Machine with Local Capabilities

2018-04-13

Thank you!