Tales of Belgium: Reasoning about Capability Machines using Logical Relations

Lau Skorstengaard

Aarhus University

Aarhus University, December 2016
Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion
Leuven
Leuven
Leuven
Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion
Why should I care about capability machines?

Current low-level protection mechanisms

- Coarse-grained compartmentalisation
- Expensive context switches
- Well suited for high-level applications
- Does not scale well
- E.g., virtual memory
Why should I care about capability machines?

**Capability machines**
- Fine-grained compartmentalisation
- Cheap compartments
- Fine-grained sharing
- Well suited for applications with need for many compartments
Capabilities

What is a capability?
Capabilities

What is a capability?

- *Unforgeable* token of authority
Capabilities

What is a capability?

▶ *Unforgeable* token of authority

What is a capability in a capability machine?
Capabilities

What is a capability?

▶ *Unforgeable* token of authority

What is a capability in a capability machine?

▶ Unforgeable pointer

*Figure: CHERI capability [1]*
Capabilities

What is a capability?

▶ *Unforgeable* token of authority

What is a capability in a capability machine?

▶ Unforgeable pointer
▶ Range of memory

Figure: CHERI capability [1]
Capabilities

What is a capability?
- *Unforgeable* token of authority

What is a capability in a capability machine?
- Unforgeable pointer
- Range of memory
- Permission

Figure: CHERI capability [1]
Capability permissions

- Read
- Write
- Execute
Capability permissions

- Read
- Write
- Execute
- Enter

When jumped to, it becomes a read and execute capability.

Cannot be used in any other way.

Used by distrusting pieces of code to cross security boundaries.

Modularisation
Capability permissions

- Read
- Write
- Execute
- Enter
  - When jumped to, it becomes a read and execute capability
  - Cannot be used in any other way
Capability permissions

- Read
- Write
- Execute
- Enter
  - When jumped to, it becomes a read and execute capability
  - Cannot be used in any other way
  - Used by distrusting pieces of code to cross security boundaries
Capability permissions

- Read
- Write
- Execute
- Enter
  - When jumped to, it becomes a read and execute capability
  - Cannot be used in any other way
  - Used by distrusting pieces of code to cross security boundaries
  - Modularisation
Capability machine instructions

- Same instructions as in a normal low-level machine
Capability machine instructions

- Same instructions as in a normal low-level machine
  - jmp, jnz, move, plus, load, store

- Instructions may require capability with certain permission.

- Capability manipulation instructions
  - lea, restrict, subseg

- No instruction generates new capability
- Manipulation of capabilities cannot result in authority amplification
Capability machine instructions

- Same instructions as in a normal low-level machine
  - `jmp`, `jnz`, `move`, `plus`, `load`, `store`
  - Instructions may require capability with certain permission.
Capability machine instructions

- Same instructions as in a normal low-level machine
  - jmp, jnz, move, plus, load, store
  - Instructions may require capability with certain permission.
- Capability manipulation instructions
Capability machine instructions

- Same instructions as in a normal low-level machine
  - jmp, jnz, move, plus, load, store
  - Instructions may require capability with certain permission.
- Capability manipulation instructions
  - lea, restrict, subseg
Capability machine instructions

- Same instructions as in a normal low-level machine
  - jmp, jnz, move, plus, load, store
  - Instructions may require capability with certain permission.

- Capability manipulation instructions
  - lea, restrict, subseg
  - No instruction generates new capability
Capability machine instructions

- Same instructions as in a normal low-level machine
  - jmp, jnz, move, plus, load, store
  - Instructions may require capability with certain permission.

- Capability manipulation instructions
  - lea, restrict, subseg
  - No instruction generates new capability
  - Manipulation of capabilities cannot result in authority amplification
Capability machine overview

- Capabilities
Capability machine overview

- Capabilities
  - Permissions
Capability machine overview

- Capabilities
  - Permissions
  - Range of authority
Capability machine overview

- Capabilities
  - Permissions
  - Range of authority
- Capability aware instructions
Capability machine overview

- Capabilities
  - Permissions
  - Range of authority
- Capability aware instructions
- Memory and registers
Capability machine overview

- Capabilities
  - Permissions
  - Range of authority
- Capability aware instructions
- Memory and registers
  - Can contain data and capabilities
Capability machine overview

- Capabilities
  - Permissions
  - Range of authority
- Capability aware instructions
- Memory and registers
  - Can contain data and capabilities
  - Capabilities tagged
Simple capability machine limitations

- No revocation of capabilities
Simple capability machine limitations

- No revocation of capabilities
- Simulating revocation of enter capabilities:
  - On jump overwrite first address with fail instruction
  - Subsequent jumps fail
- Issues:
  - Memory leak
  - Does not scale to other types of permissions
Simple capability machine limitations

- No revocation of capabilities
- Simulating revocation of enter capabilities:
  - On jump overwrite first address with fail instruction
Simple capability machine limitations

- No revocation of capabilities
- Simulating revocation of enter capabilities:
  - On jump overwrite first address with fail instruction
  - Subsequent jumps fail

Issues:
- Memory leak
- Does not scale to other types of permissions
Simple capability machine limitations

- No revocation of capabilities
- Simulating revocation of enter capabilities:
  - On jump overwrite first address with fail instruction
  - Subsequent jumps fail
- Issues:
Simple capability machine limitations

- No revocation of capabilities
- Simulating revocation of enter capabilities:
  - On jump overwrite first address with fail instruction
  - Subsequent jumps fail
- Issues:
  - Memory leak
Simple capability machine limitations

- No revocation of capabilities
- Simulating revocation of enter capabilities:
  - On jump overwrite first address with fail instruction
  - Subsequent jumps fail
- Issues:
  - Memory leak
  - Does not scale to other types of permissions
Local capabilities

- Idea: New type of capability that cannot be stored when “crossing security boundaries”
Local capabilities

- Idea: New type of capability that cannot be stored when “crossing security boundaries”
- Capabilities get two new fields:
Local capabilities

- Idea: New type of capability that cannot be stored when “crossing security boundaries”
- Capabilities get two new fields:
  - Local/(global)
Local capabilities

- Idea: New type of capability that cannot be stored when “crossing security boundaries”
- Capabilities get two new fields:
  - Local/(global)
  - Permit write local (pwl)
Local capabilities

- Idea: New type of capability that cannot be stored when “crossing security boundaries”
- Capabilities get two new fields:
  - Local/(global)
  - Permit write local (pwl)
- Only pwl-capabilities can write local capabilities.
Local capabilities

- Idea: New type of capability that cannot be stored when “crossing security boundaries”
- Capabilities get two new fields:
  - Local/(global)
  - Permit write local (pwl)
- Only pwl-capabilities can write local capabilities.
- Gives simple temporal revocation, but
Local capabilities

- Idea: New type of capability that cannot be stored when "crossing security boundaries"
- Capabilities get two new fields:
  - Local/(global)
  - Permit write local (pwl)
- Only pwl-capabilities can write local capabilities.
- Gives simple temporal revocation, but
  - requires no global pwl-capabilities
Local capabilities

- Idea: New type of capability that cannot be stored when “crossing security boundaries”
- Capabilities get two new fields:
  - Local/(global)
  - Permit write local (pwl)
- Only pwl-capabilities can write local capabilities.
- Gives simple temporal revocation, but
  - requires no global pwl-capabilities
  - enforcement depends on programming discipline.
Brussels
Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion
Formalisation - Permissions

Permissions

- To simplify matters, we only allow certain combinations of permissions

\[
\text{Perm} \overset{\text{def}}{=} \{ \ldots \}
\]
Formalisation - Permissions

Permissions

- To simplify matters, we only allow certain combinations of permissions
- No permissions,

\[
\text{Perm} \overset{\text{def}}{=} \{ \text{o, r, rw, rw, rl, rx, e, rwx, rwex}\}
\]
To simplify matters, we only allow certain combinations of permissions:

- No permissions, read only,
- Read-write, read-execute, enter,
- Read-write-execute, read-write-local-execute

\[ \text{Perm} \overset{\text{def}}{=} \{ \text{o, ro,} \} \]
Formalisation - Permissions

Permissions

▶ To simplify matters, we only allow certain combinations of permissions
▶ No permissions, read only, read-write,

\[
\text{Perm}^{\text{def}} = \{ o, \text{ro}, \text{rw}, \text{rwx}, \text{rwl}, \text{rx}, \text{e}, \text{rwx}, \text{rwlx} \}
\]
Permissions

- To simplify matters, we only allow certain combinations of permissions
- No permissions, read only, read-write, read-execute,

\[
\text{Perm} \overset{\text{def}}{=} \{ \text{o, ro, rw, rx, } \}
\]
Formalisation - Permissions

Permissions

- To simplify matters, we only allow certain combinations of permissions
- No permissions, read only, read-write, read-execute, enter,

\[
\text{Perm} \stackrel{\text{def}}{=} \{ o, \text{ro}, \text{rw}, \text{rx}, e, \text{e} \}
\]
Formalisation - Permissions

Permissions

- To simplify matters, we only allow certain combinations of permissions
- No permissions, read only, read-write, read-execute, enter, read-write-execute,

\[
\text{Perm} \overset{\text{def}}{=} \{ \text{o, ro, rw, rx, e, rwx,} \}
\]
Permissions

To simplify matters, we only allow certain combinations of permissions

- No permissions, read only, read-write, read-’write-local’
- read-execute, enter, read-write-execute,
- read-’write-local’-execute

\[
\text{Perm} \stackrel{\text{def}}{=} \{ o, \text{ro}, \text{rw}, \text{rwl}, \text{rx}, \text{e}, \text{rwx}, \text{rwlx}\}
\]
Permissions ordering

Figure: Permission hierarchy
Formalisation - Locality

**Locality**

Global ::= \{global, local\}

**Locality ordering**

global

| local

*Figure: Locality hierarchy*
Formalisation - Capabilities

Capability

\[ \text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr} \times \text{Addr} \]

Example: \[(e, \text{local}), 30, 42, 30)\]
Formalisation - Capabilities

Capability

- Permission and locality

\[
\text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr} \times \text{Addr}
\]

Example: \((e_{\text{local}}, 30, 42, 30)\)
Formalisation - Capabilities

Capability

- Permission and locality

\[
\text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global})
\]
Formalisation - Capabilities

Capability

- Permission and locality
- Range of authority

\[ \text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \]
Formalisation - Capabilities

**Capability**
- Permission and locality
- Range of authority

\[ \text{Addr} \overset{\text{def}}{=} \mathbb{N} \]

\[ \text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \]
Formalisation - Capabilities

Capability
- Permission and locality
- Range of authority

\[
\text{Addr} \overset{\text{def}}{=} \mathbb{N}
\]

\[
\text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr}
\]
Formalisation - Capabilities

Capability
- Permission and locality
- Range of authority
- Pointer

\[ \text{Addr} \overset{\text{def}}{=} \mathbb{N} \]

\[ \text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr} \]
Formalisation - Capabilities

Capability

- Permission and locality
- Range of authority
- Pointer

\[
\text{Addr} \overset{\text{def}}{=} \mathbb{N}
\]

\[
\text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr} \times \text{Addr}
\]
Formalisation - Capabilities

**Capability**

- Permission and locality
- Range of authority
- Pointer

\[
\text{Addr} \overset{\text{def}}{=} \mathbb{N}
\]

\[
\text{Cap} \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr} \times \text{Addr}
\]

Example: \(((e, \text{local}), 30, 42, 30)\)
Formalisation - Words and register file

Words

Word $\overset{\text{def}}{=} \ldots$
Formalisation - Words and register file

**Words**
- Capability

\[
\text{Word} \overset{\text{def}}{=} 
\]
Words
- Capability

\[
\text{Word} \overset{\text{def}}{=} \text{Cap}
\]
Formalisation - Words and register file

Words

- Capability
- Data (and instructions)

\[ \text{Word} \overset{def}{=} \text{Cap} \]
Formalisation - Words and register file

**Words**

- Capability
- Data (and instructions)

\[
\text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z}
\]
Words

- Capability
- Data (and instructions)
- In the real machine capabilities are tagged

\[ \text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z} \]
Formalisation - Words and register file

Words

- Capability
- Data (and instructions)
- In the real machine capabilities are tagged

\[ \text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z} \]

Register file

\[ \text{Reg} \overset{\text{def}}{=} \]

Formalisation - Words and register file

Words

- Capability
- Data (and instructions)
- In the real machine capabilities are tagged
  \[
  \text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z}
  \]

Register file

- Assume finite set of registers RegisterName \( \ni \) pc

  \[
  \text{Reg} \overset{\text{def}}{=} 
  \]
Formalisation - Words and register file

**Words**

- Capability
- Data (and instructions)
- In the real machine capabilities are tagged

\[
\text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z}
\]

**Register file**

- Assume finite set of registers RegisterName \(\ni\) pc

\[
\text{Reg} \overset{\text{def}}{=} \text{RegisterName} \rightarrow \text{Word}
\]
Formalisation - Memory and configurations

Memory

\[ \text{Mem} \stackrel{\text{def}}{=} \]
Formalisation - Memory and configurations

Memory

- Map from Addr to Word

\[ \text{Mem} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word} \]
Formalisation - Memory and configurations

Memory

- Map from Addr to Word

\[
\text{Mem} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]

Configuration

\[
\text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Mem} + \{\text{failed}\} + \{\text{halted}\} \times \text{Mem}
\]
Formalisation - Memory and configurations

Memory
- Map from Addr to Word

\[
\text{Mem} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]

Configuration
- Executable configuration

\[
\text{Conf} \overset{\text{def}}{=} 
\]
Formalisation - Memory and configurations

Memory

- Map from Addr to Word

\[
\text{Mem} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]

Configuration

- Executable configuration

\[
\text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Mem}
\]
Formalisation - Memory and configurations

Memory

- Map from Addr to Word

\[
\text{Mem} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]

Configuration

- Executable configuration
- Successfully halted configuration

\[
\text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Mem}
\]
Formalisation - Memory and configurations

Memory
- Map from Addr to Word
  \[ \text{Mem} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word} \]

Configuration
- Executable configuration
- Successfully halted configuration

\[ \text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Mem} + \{ \text{halted} \} \times \text{Mem} \]
Formalisation - Memory and configurations

Memory

- Map from Addr to Word
  \[ \text{Mem} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word} \]

Configuration

- Executable configuration
- Successfully halted configuration
- Failed configuration
  \[ \text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Mem} + \{ \text{failed} \} + \{ \text{halted} \} \times \text{Mem} \]
Formalisation - Instructions

Syntax

Instructions ::=
Formalisation - Instructions

Syntax

\[ rv ::= n \mid r \]

Instructions ::=

\[ \text{jmp} \ r \mid \text{jnz} \ r \]
\[ \text{move} \ r \ rv \mid \text{load} \ r \ r \]
\[ \text{store} \ r \ r \mid \text{plus} \ r \ rv \ rv \]
\[ \text{lea} \ r \ rv \mid \text{restrict} \ r \ r \ rv \]
\[ \text{subseg} \ r \ rv \ rv \mid \text{getp} \ r \ r \]
\[ \text{getl} \ r \ r \mid \text{getb} \ r \ r \]
\[ \text{gete} \ r \ r \mid \text{geta} \ r \ r \]
\[ \text{fail} \mid \text{halt} \]
Formalisation - Instructions

Syntax

- The normal instructions

\[ rv ::= n | r \]

Instructions \[ ::= \]
Formalisation - Instructions

Syntax

▶ The normal instructions

\[
rv ::= n \mid r
\]

Instructions ::= jmp r | jnz r rv | move r rv |
load r r | store r r | plus r rv rv
Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions

\[ \text{rv} := n \mid r \]

Instructions := \( \text{jmp } r \mid \text{jnz } r \text{rv} \mid \text{move } r \text{rv} \mid \text{load } r \text{r} \mid \text{store } r \text{r} \mid \text{plus } r \text{rv} \text{rv} \)
Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions

\[
rv ::= n \mid r
\]

\[
\text{Instructions} ::= \text{jmp } r \mid \text{jnz } r \ rv \mid \text{move } r \ rv \mid \\
\text{load } r \ r \mid \text{store } r \ r \mid \text{plus } r \ rv \ rv \mid \\
\text{lea } r \ rv \mid \text{restrict } r \ r \ rv \mid \\
\text{subseg } r \ rv \ rv \mid \\
\text{getp } r \ r \mid \text{getl } r \ r \mid \text{getb } r \ r \mid \\
\text{gete } r \ r \mid \text{geta } r \ r
\]
Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions
- Instructions to stop the machine

\[
rv ::= n \mid r
\]

Instructions ::= jmp r | jnz r rv | move r rv |
load r r | store r r | plus r rv rv |
lea r rv | restrict r r rv |
subseg r rv rv |
getp r r | getl r r | getb r r |
gete r r | geta r r
Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions
- Instructions to stop the machine

\[
rv ::= n \mid r
\]

Instructions ::= jmp r | jnz r rv | move r rv |
load r r | store r r | plus r rv rv |
lea r rv | restrict r r rv |
subseg r rv rv |
getp r r | getl r r | getb r r |
gete r r | geta r r | fail | halt
Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$
Formalisation - Operational Semantics (1)

**Execution relation**

\[
\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}
\]

\[
\text{executionAllowed}(\Phi)
\]
Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\Phi.\text{reg}(pc) = ((\text{perm}, g), \text{base}, \text{end}, a)$$

$$\text{executionAllowed}(\Phi)$$
Formalisation - Operational Semantics (1)

Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf} \]

\[ \Phi.\text{reg}(pc) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{base} \leq a \leq \text{end} \]

\[ \text{executionAllowed}(\Phi) \]
Formalisation - Operational Semantics (1)

Execution relation

→ ⊆ \((\text{Reg} \times \text{Mem}) \times \text{Conf}\)

\[
\Phi.\text{reg}(pc) = ((perm, g), base, end, a)\\
\text{base} \leq a \leq \text{end} \quad perm \in \{\text{rx}, \text{rwx}, \text{rwlx}\}\\
\text{executionAllowed}(\Phi)
\]
Formalisation - Operational Semantics (1)

Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf} \]

\[ \Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \]
\[ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwx}l\} \]

\[ \text{executionAllowed}(\Phi) \]

\[ \neg \text{executionAllowed}(\Phi) \]

\[ \Phi \rightarrow \]

\[ \Phi \rightarrow \]
Formalisation - Operational Semantics (1)

**Execution relation**

\[ \rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf} \]

\[ \Phi.\text{reg}(pc) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{ \text{rx}, \text{rwx}, \text{rwlx} \} \]

\[ \text{executionAllowed}(\Phi) \]

\[ \neg \text{executionAllowed}(\Phi) \]

\[ \Phi \rightarrow \text{failed} \]
Formalisation - Operational Semantics (1)

**Execution relation**

\[ \rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf} \]

\[ \Phi.\text{reg}(pc) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwlx}\} \]

\[ \text{executionAllowed}(\Phi) \]

\[ \neg \text{executionAllowed}(\Phi) \]

\[ \Phi \rightarrow \text{failed} \]

\[ \text{executionAllowed}(\Phi) \]

\[ \Phi \rightarrow \]
Formalisation - Operational Semantics (1)

Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf} \]

\[ \Phi.\text{reg}(pc) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

base \leq a \leq end \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwlx}\}

\]

\[ \text{executionAllowed}(\Phi) \]

\[ \neg \text{executionAllowed}(\Phi) \quad \Phi \rightarrow \text{failed} \]

\[ \text{executionAllowed}(\Phi) \quad i = \Phi.\text{mem}(a) \quad \Phi \rightarrow i \]
Formalisation - Operational Semantics (1)

**Execution relation**

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\Phi.\text{reg}(pc) = ((\text{perm}, g), \text{base}, \text{end}, a)$$

$$\text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx, rwx, rwx}\}$$

$$\text{executionAllowed}(\Phi)$$

$$\neg \text{executionAllowed}(\Phi)$$

$$\Phi \rightarrow \text{failed}$$

$$\text{executionAllowed}(\Phi) \quad i = \Phi.\text{mem}(a)$$

$$\Phi \rightarrow \llbracket i \rrbracket(\Phi)$$
\[\text{[store } r_1 \ r_2\text{]}(\Phi) = \]
$w = \Phi.\text{reg}(r_2)$

\[
\begin{array}{c}
\llbracket \text{store } r_1 \ r_2 \rrbracket (\Phi) = \\
\Phi[\text{mem}. r_1 \mapsto w]
\end{array}
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{[store } r_1 \ r_2\text{]}(\Phi) = \Phi[\text{mem}.a \mapsto w] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \]
\[ \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]
\[ \text{perm} \in \{\text{rw, rwl, rwx, rwlx}\} \]

\[
\begin{array}{l}
[\text{store } r_1 \ r_2](\Phi) = \\
\Phi[\text{mem}.a \mapsto w]
\end{array}
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \]

\[ [\text{store } r_1 \ r_2](\Phi) = \Phi[\text{mem}.a \mapsto w] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwxl}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((-, \text{local}), -, -, -) \]

\[ \text{J} \text{store } r_1 \ r_2 \text{K}(\Phi) = \Phi[\text{mem} . a \mapsto w] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((_, \text{local}), _, _, _) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\[
\begin{array}{l}
\text{[store } r_1 \ r_2\text{]}(\Phi) = \\
\Phi[\text{mem}.a \mapsto w]
\end{array}
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]
\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]
\[ \text{base} \leq a \leq \text{end} \quad w = ((\_ , \text{local}), \_ , \_ , \_ ) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\[ [\text{store } r_1 \ r_2] (\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w]) \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((\_ , \text{local}), \_ , \_ , \_ ) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\[ [\text{store } r_1 \ r_2](\Phi) = \text{updatePc}(\Phi[\text{mem} . a \mapsto w]) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg}.\text{pc} \mapsto \_] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rxw}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((-, \text{local}), -, -, -) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\[ [\text{store } r_1 \ r_2](\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w]) \]

\[ \Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg}.\text{pc} \mapsto ] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((-, \text{local}), - , - , -) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\( [\text{store } r_1 \ r_2] (\Phi) = \text{updatePc}(\Phi[\text{mem.a} \mapsto w]) \)

\[ \Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg.pc} \mapsto \quad ] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((\_ , \text{local}), \_ , \_ , \_ ) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\[ \text{[store } r_1 \ r_2 \text{]}(\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w]) \]

\[ \Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg.p\text{c}} \mapsto \text{newPc}] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((-, \text{local}), -, -, -) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\[ [\text{store } r_1 r_2](\Phi) = \text{updatePc}(\Phi[\text{mem.a} \mapsto w]) \]

\[ [\text{restrict } r_1 r_2 r_3] = \Phi[\text{reg.r}_1 \mapsto c] \]

\[ \Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg.pc} \mapsto \text{newPc}] \]
Formalisation - Operational Semantics (2)

\[
\begin{align*}
w &= \Phi.\text{reg}(r_2) \\
\Phi.\text{reg}(r_1) &= ((\text{perm}, g), \text{base}, \text{end}, a) \\
\text{perm} &\in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \\
\text{base} &\leq a \leq \text{end} \\
w &= ((\_, \text{local}), \_, \_, \_) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \\
\end{align*}
\]

\[
\llbracket \text{store } r_1 \ r_2 \rrbracket (\Phi) = \text{updatePc}(\Phi[\text{mem} \cdot a \mapsto w])
\]

\[
\Phi.\text{reg}(r_2) = (\text{permPair}, \text{base}, \text{end}, a)
\]

\[
\llbracket \text{restrict } r_1 \ r_2 \ r_3 \rrbracket = \Phi[\text{reg} \cdot r_1 \mapsto c]
\]

\[
\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\
\text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \\
\text{updatePc}(\Phi) = \Phi[\text{reg} \cdot \text{pc} \mapsto \text{newPc}]
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw, rwl, rwx, rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((-, \text{local}), -, -, -) \Rightarrow \text{perm} \in \{\text{rwl, rwlx}\} \]

\[ \left[\text{store } r_1 \ r_2\right](\Phi) = \text{updatePc}(\Phi[\text{mem.}a \mapsto w]) \]

\[ \Phi.\text{reg}(r_2) = (\text{permPair}, \text{base}, \text{end}, a) \]

\[ \text{newPermPair} = \text{decodePermPair}(\Phi, r_3) \]

\[ \left[\text{restrict } r_1 \ r_2 \ r_3\right] = \Phi[\text{reg.}r_1 \mapsto c] \]

\[ \Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg.}pc \mapsto \text{newPc}] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((\text{local}, \text{local}), \text{-}, \text{-}, \text{-}) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\[ \text{store } r_1 r_2](\Phi) = \text{updatePc}(\Phi[\text{mem.}a \mapsto w]) \]

\[ \Phi.\text{reg}(r_2) = (\text{permPair}, \text{base}, \text{end}, a) \]

\[ \text{newPermPair} = \text{decodePermPair}(\Phi, r_3) \]

\[ \text{newPermPair} \sqsubseteq \text{permPair} \]

\[ \text{restrict } r_1 r_2 r_3] = \Phi[\text{reg.}r_1 \mapsto c] \]

\[ \Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg.}\text{pc} \mapsto \text{newPc}] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((\_ , \text{local}), \_ , \_ , \_ ) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\} \]

\[ \text{store } r_1 \, r_2 \] (\( \Phi \)) = \text{updatePc}(\Phi[\text{mem}\.a \mapsto \! w])

\[ \Phi.\text{reg}(r_2) = (\text{permPair}, \text{base}, \text{end}, a) \]

\[ \text{newPermPair} = \text{decodePermPair}(\Phi, r_3) \]

\[ \text{newPermPair} \sqsubseteq \text{permPair} \quad c = (\text{newPermPair}, \text{base}, \text{end}, a) \]

\[ \text{restrict } r_1 \, r_2 \, r_3 \] = \( \Phi[\text{reg}.r_1 \mapsto c] \)

\[ \Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg}.\text{pc} \mapsto \text{newPc}] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = (( perm, g), base, end, a) \]

\[ \text{perm} \in \{ \text{rw}, \text{rwl}, \text{rwx}, \text{rwlx} \} \]

\[ \text{base} \leq a \leq \text{end} \quad w = ((-, \text{local}), -, -, -) \Rightarrow \text{perm} \in \{ \text{rwl}, \text{rwlx} \} \]

\[ \text{\{store \ } r_1 \ r_2 \text{\}}(\Phi) = \text{updatePc}(\Phi[\text{mem.a} \mapsto w]) \]

\[ \Phi.\text{reg}(r_2) = ( \text{permPair}, base, end, a) \]

\[ \text{newPermPair} = \text{decodePermPair}(\Phi, r_3) \]

\[ \text{newPermPair} \sqsubseteq \text{permPair} \quad c = (\text{newPermPair}, base, end, a) \]

\[ \text{\{restrict \ } r_1 \ r_2 \ r_3 \text{\}} = \text{updatePc}(\Phi[\text{reg.r_1} \mapsto c]) \]

\[ \Phi.\text{reg}(\text{pc}) = (( perm, g), base, end, a) \]

\[ \text{newPc} = ( perf, base, end, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg.pc} \mapsto \text{newPc}] \]
Need a *failed* case for each of the rules
Formalisation - Operational Semantics (3)

- Need a *failed* case for each of the rules
- The operational semantics of the remaining instructions is defined in a similar fashion
Antwerp
Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion
The “awkward” example

\[
g = \text{fun } _ = \Rightarrow \\
\quad \text{let } x = 0 \text{ in} \\
\quad \text{fun } \text{adv } \Rightarrow \\
\quad \quad x := 0; \\
\quad \quad \text{adv}(); \\
\quad \quad x := 1; \\
\quad \quad \text{adv}(); \\
\quad \quad \text{assert}(x == 1)
\]
The “awkward” example

```ocaml
let x = 0 in
fun adv =>
x := 0;
adv();
x := 1;
adv();
assert(x == 1)
```

Show for any reasonable `adv` that the assertion never fails for `adv(g)`. 
The “awkward” example

```haskell
let x = 0 in
fun adv =>
  x := 0;
  adv();
  x := 1;
  adv();
  assert(x == 1)
```

- Show for any reasonable `adv` that the assertion never fails for `adv(g)`.
- Need to define some macros to make a readable translation.
Macros

- `crtcls [(x_1, r_1), ..., (x_n, r_n)] r`
Macros

- `crtcls [(x_1, r_1), \ldots, (x_n, r_n)] r`
  - creates closure
Macros

- `crtcls [(x_1, r_1), \ldots, (x_n, r_n)] r`
  - creates closure
  - $x_1, \ldots, x_n$ available in program
Macros

- `crtcls [(x₁, r₁), ..., (xₙ, rₙ)] r`
  - creates closure
  - `x₁, ..., xₙ` available in program
  - `r` capability for program
Macros

- `crtcls [(x_1, r_1), \ldots, (x_n, r_n)] r`
  - creates closure
  - $x_1, \ldots, x_n$ available in program
  - $r$ capability for program

- `assert r_1 r_2`
  - check whether $r_1$ and $r_2$ contain the same value
  - if so: continue execution
  - if not: set assertion flag and halt
Macros

- \texttt{crtcls} \([ (x_1, r_1), \ldots, (x_n, r_n) ] \ r \)
  - creates closure
  - \(x_1, \ldots, x_n\) available in program
  - \(r\) capability for program

- \texttt{assert} \(rv_1 \ rv_2\)
  - check whether \(rv_1\) and \(rv_2\) contains the same value
Macros

- `crtcls [(x_1, r_1), \ldots, (x_n, r_n)] r`
  - creates closure
  - $x_1, \ldots, x_n$ available in program
  - $r$ capability for program

- `assert rv_1 rv_2`
  - check whether $rv_1$ and $rv_2$ contain the same value
    - if so: continue execution
Macros

- `crtcls [(x_1, r_1), \ldots, (x_n, r_n)] r`
  - creates closure
  - `x_1, \ldots, x_n` available in program
  - `r` capability for program

- `assert r_{v1} r_{v2}`
  - check whether `r_{v1}` and `r_{v2}` contains the same value
    - if so: continue execution
    - if not: set assertion flag and halt
Macros

- **crtcls** \([ (x_1, r_1), \ldots, (x_n, r_n) ] r\)
  - creates closure
  - \(x_1, \ldots, x_n\) available in program
  - \(r\) capability for program

- **assert** \(r v_1\) \(r v_2\)
  - check whether \(r v_1\) and \(r v_2\) contains the same value
    - if so: continue execution
    - if not: set assertion flag and halt

- **malloc** \(r\) \(n\)
Macros

- **crtcls** \[ (x_1, r_1), \ldots, (x_n, r_n) \] \( r \)
  - creates closure
  - \( x_1, \ldots, x_n \) available in program
  - \( r \) capability for program

- **assert** \( rv_1 \) \( rv_2 \)
  - check whether \( rv_1 \) and \( rv_2 \) contains the same value
    - if so: continue execution
    - if not: set assertion flag and halt

- **malloc** \( r \) \( n \)
  - allocates a *fresh* piece of memory of size \( n \)
Macros

- **crtcls** \([(x_1, r_1), \ldots, (x_n, r_n)] r\)
  - creates closure
  - \(x_1, \ldots, x_n\) available in program
  - \(r\) capability for program

- **assert** \(rv_1\) \(rv_2\)
  - check whether \(rv_1\) and \(rv_2\) contains the same value
    - if so: continue execution
    - if not: set assertion flag and halt

- **malloc** \(r\) \(n\)
  - allocates a *fresh* piece of memory of size \(n\)
  - leaves a global capability with rwx permissions in register \(r\)
The “awkward” example (naive translation)

\[
g = \text{fun } _ \Rightarrow \\
\text{let } x = 0 \text{ in} \\
\text{fun } \text{adv } \Rightarrow \\
x := 0; \\
\text{adv}(); \\
x := 1; \\
\text{adv}(); \\
\text{assert}(x == 1)
\]
The “awkward” example (naive translation)

\[
g = \text{fun } _\_ \Rightarrow \\
    \text{let } x = 0 \text{ in} \\
    \text{fun } \text{adv} \Rightarrow \\
    \quad x := 0; \\
    \quad \text{adv}(); \\
    \quad x := 1; \\
    \quad \text{adv}(); \\
    \quad \text{assert}(x == 1)
\]

\[
g: \quad \text{malloc } r_2 1 \\
    \text{store } r_2 0 \\
    \text{move } \text{pc } r_3 \\
    \text{lea } r_3 \ldots \\
    \text{crtcls } [(x, r_2)] r_3 \\
    \text{jmp } r_0
\]
The “awkward” example (naive translation)

```plaintext
\[
g = \text{fun } _ \Rightarrow \text{let } x = 0 \text{ in } \text{fun } \text{adv } \Rightarrow \text{let } x = 0; \text{adv}(); \text{adv>(); x} = 1; \text{adv}(); \text{assert}(x == 1)
\]
```

```plaintext
\[
g: \text{malloc } r_2 1 \\
\text{store } r_2 0 \\
\text{move pc } r_3 \\
\text{lea } r_3 ... \\
\text{crtcls } [(x, r_2)] r_3 \\
\text{jmp } r_0
\]
```

```plaintext
\[
f: \text{store } x 0 \\
\text{jmp } r_1 \\
\text{store } x 1 \\
\text{jmp } r_1 \\
\text{load } r_1 x \\
\text{assert } r_1 1 \\
\text{jmp } r_0
\]
```
The “awkward” example (naive translation)

```
| g = fun _ =>
| let x = 0 in
| fun adv =>
|   x := 0;
|   adv();
|   x := 1;
|   adv();
|   assert (x == 1)

| f:          store x 0
|            jmp r_1 !
|            store x 1
|            jmp r_1 !
|            load r_1 x
|            assert r_1 1
|            jmp r_0
```

```
g:   malloc r_2 1
store r_2 0
move pc r_3
lea r_3 ...
.crtcls [(x, r_2)] r_3
jmp r_0
```
Stack and stack capability

- local rwlx-capability
Stack and stack capability

- local rwx-capability
- Stack capability always points to the top element of the stack
Stack and stack capability

- local rwlx-capability
- Stack capability always points to the top element of the stack
- Only place one can store local capabilities
Stack and stack capability

- local rwlx-capability
- Stack capability always points to the top element of the stack
- Only place one can store local capabilities
- When a stack is available, we assume it is in register $r_{stk}$
Macros (1)

\texttt{scall \ r(\bar{r}_{\text{args}}, \bar{r}_{\text{priv}})}

- $\bar{r}_{\text{args}}$ list of argument registers
- $\bar{r}_{\text{priv}}$ list of “private” registers
- $\ r$ register to jump to

\texttt{scall} does the following:

- Push the restore code to the stack.
- "private" registers to the stack.
- return address capability
- stack capability
- Create protected return pointer
- Restrict stack capability to unused part
- Clear the part of the stack we release control over
- Clear unused registers
- Jump to $r$

Upon return: Run the on stack restoration code

Return address in caller-code: Restore "private" state
Macros (1)

\[
\text{scall } r(\bar{r}_{\text{args}}, \bar{r}_{\text{priv}})
\]

- \(\bar{r}_{\text{args}}\) list of argument registers
- \(\bar{r}_{\text{priv}}\) list of “private” registers
- \(r\) register to jump to
- scall does the following:
Macros (1)

\texttt{scall \, r(\overline{r}_{\text{args}}, \overline{r}_{\text{priv}})}

\begin{itemize}
\item \overline{r}_{\text{args}} list of argument registers
\item \overline{r}_{\text{priv}} list of “private” registers
\item \texttt{r} register to jump to
\item \texttt{scall} does the following:
  \begin{itemize}
  \item Push
    \begin{itemize}
    \item the restore code to the stack.
    \item “private” registers to the stack.
    \item return address capability
    \item stack capability
    \end{itemize}
  \end{itemize}
\end{itemize}
Macros (1)

\texttt{scall} \( r(\bar{r}_{\text{args}}, \bar{r}_{\text{priv}}) \)

- \( \bar{r}_{\text{args}} \) list of argument registers
- \( \bar{r}_{\text{priv}} \) list of “private” registers
- \( r \) register to jump to
- \texttt{scall} does the following:
  - Push
    - the restore code to the stack.
    - “private” registers to the stack.
    - return address capability
    - stack capability
  - Create protected return pointer
Macros (1)

\[ \text{scall } r(\bar{r}_{\text{args}}, \bar{r}_{\text{priv}}) \]

- \( \bar{r}_{\text{args}} \) list of argument registers
- \( \bar{r}_{\text{priv}} \) list of “private” registers
- \( r \) register to jump to
- \text{scall} does the following:
  - Push
    - the restore code to the stack.
    - “private” registers to the stack.
    - return address capability
    - stack capability
  - Create protected return pointer
  - Restrict stack capability to unused part
scall \( r(\bar{r}_{\text{args}}, \bar{r}_{\text{priv}}) \)

- \( \bar{r}_{\text{args}} \) list of argument registers
- \( \bar{r}_{\text{priv}} \) list of “private” registers
- \( r \) register to jump to
- scall does the following:
  - Push
    - the restore code to the stack.
    - “private” registers to the stack.
    - return address capability
    - stack capability
  - Create protected return pointer
  - Restrict stack capability to unused part
  - Clear the part of the stack we release control over
  - Clear unused registers
Macros (1)

```latex
\text{scall } r(\bar{r}_{\text{args}}, \bar{r}_{\text{priv}})
```

- $\bar{r}_{\text{args}}$ list of argument registers
- $\bar{r}_{\text{priv}}$ list of “private” registers
- $r$ register to jump to
- scall does the following:
  - Push
    - the restore code to the stack.
    - “private” registers to the stack.
    - return address capability
    - stack capability
  - Create protected return pointer
  - Restrict stack capability to unused part
  - Clear the part of the stack we release control over
  - Clear unused registers
  - Jump to $r$
```
scall \( r(\bar{r}_{\text{args}}, \bar{r}_{\text{priv}}) \)

- \( \bar{r}_{\text{args}} \) list of argument registers
- \( \bar{r}_{\text{priv}} \) list of “private” registers
- \( r \) register to jump to
- scall does the following:
  - Push
    - the restore code to the stack.
    - “private” registers to the stack.
    - return address capability
    - stack capability
  - Create protected return pointer
  - Restrict stack capability to unused part
  - Clear the part of the stack we release control over
  - Clear unused registers
  - Jump to \( r \)
  - Upon return: Run the on stack restoration code
Macros (1)

\texttt{scall} \, r(\bar{r}_{args}, \bar{r}_{priv})

- $\bar{r}_{args}$ list of argument registers
- $\bar{r}_{priv}$ list of “private” registers
- $r$ register to jump to
- \texttt{scall} does the following:
  - Push
    - the restore code to the stack.
    - “private” registers to the stack.
    - return address capability
    - stack capability
  - Create protected return pointer
  - Restrict stack capability to unused part
  - Clear the part of the stack we release control over
  - Clear unused registers
  - Jump to $r$
  - Upon return: Run the on stack restoration code
  - Return address in caller-code: Restore “private” state
The “awkward” example (naive translation)

```plaintext
The “awkward” example (naive translation)

\[
g = \text{fun } _ \Rightarrow \\
  \text{let } x = 0 \text{ in } \\
  \text{fun } \text{adv } \Rightarrow \\
  x := 0; \\
  \text{adv}(); \\
  x := 1; \\
  \text{adv}(); \\
  \text{assert}(x == 1)
\]

f:  
  \text{store } x 0 \\
  \text{jmp } r_1 \\
  \text{store } x 1 \\
  \text{jmp } r_1 \\
  \text{load } r_1 x \\
  \text{assert } r_1 1 \\
  \text{jmp } r_0

\[
g:  \text{malloc } r_2 1 \\
  \text{store } r_2 0 \\
  \text{move } \text{pc } r_3 \\
  \text{lea } r_3 \ldots \\
  \text{crtcls } [(x, r_2)] r_3 \\
  \text{jmp } r_0
\]
```
The “awkward” example (naive translation)

\[
\begin{align*}
g &= \text{fun } \_ \Rightarrow \\
&\quad \text{let } x = 0 \text{ in} \\
&\quad \text{fun } \text{adv } \Rightarrow \\
&\quad \quad x := 0; \\
&\quad \quad \text{adv}(); \\
&\quad \quad x := 1; \\
&\quad \quad \text{adv}(); \\
&\quad \quad \text{assert}(x == 1)
\end{align*}
\]

\[
g: \quad \text{malloc } r_2 1 \\
\quad \text{store } r_2 0 \\
\quad \text{move pc } r_3 \\
\quad \text{lea } r_3 \ldots \\
\quad \text{crtcls } [(x, r_2)] r_3 \\
\quad \text{jmp } r_0
\]

\[
f: \quad \text{store } x 0 \\
\quad \text{SCALL } r_1([], [r_0, r_1]) \\
\quad \text{store } x 1 \\
\quad \text{SCALL } r_1([], [r_0]) \\
\quad \text{load } r_1 x \\
\quad \text{assert } r_1 1 \\
\quad \text{jmp } r_0
\]
The “awkward” example (naive translation)

\[
g = \text{fun } _ => \\
    \text{let } x = 0 \text{ in} \\
    \text{fun } \text{adv } => \\
    x := 0; \\
    \text{adv}(); \\
    x := 1; \\
    \text{adv}(); \\
    \text{assert}(x == 1)
\]

\[
f: \\
\begin{align*}
\text{store } x 0 \\
\text{SCALL } r_1([], [r_0, r_1]) \\
\text{store } x 1 \\
\text{SCALL } r_1([], [r_0]) \\
\text{LOAD } r_1 x \\
\text{assert } r_1 1 \\
\text{JMP } r_0 !
\end{align*}
\]

\[
g: \\
\begin{align*}
\text{MALLOC } r_2 1 \\
\text{STORE } r_2 0 \\
\text{MOVE } PC r_3 \\
\text{LEA } r_3 \ldots \\
\text{CRTCLS } [(x, r_2)] r_3 \\
\text{JMP } r_0 !
\end{align*}
\]
Macros (2)

- mclear $r$

- rclear $\overline{r}$

Clear all memory cells capability in register $r$ has authority over.

Clear all the registers in $\overline{r}$. 
Macros (2)

- \texttt{mclear }\texttt{r}
  - Clear all memory cells capability in register \texttt{r} has authority over.
Macros (2)

- **mc**lear $r$
  - Clear all memory cells capability in register $r$ has authority over.
- **r**clear $\bar{r}$
Macros (2)

- **mclear** $r$
  - Clear all memory cells capability in register $r$ has authority over.

- **rclear** $\bar{r}$
  - Clear all the registers in $\bar{r}$. 
The “awkward” example (naive translation)

\[ g = \text{fun } _ => \\
\quad \text{let } x = 0 \text{ in} \\
\quad \text{fun } \text{adv } => \\
\quad \quad x := 0; \\
\quad \quad \text{adv}(); \\
\quad \quad x := 1; \\
\quad \quad \text{adv}(); \\
\quad \quad \text{assert}(x == 1) \]

\[ f: \text{ store } x 0 \\
\quad \text{ scall } r_1([], [r_0, r_1]) \\
\quad \text{ store } x 1 \\
\quad \text{ scall } r_1([], [r_0]) \\
\quad \text{ load } r_1 x \\
\quad \text{ assert } r_1 1 \\
\quad \text{ jmp } r_0 \]

g: \text{ malloc } r_2 1 \\
\quad \text{ store } r_2 0 \\
\quad \text{ move } pc r_3 \\
\quad \text{ lea } r_3 ... \\
\quad \text{crtcls } [(x, r_2)] \ r_3 \\
\quad \text{ jmp } r_0 \\
\quad \text{ jmp } r_0 \]
The “awkward” example (naive translation)

```plaintext
\[
g = \text{fun } _\Rightarrow \\
    \quad \text{let } x = 0 \text{ in}
    \quad \text{fun } \text{adv } \Rightarrow \\
    \quad \quad x := 0;
    \quad \quad \text{adv}();
    \quad \quad x := 1;
    \quad \quad \text{adv}();
    \quad \quad \text{assert}(x == 1)
\]
```

```
g:         \text{malloc } r_2 \ 1
           \text{store } r_2 0
           \text{move } pc \ r_3
           \text{lea } r_3 \ldots
           \text{crtcls } [(x, r_2)] \ r_3
           \text{rclear } \text{RN} \setminus \{pc, r_0, r_1\}
           \text{jmp } r_0
```

```
f:         \text{store } x 0
           \text{scall } r_1([], [r_0, r_1])
           \text{store } x 1
           \text{scall } r_1([], [r_0])
           \text{load } r_1 x
           \text{assert } r_1 \ 1
           \text{rclear } \text{RN} \setminus \{r_0, pc\}
           \text{jmp } r_0
```
The "awkward" example (naive translation)

\[
g = \text{fun } \_ \Rightarrow \\
\quad \text{let } x = 0 \text{ in} \\
\quad \text{fun } \text{adv } \Rightarrow \\
\quad \quad x := 0; \\
\quad \quad \text{adv}(); \\
\quad \quad x := 1; \\
\quad \quad \text{adv}(); \\
\quad \quad \text{assert}(x == 1)
\]

\[
f : \quad \text{store } x 0 \\
\quad \text{scall } r_1([], [r_0, r_1]) \\
\quad \text{store } x 1 \\
\quad \text{scall } r_1([], [r_0]) \\
\quad \text{load } r_1 x \\
\quad \text{assert } r_1 1 \\
\quad \text{mclear } r_{stk} \\
\quad \text{rclear } RN \setminus \{r_0, pc\} \\
\quad \text{jmp } r_0
\]
The “awkward” example (naive translation)

```plaintext
def g(t) =>
    let x = 0 in
    fun adv =>
        x := 0;
        adv();
        x := 1;
        adv();
        assert(x == 1)
```

```
g:     malloc r2 1
         store r2 0
         move pc r3
         lea r3 ...
        crtcls[(x, r2)] r3
        rclear RN \ {pc, r0, r1}
        jmp r0
```

```
f:     store x 0 !
        scall r1([], [r0, r1])
        store x 1
        scall r1([], [r0])
        load r1 x
        assert r1 1
        mclear rstk
        rclear RN \ {r0, pc}
        jmp r0
```
Macros (3)

▶ reqglob \( r \)

▶ if the word in register \( r \) is not a global capability, then fail

▶ otherwise continue execution

▶ prepstack \( r \)

▶ if the word in register \( r \) is not an \( rwlx \)-capability, then fail

▶ otherwise set pointer of capability to point just below range of authority
Macros (3)

- reqglob \( r \)
  - if the word in register \( r \) is not a global capability, then fail
Macros (3)

- reqglob $r$
  - if the word in register $r$ is not a global capability, then fail
  - otherwise continue execution
Macros (3)

- reqglob \(r\)
  - if the word in register \(r\) is not a global capability, then fail
  - otherwise continue execution

- prepstack \(r\)
Macros (3)

- **reqglob** $r$
  - if the word in register $r$ is not a global capability, then fail
  - otherwise continue execution

- **prepstack** $r$
  - if the word in register $r$ is not an *rwlx*-capability, then fail
Macros (3)

- reqglob \( r \)
  - if the word in register \( r \) is not a global capability, then fail
  - otherwise continue execution

- prepstack \( r \)
  - if the word in register \( r \) is not an \( \text{rwx} \)-capability, then fail
  - otherwise set pointer of capability to point just below range of authority
The “awkward” example (final version)

\[
\begin{align*}
g & = \text{fun } _z \Rightarrow \\
& \quad \text{let } x = 0 \text{ in} \\
& \quad \text{fun } \text{adv } \Rightarrow \\
& \quad \quad x := 0; \\
& \quad \quad \text{adv}(); \\
& \quad \quad x := 1; \\
& \quad \quad \text{adv}(); \\
& \quad \quad \text{assert}(x == 1) \\
\end{align*}
\]

\[
\begin{align*}
g :& \quad \text{malloc } r_2 \ 1 \\
& \quad \text{store } r_2 \ 0 \\
& \quad \text{move } pc \ r_3 \\
& \quad \text{lea } r_3 \ldots \\
& \quad \text{crtcls }[(x, r_2)] \ r_3 \\
& \quad \text{rclear } \text{RN} \setminus \{pc, r_0, r_1\} \\
& \quad \text{jmp } r_0 \\
\end{align*}
\]

\[
\begin{align*}
f :& \quad \text{store } x \ 0 \\
& \quad \text{scall } r_1([], [r_0, r_1]) \\
& \quad \text{store } x \ 1 \\
& \quad \text{scall } r_1([], [r_0]) \\
& \quad \text{load } r_1 \ x \\
& \quad \text{assert } r_1 \ 1 \\
& \quad \text{mclear } r_{stk} \\
& \quad \text{rclear } \text{RN} \setminus \{r_0, pc\} \\
& \quad \text{jmp } r_0 \\
\end{align*}
\]
The “awkward” example (final version)

\[
g = \text{fun } \_ \Rightarrow \\
\quad \text{let } x = 0 \text{ in} \\
\quad \text{fun } \text{adv } \Rightarrow \\
\quad \quad x := 0; \\
\quad \quad \text{adv}(); \\
\quad \quad x := 1; \\
\quad \quad \text{adv}(); \\
\quad \quad \text{assert}(x == 1)
\]

\[
g: \quad \text{malloc } r_2 1 \\
\text{store } r_2 0 \\
\text{move } pc \ r_3 \\
\text{lea } r_3 \ldots \\
\text{crtcls} [(x, r_2)] \ r_3 \\
\text{rclear } \text{RN} \setminus \{pc, r_0, r_1\} \\
\text{jmp } r_0
\]

\[
f: \\
\text{prepstack } r_{stk} \\
\text{store } x 0 \\
\text{scall } r_1([], [r_0, r_1]) \\
\text{store } x 1 \\
\text{scall } r_1([], [r_0]) \\
\text{load } r_1 x \\
\text{assert } r_1 1 \\
\text{mclear } r_{stk} \\
\text{rclear } \text{RN} \setminus \{r_0, pc\} \\
\text{jmp } r_0
\]
The “awkward” example (final version)

```
g = fun _ =>
    let x = 0 in
    fun adv =>
        x := 0;
        adv();
        x := 1;
        adv();
        assert(x == 1)

f:
    reqglob r1
    prepstack r_{stk}
    store x 0
    scall r_1([], [r_0, r_1])
    store x 1
    scall r_1([], [r_0])
    load r_1 x
    assert r_1 1
    mclear r_{stk}
    rclear RN \ {r_0, pc}
    jmp r_0
```

```
Stack discipline

- *Always* clear the unused stack before transferring control to untrusted code.
Stack discipline

- *Always* clear the unused stack before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities on the stack.
Stack discipline

- *Always* clear the unused stack before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities on the stack.
  - Prevent adversary from “hiding” local capability on the stack.
Stack discipline

- *Always* clear the unused stack before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities on the stack.
  - Prevent adversary from “hiding” local capability on the stack
- If stack capability untrusted, then

- Only use it if it is *rwlx*.
- Make it point just below range of authority.
- If it looks like a stack, works like a stack, and quacks like a stack, then it is probably a stack.
- In the presence of an untrusted stack capability, only use global callbacks.
Stack discipline

- *Always* clear the unused stack before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities on the stack.
  - Prevent adversary from “hiding” local capability on the stack.
- If stack capability untrusted, then
  - only use it if it is `rwx`
Stack discipline

- Always clear the unused stack before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities on the stack.
  - Prevent adversary from “hiding” local capability on the stack.
- If stack capability untrusted, then
  - only use it if it is rw\l x
  - make it point just below range of authority.
Stack discipline

- *Always* clear the unused stack before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities on the stack.
  - Prevent adversary from “hiding” local capability on the stack.
- If stack capability untrusted, then
  - only use it if it is *rwx*
  - make it point just below range of authority.
  - If it looks like a stack, works like a stack, and quacks like a stack, then it is probably a stack.
Stack discipline

- *Always* clear the unused stack before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities on the stack.
  - Prevent adversary from “hiding” local capability on the stack.
- If stack capability untrusted, then
  - only use it if it is *rwx*
  - make it point just below range of authority.
  - If it looks like a stack, works like a stack, and quacks like a stack, then it is probably a stack.
- In the presence of an untrusted stack capability, only use global callbacks.
Register discipline

- *Always* clear non-argument registers before transferring control to untrusted code.
Register discipline

- *Always* clear non-argument registers before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities.
Register discipline

- *Always* clear non-argument registers before transferring control to untrusted code.
  - Prevent unintentionally leaking capabilities.
  - Prevent adversary from “hiding” local capability in a register.
Bruges
Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion
Worlds

World
Worlds

World

Memory
Worlds

World

Memory
Worlds

World

Memory
Worlds
Worlds

World

Memory
Worlds

- World : \( \mathbb{N} \xrightarrow{\text{fin}} \text{Region} \)
Worlds

- World : $\mathbb{N}^{\text{fin}} \rightarrow \text{Region}$
- Three kinds of regions:
Worlds

- World : $\mathbb{N}^{\text{fin}} \rightarrow \text{Region}$
- Three kinds of regions:
  - **permanent** Models parts of memory that global and local capabilities can govern.
  - temporary Models parts of memory that only local capabilities can govern.
  - revoked Masking of region.

Two future world relations:
- public Extensional (⊒ _pub_)
- private Extensional and temporary regions can be revoked! (⊒ _priv_).
Worlds

- World: \( \mathbb{N} \xrightarrow{\text{fin}} \text{Region} \)

- Three kinds of regions:
  - **permanent**: Models parts of memory that global and local capabilities can govern.
  - **temporary**: Models parts of memory that only local capabilities can govern.
Worlds

- World : $\mathbb{N} \xrightarrow{\text{fin}} \text{Region}$
- Three kinds of regions:
  - **permanent**: Models parts of memory that global and local capabilities can govern.
  - **temporary**: Models parts of memory that only local capabilities can govern.
  - **revoked**: Masking of region.
Worlds

- World : $\mathbb{N} \xrightarrow{\text{fin}} \text{Region}$
- Three kinds of regions:
  - permanent Models parts of memory that global and local capabilities can govern.
  - temporary Models parts of memory that only local capabilities can govern.
  - revoked Masking of region.
- Two future world relations
Worlds

- World : $\mathbb{N} \xrightarrow{\text{fin}} \text{Region}$

- Three kinds of regions:
  - permanent: Models parts of memory that global and local capabilities can govern.
  - temporary: Models parts of memory that only local capabilities can govern.
  - revoked: Masking of region.

- Two future world relations
  - public: Extensional ($\equiv^{\text{pub}}$)
Worlds

- World: \( \mathbb{N} \xrightarrow{\text{fin}} \text{Region} \)

- Three kinds of regions:
  - **permanent**: Models parts of memory that global and local capabilities can govern.
  - **temporary**: Models parts of memory that only local capabilities can govern.
  - **revoked**: Masking of region.

- Two future world relations
  - **public**: Extensional (\( \sqsubseteq^{\text{pub}} \))
  - **private**: Extensional and temporary regions can be revoked! (\( \sqsubseteq^{\text{priv}} \))
Regions

Region

- Diagram of a region inside a square
Regions

Region

Memory cell
Regions

Region

Memory cell
Regions

Region

Memory cell
Regions

Region

Memory cell
Regions

Region

Memory cell
Regions

Region

Memory cell
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
  - \(s\), current state

Future Worlds

- In \(\sqsubseteq_{pub}\), existing regions are allowed to make transitions in \(\phi_{pub}\)
- In \(\sqsubseteq_{priv}\), existing regions are allowed to make transitions in \(\phi\)
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
  - \(s\), current state
  - \(\phi_{pub} : \text{State}^2\), reflexive and transitive
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
  - \(s\), current state
  - \(\phi_{pub}: \text{State}^2\), reflexive and transitive
  - \(\phi \supseteq \phi_{pub}\), reflexive and transitive
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
  - \(s\), current state
  - \(\phi_{pub} : \text{State}^2\), reflexive and transitive
  - \(\phi \supseteq \phi_{pub}\), reflexive and transitive
  - \(H : \text{State} \rightarrow \text{World} \xrightarrow{\text{mon}} \text{Pred(MemSegment)}\), state interpretation function
  - \(\text{Future Worlds}
    \begin{align*}
    \text{In } \sqsubseteq \text{pub}, & \text{existing regions are allowed to make transitions in } \phi_{pub} \\
    \text{In } \sqsubseteq \text{priv}, & \text{existing regions are allowed to make transitions in } \phi
    \end{align*}
  \)
Regions

- Regions are transition systems \((v, s, \phi_{\text{pub}}, \phi, H)\)
  - \(v\), view
  - \(s\), current state
  - \(\phi_{\text{pub}} : \text{State}^2\), reflexive and transitive
  - \(\phi \supseteq \phi_{\text{pub}}\), reflexive and transitive
  - \(H : \text{State} \rightarrow \text{World} \xrightarrow{\text{mon}} \text{Pred(MemSegment)}, \text{state interpretation function}\)
    - if region permanent, then \(H\) monotone w.r.t. \(\sqsupseteq_{\text{priv}}\)
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
  - \(s\), current state
  - \(\phi_{pub} : \text{State}^2\), reflexive and transitive
  - \(\phi \supseteq \phi_{pub}\), reflexive and transitive
  - \(H : \text{State} \rightarrow \text{World} \xrightarrow{\text{mon}} \text{Pred(MemSegment)}\), state interpretation function
    - if region permanent, then \(H\) monotone w.r.t. \(\sqsupseteq^{priv}\)
    - if region temporary, then \(H\) monotone w.r.t. \(\sqsupseteq^{pub}\)
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
  - \(s\), current state
  - \(\phi_{pub}: \text{State}^2\), reflexive and transitive
  - \(\phi \supseteq \phi_{pub}\), reflexive and transitive
  - \(H: \text{State} \rightarrow \text{World} \xrightarrow{\text{mon}} \text{Pred(MemSegment)}\), state interpretation function
    - if region permanent, then \(H\) monotone w.r.t. \(\sqsupseteq_{\text{priv}}\)
    - if region temporary, then \(H\) monotone w.r.t. \(\sqsupseteq_{\text{pub}}\)

- Future Worlds
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
  - \(s\), current state
  - \(\phi_{pub}: \text{State}^2\), reflexive and transitive
  - \(\phi \supseteq \phi_{pub}\), reflexive and transitive
  - \(H: \text{State} \rightarrow \text{World} \xrightarrow{mon} \text{Pred(MemSegment)}, \text{state interpretation function}\)
    - if region permanent, then \(H\) monotone w.r.t. \(\sqsubseteq^{priv}\)
    - if region temporary, then \(H\) monotone w.r.t. \(\sqsubseteq^{pub}\)

- Future Worlds
  - In \(\sqsubseteq^{pub}\), existing regions are allowed to make transitions in \(\phi_{pub}\)
Regions

- Regions are transition systems \((v, s, \phi_{pub}, \phi, H)\)
  - \(v\), view
  - \(s\), current state
  - \(\phi_{pub} : \text{State}^2\), reflexive and transitive
  - \(\phi \supseteq \phi_{pub}\), reflexive and transitive
  - \(H : \text{State} \rightarrow \text{World} \xrightarrow{\text{mon}} \text{Pred(MemSegment)}\), state interpretation function
    - if region permanent, then \(H\) monotone w.r.t. \(\sqsupseteq_{\text{priv}}\)
    - if region temporary, then \(H\) monotone w.r.t. \(\sqsupseteq_{\text{pub}}\)

- Future Worlds
  - In \(\sqsupseteq_{\text{pub}}\), existing regions are allowed to make transitions in \(\phi_{pub}\)
  - In \(\sqsupseteq_{\text{priv}}\), existing regions are allowed to make transitions in \(\phi\)
Logical Relation

- Step indexed
Logical Relation

- Step indexed
  - but in the following, the steps are omitted.
Observation relation

- Executable configurations that produce desired results.

\[ O : \text{World} \rightarrow \text{Pred}(\text{Reg} \times \text{MemSegment}) \]

\[ O(W) \overset{\text{def}}{=} \{(\text{reg}, \text{ms}) \mid \text{\...} \]
Observation relation

- Executable configurations that produce desired results.

\[ \mathcal{O} : \text{World} \rightarrow \text{Pred}(\text{Reg} \times \text{MemSegment}) \]

\[ \mathcal{O}(\mathcal{W}) \overset{\text{def}}{=} \{ (\text{reg}, \text{ms}) \mid \forall \text{ms}_f, \text{mem}' \cdot \]

\[ (\text{reg}, \text{ms} \oplus \text{ms}_f) \rightarrow (\text{halted}, \text{mem}') \]
Observation relation

- Executable configurations that produce desired results.

\[ O : \text{World} \rightarrow \text{Pred}(\text{Reg} \times \text{MemSegment}) \]

\[ O(W) \overset{\text{def}}{=} \{ (\text{reg}, ms) \mid \forall ms_f, \text{mem}' . \]

\[ (\text{reg}, ms \oplus ms_f) \rightarrow (\text{halted}, \text{mem}') \]

\[ \Rightarrow \exists W' \supseteq^{\text{priv}} W . \exists ms_r, ms' . \]

\[ \text{mem}' = ms' \oplus ms_r \oplus ms_f \land \]

\[ ms' : W' \} \]
Register-file relation

- Register-files with “well-behaved” words.
- On jump, the contents of the register-file can be seen as the arguments.

\[ R : \text{World} \xrightarrow{\text{mon}} \text{Pred(Reg)} \]

\[ R(W) \overset{\text{def}}{=} \]
Register-file relation

- Register-files with “well-behaved” words.
- On jump, the contents of the register-file can be seen as the arguments.

\[
\mathcal{R} : \text{World} \xrightarrow{\text{mon}} \text{Pred(Reg)} \\
\mathcal{R}(W) \overset{\text{def}}{=} \{ \text{reg} | \forall r \in \text{RegisterName} \setminus \{ \text{pc} \}. \text{reg}(r) \in \mathcal{V}(W) \}\]
Expression relation

- Words that produce admissible results when "executed".

\[ \mathcal{E} : \text{World} \rightarrow \text{Pred(Word)} \]

\[ \mathcal{E}(\mathcal{W}) \overset{\text{def}}{=} \]

\[ \forall \text{reg} \in \mathcal{R}(\mathcal{W}), \forall \text{ms} : \text{W}, (\text{reg}[\text{pc} \mapsto \text{pc}], \text{ms}) \in \text{O}(\mathcal{W}) \]
Expression relation

- Words that produce admissible results when “executed”.

\[ E : \text{World} \rightarrow \text{Pred(Word)} \]

\[ E(W) \overset{\text{def}}{=} \{ c \mid \forall \text{reg} \in \mathcal{R}(W). \forall \text{ms} : W. \} \]
Expression relation

Words that produce admissible results when “executed”.

\[ \mathcal{E} : \text{World} \rightarrow \text{Pred(Word)} \]

\[ \mathcal{E}(W) \overset{\text{def}}{=} \{ c \mid \forall \text{reg} \in \mathcal{R}(W). \]

\[ \forall \text{ms : W}. \]

\[ (\text{reg}[pc \mapsto pc], \text{ms}) \in \mathcal{O}(W) \} \]
Value relation

\( \forall \)-relation

- All integers (data) are in the \( \forall \)-relation
Value relation

\(\n\)-relation

- All integers (data) are in the \(\n\)-relation
- For capabilities, define a condition for each kind of permission it grants
Value relation

\( \mathcal{V} \)-relation

- All integers (data) are in the \( \mathcal{V} \)-relation
- For capabilities, define a condition for each kind of permission it grants
- Global capabilities
Value relation

\( \nabla \)-relation

- All integers (data) are in the \( \nabla \)-relation
- For capabilities, define a condition for each kind of permission it grants
- Global capabilities
  - Must respect \( \sqsubset^\text{priv} \)

\( \sqsubseteq \)-relation

- Authority only over memory segments governed by permanent region.

\( \sqsubset \)-relation

- Authority over memory segments governed by either permanent or temporary regions.
Value relation

\( \forall \)-relation

- All integers (data) are in the \( \forall \)-relation
- For capabilities, define a condition for each kind of permission it grants
- Global capabilities
  - Must respect \( \sqsubseteq^{\text{priv}} \)
  - Authority only over memory segments governed by permanent region.
Value relation

\( \forall \)-relation

- All integers (data) are in the \( \forall \)-relation
- For capabilities, define a condition for each kind of permission it grants
- Global capabilities
  - Must respect \( \sqsubseteq^{\text{priv}} \)
  - Authority only over memory segments governed by permanent region.
- Local capabilities
Value relation

\( \forall \)-relation

- All integers (data) are in the \( \forall \)-relation
- For capabilities, define a condition for each kind of permission it grants
- Global capabilities
  - Must respect \( \sqsupseteq^{\text{priv}} \)
  - Authority only over memory segments governed by permanent region.
- Local capabilities
  - Must respect \( \sqsupseteq^{\text{pub}} \)
Value relation

\( \forall \)-relation

- All integers (data) are in the \( \forall \)-relation
- For capabilities, define a condition for each kind of permission it grants
- Global capabilities
  - Must respect \( \sqsupseteq^{\text{priv}} \)
  - Authority only over memory segments governed by permanent region.
- Local capabilities
  - Must respect \( \sqsupseteq^{\text{pub}} \)
  - Authority over memory segments governed by either permanent or temporary regions
Read condition

\[ readCondition(g, W, base, end) = \{(base, end) \mid \exists r \in localityReg(g, W). \exists [base', end'] \supseteq [base, end]. \exist (W(r) \subset _{base', end'}^{pwl}) \} \]
Read condition

\[
readCondition(g, W, base, end) = \\
\{ (base, end) | \exists r \in localityReg(g, W). \\
\quad \exists [base', end'] \supseteq [base, end]. \\
\quad W(r) \subset \iota_{base', end'}^{pwl} \}
\]

\[
\iota_{base, end}^{pwl} \overset{\text{def}}{=} (\text{temp}, 1, =, =, H_{base, end}^{pwl})
\]
Read condition

\[
\text{readCondition}(g, W, \text{base}, \text{end}) = \\
\{(\text{base}, \text{end}) \mid \exists r \in \text{localityReg}(g, W). \\
\exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]. \\
W(r) \subseteq \iota_{\text{base}', \text{end}'}^{\text{pw}}\}
\]

\[
\iota_{\text{base}, \text{end}}^{\text{pw}} \overset{\text{def}}{=} (\text{temp}, 1, =, =, H_{\text{base}, \text{end}}^{\text{pw}})
\]

\[
H_{\text{pw}}^{\text{pw}} : \text{Addr}^2 \to \text{State} \to (\text{Wor}_\equiv^{\text{pub}} \overset{\text{mon}, \text{ne}}{\longrightarrow} \text{Pred} (\text{MemSegment})))
\]

\[
H_{\text{base}, \text{end}}^{\text{pw}} \ s \ \hat{W} \overset{\text{def}}{=} \left\{ ms \mid \text{dom}(ms) = [\text{base}, \text{end}] \wedge \\
\forall a \in [\text{base}, \text{end}]. ms(a) \in \mathcal{V}(\hat{W}) \right\}
\]
Write condition

\[ \text{writeCondition}(\nu, g, W, \text{base}, \text{end}) = \]
\[ \{(\text{base}, \text{end}) \mid \exists r \in \text{localityReg}(g, W). \]
\[ \exists[\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]. \]
\[ W(r) \supseteq \nu_{\text{base}', \text{end}'} \} \]
Write condition

\[
\text{writeCondition}(\iota, g, W, base, end) = \\
\{(base, end) \mid \exists r \in \text{localityReg}(g, W). \\
\exists [base', end'] \supset [base, end]. \\
W(r) \supset \iota_{base', end'}\}
\]

\[
\iota^{\text{nwl}}_{\text{base,end}} \overset{\text{def}}{=} (\text{temp}, 1, =, =, H^{\text{nwl}}_{\text{start,end}})
\]
Write condition

\[
\text{writeCondition}(\nu, g, W, \text{base}, \text{end}) = \\
\{ (\text{base}, \text{end}) \mid \exists r \in \text{localityReg}(g, W). \\
\exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]. \\
W(r) \supseteq \nu_{\text{base}', \text{end}'} \}
\]

\[
\nu_{\text{base}, \text{end}} \overset{\text{def}}{=} (\text{temp}, 1, =, =, H_{\text{start}, \text{end}}^{\text{nwl}})
\]

\[
H_{\text{nwl}}^{\text{nwl}} : \text{Addr}^{2} \rightarrow \text{State} \rightarrow \text{Pred(\text{MemSegment})}
\]

\[
H_{\text{base}, \text{end}}^{\text{nwl}} s \overset{\text{def}}{=} \left\{ ms \mid \text{dom}(ms) = [\text{base}, \text{end}] \wedge \\
\forall a \in [\text{base}, \text{end}]. \\
ms(a) \in \mathcal{V}(\text{revokeTemp}(\hat{W})) \right\}
\]
Conditions on execution

\[
\text{executeCondition}(g, W, \text{perm}, \text{base}, \text{end}) = \\
\left\{ \left(\text{perm}, \text{base}, \text{end}\right) \mid \forall W' \supseteq W . \\
\quad \forall a \in [\text{base}, \text{end}] . \\
\quad \quad ((\text{perm}, g), \text{base}, \text{end}, a) \in E(W') \right\}
\]
Conditions on execution

\[
\text{executeCondition}(g, W, \text{perm}, \text{base}, \text{end}) = \\
\left\{ (\text{perm}, \text{base}, \text{end}) \mid \forall W' \sqsupseteq W. \\
\quad \forall a \in [\text{base}, \text{end}]. \\
\quad \quad ((\text{perm}, g), \text{base}, \text{end}, a) \in E(W') \right\}
\]

where \( g = \text{local} \Rightarrow \sqsupseteq = \sqsupseteq^{\text{pub}} \)

and \( g = \text{global} \Rightarrow \sqsupseteq = \sqsupseteq^{\text{priv}} \)
Conditions on execution

\[
\text{executeCondition}(g, W, \text{perm}, \text{base}, \text{end}) = \\
\{ (\text{perm}, \text{base}, \text{end}) \mid \forall W' \sqsubseteq W. \\
\quad \forall a \in [\text{base}, \text{end}]. \\
\quad \quad ((\text{perm}, g), \text{base}, \text{end}, a) \in \mathcal{E}(W') \}\n\]

\[
\text{enterCondition}(g, W, \text{base}, \text{end}, a) = \\
\{ (\text{base}, \text{end}, a) \mid \forall W' \sqsubseteq W. \\
\quad ((rx, g), \text{base}, \text{end}, a) \in \mathcal{E}(W') \}\n\]

where \( g = \text{local} \Rightarrow \sqsubseteq = \sqsubseteq^{\text{pub}} \)

and \( g = \text{global} \Rightarrow \sqsubseteq = \sqsubseteq^{\text{priv}} \)
Value relation

\[ \forall : \text{World} \xrightarrow{\text{mon}} \text{Pred(Word)} \]

\[ \forall(W) \overset{\text{def}}{=} \]

Value relation

\[ V : \text{World} \xrightarrow{\text{mon}} \text{Pred(Word)} \]

\[ V(\mathcal{W}) \overset{\text{def}}{=} \{ i \mid i \in \mathbb{Z} \} \cup \]

Value relation

\[ \mathcal{V} : \text{World} \xrightarrow{\text{mon}} \text{Pred(Word)} \]

\[ \mathcal{V}(\mathcal{W}) \overset{\text{def}}{=} \{ i \mid i \in \mathbb{Z} \} \cup \]

\[ \{(((o, g), \text{base}, \text{end}, a)) \} \cup \]
Value relation

\[ \forall : \text{World} \xrightarrow{\text{mon}} \text{Pred(Word)} \]

\[ \forall(W) \overset{\text{def}}{=} \{ i \mid i \in \mathbb{Z} \} \cup \]

\[ \{ ((o, g), \text{base}, \text{end}, a) \} \cup \]

\[ \{ ((e, g), \text{base}, \text{end}, a) \mid \text{enterCondition}(g, W, \text{base}, \text{end}, a) \} \cup \]
Value relation

\[ \forall : \text{World} \xrightarrow{\text{mon}} \text{Pred} (\text{Word}) \]

\[ \forall (W) \overset{\text{def}}{=} \{ i \mid i \in \mathbb{Z} \} \cup \]

\[ \{ ((o, g), \text{base}, \text{end}, a) \} \cup \]

\[ \{ ((e, g), \text{base}, \text{end}, a) \mid \text{enterCondition}(g, W, \text{base}, \text{end}, a) \} \cup \]

\[ \{ ((rwlx, g), \text{base}, \text{end}, a) \mid \text{readCondition}(g, W, \text{base}, \text{end}) \wedge \]

\[ \text{writeCondition}(\nu^{\text{pwl}}, g, W, \text{base}, \text{end}) \wedge \]

\[ \text{executeCondition}(g, W, rwlx, \text{base}, \text{end}) \} \cup \]
Value relation

\[
\begin{align*}
\forall : \text{World} & \xrightarrow{\text{mon}} \text{Pred(Word)} \\
\forall(W) & \overset{\text{def}}{=} \{ i \mid i \in \mathbb{Z} \} \cup \\
& \quad \{ ((o, g), \text{base}, \text{end}, a) \} \cup \\
& \quad \{ ((e, g), \text{base}, \text{end}, a) \mid \text{enterCondition}(g, W, \text{base}, \text{end}, a) \} \cup \\
& \quad \{ ((rwlx, g), \text{base}, \text{end}, a) \mid \text{readCondition}(g, W, \text{base}, \text{end}) \land \\
& \quad \quad \text{writeCondition}(i^{\text{pwl}}, g, W, \text{base}, \text{end}) \land \\
& \quad \quad \text{executeCondition}(g, W, rwlx, \text{base}, \text{end}) \} \cup \\
& \ldots
\end{align*}
\]
Lemma (FTLR)

For all $W \in \text{World}$ and $c \in \text{Caps}$,

$$c \in \mathcal{E}(W).$$
Lemma (FTLR)

For all \( W \in \text{World} \) and \( c \in \text{Caps} \),

\[ c \in \mathcal{E}(W). \]

- The \( \text{pc} \)-register can be accessed like any other register
Lemma (FTLR)

For all $W \in \text{World}$ and $c \in \text{Caps}$,

$$c \in \mathcal{E}(W).$$

- The $pc$-register can be accessed like any other register
- Capability must behave when used for read/write
Lemma (FTLR)

For all $W \in \text{World}$, $g \in \text{Global}$, $\text{perm} \in \text{Perm}$, and $\text{base}$, $\text{end}$, $a \in \text{Addr}$,

if

\[ \text{perm} = \text{rx} \text{ and readCondition}(g, W, \text{base}, \text{end}), \]

or

\[ \text{perm} = \text{rwx} \text{ and readCondition}(W, \text{base}, \text{end}) \]

and $\text{writeCondition}(\nu^{\text{nwl}}, g, W, \text{base}, \text{end})$

or

\[ \ldots \]

then

\[ (\text{perm}, \text{base}, \text{end}, a) \in \mathcal{E}(W). \]
The awkward example

- Using the logical relation, we can prove well-bracketedness for the awkward example.

```ocaml
g = fun _ =>
    let x = 0 in
    fun adv =>
        x := 0;
        adv();
        x := 1;
        adv();
        assert(x == 1)
```
The awkward example

- Using the logical relation, we can prove well-bracketedness for the awkward example.
- The proof will have to wait for another time.

```
g = fun _ =>
    let x = 0 in
    fun adv =>
      x := 0;
      adv();
      x := 1;
      adv();
      assert(x == 1)
```
Namur
Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion
Conclusion

- With a simple capability system and reasonable conventions, we can enforce well-bracketedness.
- Using known logical relation techniques, we can reason about programs for a simple capability machine.
Questions?
Questions?

- Chocolates in the kitchen.