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

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?

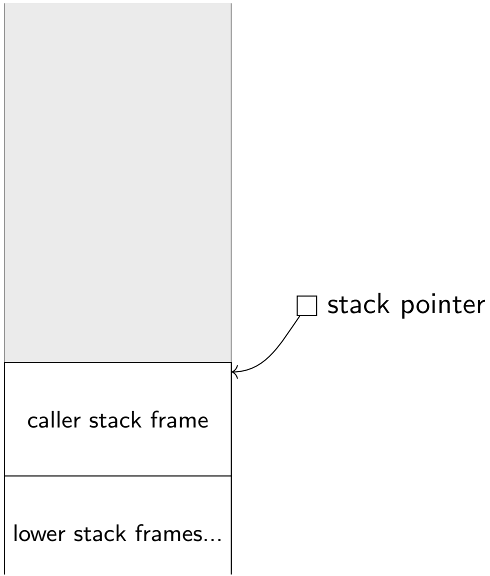
```
let x = ref 0 in
  λf.(x := 0;
      f();
      x := 1;
      f();
      assert(!x == 1))
```

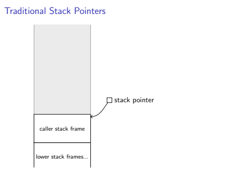

What Does This Program Do?

```
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.

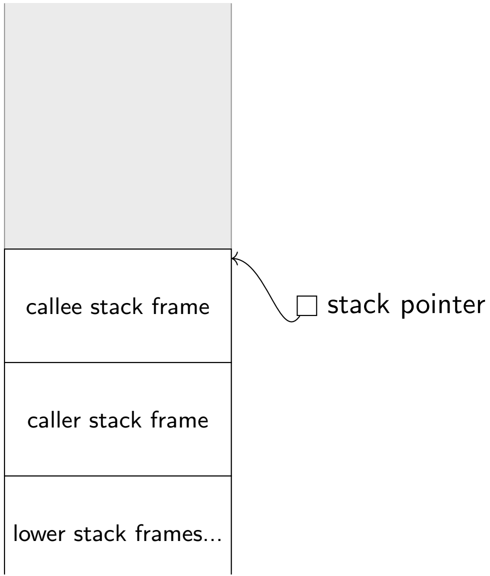
Traditional Stack Pointers

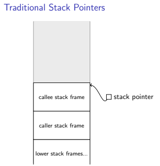




1. Simply put, a caller calls a function which
2. pushes a new stack frame on the stack the callee uses for its execution.
3. When the callee is done, then it returns to the caller by popping its stack frams.

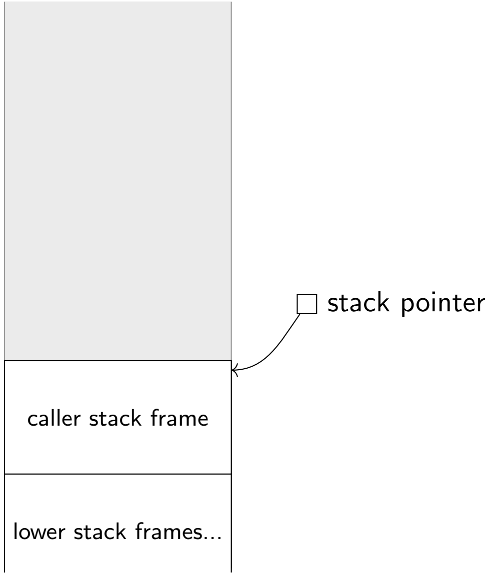
Traditional Stack Pointers

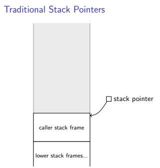




1. Simply put, a caller calls a function which
2. pushes a new stack frame on the stack the callee uses for its execution.
3. When the callee is done, then it returns to the caller by popping its stack frams.

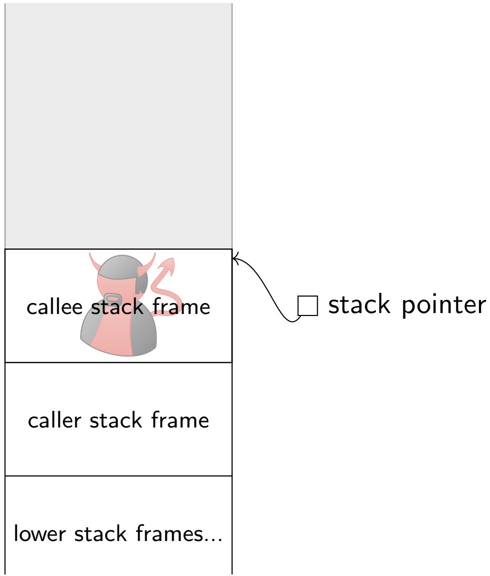
Traditional Stack Pointers

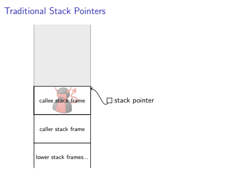




1. Simply put, a caller calls a function which
2. pushes a new stack frame on the stack the callee uses for its execution.
3. When the callee is done, then it returns to the caller by popping its stack frams.

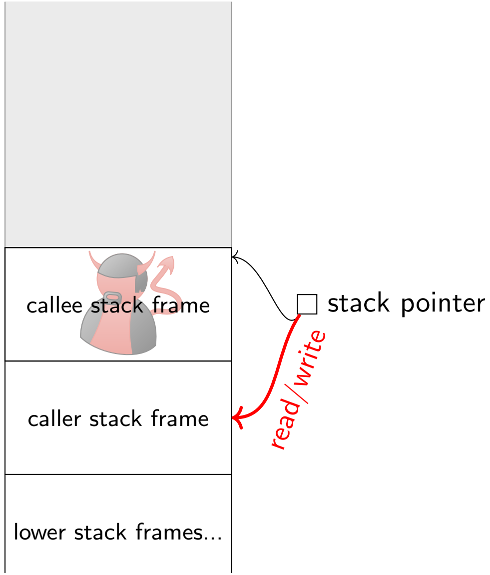
Traditional Stack Pointers

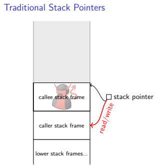




1. If callee (evil) assembly code with no intention to follow the CC, then there are multiple ways for them to break things:

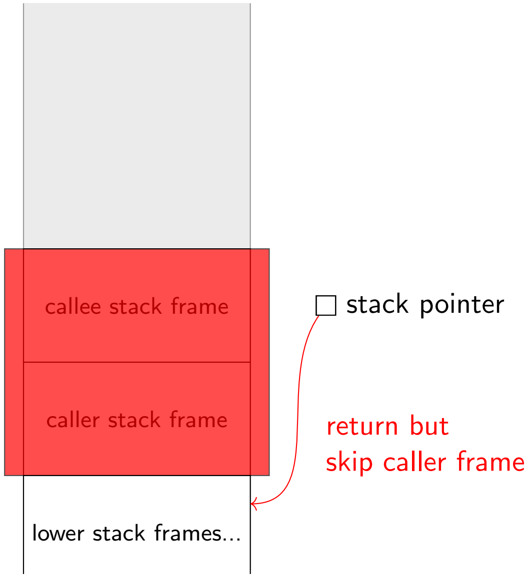
Traditional Stack Pointers

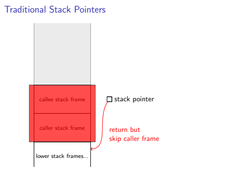




1. If callee (evil) assembly code with no intention to follow the CC, then there are multiple ways for them to break things:
2. Read or write directly from or to the caller's stack frame, breaking local-state encapsulation

Traditional Stack Pointers

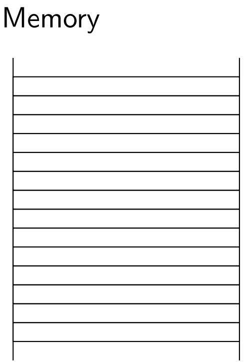


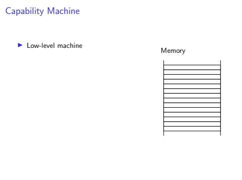


1. If callee (evil) assembly code with no intention to follow the CC, then there are multiple ways for them to break things:
2. Read or write directly from or to the caller's stack frame, breaking local-state encapsulation
3. Skip the caller's stack frame and return to one further down breaking well-bracketedness.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

Capability Machine

- Low-level machine



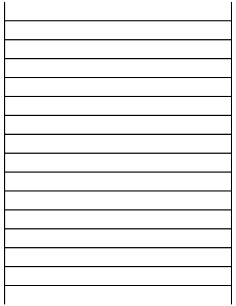


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.

Capability Machine

- Low-level machine
- Capabilities replace pointers

Memory

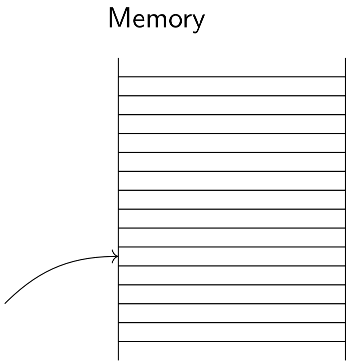


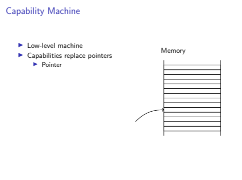
Capability Machine
- Low-level machine
- Capabilities replace pointers
Memory

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

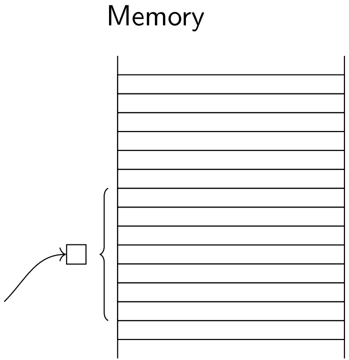


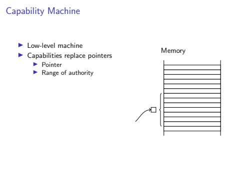


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.

Capability Machine

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

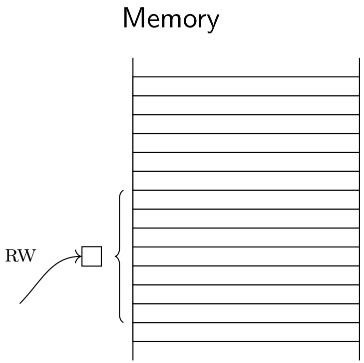


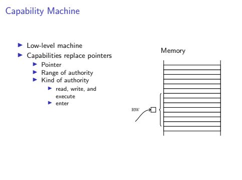


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.

Capability Machine

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

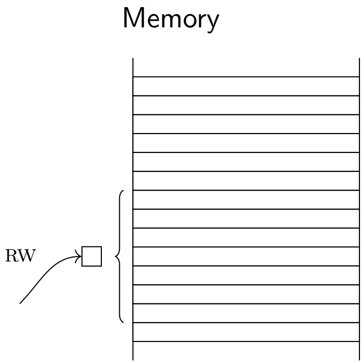


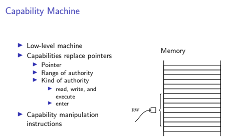


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.
4. Roughly two kinds of capabilities:
5. Memory capabilities, allows you to do all the standard memory operations.
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

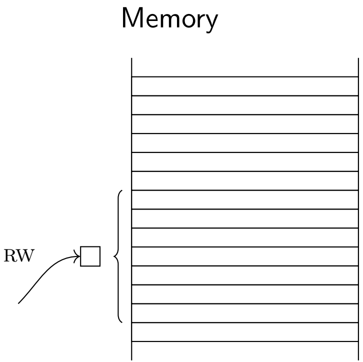


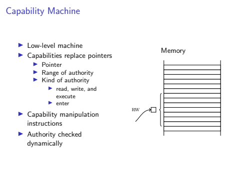


1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.
4. Roughly two kinds of capabilities:
5. Memory capabilities, allows you to do all the standard memory operations.
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

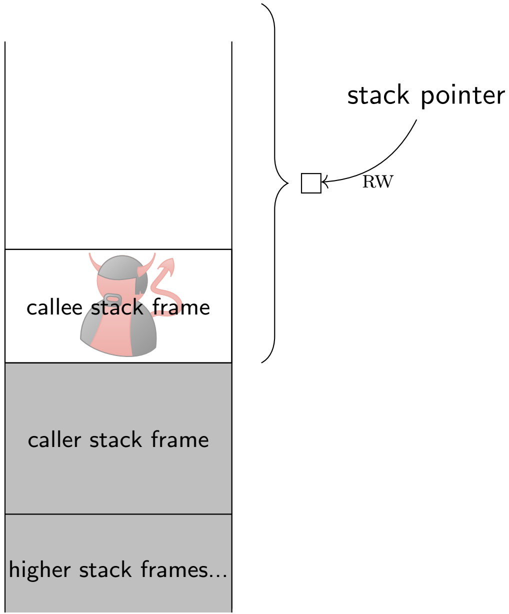
- Low-level machine
- Capabilities replace pointers
  - Pointer
  - Range of authority
  - Kind of authority
    - read, write, and execute
    - enter
- Capability manipulation instructions
- Authority checked dynamically

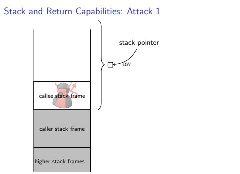




1. Capability machines are low-level machines proposed in the systems community.
2. For instance, the CHERI OS operates on one.
3. Has all the instructions we expect, load, store, jmp, etc.
4. Roughly two kinds of capabilities:
5. Memory capabilities, allows you to do all the standard memory operations.
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.

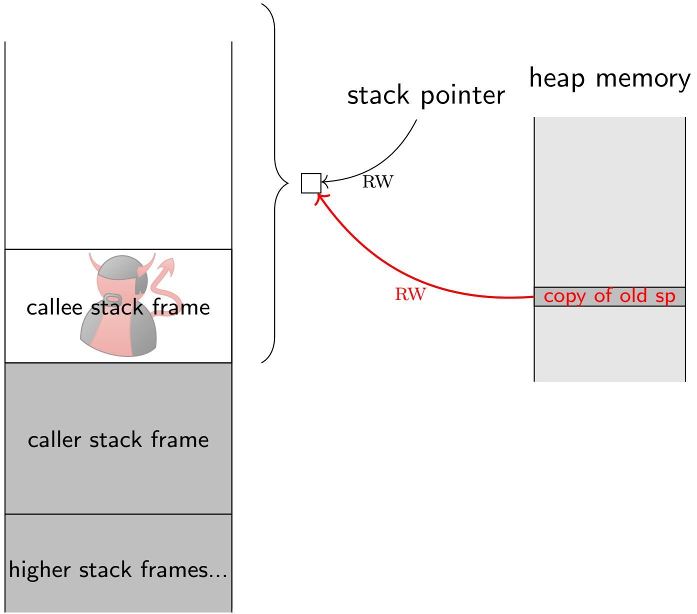
Stack and Return Capabilities: Attack 1

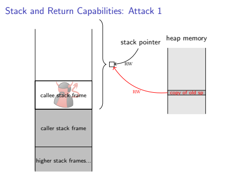




1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.

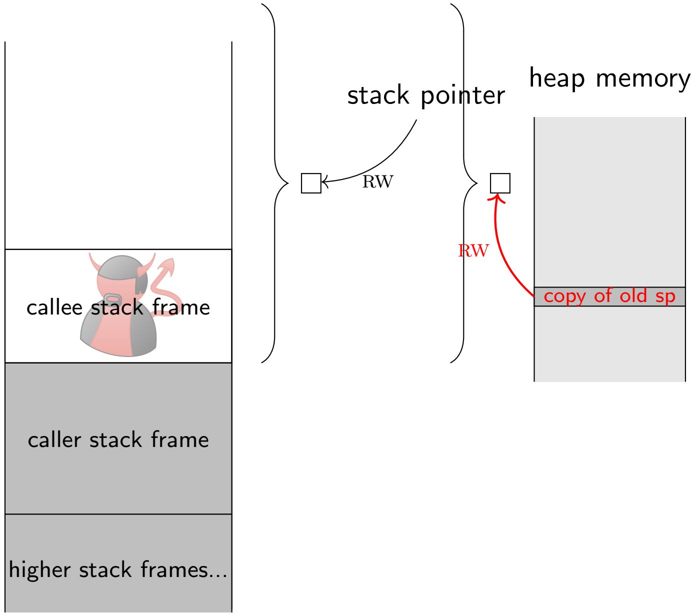
Stack and Return Capabilities: Attack 1

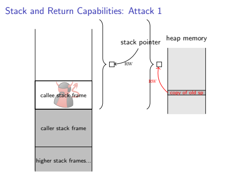




1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.

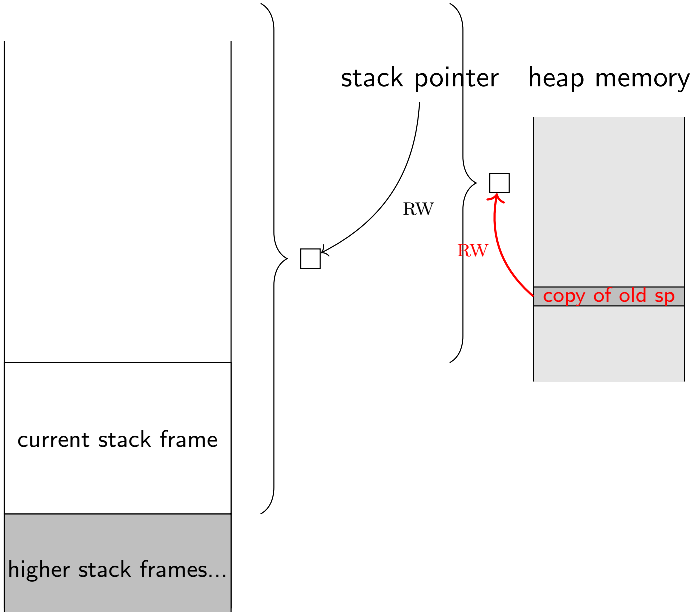
Stack and Return Capabilities: Attack 1

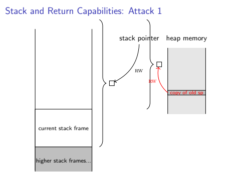




1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.
3. Upon return the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

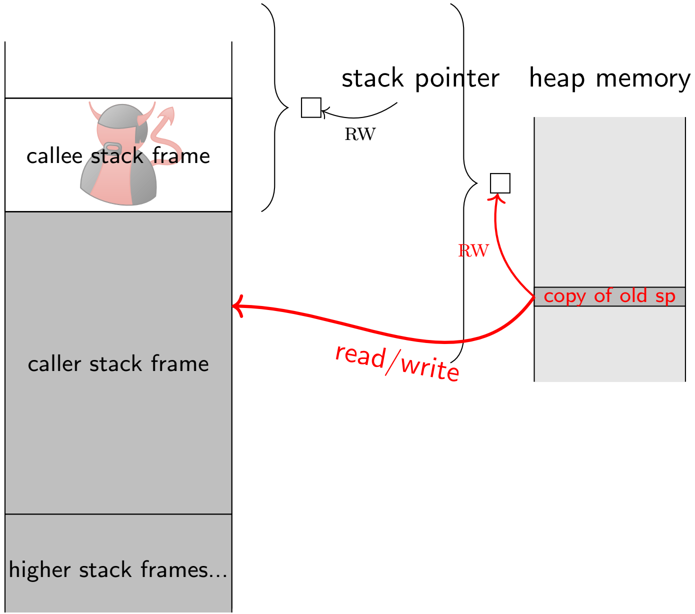
Stack and Return Capabilities: Attack 1

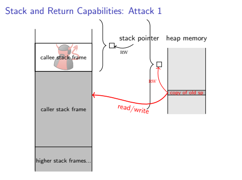




1. Let's see how this changes things: Now the untrusted code cannot immediately read from the caller stack frame, because the stack capability does not have authority over that part of memory.
2. There is nothing that prevents the untrusted code from storing the stk ptr on the heap.
3. Upon return the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

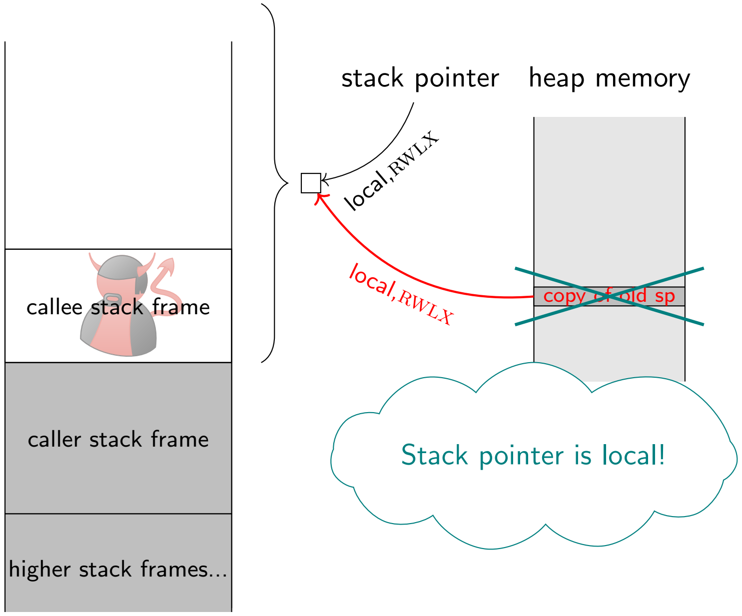
Local Capabilities

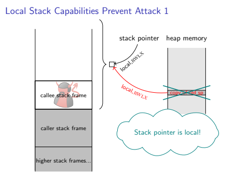
CHERI inspired
- Capabilities tagged with locality (local or global)
- New *write-local* authority
- Local capabilities can only be stored by capabilities with *write-local* permission

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.

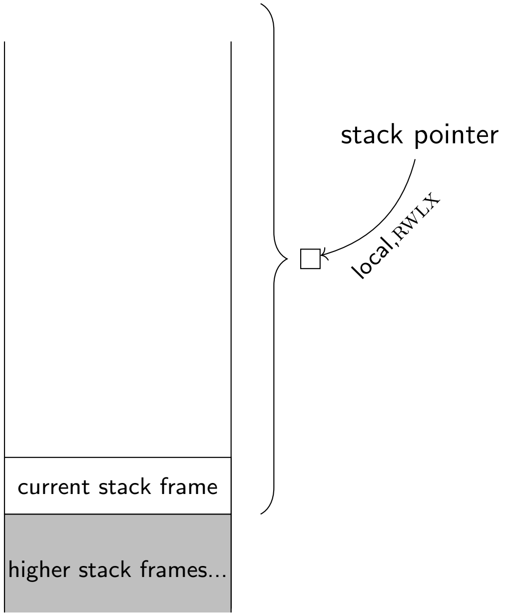
Local Stack Capabilities Prevent Attack 1

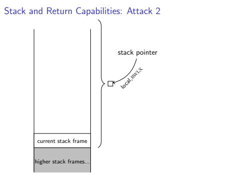




In the attack from before, when the attacker attempts to store the stack capability on the heap, then the machine checks that we have the correct authority to perform the operation. Assuming we only have global capabilities for the heap, it cannot have write-local authority, due to the assumption on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

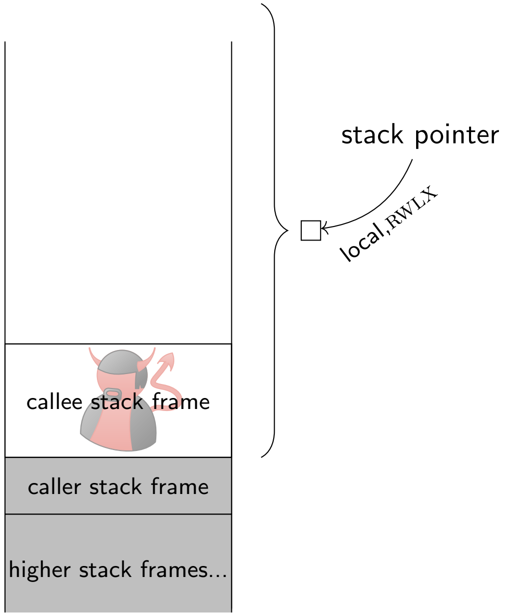
Stack and Return Capabilities: Attack 2

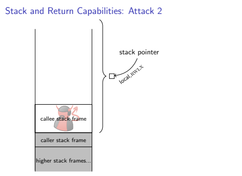




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.

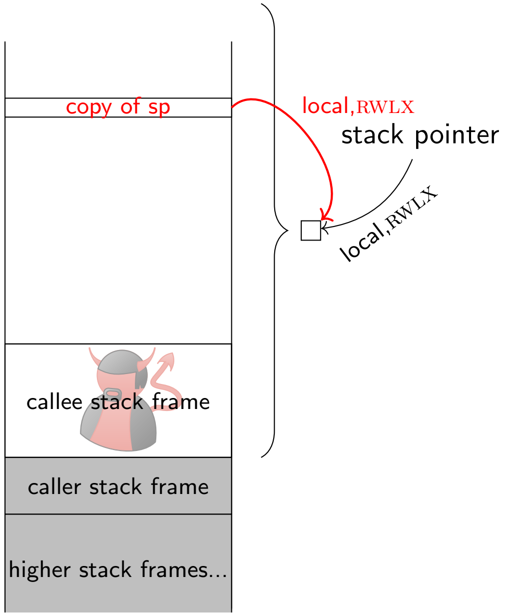
Stack and Return Capabilities: Attack 2

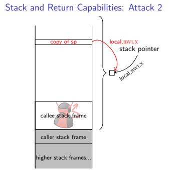




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.

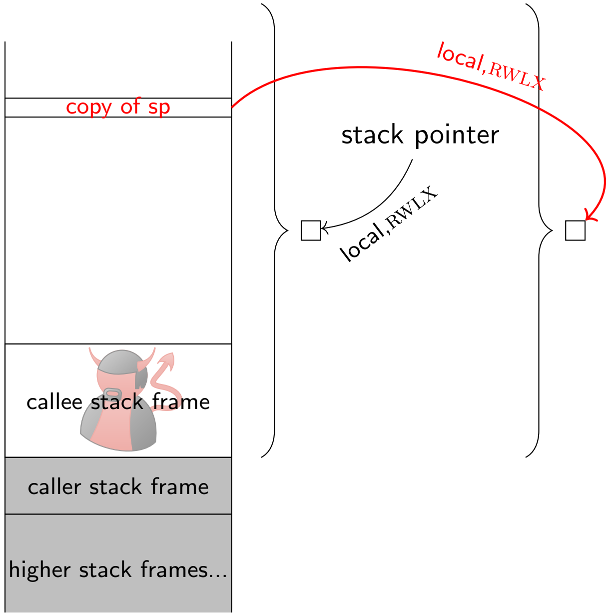
Stack and Return Capabilities: Attack 2

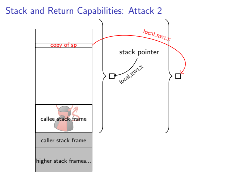




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.

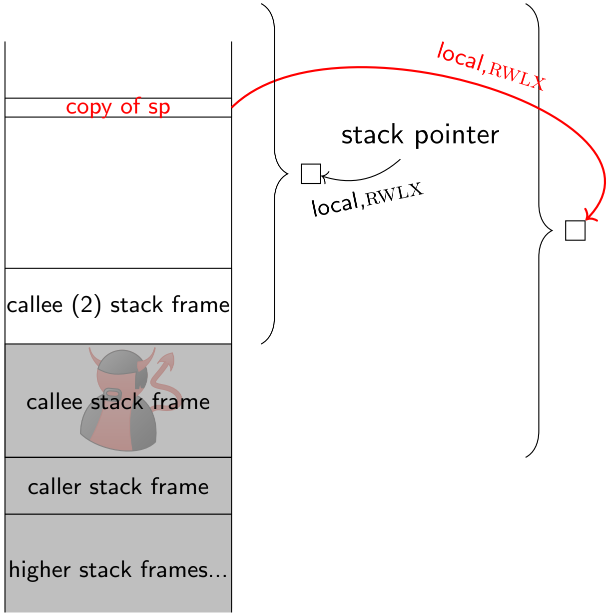
Stack and Return Capabilities: Attack 2

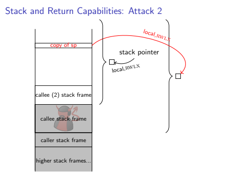




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.

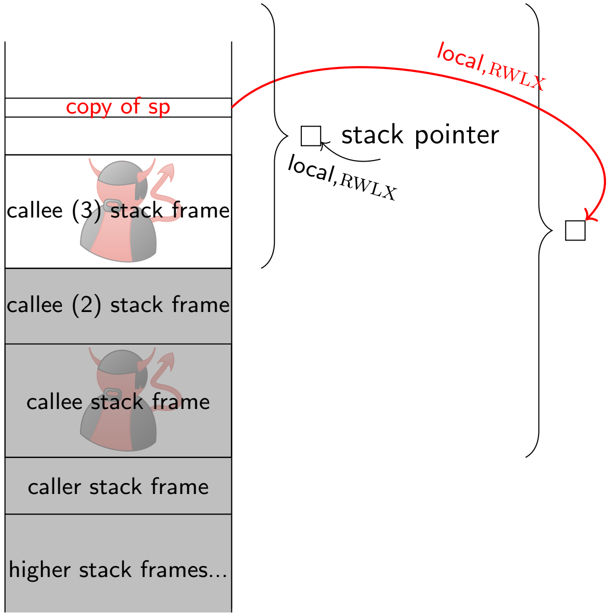
Stack and Return Capabilities: Attack 2

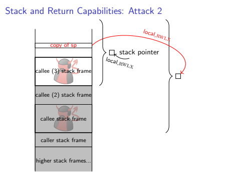




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.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

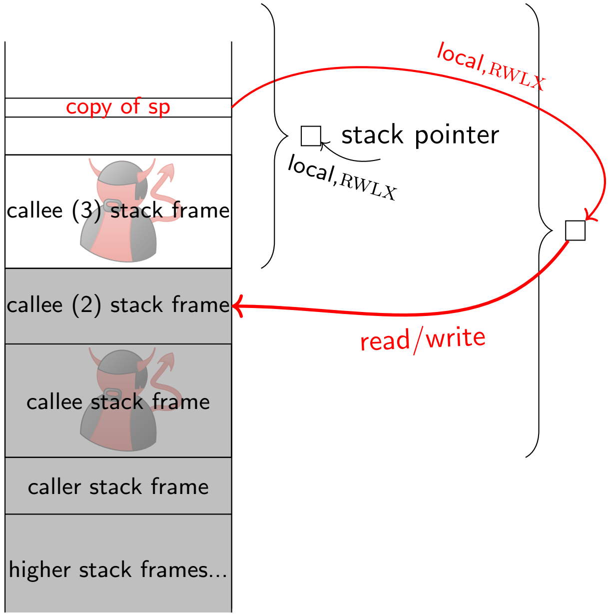
Stack and Return Capabilities: Attack 2

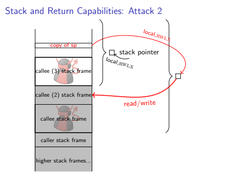




1. While this prevents attack 1, we are not quite safe done.
2. Trusted caller calls untrusted code.
3. untrusted code stores the stack pointer on the stack.
4. stack pointer local, but stack pointer has write local permission, so no problem.
5. untrusted code calls some trusted code with a callback.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

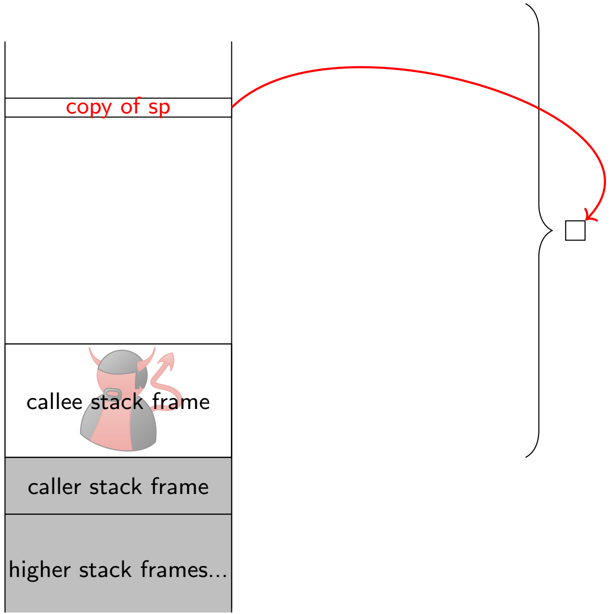
Calling Convention (Continued)

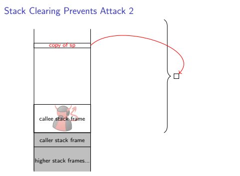
. . .

- 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

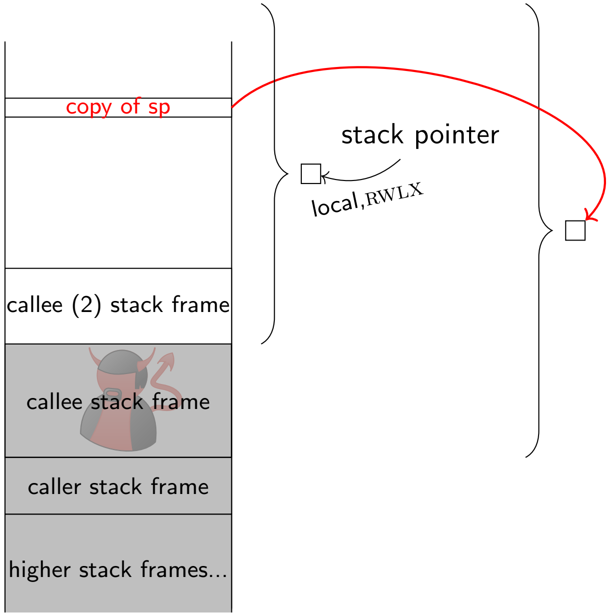
Stack Clearing Prevents Attack 2

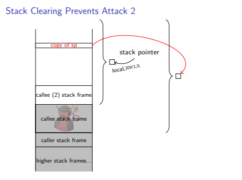




1. Let's see that the addition to the CC prevents attack 2.
2. The untrusted code has been called. It calls the well-behaved code.
3. The well-behaved code does its thing, but this time it clears the stack overwritting the old stack pointer the untrusted code had saved for later.
4. The untrusted code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

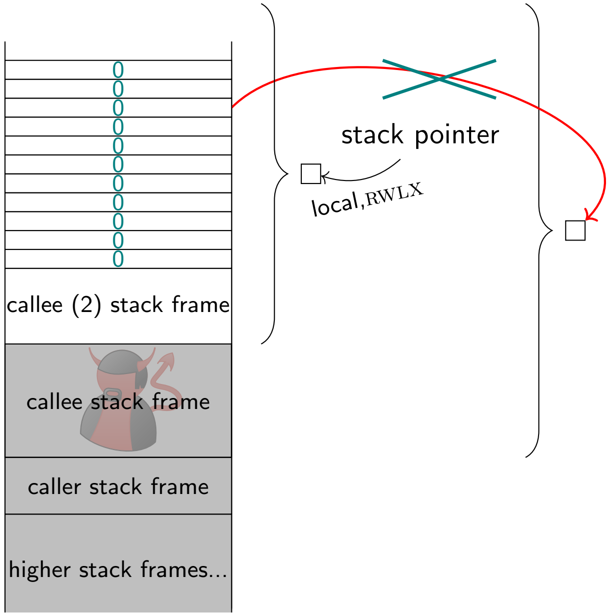
Stack Clearing Prevents Attack 2

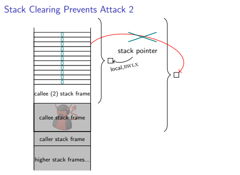




1. Let's see that the addition to the CC prevents attack 2.
2. The untrusted code has been called. It calls the well-behaved code.
3. The well-behaved code does its thing, but this time it clears the stack overwritting the old stack pointer the untrusted code had saved for later.
4. The untrusted code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

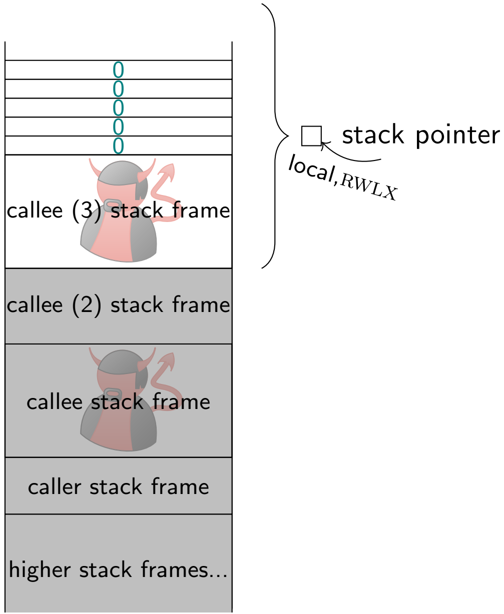
Stack Clearing Prevents Attack 2

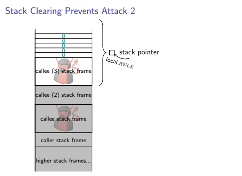




1. Let's see that the addition to the CC prevents attack 2.
2. The untrusted code has been called. It calls the well-behaved code.
3. The well-behaved code does its thing, but this time it clears the stack overwritting the old stack pointer the untrusted code had saved for later.
4. The untrusted code 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

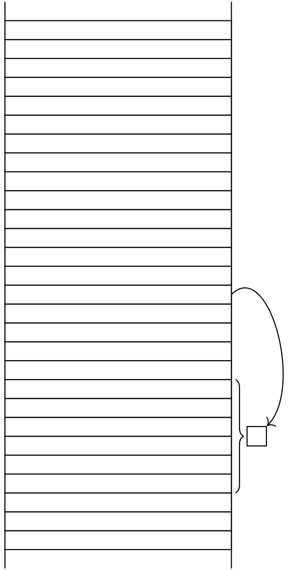
- How can we be sure the calling convention works?
- Unary step-indexed Kripke logical relation over recursive worlds
  - Statement of guarantees provided by the capability machine

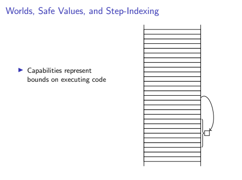
Formalizing the Guarantees of a Capability Machine
- How can we be sure the calling convention works?
- Unary step-indexed Kripke logical relation over recursive worlds
  - Statement of guarantees provided by the capability machine

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

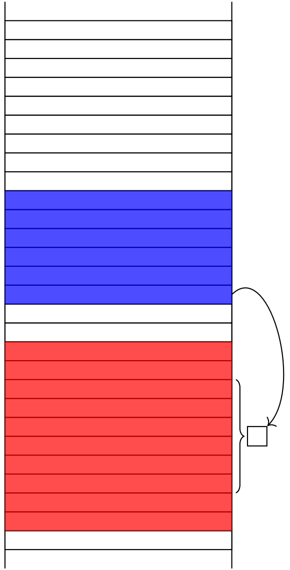


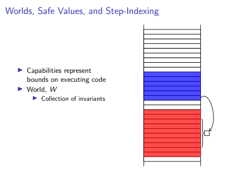


1. Compared to standard assembly language, capabilities executing code has access to represent bound.
2. That is, the capabilities the executing code has access to.
3. What exactly is the bound on. In the system we have considered, no I/O, so memory invariants.
4. Take what the program does into account - allows more fine-grained then simply "read/write"

Worlds, Safe Values, and Step-Indexing

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

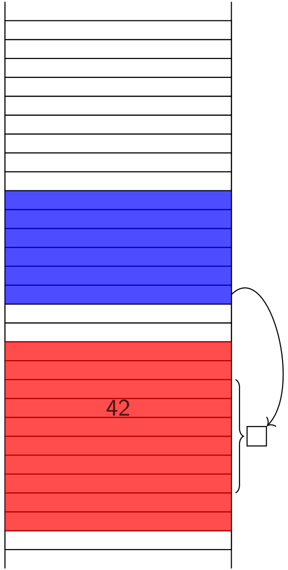


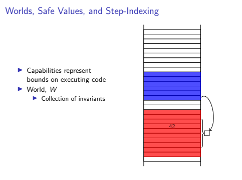


1. Compared to standard assembly language, capabilities executing code has access to represent bound.
2. That is, the capabilities the executing code has access to.
3. What exactly is the bound on. In the system we have considered, no I/O, so memory invariants.
4. Take what the program does into account - allows more fine-grained then simply "read/write"
5. The collection of invariants is what we call the world (and it can bethought of as a model of the memory)
6. Here colored region represents invariant. A simple invariant could be a specific address contains 42.

Worlds, Safe Values, and Step-Indexing

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

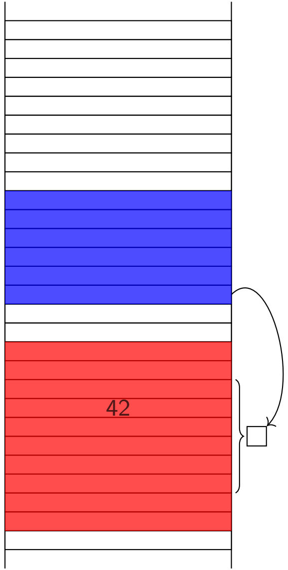


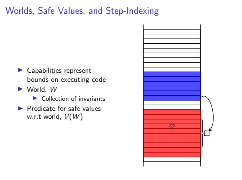


1. Compared to standard assembly language, capabilities executing code has access to represent bound.
2. That is, the capabilities the executing code has access to.
3. What exactly is the bound on. In the system we have considered, no I/O, so memory invariants.
4. Take what the program does into account - allows more fine-grained then simply "read/write"
5. The collection of invariants is what we call the world (and it can bethought of as a model of the memory)
6. Here colored region represents invariant. A simple invariant could be a specific address contains 42.
7. We are interested in all the words on the machine that preserve these invariants, so

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, $\mathcal{V}(W)$

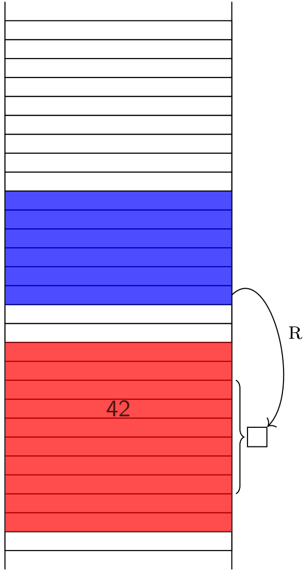


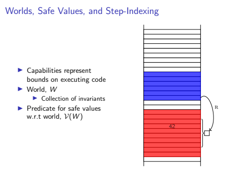


1. We are interested in all the words on the machine that preserve these invariants, so
2. We define a predicate with theses values. Whether a capability violates invariants obviously depend on the invariants, so the predicate has to be world-indexed.
3. Let's see some examples:

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, $\mathcal{V}(W)$

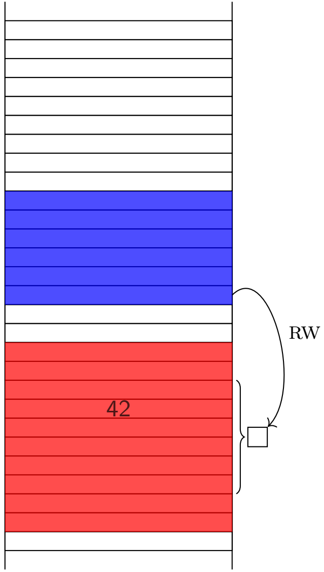


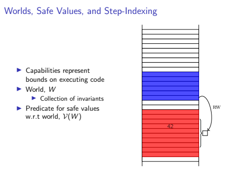


1. We are interested in all the words on the machine that preserve these invariants, so
2. We define a predicate with theses values. Whether a capability violates invariants obviously depend on the invariants, so the predicate has to be world-indexed.
3. Let's see some examples:
4. If the capability on the slide has read authority, then it cannot violate the simple invariant.

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, $\mathcal{V}(W)$

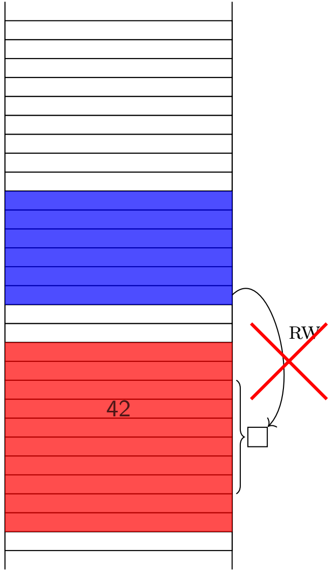


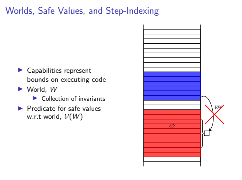


1. We are interested in all the words on the machine that preserve these invariants, so
2. We define a predicate with theses values. Whether a capability violates invariants obviously depend on the invariants, so the predicate has to be world-indexed.
3. Let's see some examples:
4. If the capability on the slide has read authority, then it cannot violate the simple invariant.
5. If the capability on the slide also has write authority, then it can violate the invariant by simply overwriting that address with a different number. Not safe.

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, $\mathcal{V}(W)$

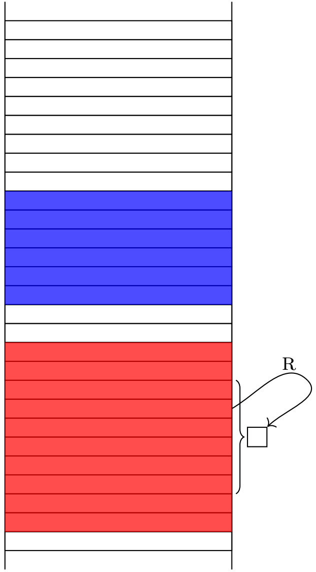


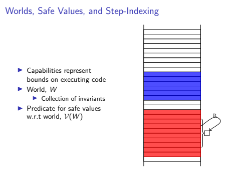


1. We are interested in all the words on the machine that preserve these invariants, so
2. We define a predicate with theses values. Whether a capability violates invariants obviously depend on the invariants, so the predicate has to be world-indexed.
3. Let's see some examples:
4. If the capability on the slide has read authority, then it cannot violate the simple invariant.
5. If the capability on the slide also has write authority, then it can violate the invariant by simply overwriting that address with a different number. Not safe.
6. Generally speaking a capability that can read is safe when it only can read safe words. What happens when it is stored at an address that it has authority over itself?
7. It is safe only if it can read only safe values which requires it to be safe.

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, $\mathcal{V}(W)$

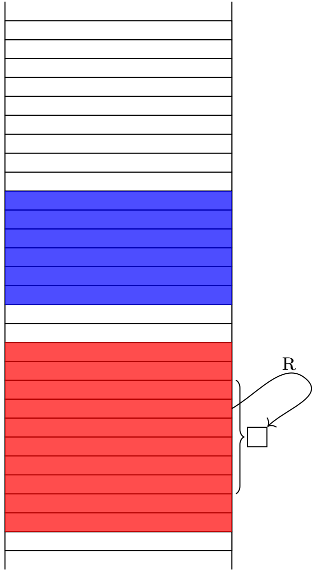


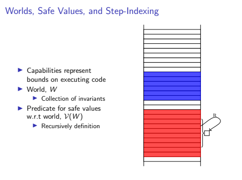


1. It is safe only if it can read only safe values which requires it to be safe.
2. Need to take a fixed-point. Made possible by step-indexing.

Worlds, Safe Values, and Step-Indexing

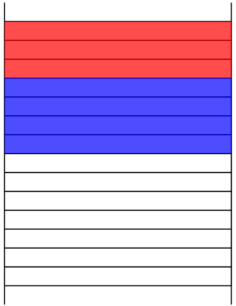
- Capabilities represent bounds on executing code
- World, W
  - Collection of invariants
- Predicate for safe values w.r.t world, $\mathcal{V}(W)$
  - Recursively definition



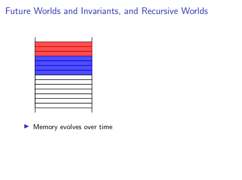


1. It is safe only if it can read only safe values which requires it to be safe.
2. Need to take a fixed-point. Made possible by step-indexing.
3. Related to similar issue for languages with recursive types and ML-like references.

Future Worlds and Invariants, and Recursive Worlds

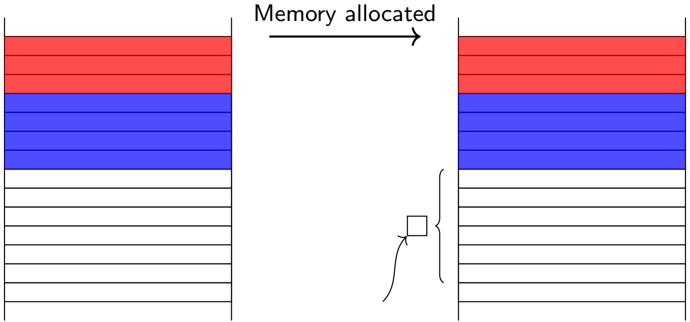


- Memory evolves over time

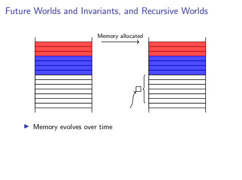


1. Like languages with ML-references, memory changes over time. Example, if we are in this memory with two invariants and more is memory allocation. World need to cope with this.

Future Worlds and Invariants, and Recursive Worlds

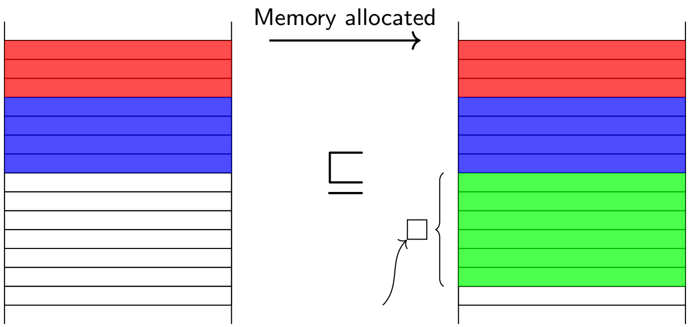


- Memory evolves over time

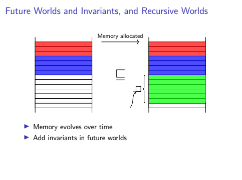


1. Like languages with ML-references, memory changes over time. Example, if we are in this memory with two invariants and more is memory allocation. World need to cope with this.
2. Notion called future worlds. Allow us to add invariants.

Future Worlds and Invariants, and Recursive Worlds

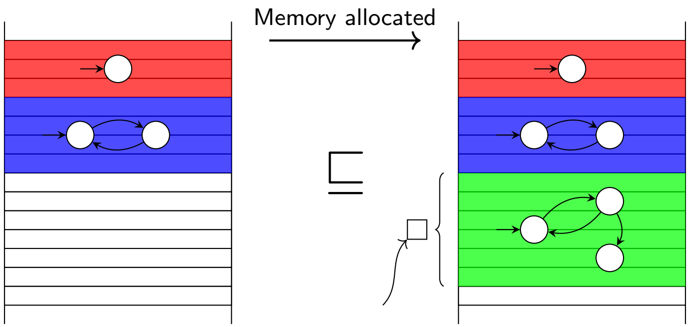


- Memory evolves over time
- Add invariants in future worlds

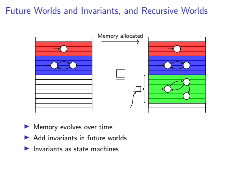


1. Like languages with ML-references, memory changes over time. Example, if we are in this memory with two invariants and more is memory allocation. World need to cope with this.
2. Notion called future worlds. Allow us to add invariants.
3. Memory may be repurposed. Satic invariants don't model this. Alle invariants have an internal state machine. State machine progress in future worlds.

Future Worlds and Invariants, and Recursive Worlds

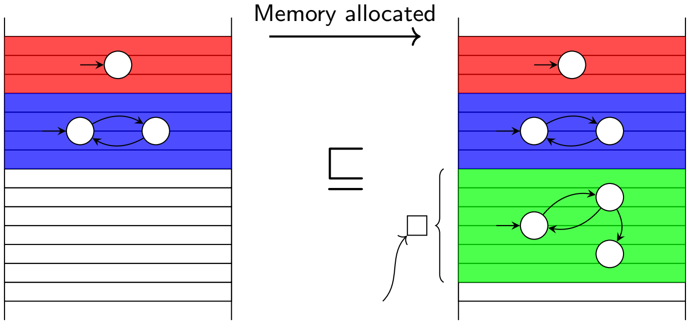


- Memory evolves over time
- Add invariants in future worlds
- Invariants as state machines



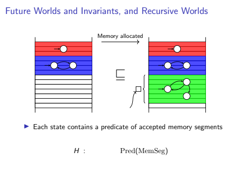
1. Like languages with ML-references, memory changes over time. Example, if we are in this memory with two invariants and more is memory allocation. World need to cope with this.
2. Notion called future worlds. Allow us to add invariants.
3. Memory may be repurposed. Satic invariants don't model this. Alle invariants have an internal state machine. State machine progress in future worlds.
4. Each state is associated with a predicate of all memories that respect the invariant.
5. Safety monotone wrt worlds, Kripke logical relation

Future Worlds and Invariants, and Recursive Worlds



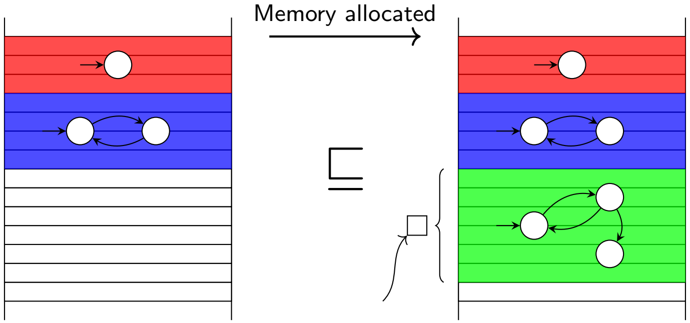
- Each state contains a predicate of accepted memory segments

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



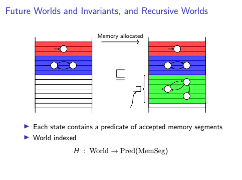
1. Like languages with ML-references, memory changes over time. Example, if we are in this memory with two invariants and more is memory allocation. World need to cope with this.
2. Notion called future worlds. Allow us to add invariants.
3. Memory may be repurposed. Satic invariants don't model this. Alle invariants have an internal state machine. State machine progress in future worlds.
4. Each state is associated with a predicate of all memories that respect the invariant.
5. Safety monotone wrt worlds, Kripke logical relation
6. We want to express "all memories with safe values". World dependent, so the predicate needs to be world indexed.

Future Worlds and Invariants, and Recursive Worlds



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

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



1. Like languages with ML-references, memory changes over time. Example, if we are in this memory with two invariants and more is memory allocation. World need to cope with this.
2. Notion called <u>future worlds</u>. Allow us to <u>add invariants</u>.
3. Memory may be repurposed. Satic invariants don't model this. Alle invariants have an internal state machine. <u>State machine progress in future worlds</u>.
4. Each state is associated with a predicate of all memories that respect the invariant.
5. <u>Safety monotone wrt worlds</u>, <u>Kripke logical relation</u>
6. We want to express "all memories with safe values". World dependent, so the predicate needs to be world indexed.
7. Worlds with invariants with state machines with predicates that are world indexed - circular definition.
8. Resolved using standard techniques from the litterature (essentially advanced step-indexing).

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)

1. Now consider how local capabilities affect all this.
2. Consider this simple example, first f is called with capability c as an argument. Then f is called with unit.
3. What may we assume about c in the second invocation of f?
4. Depends on c!

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

1. (Cont) Depends on c!
2. If c is global, then it can be stored on the heap, so it needs to remain safe for the remainder of the execution. When f is invoked c must still be safe.

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

1. (Cont) Depends on c!
2. If c is global, then it can be stored on the heap, so it needs to remain safe for the remainder of the execution. When f is invoked c must still be safe.
3. If c local, then CC dictates clear all the places c may reside, so in the second invocation c need not remain safe.
4. Need two future world relations. In both, global capabilities must remain safe. In one local capabilities need not. We have public and private future world relation.

Local Capabilities

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

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

- `c` global $\Rightarrow$ available in second invocation of `f`
- `c` local $\Rightarrow$ not available in second invocation of `f`

Lemma (Double monotonicity of safety predicate)

- *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 safety predicate)
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′).

1. This double monotonicity lemma expresses the assumptions we can make, namely in public future world all capabilities remain valid and in private future worlds local capabilities need not remain valid.
2. In the example this means that if c is local, then it is okay to invoke f a second time in a private future world as c need not be safe anymore.
3. On the other hand, if c is global, then the invokation of f must be in a public future world, so c remains safe.
4. Related to public private future worlds, state machines with pub/priv transitions

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

1. Now for the formal statement about guarantees
2. readCond: only safe values in the interval.
3. E-relation: when capability used as the pc with register file and memory respecting the world, then the execution respects the memory invariants.
4. In other words, take an arbitrary capability that only has access to safe values, then the memory invariants are preserved when we use it for execution.
5. The instructions it execute don't matter only the authority it can use.

“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))
```


1. known from the litterature
2. in ML difficult to reason about as callback f can be the closure it self. (so x can be either)
3. assert may fail if calls not well-bracketed
4. can do more things to attack well-bracketedness low-level. Context need not follow CC, so well-behaved code cannot rely on behavior of untrusted code.
5. We have proven a faithful translation correct. That is the assert never fails. Notice, dynamic checks, so machine can fail, but we set it up, so we can distinguish this from assertion failure.
6. More semantic statement of guarantees allow us to prove it.

Conclusion

- Capability machines can guarantee properties of high-level languages.

Conclusion

- Capability machines can guarantee properties of high-level languages.

Conclusion

- Capability machines can guarantee properties of high-level languages.
- We developed a calling convention that ensures well-bracketedness and local-state encapsulation.

Conclusion

- Capability machines can guarantee properties of high-level languages.
- We developed a calling convention that ensures well-bracketedness and local-state encapsulation.

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.

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.

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.

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.

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.

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.

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!