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

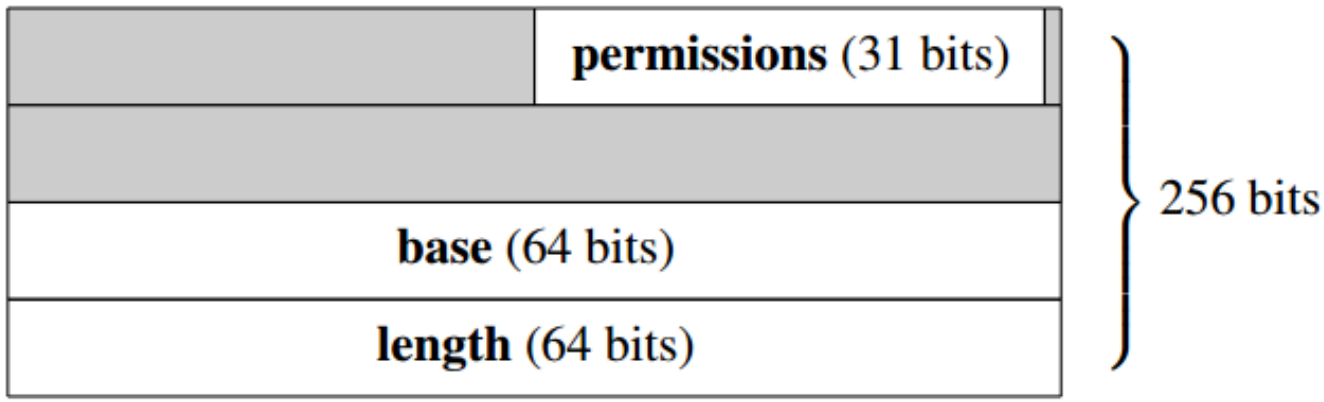


Figure: CHERI capability [1]

Capabilities

What is a capability?

- *Unforgeable* token of authority

What is a capability in a capability machine?

- Unforgeable pointer
- Range of memory

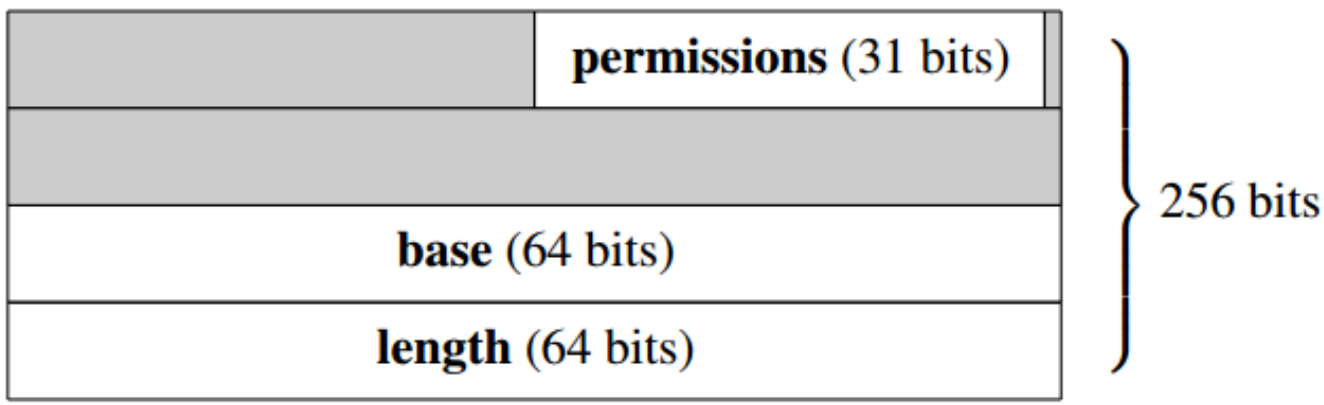


Figure: CHERI capability [1]

Capabilities

What is a capability?

- *Unforgeable* token of authority

What is a capability in a capability machine?

- Unforgeable pointer
- Range of memory
- Permission

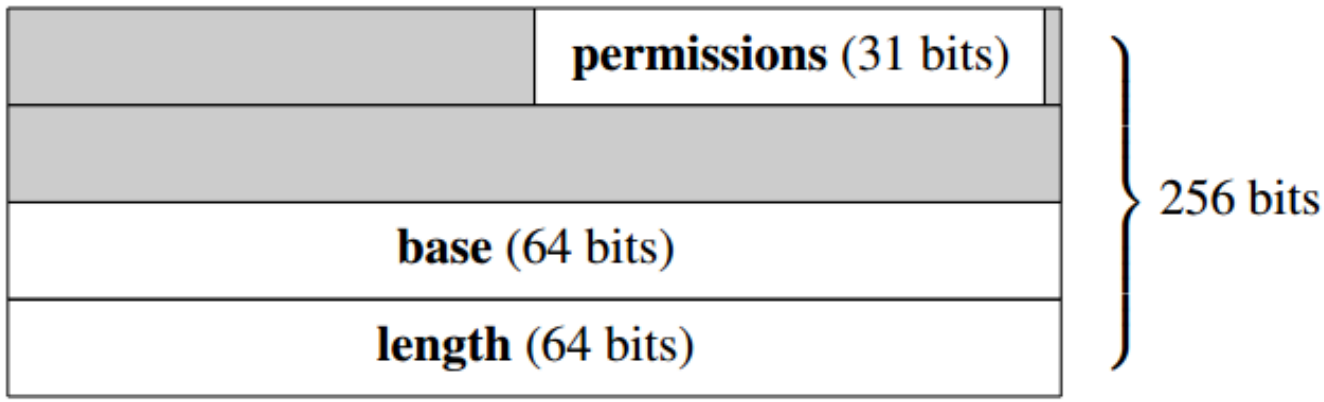


Figure: CHERI capability [1]

Capability permissions

- Read
- Write
- Execute

Capability permissions

- Read
- Write
- Execute
- Enter

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`

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:

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

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

$$\mathrm{Perm} \stackrel{def}{=} \{ \qquad \}$$

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \qquad\qquad\qquad\qquad\qquad\qquad \}$$

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \mathrm{ro}, \qquad\qquad\qquad\qquad\qquad \}$$

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \mathrm{ro}, \mathrm{rw}, \qquad\qquad\qquad\qquad \}$$

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \mathrm{ro}, \mathrm{rw}, \qquad \mathrm{rx}, \qquad\qquad \}$$

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \mathrm{ro}, \mathrm{rw}, \qquad \mathrm{rx}, \mathrm{e}, \qquad\qquad \}$$

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,

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \mathrm{ro}, \mathrm{rw}, \qquad \mathrm{rx}, \mathrm{e}, \mathrm{rwx}, \qquad \}$$

Formalisation - Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \mathrm{ro}, \mathrm{rw}, \mathrm{rwl}, \mathrm{rx}, \mathrm{e}, \mathrm{rwx}, \mathrm{rwlx}\}$$

Formalisation - Permissions

Permissions ordering

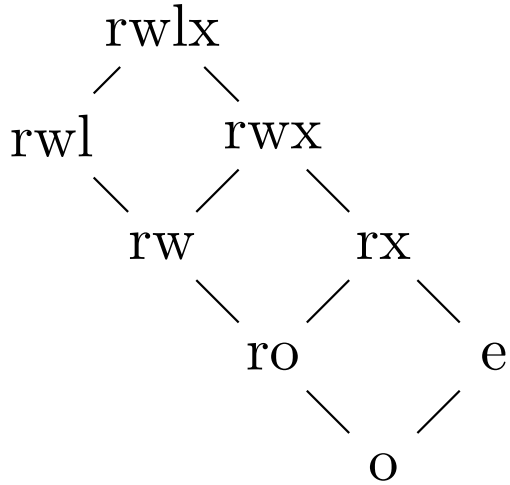


Figure: Permission hierarchy

Formalisation - Locality

Locality

$$\text{Global} ::= \{\text{global}, \text{local}\}$$

Locality ordering

$$\begin{array}{c} \text{global} \\ | \\ \text{local} \end{array}$$

Figure: Locality hierarchy

Formalisation - Capabilities

Capability

$$\mathrm{Cap} \stackrel{def}{=}$$

Formalisation - Capabilities

Capability

- Permission and locality

$$\mathrm{Cap} \stackrel{def}{=}$$

Formalisation - Capabilities

Capability

- Permission and locality

$$\mathrm{Cap} \stackrel{def}{=} (\mathrm{Perm} \times \mathrm{Global})$$

Formalisation - Capabilities

Capability

- Permission and locality
- Range of authority

$$\mathrm{Cap} \stackrel{def}{=} (\mathrm{Perm} \times \mathrm{Global})$$

Formalisation - Capabilities

Capability

- Permission and locality
- Range of authority

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} (\mathrm{Perm} \times \mathrm{Global})$$

Formalisation - Capabilities

Capability

- Permission and locality
- Range of authority

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} (\mathrm{Perm} \times \mathrm{Global}) \times \mathrm{Addr} \times \mathrm{Addr}$$

Formalisation - Capabilities

Capability

- Permission and locality
- Range of authority
- Pointer

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} (\mathrm{Perm} \times \mathrm{Global}) \times \mathrm{Addr} \times \mathrm{Addr}$$

Formalisation - Capabilities

Capability

- Permission and locality
- Range of authority
- Pointer

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} (\mathrm{Perm} \times \mathrm{Global}) \times \mathrm{Addr} \times \mathrm{Addr} \times \mathrm{Addr}$$

Formalisation - Capabilities

Capability

- Permission and locality
- Range of authority
- Pointer

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} (\mathrm{Perm} \times \mathrm{Global}) \times \mathrm{Addr} \times \mathrm{Addr} \times \mathrm{Addr}$$

Example: $((\mathrm{e}, \mathrm{local}), 30, 42, 30)$

Formalisation - Words and register file

Words

$$\mathrm{Word} \stackrel{def}{=}$$

Formalisation - Words and register file

Words

- Capability

$$\text{Word} \stackrel{def}{=}$$

Formalisation - Words and register file

Words

- Capability

$$\mathrm{Word} \stackrel{def}{=} \mathrm{Cap}$$

Formalisation - Words and register file

Words

- Capability
- Data (and instructions)

$$\mathrm{Word} \stackrel{def}{=} \mathrm{Cap}$$

Formalisation - Words and register file

Words

- Capability
- Data (and instructions)

$$\mathrm{Word} \stackrel{def}{=} \mathrm{Cap} + \mathbb{Z}$$

Formalisation - Words and register file

Words

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

$$\mathrm{Word} \stackrel{def}{=} \mathrm{Cap} + \mathbb{Z}$$

Formalisation - Words and register file

Words

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

$$\mathrm{Word} \stackrel{def}{=} \mathrm{Cap} + \mathbb{Z}$$

Register file

$$\mathrm{Reg} \stackrel{def}{=}$$

Formalisation - Words and register file

Words

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

$$\mathrm{Word} \stackrel{def}{=} \mathrm{Cap} + \mathbb{Z}$$

Register file

- Assume finite set of registers $\mathrm{RegisterName} \ni \mathrm{pc}$

$$\mathrm{Reg} \stackrel{def}{=}$$

Formalisation - Words and register file

Words

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

$$\mathrm{Word} \stackrel{def}{=} \mathrm{Cap} + \mathbb{Z}$$

Register file

- Assume finite set of registers $\mathrm{RegisterName} \ni \mathrm{pc}$

$$\mathrm{Reg} \stackrel{def}{=} \mathrm{RegisterName} \to \mathrm{Word}$$

Formalisation - Memory and configurations

Memory

$$\mathrm{Mem} \stackrel{def}{=}$$

Formalisation - Memory and configurations

Memory

- Map from Addr to Word

$$\mathrm{Mem} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Formalisation - Memory and configurations

Memory

- Map from Addr to Word

$$\mathrm{Mem} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

$$\mathrm{Conf} \stackrel{def}{=}$$

Formalisation - Memory and configurations

Memory

- Map from Addr to Word

$$\mathrm{Mem} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration

$$\mathrm{Conf} \stackrel{def}{=}$$

Formalisation - Memory and configurations

Memory

- Map from Addr to Word

$$\mathrm{Mem} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration

$$\mathrm{Conf} \stackrel{def}{=} \mathrm{Reg} \times \mathrm{Mem}$$

Formalisation - Memory and configurations

Memory

- Map from Addr to Word

$$\mathrm{Mem} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration
- Successfully halted configuration

$$\mathrm{Conf} \stackrel{def}{=} \mathrm{Reg} \times \mathrm{Mem}$$

Formalisation - Memory and configurations

Memory

- Map from Addr to Word

$$\mathrm{Mem} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration
- Successfully halted configuration

$$\mathrm{Conf} \stackrel{def}{=} \mathrm{Reg} \times \mathrm{Mem} \qquad + \{\mathit{halted}\} \times \mathrm{Mem}$$

Formalisation - Memory and configurations

Memory

- Map from Addr to Word

$$\mathrm{Mem} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration
- Successfully halted configuration
- Failed configuration

$$\mathrm{Conf} \stackrel{def}{=} \mathrm{Reg} \times \mathrm{Mem} + \{\mathit{failed}\} + \{\mathit{halted}\} \times \mathrm{Mem}$$

Formalisation - Instructions

Syntax

$$\text{Instructions} \quad ::=$$

Formalisation - Instructions

Syntax

$$
\begin{array}{rcl}
rv & ::= & n \mid r \\
\text{Instructions} & ::= &
\end{array}
$$

Formalisation - Instructions

Syntax

- The normal instructions

$$\begin{array}{rcl} rv & ::= & n \mid r \\ \text{Instructions} & ::= & \end{array}$$

Formalisation - Instructions

Syntax

- The normal instructions

$$
\begin{array}{rcl}
rv & ::= & n \mid r \\
\text{Instructions} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rv \mid \texttt{move}\ r\ rv \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rv\ rv
\end{array}
$$

Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions

$$
\begin{array}{rcl}
rv & ::= & n \mid r \\
\text{Instructions} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rv \mid \texttt{move}\ r\ rv \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rv\ rv
\end{array}
$$

Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions

$$
\begin{array}{rcl}
rv & ::= & n \mid r \\
\text{Instructions} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rv \mid \texttt{move}\ r\ rv \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rv\ rv \mid \\
 & & \texttt{lea}\ r\ rv \mid \texttt{restrict}\ r\ r\ rv \mid \\
 & & \texttt{subseg}\ r\ rv\ rv \mid \\
 & & \texttt{getp}\ r\ r \mid \texttt{getl}\ r\ r \mid \texttt{getb}\ r\ r \mid \\
 & & \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r
\end{array}
$$

Formalisation - Instructions

Syntax

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

$$
\begin{array}{rcl}
rv & ::= & n \mid r \\
\text{Instructions} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rv \mid \texttt{move}\ r\ rv \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rv\ rv \mid \\
 & & \texttt{lea}\ r\ rv \mid \texttt{restrict}\ r\ r\ rv \mid \\
 & & \texttt{subseg}\ r\ rv\ rv \mid \\
 & & \texttt{getp}\ r\ r \mid \texttt{getl}\ r\ r \mid \texttt{getb}\ r\ r \mid \\
 & & \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r
\end{array}
$$

Formalisation - Instructions

Syntax

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

$$
\begin{array}{rcl}
rv & ::= & n \mid r \\
\text{Instructions} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rv \mid \texttt{move}\ r\ rv \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rv\ rv \mid \\
 & & \texttt{lea}\ r\ rv \mid \texttt{restrict}\ r\ r\ rv \mid \\
 & & \texttt{subseg}\ r\ rv\ rv \mid \\
 & & \texttt{getp}\ r\ r \mid \texttt{getl}\ r\ r \mid \texttt{getb}\ r\ r \mid \\
 & & \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Formalisation - Operational Semantics (1)

Execution relation

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

Formalisation - Operational Semantics (1)

Execution relation

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

$$\frac{}{\mathit{executionAllowed}(\Phi)}$$

Formalisation - Operational Semantics (1)

Execution relation

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

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

$$\frac{}{\mathit{executionAllowed}(\Phi)}$$

Formalisation - Operational Semantics (1)

Execution relation

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

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end}}{\mathit{executionAllowed}(\Phi)}$$

Formalisation - Operational Semantics (1)

Execution relation

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

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}, \mathrm{rwlx}\}}{\mathit{executionAllowed}(\Phi)}$$

Formalisation - Operational Semantics (1)

Execution relation

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

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}, \mathrm{rwlx}\}}{\mathit{executionAllowed}(\Phi)}$$

$$\frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow}$$

Formalisation - Operational Semantics (1)

Execution relation

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

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}, \mathrm{rwlx}\}}{\mathit{executionAllowed}(\Phi)}$$

$$\frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow \mathit{failed}}$$

Formalisation - Operational Semantics (1)

Execution relation

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

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}, \mathrm{rwlx}\}}{\mathit{executionAllowed}(\Phi)}$$

$$\frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow \mathit{failed}}$$

$$\frac{\mathit{executionAllowed}(\Phi)}{\Phi \rightarrow}$$

Formalisation - Operational Semantics (1)

Execution relation

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

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}, \mathrm{rwlx}\}}{\mathit{executionAllowed}(\Phi)}$$

$$\frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow \mathit{failed}}$$

$$\frac{\mathit{executionAllowed}(\Phi) \qquad i = \Phi.\mathrm{mem}(a)}{\Phi \rightarrow \ i}$$

Formalisation - Operational Semantics (1)

Execution relation

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

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}, \mathrm{rwlx}\}}{\mathit{executionAllowed}(\Phi)}$$

$$\frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow \mathit{failed}}$$

$$\frac{\mathit{executionAllowed}(\Phi) \qquad i = \Phi.\mathrm{mem}(a)}{\Phi \rightarrow [\![i]\!](\Phi)}$$

Formalisation - Operational Semantics (2)

$$[\![\texttt{store}\ r_1\ r_2]\!](\Phi) =$$

Formalisation - Operational Semantics (2)

$$w = \Phi.\mathrm{reg}(r_2)$$

$$[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \qquad \Phi[\mathrm{mem}.r_1 \mapsto w]$$

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{reg}(r_2) \qquad \Phi.\mathrm{reg}(r_1) = (perm, base, end, a)}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \qquad \Phi[\mathrm{mem}.a \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \qquad \Phi[\mathrm{mem}.a \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \qquad \Phi[\mathrm{mem}.a \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \Phi[\mathrm{mem}.a \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow \mathit{perm} \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \qquad \Phi[\mathrm{mem}.a \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow \mathit{perm} \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \mathit{updatePc}(\Phi[\mathrm{mem}.a \mapsto w])}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow \mathit{perm} \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \mathit{updatePc}(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{}{\mathit{updatePc}(\Phi) = \Phi[\mathrm{reg.pc} \mapsto \qquad\qquad]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow \mathit{perm} \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \mathit{updatePc}(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a)}{\mathit{updatePc}(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto \qquad\qquad]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow \mathit{perm} \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \mathit{updatePc}(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ \mathit{newPc} = (\mathit{perm}, \mathit{base}, \mathit{end}, a + 1) \end{array}}{\mathit{updatePc}(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto \qquad\qquad]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow \mathit{perm} \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \mathit{updatePc}(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ \mathit{newPc} = (\mathit{perm}, \mathit{base}, \mathit{end}, a + 1) \end{array}}{\mathit{updatePc}(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto \mathit{newPc}]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow \mathit{perm} \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \mathit{updatePc}(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad\qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ \mathit{newPc} = (\mathit{perm}, \mathit{base}, \mathit{end}, a + 1) \end{array}}{\mathit{updatePc}(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto \mathit{newPc}]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (perm, base, end, a) \qquad perm \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ base \leq a \leq end \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow perm \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\Phi.\mathrm{reg}(r_2) = (permPair, base, end, a)$$

$$\frac{}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad\qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(\mathrm{pc}) = ((perm, g), base, end, a) \\ newPc = (perm, base, end, a + 1) \end{array}}{updatePc(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto newPc]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (perm, base, end, a) \qquad perm \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ base \leq a \leq end \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow perm \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(r_2) = (permPair, base, end, a) \\ newPermPair = decodePermPair(\Phi, r_3) \end{array}}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad\qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(\mathrm{pc}) = ((perm, g), base, end, a) \\ newPc = (perm, base, end, a + 1) \end{array}}{updatePc(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto newPc]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (perm, base, end, a) \qquad perm \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ base \leq a \leq end \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow perm \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(r_2) = (permPair, base, end, a) \\ newPermPair = decodePermPair(\Phi, r_3) \\ newPermPair \sqsubseteq permPair \end{array}}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(\mathrm{pc}) = ((perm, g), base, end, a) \\ newPc = (perm, base, end, a + 1) \end{array}}{updatePc(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto newPc]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (perm, base, end, a) \qquad perm \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ base \leq a \leq end \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow perm \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(r_2) = (permPair, base, end, a) \\ newPermPair = decodePermPair(\Phi, r_3) \\ newPermPair \sqsubseteq permPair \qquad c = (newPermPair, base, end, a) \end{array}}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(\mathrm{pc}) = ((perm, g), base, end, a) \\ newPc = (perm, base, end, a + 1) \end{array}}{updatePc(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto newPc]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\mathrm{reg}(r_2) \\ \Phi.\mathrm{reg}(r_1) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{rw}, \mathrm{rwl}, \mathrm{rwx}, \mathrm{rwlx}\} \\ \mathit{base} \leq a \leq \mathit{end} \qquad w = ((\_, \mathrm{local}), \_, \_, \_) \Rightarrow \mathit{perm} \in \{\mathrm{rwl}, \mathrm{rwlx}\} \end{array}}{[\![\texttt{store}\ r_1\ r_2]\!](\Phi) = \mathit{updatePc}(\Phi[\mathrm{mem}.a \mapsto w])}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(r_2) = (\mathit{permPair}, \mathit{base}, \mathit{end}, a) \\ \mathit{newPermPair} = \mathit{decodePermPair}(\Phi, r_3) \\ \mathit{newPermPair} \sqsubseteq \mathit{permPair} \qquad c = (\mathit{newPermPair}, \mathit{base}, \mathit{end}, a) \end{array}}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto c])}$$

$$\frac{\begin{array}{c} \Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ \mathit{newPc} = (\mathit{perm}, \mathit{base}, \mathit{end}, a + 1) \end{array}}{\mathit{updatePc}(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto \mathit{newPc}]}$$

Formalisation - Operational Semantics (3)

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

The “awkward” example

```
g = fun _ =>
      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

```
g = fun _ =>
      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` $[(x_1, r_1), \ldots, (x_n, r_n)]$ 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` $[(x_1, r_1), \ldots, (x_n, r_n)]$ r
  - creates closure
  - $x_1, \ldots, x_n$ 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` rv_1 rv_2

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

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

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

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

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 permission in register r

The “awkward” example (naive translation)

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

The "awkward" example (naive translation)

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

---

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

The "awkward" example (naive translation)

```
g = fun _ =>
      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
      jmp r0
```

```
f :   store x 0
      jmp r1
      store x 1
      jmp r1
      load r1 x
      assert r1 1
      jmp r0
```

The "awkward" example (naive translation)

```
g = fun _ =>
      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
      jmp r0
```

```
f :   store x 0
      jmp r1   !
      store x 1
      jmp r1   !
      load r1 x
      assert r1 1
      jmp r0
```

Stack and stack capability

- local rwlx-capability

Stack and stack capability

- local rwlx-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)

`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

Macros (1)

`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
- `scall` does the following:

Macros (1)

`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
- `scall` does the following:
  - Push
    - the restore code to the stack.
    - "private" registers to the stack.
    - return address capability
    - stack capability

Macros (1)

`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
- `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)

`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
- `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

Macros (1)

`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
- `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)

`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
- `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

Macros (1)

`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
- `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)

`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
- `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)

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

---

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

f : store x 0
jmp r_1
store x 1
jmp r_1
load r_1 x
assert r_1 1
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)
```

---

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

f : 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
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)
```

---

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 !

f : 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
jmp r_0 !

Macros (2)

- `mclear` *r*

Macros (2)

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

Macros (2)

- `mclear` r
  - Clear all memory cells capability in register r has authority over.
- `rclear` $\bar{r}$

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

---

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

f : 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

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)
```

---

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

f : `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

`rclear` $\mathrm{RN} \setminus \{r_0, \mathrm{pc}\}$
`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)
```

---

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

f : 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 $\mathrm{RN} \setminus \{r_0, \mathrm{pc}\}$
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)
```

---

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

f : `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` $\mathrm{RN} \setminus \{r_0, \mathrm{pc}\}$
`jmp` r_0

Macros (3)

- `reqglob` *r*

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 rwlx-capability, then fail
  - otherwise set pointer of capability to point just below range of authority

The "awkward" example (final version)

```
g = fun _ =>
      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 r_stk
rclear RN \ {r0, pc}
jmp r0
```

The "awkward" example (final version)

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

---

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

f :
$\texttt{prepstack}\ r_{stk}$
$\texttt{store}\ x\ 0$
$\texttt{scall}\ r_1([], [r_0, r_1])$
$\texttt{store}\ x\ 1$
$\texttt{scall}\ r_1([], [r_0])$
$\texttt{load}\ r_1\ x$
$\texttt{assert}\ r_1\ 1$
$\texttt{mclear}\ r_{stk}$
$\texttt{rclear}\ \mathrm{RN} \setminus \{r_0, \mathrm{pc}\}$
$\texttt{jmp}\ r_0$

The "awkward" example (final version)

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

---

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

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` $\mathrm{RN} \setminus \{r_0, \mathrm{pc}\}$
`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

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

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.

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.

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.

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

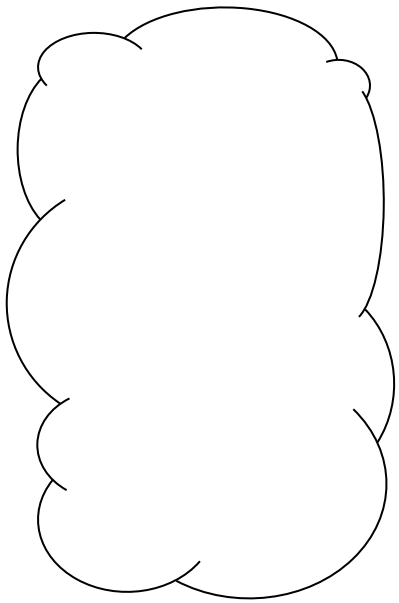
Example, Macros, and Stack Discipline

Logical Relation

Conclusion

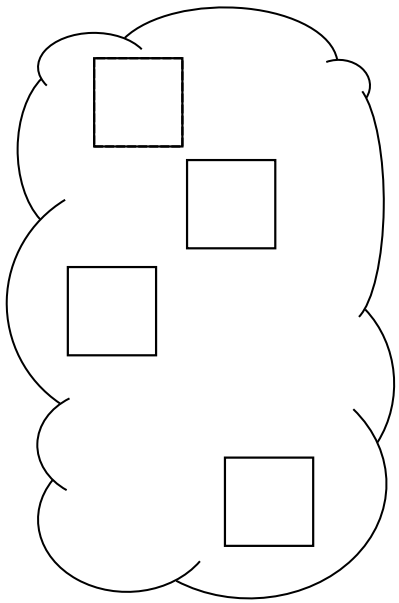
Worlds

World



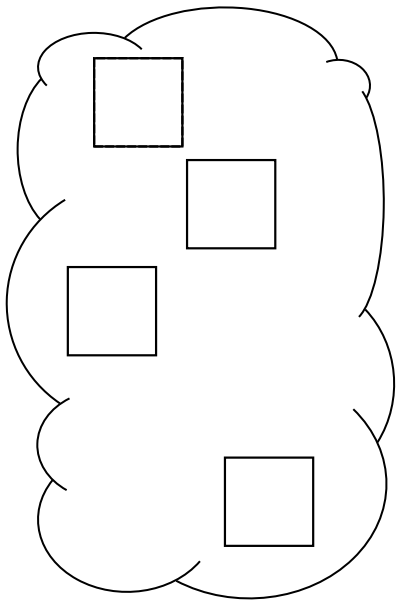
Worlds

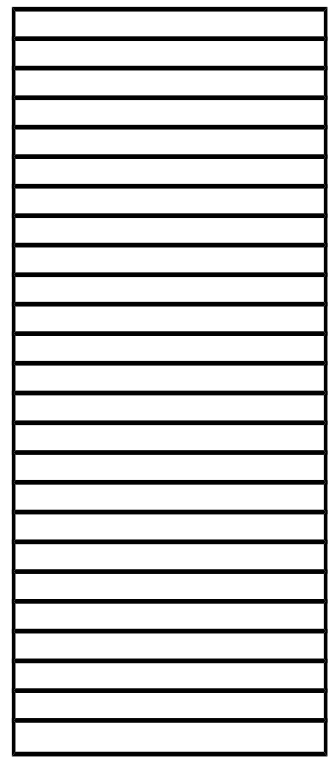
World



Worlds

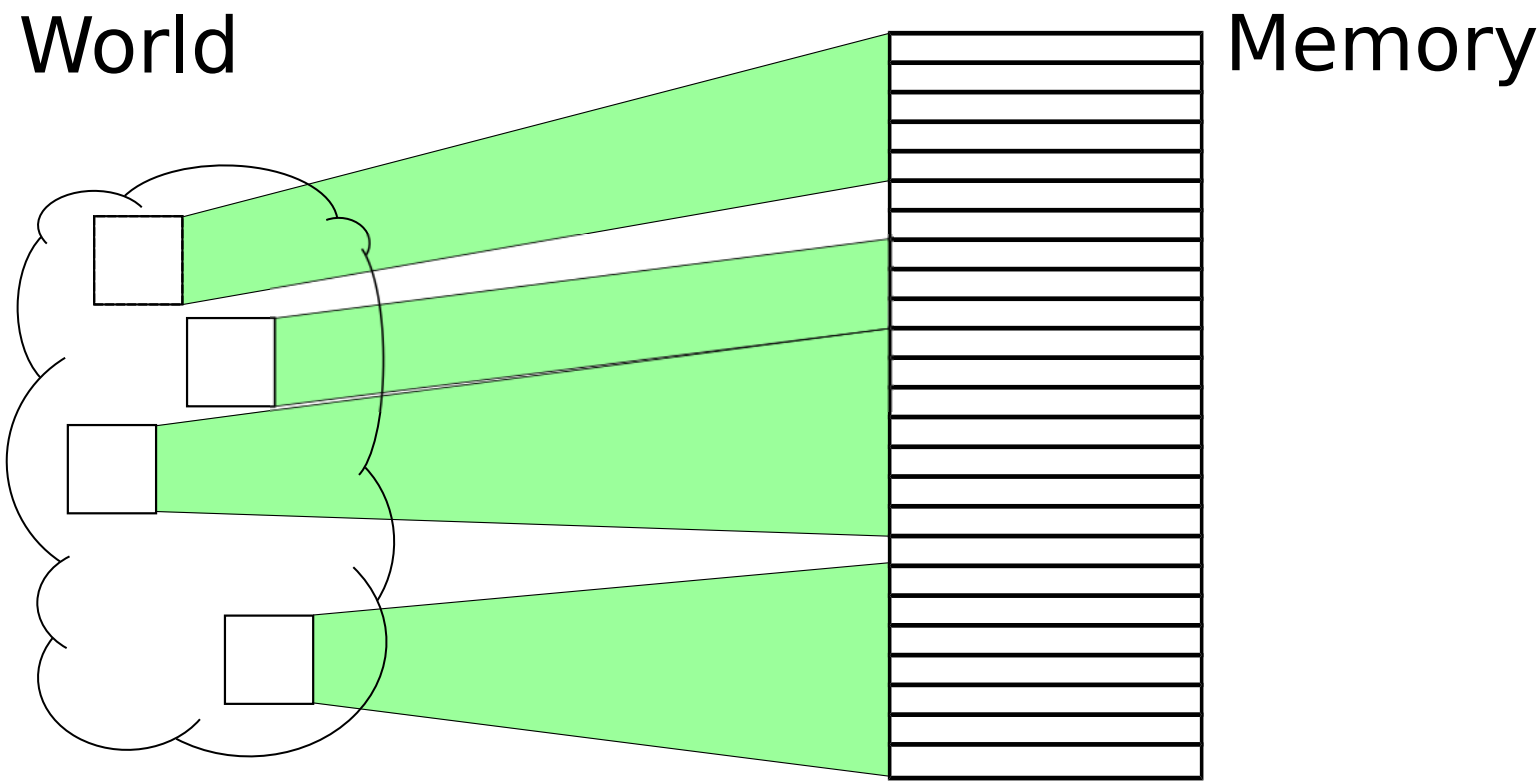
World



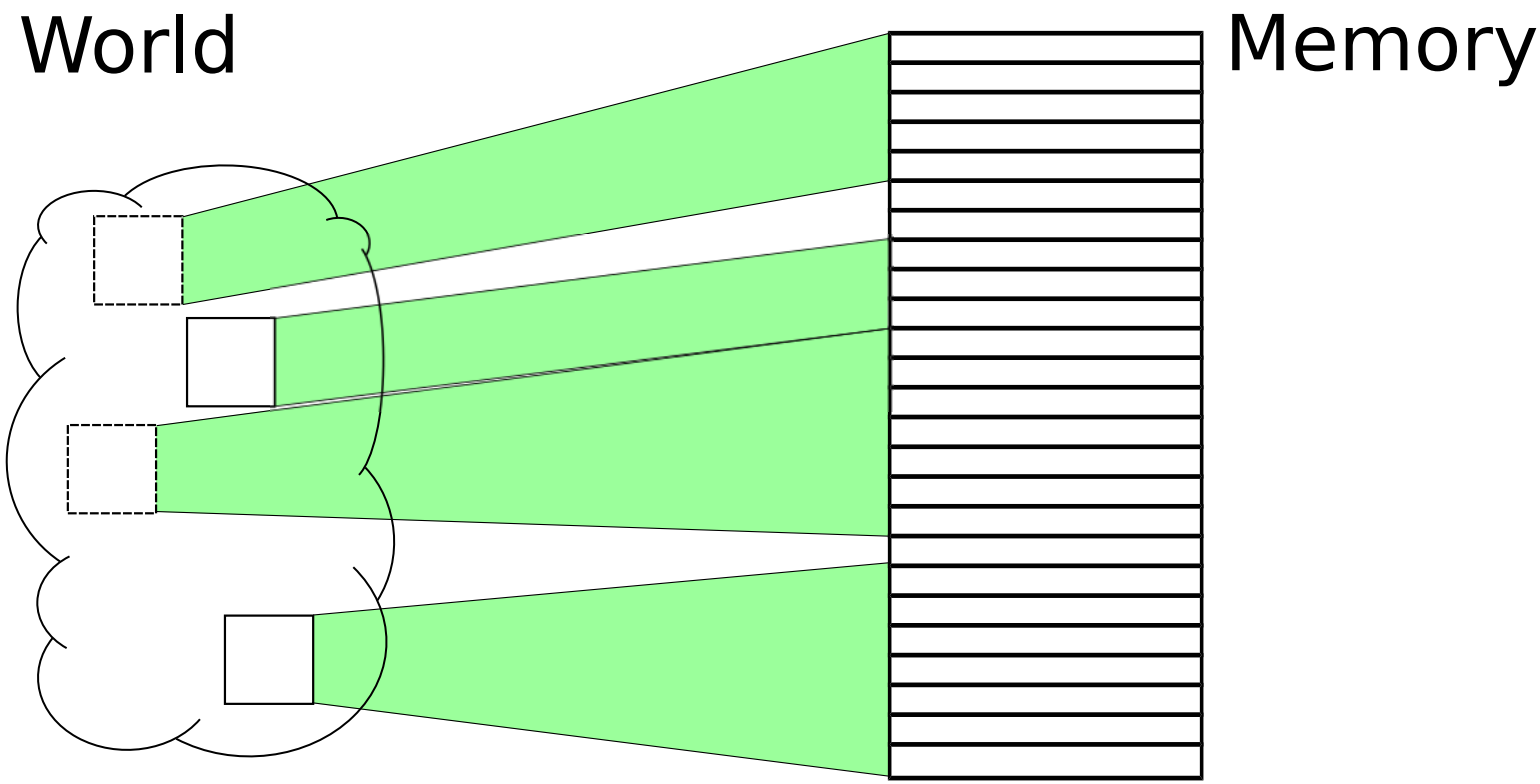


Memory

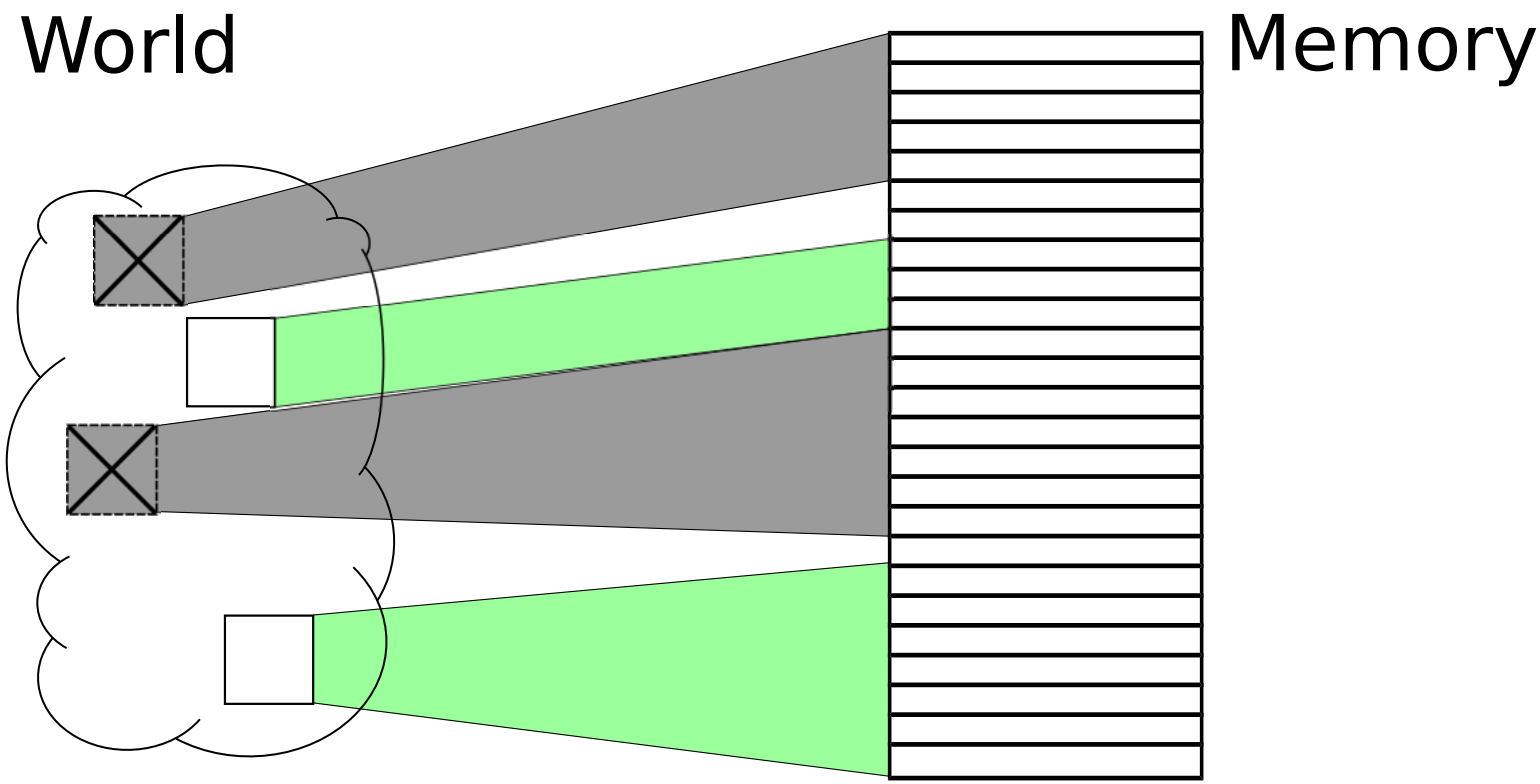
Worlds



Worlds

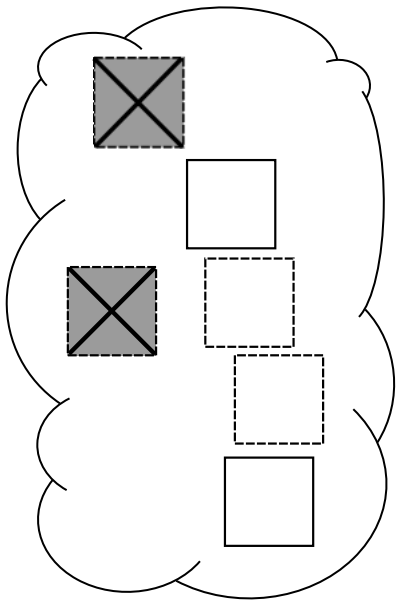


Worlds

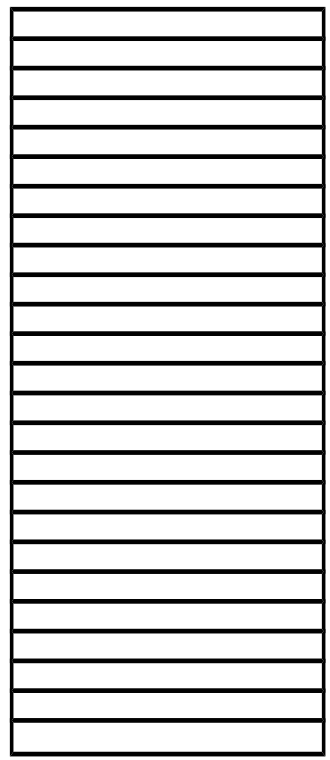


Worlds

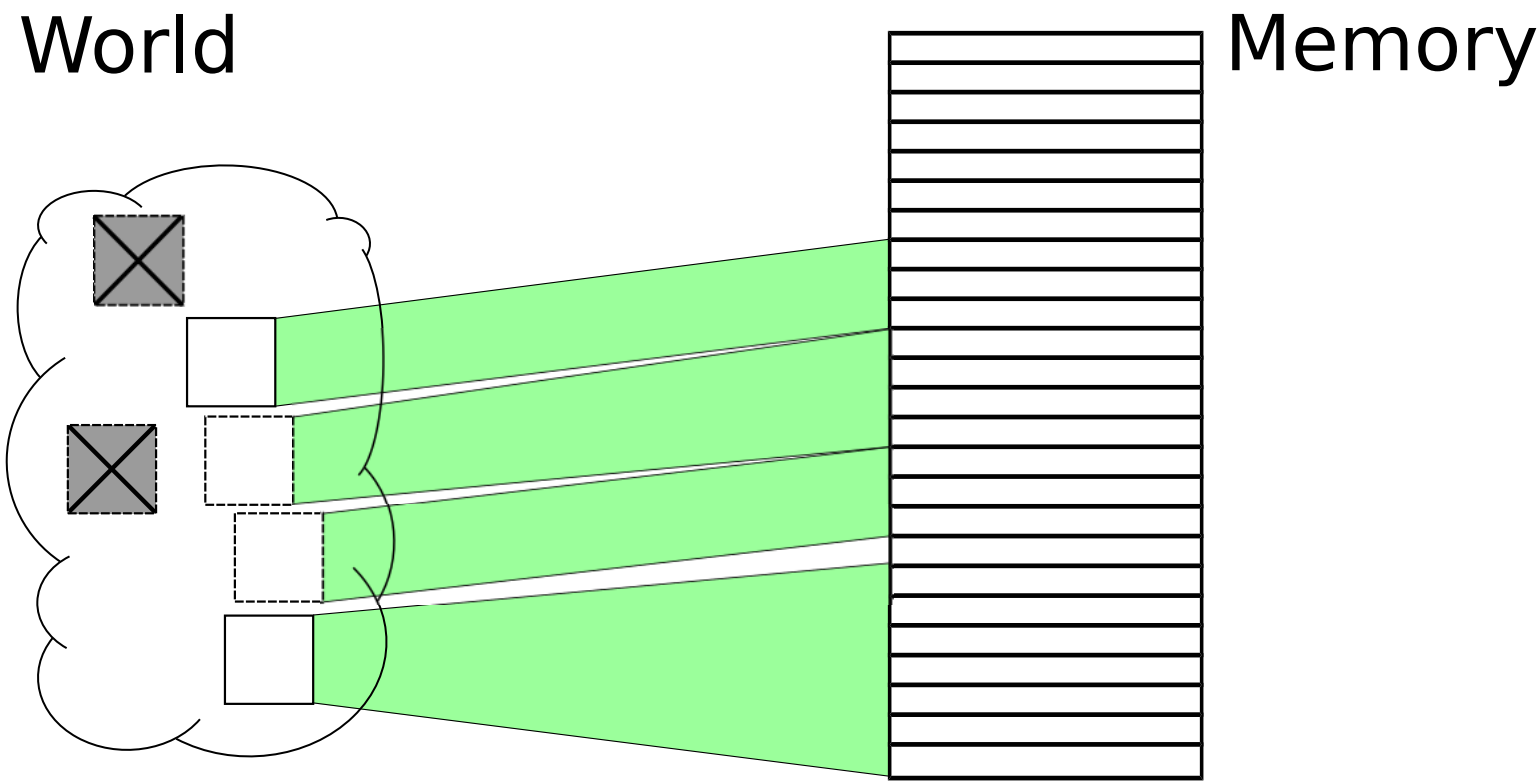
World



Memory



Worlds



Worlds

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

Worlds

- $\mathrm{World} : \mathbb{N} \xrightarrow{fin} \mathrm{Region}$
- Three kinds of regions:

Worlds

- $\mathrm{World} : \mathbb{N} \xrightarrow{fin} \mathrm{Region}$
- Three kinds of regions:

  permanent Models parts of memory that global and local capabilities can govern.

Worlds

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

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

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

- $\mathrm{World} : \mathbb{N} \xrightarrow{fin} \mathrm{Region}$
- 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 ($\sqsupseteq^{pub}$)

Worlds

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

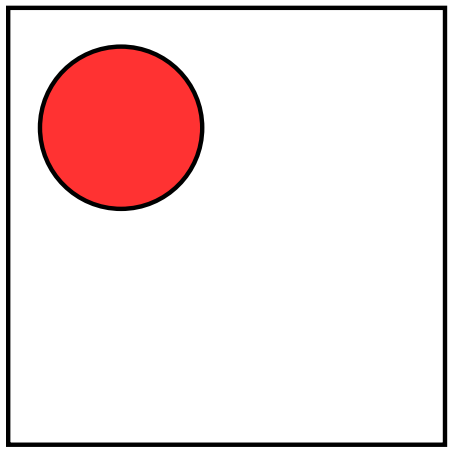
- Two future world relations

  public Extensional ($\sqsupseteq^{pub}$)

  private Extensional and temporary regions can be revoked! ($\sqsupseteq^{priv}$)

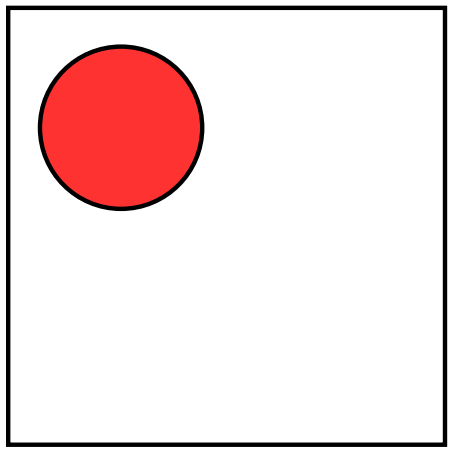
Regions

Region



Regions

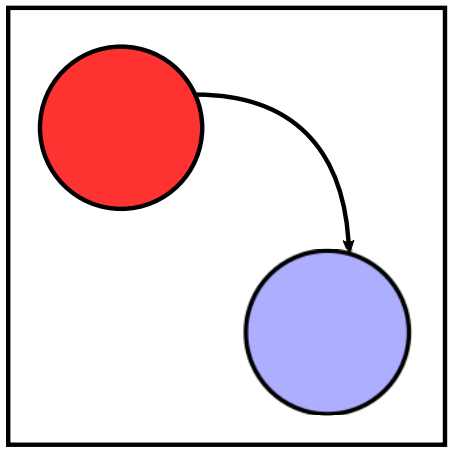
Region



Memory cell

Regions

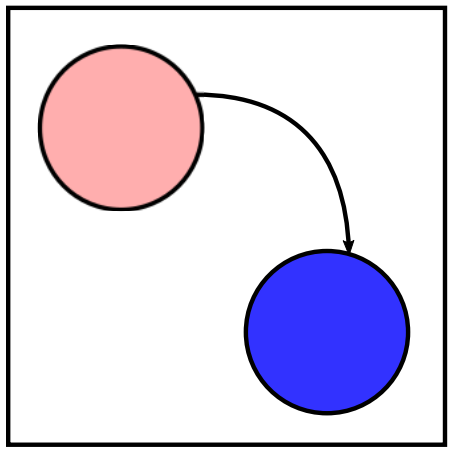
Region



Memory cell

Regions

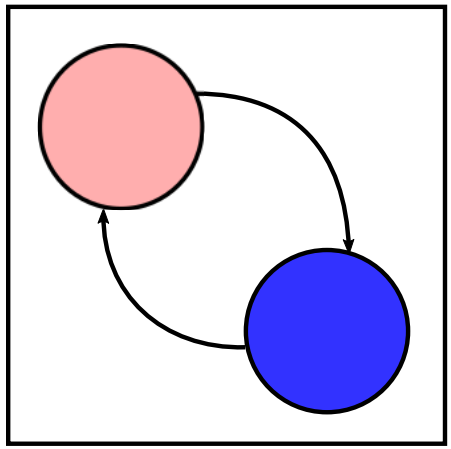
Region



Memory cell

Regions

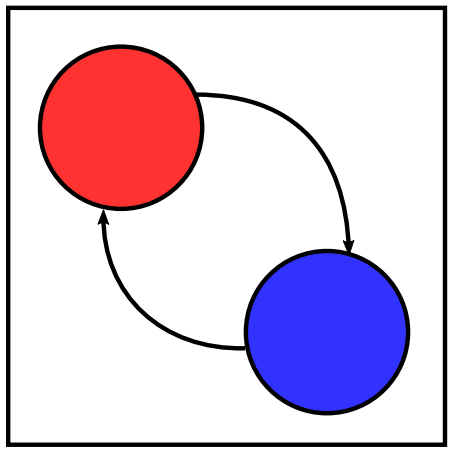
Region



Memory cell

Regions

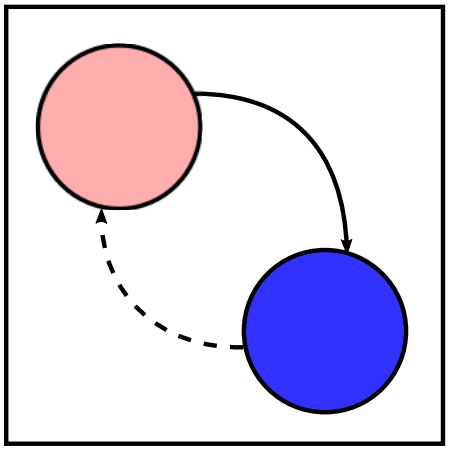
Region



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

Regions

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

Regions

- Regions are transition systems $(v, s, \phi_{pub}, \phi, H)$
  - v, view
  - s, current state
  - $\phi_{pub} : \mathrm{State}^2$, 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
  - $\phi_{pub} : \mathrm{State}^2$, reflexive and transitive
  - $\phi \supseteq \phi_{pub}$, reflexive and transitive
  - $H : \mathrm{State} \to \mathrm{World} \xrightarrow{mon} \mathrm{Pred}(\mathrm{MemSegment})$, state interpretation function

Regions

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

Regions

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

Regions

- Regions are transition systems $(v, s, \phi_{pub}, \phi, H)$
  - v, view
  - s, current state
  - $\phi_{pub} : \mathrm{State}^2$, reflexive and transitive
  - $\phi \supseteq \phi_{pub}$, reflexive and transitive
  - $H : \mathrm{State} \to \mathrm{World} \xrightarrow{mon} \mathrm{Pred}(\mathrm{MemSegment})$, state interpretation function
    - if region permanent, then H monotone w.r.t. $\sqsupseteq^{priv}$
    - if region temporary, then H monotone w.r.t. $\sqsupseteq^{pub}$
- Future Worlds
  - In $\sqsupseteq^{pub}$, existing regions are allowed to make transitions in ϕ_{pub}

Regions

- Regions are transition systems $(v, s, \phi_{pub}, \phi, H)$
  - v, view
  - s, current state
  - $\phi_{pub} : \mathrm{State}^2$, reflexive and transitive
  - $\phi \supseteq \phi_{pub}$, reflexive and transitive
  - $H : \mathrm{State} \to \mathrm{World} \xrightarrow{mon} \mathrm{Pred}(\mathrm{MemSegment})$, state interpretation function
    - if region permanent, then H monotone w.r.t. $\sqsupseteq^{priv}$
    - if region temporary, then H monotone w.r.t. $\sqsupseteq^{pub}$
- Future Worlds
  - In $\sqsupseteq^{pub}$, existing regions are allowed to make transitions in ϕ_{pub}
  - In $\sqsupseteq^{priv}$, existing regions are allowed to make transitions in ϕ

Logical Relation

- Step indexed

Logical Relation

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

Observation relation

- Executable configurations that produce desired results.

$$\mathcal{O} : \mathrm{World} \to \mathrm{Pred}(\mathrm{Reg} \times \mathrm{MemSegment})$$

$$\mathcal{O}(W) \stackrel{def}{=} \{(reg, ms) \mid$$

Observation relation

- Executable configurations that produce desired results.

$$\mathcal{O} : \mathrm{World} \to \mathrm{Pred}(\mathrm{Reg} \times \mathrm{MemSegment})$$

$$\mathcal{O}(W) \stackrel{def}{=} \{(reg, ms) \mid \forall ms_f, mem'. \\ (reg, ms \uplus ms_f) \to (halted, mem')$$

Observation relation

- Executable configurations that produce desired results.

$$\mathcal{O} : \mathrm{World} \to \mathrm{Pred}(\mathrm{Reg} \times \mathrm{MemSegment})$$

$$\begin{aligned}\mathcal{O}(W) \stackrel{def}{=} \{(reg, ms) \mid\ & \forall ms_f, mem'. \\ & (reg, ms \uplus ms_f) \to (halted, mem') \\ & \Rightarrow \exists W' \sqsupseteq^{priv} W. \exists ms_r, ms'. \\ & mem' = ms' \uplus ms_r \uplus ms_f \wedge \\ & ms' : W'\}\end{aligned}$$

Register-file relation

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

$$\mathcal{R} : \mathrm{World} \xrightarrow[\sqsupseteq^{pub}]{mon} \mathrm{Pred}(\mathrm{Reg})$$

$$\mathcal{R}(W) \stackrel{def}{=}$$

Register-file relation

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

$$\mathcal{R} : \mathrm{World} \xrightarrow[\sqsupseteq^{pub}]{mon} \mathrm{Pred}(\mathrm{Reg})$$

$$\mathcal{R}(W) \stackrel{def}{=} \{reg \mid \forall r \in \mathrm{RegisterName} \setminus \{\mathrm{pc}\}. \\ reg(r) \in \mathcal{V}(W)\}$$

Expression relation

- Words that produce admissable results when "executed".

$$\mathcal{E} : \mathrm{World} \to \mathrm{Pred}(\mathrm{Word})$$

$$\mathcal{E}(W) \stackrel{def}{=}$$

Expression relation

- Words that produce admissable results when "executed".

$$\mathcal{E} : \mathrm{World} \to \mathrm{Pred}(\mathrm{Word})$$

$$\mathcal{E}(W) \stackrel{def}{=} \{c \mid \forall reg \in \mathcal{R}(W).$$
$$\forall ms : W.$$

Expression relation

- Words that produce admissable results when "executed".

$$\mathcal{E} : \mathrm{World} \to \mathrm{Pred}(\mathrm{Word})$$

$$\mathcal{E}(W) \stackrel{def}{=} \{c \mid \forall reg \in \mathcal{R}(W).\ \forall ms : W.\ (reg[\mathrm{pc} \mapsto pc], ms) \in \mathcal{O}(W)\}$$

Value relation

$\mathcal{V}$-relation

- All integers (data) are in the $\mathcal{V}$-relation

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

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

$\mathcal{V}$-relation

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

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
  - Must respect $\sqsupseteq^{priv}$
  - Authority only over memory segments governed by permanent region.

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
  - Must respect $\sqsupseteq^{priv}$
  - Authority only over memory segments governed by permanent region.
- Local capabilities

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
  - Must respect $\sqsupseteq^{priv}$
  - Authority only over memory segments governed by permanent region.
- Local capabilities
  - Must respect $\sqsupseteq^{pub}$

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
  - Must respect $\sqsupseteq^{priv}$
  - Authority only over memory segments governed by permanent region.
- Local capabilities
  - Must respect $\sqsupseteq^{pub}$
  - Authority over memory segments governed by either permanent or temporary regions

Read condition

$$
\begin{aligned}
&\mathit{readCondition}(g, W, \mathit{base}, \mathit{end}) = \\
&\quad \{(\mathit{base}, \mathit{end}) \mid \exists r \in \mathit{localityReg}(g, W). \\
&\qquad\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\
&\qquad\qquad\quad W(r) \subset \iota^{\mathit{pwl}}_{\mathit{base}', \mathit{end}'}\}
\end{aligned}
$$

Read condition

$$
\begin{aligned}
&\mathit{readCondition}(g, W, \mathit{base}, \mathit{end}) = \\
&\quad \{(\mathit{base}, \mathit{end}) \mid \exists r \in \mathit{localityReg}(g, W). \\
&\qquad\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\
&\qquad\qquad\quad W(r) \subset \iota^{\mathit{pwl}}_{\mathit{base}', \mathit{end}'}\}
\end{aligned}
$$

$$
\iota^{\mathit{pwl}}_{\mathit{base}, \mathit{end}} \stackrel{\mathit{def}}{=} (\mathrm{temp}, 1, =, =, H^{\mathit{pwl}}_{\mathit{base}, \mathit{end}})
$$

Read condition

$$
\begin{aligned}
&\mathit{readCondition}(g, W, \mathit{base}, \mathit{end}) = \\
&\quad \{(\mathit{base}, \mathit{end}) \mid \exists r \in \mathit{localityReg}(g, W). \\
&\qquad\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\
&\qquad\qquad\quad W(r) \subset \iota^{\mathit{pwl}}_{\mathit{base}', \mathit{end}'}\}
\end{aligned}
$$

$$
\iota^{\mathit{pwl}}_{\mathit{base},\mathit{end}} \stackrel{\mathit{def}}{=} (\mathrm{temp}, 1, =, =, H^{\mathit{pwl}}_{\mathit{base},\mathit{end}})
$$

$$
H^{\mathit{pwl}} : \mathrm{Addr}^2 \to \mathrm{State} \to (\mathrm{Wor}_{\sqsupseteq^{\mathit{pub}}} \xrightarrow{\mathit{mon},\ \mathit{ne}} \mathrm{Pred}(\mathrm{MemSegment}))
$$

$$
H^{\mathit{pwl}}_{\mathit{base},\mathit{end}}\ s\ \hat{W} \stackrel{\mathit{def}}{=} \left\{ \mathit{ms} \;\middle|\; \begin{array}{l} \mathrm{dom}(\mathit{ms}) = [\mathit{base}, \mathit{end}] \wedge \\ \forall a \in [\mathit{base}, \mathit{end}].\ \mathit{ms}(a) \in \mathcal{V}(\hat{W}) \end{array} \right\}
$$

Write condition

$$
\begin{aligned}
&\mathit{writeCondition}(\iota, g, W, \mathit{base}, \mathit{end}) = \\
&\quad \{(\mathit{base}, \mathit{end}) \mid \exists r \in \mathit{localityReg}(g, W). \\
&\qquad\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\
&\qquad\qquad\quad W(r) \supset \iota_{\mathit{base}', \mathit{end}'} \}
\end{aligned}
$$

Write condition

$$\begin{aligned} &\mathit{writeCondition}(\iota, g, W, \mathit{base}, \mathit{end}) = \\ &\quad \{(\mathit{base}, \mathit{end}) \mid \exists r \in \mathit{localityReg}(g, W). \\ &\qquad\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\ &\qquad\qquad\quad W(r) \supset \iota_{\mathit{base}', \mathit{end}'} \} \end{aligned}$$

$$\iota^{\mathit{nwl}}_{\mathit{base},\mathit{end}} \stackrel{\mathit{def}}{=} (\mathrm{temp}, 1, =, =, H^{\mathit{nwl}}_{\mathit{start},\mathit{end}})$$

Write condition

$$\begin{aligned} &\mathit{writeCondition}(\iota, g, W, \mathit{base}, \mathit{end}) = \\ &\quad \{(\mathit{base}, \mathit{end}) \mid \exists r \in \mathit{localityReg}(g, W). \\ &\qquad\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\ &\qquad\qquad\quad W(r) \supset \iota_{\mathit{base}', \mathit{end}'} \} \end{aligned}$$

$$\iota^{\mathit{nwl}}_{\mathit{base},\mathit{end}} \stackrel{\mathit{def}}{=} (\mathrm{temp}, 1, =, =, H^{\mathit{nwl}}_{\mathit{start},\mathit{end}})$$

$$H^{\mathit{nwl}} : \mathrm{Addr}^2 \to \mathrm{State} \to (\mathrm{Wor}_{\sqsupseteq^{\mathit{priv}}} \xrightarrow{\mathit{mon},\ \mathit{ne}} \mathrm{Pred}(\mathrm{MemSegment}))$$

$$H^{\mathit{nwl}}_{\mathit{base},\mathit{end}}\ s\ \hat{W} \stackrel{\mathit{def}}{=} \left\{ \mathit{ms} \;\middle|\; \begin{array}{l} \mathrm{dom}(\mathit{ms}) = [\mathit{base}, \mathit{end}] \wedge \\ \forall a \in [\mathit{base}, \mathit{end}]. \\ \quad \mathit{ms}(a) \in \mathcal{V}(\mathit{revokeTemp}(\hat{W})) \end{array} \right\}$$

Conditions on execution

$$
\begin{aligned}
&\mathit{executeCondition}(g, W, \mathit{perm}, \mathit{base}, \mathit{end}) = \\
&\quad \{(\mathit{perm}, \mathit{base}, \mathit{end}) \mid \forall W' \sqsupseteq W. \\
&\qquad\qquad \forall a \in [\mathit{base}, \mathit{end}]. \\
&\qquad\qquad\quad ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \in \mathcal{E}(W')\}
\end{aligned}
$$

Conditions on execution

$$executeCondition(g, W, perm, base, end) =$$
$$\{(perm, base, end) \mid \forall W' \sqsupseteq W.$$
$$\forall a \in [base, end].$$
$$((perm, g), base, end, a) \in \mathcal{E}(W')\}$$

where $g = \mathrm{local} \Rightarrow \sqsupseteq = \sqsupseteq^{pub}$

and $g = \mathrm{global} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}$

Conditions on execution

$$\begin{aligned}
&\mathit{executeCondition}(g, W, \mathit{perm}, \mathit{base}, \mathit{end}) = \\
&\quad \{(\mathit{perm}, \mathit{base}, \mathit{end}) \mid \forall W' \sqsupseteq W. \\
&\qquad\qquad \forall a \in [\mathit{base}, \mathit{end}]. \\
&\qquad\qquad\quad ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \in \mathcal{E}(W')\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{enterCondition}(g, W, \mathit{base}, \mathit{end}, a) = \\
&\quad \{(\mathit{base}, \mathit{end}, a) \mid \forall W' \sqsupseteq W. \\
&\qquad\qquad ((\mathrm{rx}, g), \mathit{base}, \mathit{end}, a) \in \mathcal{E}(W')\}
\end{aligned}$$

where $g = \mathrm{local} \Rightarrow \sqsupseteq = \sqsupseteq^{\mathit{pub}}$

and $g = \mathrm{global} \Rightarrow \sqsupseteq = \sqsupseteq^{\mathit{priv}}$

Value relation

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

$$\mathcal{V}(W) \overset{def}{=}$$

Value relation

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

$$\mathcal{V}(W) \stackrel{def}{=} \{i \mid i \in \mathbb{Z}\} \cup$$

Value relation

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

$$\begin{aligned}\mathcal{V}(W) \stackrel{def}{=} & \{i \mid i \in \mathbb{Z}\} \cup \\ & \{((o, g), base, end, a)\} \cup\end{aligned}$$

Value relation

$$\mathcal{V} : \mathrm{World} \xrightarrow[\sqsupseteq^{pub}]{mon} \mathrm{Pred}(\mathrm{Word})$$

$$\begin{aligned}\mathcal{V}(W) \stackrel{def}{=} & \{i \mid i \in \mathbb{Z}\} \cup \\ & \{((\mathrm{o}, g), base, end, a)\} \cup \\ & \{((\mathrm{e}, g), base, end, a) \mid \\ & \quad enterCondition(g, W, base, end, a)\} \cup\end{aligned}$$

Value relation

$$\mathcal{V} : \mathrm{World} \xrightarrow[\sqsupseteq^{pub}]{mon} \mathrm{Pred}(\mathrm{Word})$$

$$\begin{aligned}
\mathcal{V}(W) \stackrel{def}{=} & \{i \mid i \in \mathbb{Z}\} \cup \\
& \{((\mathrm{o}, g), base, end, a)\} \cup \\
& \{((\mathrm{e}, g), base, end, a) \mid \\
& \quad enterCondition(g, W, base, end, a)\} \cup \\
& \{((\mathrm{rwlx}, g), base, end, a) \mid \\
& \quad readCondition(g, W, base, end) \wedge \\
& \quad writeCondition(\iota^{pwl}, g, W, base, end) \wedge \\
& \quad executeCondition(g, W, \mathrm{rwlx}, base, end)\} \cup
\end{aligned}$$

Value relation

$$\mathcal{V} : \mathrm{World} \xrightarrow[\sqsupseteq^{pub}]{mon} \mathrm{Pred}(\mathrm{Word})$$

$$\begin{aligned}
\mathcal{V}(W) \stackrel{def}{=}\ & \{i \mid i \in \mathbb{Z}\} \cup \\
& \{((\mathrm{o}, g), base, end, a)\} \cup \\
& \{((\mathrm{e}, g), base, end, a) \mid \\
& \quad enterCondition(g, W, base, end, a)\} \cup \\
& \{((\mathrm{rwlx}, g), base, end, a) \mid \\
& \quad readCondition(g, W, base, end) \wedge \\
& \quad writeCondition(\iota^{pwl}, g, W, base, end) \wedge \\
& \quad executeCondition(g, W, \mathrm{rwlx}, base, end)\} \cup \\
& \quad \ldots
\end{aligned}$$

Fundamental Theorem of Logical Relations (FTLR)

Lemma (FTLR)

For all $W \in \mathrm{World}$ *and* $c \in \mathit{Caps}$,

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

Fundamental Theorem of Logical Relations (FTLR)

Lemma (FTLR)

For all $W \in \mathrm{World}$ *and* $c \in \mathit{Caps}$,

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

- The pc-register can be accessed like any other register

Fundamental Theorem of Logical Relations (FTLR)

Lemma (FTLR)

For all $W \in \mathrm{World}$ *and* $c \in \mathit{Caps}$,

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

- The pc-register can be accessed like any other register
- Capability must behave when used for read/write

Fundamental Theorem of Logical Relations (FTLR)

Lemma (FTLR)

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

$$\mathit{perm} = \mathrm{rx} \text{ and } \mathit{readCondition}(g, W, \mathit{base}, \mathit{end}),$$

or

$$\mathit{perm} = \mathrm{rwx} \text{ and } \mathit{readCondition}(W, \mathit{base}, \mathit{end})$$
$$\text{and } \mathit{writeCondition}(\iota^{\mathit{nwl}}, g, W, \mathit{base}, \mathit{end})$$

or

$$\ldots$$

then

$$(\mathit{perm}, \mathit{base}, \mathit{end}, a) \in \mathcal{E}(W).$$

The awkward example

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

```
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 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.

References

[1] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *International Symposium on Computer Architecuture*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.