Implementing a Capability Machine model into Iris

Aïna Linn Georges Alix Trieu Lars Birkedal

Aarhus University

ageorges@cs.au.dk

December 23, 2019

Introduction

- Capability machines allow for fine grained control over pointer permissions
- Good target for secure compilation
- In particular: we are interested in enforcing certain higher level abstractions such as *local state encapsulation* as *well-bracketed control flow* at the lowest level of the machine
- We need tools to reason about these subtle properties in a language that does not enforce them
- These tools are elaborate and complex: we want to mechanize them, and facilitate the process of using them

Introduction

- Capability machines allow for fine grained control over pointer permissions
- Good target for secure compilation
- In particular: we are interested in enforcing certain higher level abstractions such as *local state encapsulation* as *well-bracketed control flow* at the lowest level of the machine
- We need tools to reason about these subtle properties in a language that does not enforce them
- These tools are elaborate and complex: we want to mechanize them, and facilitate the process of using them

Introduction

- Capability machines allow for fine grained control over pointer permissions
- Good target for secure compilation
- In particular: we are interested in enforcing certain higher level abstractions such as *local state encapsulation* as *well-bracketed control flow* at the lowest level of the machine
- We need tools to reason about these subtle properties in a language that does not enforce them
- These tools are elaborate and complex: we want to mechanize them, and facilitate the process of using them

Introduction

- Capability machines allow for fine grained control over pointer permissions
- Good target for secure compilation
- In particular: we are interested in enforcing certain higher level abstractions such as *local state encapsulation* as *well-bracketed control flow* at the lowest level of the machine
- We need tools to reason about these subtle properties in a language that does not enforce them
- These tools are elaborate and complex: we want to mechanize them, and facilitate the process of using them

Introduction

- Capability machines allow for fine grained control over pointer permissions
- Good target for secure compilation
- In particular: we are interested in enforcing certain higher level abstractions such as *local state encapsulation* as *well-bracketed control flow* at the lowest level of the machine
- We need tools to reason about these subtle properties in a language that does not enforce them
- These tools are elaborate and complex: we want to mechanize them, and facilitate the process of using them

Overview

Capability Machines

Reasoning about Capability Safety

Program Logic

A Unary Logical Relation for Reasoning about Semantic Properties of an Untyped Language

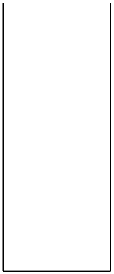
The Fundamental Theorem of Logical Relations

Reasoning about Unknown Code

Capability Machines

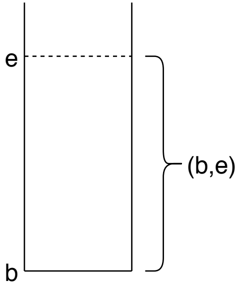
Capability Machine

Capability: An unforgeable token of authority



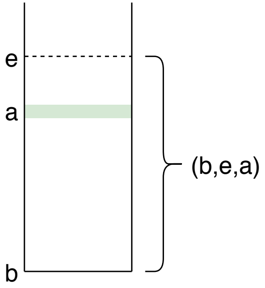
Capability Machine

Capability: An unforgeable token of authority



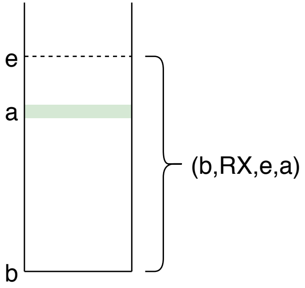
Capability Machine

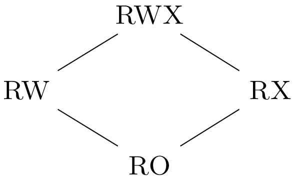
Capability: An unforgeable token of authority



Capability Machine

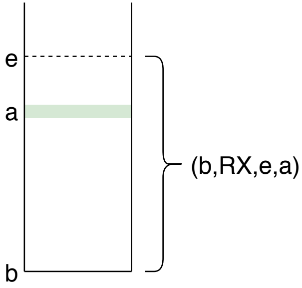
Capability: An unforgeable token of authority

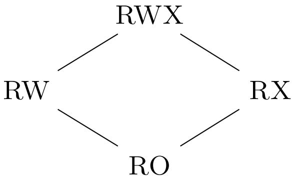




Capability Machine

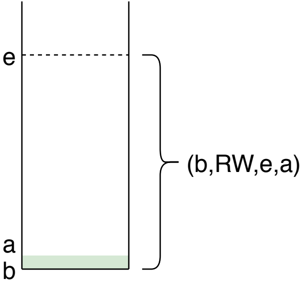
Capability: An unforgeable token of authority





Enforcing Local Stack Encapsulation using Capabilities

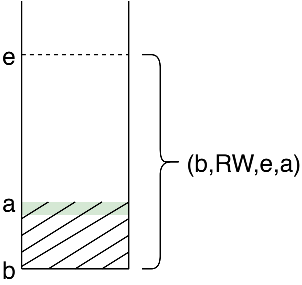
Local State Encapsulation




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
halt
```

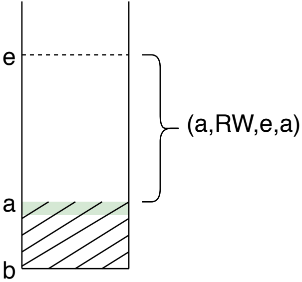
Capability Safety

Local State Encapsulation




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
halt
```

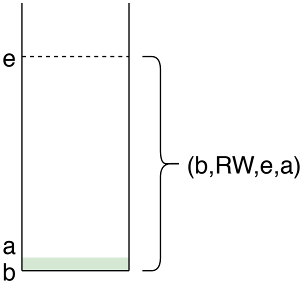
Local State Encapsulation




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
halt
```

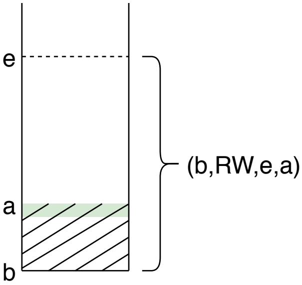
Enforcing Well Bracketed Control Flow using Capabilities

Well Bracketed Control Flow



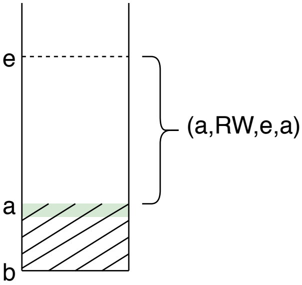

```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

Well Bracketed Control Flow




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

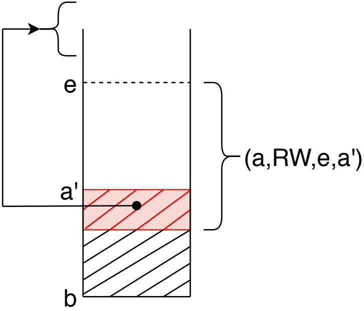
Well Bracketed Control Flow




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

Capability Safety

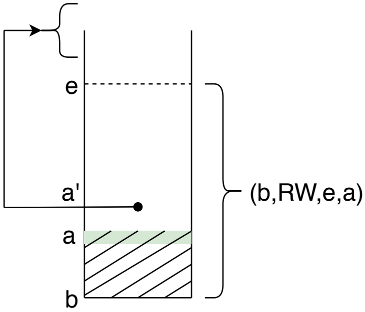
Well Bracketed Control Flow




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

Capability Safety

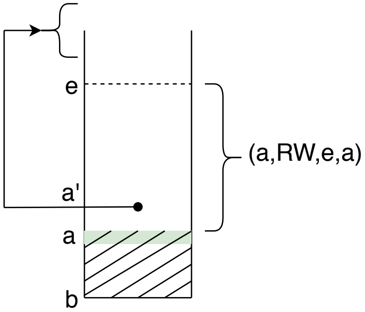
Well Bracketed Control Flow




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

Capability Safety

Well Bracketed Control Flow




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

Local Capabilities

Local Capabilities

(p,**Local**,b,e,a)

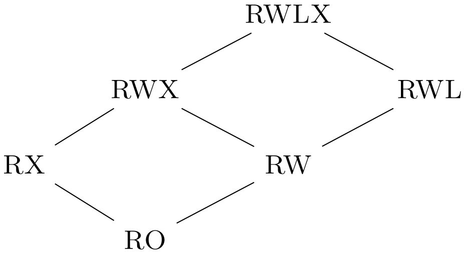
well-bracketed

(p,**Global**,b,e,a)

not well-bracketed

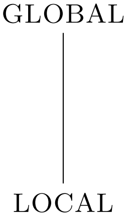
Local Capabilities

(p,**Local**,b,e,a)



well-bracketed

(p,**Global**,b,e,a)

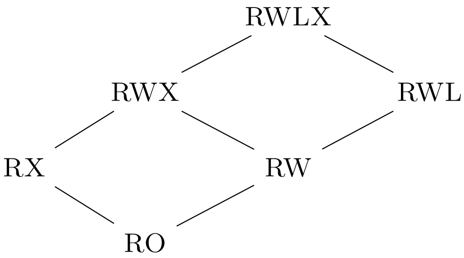


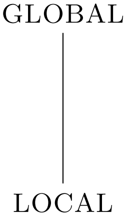
not well-bracketed

Local Capabilities

(p,**Local**,b,e,a)

(p,**Global**,b,e,a)

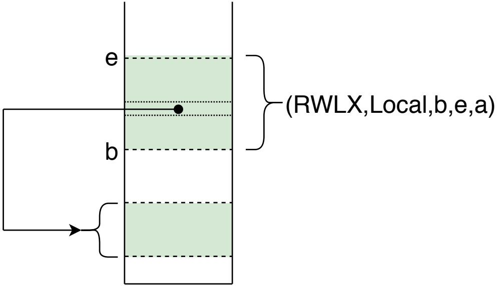




well-bracketed

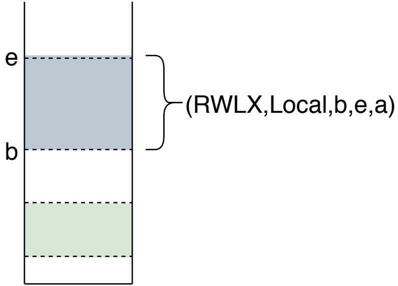
not well-bracketed

Calling Convention



`r_stk`	$(RWLX, Local, b, e, a)$

Calling Convention



`r_stk`	$(RWLX, Local, b, e, a)$

Reasoning about Capability Safety

Expressing Capability Safety

- using a Program Logic
- using a logical relation to capture invariants on the type system
- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**

Expressing Capability Safety

- using a Program Logic
- using a logical relation to capture invariants on the type system
- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**

Expressing Capability Safety

- using a Program Logic
- using a logical relation to capture invariants on the type system
- using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language

Expressing Capability Safety

- using a Program Logic
- using a logical relation to capture invariants on the type system
- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**

Step-indexed Kripke Logical Relation

$$\mathcal{V}(W) \triangleq \{n, (RW, g, b, e, a) | \cdots\} \cup \cdots$$

- World-circularity problem
  - Step indexing
- The world may evolve: we need future world relation
  - Local capabilities are revoked whereas Global capabilities are not, *the relation needs to model this distinction:*

$$\sqsupseteq_{pub} \quad \textit{and} \quad \sqsupseteq_{priv}$$

Step-indexed Kripke Logical Relation

$$\mathcal{V}(W) \triangleq \{n, (RW, g, b, e, a) | \cdots\} \cup \cdots$$

- World-circularity problem
  - Step indexing
- The world may evolve: we need future world relation
  - Local capabilities are revoked whereas Global capabilities are not, *the relation needs to model this distinction:*

$$\sqsupseteq_{pub} \qquad \textit{and} \qquad \sqsupseteq_{priv}$$

Step-indexed Kripke Logical Relation

$$\mathcal{V}(W) \triangleq \{n, (RW, g, b, e, a) | \exists r, W(r) = \iota_{[b,e]}\} \cup \cdots$$

- World-circularity problem
  - Step indexing
- The world may evolve: we need future world relation
  - Local capabilities are revoked whereas Global capabilities are not, *the relation needs to model this distinction:*

$$\sqsupseteq_{pub} \qquad \textit{and} \qquad \sqsupseteq_{priv}$$

Step-indexed Kripke Logical Relation

$$\mathcal{V}(W) \triangleq \{n, (RW, g, b, e, a) | \exists r, W(r) = \iota_{[b,e]}\} \cup \cdots$$

- World-circularity problem
  - Step indexing
- The world may evolve: we need future world relation
  - Local capabilities are revoked whereas Global capabilities are not, *the relation needs to model this distinction:*

$$\sqsupseteq_{pub} \qquad \textit{and} \qquad \sqsupseteq_{priv}$$

Step-indexed Kripke Logical Relation

$$\mathcal{V}(W) \triangleq \{n, (RW, g, b, e, a) | \exists r, W(r) \stackrel{n}{=} \iota_{[b,e]}\} \cup \cdots$$

- World-circularity problem
  - Step indexing
- The world may evolve: we need future world relation
  - Local capabilities are revoked whereas Global capabilities are not, *the relation needs to model this distinction:*

$$\sqsupseteq_{pub} \quad \textit{and} \quad \sqsupseteq_{priv}$$

Step-indexed Kripke Logical Relation

$$\mathcal{V}(W) \triangleq \{n, (RW, g, b, e, a) | \exists r, W(r) \stackrel{n}{=} \iota_{[b,e]}\} \cup \cdots$$

- World-circularity problem
  - Step indexing
- The world may evolve: we need future world relation
  - Local capabilities are revoked whereas Global capabilities are not, *the relation needs to model this distinction*:

$$\sqsupseteq_{pub} \qquad \textit{and} \qquad \sqsupseteq_{priv}$$

Step-indexed Kripke Logical Relation

$$\mathcal{V}(W) \triangleq \{n, (RW, g, b, e, a) | \exists r, W(r) \stackrel{n}{=} \iota_{[b,e]}\} \cup \cdots$$

- World-circularity problem
  - Step indexing
- The world may evolve: we need future world relation
  - Local capabilities are revoked whereas Global capabilities are not, *the relation needs to model this distinction*:

$$\sqsupseteq_{pub} \qquad \textit{and} \qquad \sqsupseteq_{priv}$$

Expressing Capability Safety in Iris - an Iris primer

Iris: Higher-order Concurrent Separation Logic Framework

- Foundational
- Implemented in Coq – equipped with an interactive proof mode
- Framework – embed any language and its operational semantics into Iris
- Comes equipped with:
  - Invariants
  - Ghost state

We can take advantage of Iris' step-indexed model and invariants to mechanize step-indexed Kripke logical relations with recursive worlds in a succinct and elegant way.

Expressing Capability Safety in Iris - an Iris primer

Iris: Higher-order Concurrent Separation Logic Framework

- Foundational
- Implemented in Coq – equipped with an interactive proof mode
- Framework – embed any language and its operational semantics into Iris
- Comes equipped with:
  - Invariants
  - Ghost state

We can take advantage of Iris' step-indexed model and invariants to mechanize step-indexed Kripke logical relations with recursive worlds in a succinct and elegant way.

Expressing Capability Safety in Iris - an Iris primer

Iris: Higher-order Concurrent Separation Logic Framework

- Foundational
- Implemented in Coq – equipped with an interactive proof mode
- Framework – embed any language and its operational semantics into Iris
- Comes equipped with:
  - Invariants
  - Ghost state

We can take advantage of Iris' step-indexed model and invariants to mechanize step-indexed Kripke logical relations with recursive worlds in a succinct and elegant way.

Expressing Capability Safety in Iris - an Iris primer

Iris: Higher-order Concurrent Separation Logic Framework

- Foundational
- Implemented in Coq – equipped with an interactive proof mode
- Framework – embed any language and its operational semantics into Iris
- Comes equipped with:
  - Invariants
  - Ghost state

We can take advantage of Iris' step-indexed model and invariants to mechanize step-indexed Kripke logical relations with recursive worlds in a succinct and elegant way.

Expressing Capability Safety in Iris - an Iris primer

Iris: Higher-order Concurrent Separation Logic Framework

- Foundational
- Implemented in Coq – equipped with an interactive proof mode
- Framework – embed any language and its operational semantics into Iris
- Comes equipped with:
  - Invariants
  - Ghost state

We can take advantage of Iris' step-indexed model and invariants to mechanize step-indexed Kripke logical relations with recursive worlds in a succinct and elegant way.

Expressing Capability Safety in Iris - an Iris primer

Iris: Higher-order Concurrent Separation Logic Framework

- Foundational
- Implemented in Coq – equipped with an interactive proof mode
- Framework – embed any language and its operational semantics into Iris
- Comes equipped with:
  - Invariants
  - Ghost state

We can take advantage of Iris' step-indexed model and invariants to mechanize step-indexed Kripke logical relations with recursive worlds in a succinct and elegant way.

Expressing Capability Safety in Iris

Roadmap

- embed the language into Iris
- define a program logic by proving Hoare Triples
- define the logical relation – using Iris tools to solve the world circularity problem
- prove the fundamental theorem of logical relations
- use the logical relation to prove examples that rely on local state encapsulation and well-bracketed control flow with calls to unknown adversary

Roadmap

- embed the language into Iris
- define a program logic by proving Hoare Triples
- define the logical relation – using Iris tools to solve the world circularity problem
- prove the fundamental theorem of logical relations
- use the logical relation to prove examples that rely on local state encapsulation and well-bracketed control flow with calls to unknown adversary

Expressing Capability Safety in Iris

Roadmap

- embed the language into Iris
- define a program logic by proving Hoare Triples
- define the logical relation – using Iris tools to solve the world circularity problem
- prove the fundamental theorem of logical relations
- use the logical relation to prove examples that rely on local state encapsulation and well-bracketed control flow with calls to unknown adversary

Roadmap

- embed the language into Iris
- define a program logic by proving Hoare Triples
- define the logical relation – using Iris tools to solve the world circularity problem
- prove the fundamental theorem of logical relations
- use the logical relation to prove examples that rely on local state encapsulation and well-bracketed control flow with calls to unknown adversary

Roadmap

- embed the language into Iris
- define a program logic by proving Hoare Triples
- define the logical relation – using Iris tools to solve the world circularity problem
- prove the fundamental theorem of logical relations
- use the logical relation to prove examples that rely on local state encapsulation and well-bracketed control flow with calls to unknown adversary

Expressing Capability Safety in Iris

Roadmap

- embed the language into Iris
- define a program logic by proving Hoare Triples
- define the logical relation – using Iris tools to solve the world circularity problem
- prove the fundamental theorem of logical relations
- use the logical relation to prove examples that rely on local state encapsulation and well-bracketed control flow with calls to unknown adversary

Program Logic

Abstract Instructions

$$(reg, mem) \to (reg', mem')$$

- Instr Executable
- Instr Halted $\to$ HaltedV
- Instr Failed $\to$ FailedV

Abstract Instructions

$$(reg, mem) \to (reg', mem')$$

- Instr Executable
- Instr Halted $\to$ HaltedV
- Instr Failed $\to$ FailedV

Abstract Instructions

$$(reg, mem) \rightarrow (reg', mem')$$

- Instr Executable
- Instr Halted $\rightarrow$ HaltedV
- Instr Failed $\rightarrow$ FailedV

Abstract Instructions

$$(reg, mem) \rightarrow (reg', mem')$$

- Instr Executable
- Instr Halted $\rightarrow$ HaltedV
- Instr Failed $\rightarrow$ FailedV

Abstract Instructions

$$(reg, mem) \rightarrow (reg', mem')$$

- Instr Executable
- Instr Halted $\rightarrow$ HaltedV
- Instr Failed $\rightarrow$ FailedV

A Capability Points-to Predicate

Points-to Predicate with Permissions

$$a \mapsto_a [RWL]w$$

Points-to Predicate with Permissions

$$a \mapsto_a [RWL]w \Longrightarrow\!\!\!\!* \; a \mapsto_a [RWL]((p, Local), b, e, l)$$

Points-to Predicate with Permissions

$$
\begin{aligned}
a \mapsto_a [RWL]w &\Longrightarrow\!\!\!\ast\ a \mapsto_a [RWL]((p, Local), b, e, l)\\
&\Longrightarrow\!\!\!\ast\ a \mapsto_a [RW]((p, Local), b, e, l)
\end{aligned}
$$

Points-to Predicate with Permissions

$$
\begin{aligned}
a \mapsto_a [RWL]w &\Longrightarrow\!\!\ast\; a \mapsto_a [RWL]((p, Local), b, e, l) \\
&\Longrightarrow\!\!\ast\; a \mapsto_a [RW]((p, Local), b, e, l) \\
&\not\Longrightarrow\!\!\ast\; a \mapsto_a [RW]((p', Local), b', e', l')
\end{aligned}
$$

A Unary Logical Relation for Reasoning about Semantic Properties of an Untyped Language

The Value Relation

A unary logical relation of an un-typed language

$$\mathcal{V} : \textit{Word} \rightarrow \textit{iProp}\ \Sigma$$

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

$$\mathcal{V}((\textsc{rw}, g), b, e, a) \triangleq \mathop{\Large *}_{a \in [b,e]} \boxed{\exists w, a \mapsto_a [RW]w \twoheadrightarrow \mathcal{V}(w)}$$

The Value Relation

A unary logical relation of an un-typed language

$$\mathcal{V} : \mathit{Word} \rightarrow \mathit{iProp}\ \Sigma$$

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

$$\mathcal{V}((\text{RW}, g), b, e, a) \triangleq \mathop{\ast}_{a \in [b,e]} \boxed{\exists w, a \mapsto_a [RW]w \twoheadrightarrow \mathcal{V}(w)}$$

A unary logical relation of an un-typed language

$$\mathcal{V} : \mathit{Word} \rightarrow \mathit{iProp}\ \Sigma$$

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

$$\mathcal{V}((\textsc{rw}, g), b, e, a) \triangleq \mathop{\ast}_{a \in [b,e]} \boxed{\exists w, a \mapsto_a [RW] w \ast \mathcal{V}(w)}$$

A unary logical relation of an un-typed language

$$\mathcal{V} : \mathit{Word} \rightarrow \mathit{iProp}\ \Sigma$$

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

$$\mathcal{V}((\textsc{rw}, g), b, e, a) \triangleq \mathop{\ast}_{a \in [b,e]} \boxed{\exists w, a \mapsto_a [RW]w \twoheadrightarrow \mathcal{V}(w)}$$

The Value Relation

A unary logical relation of an un-typed language

$$\mathcal{V} : \mathit{Word} \to \mathit{iProp}\ \Sigma$$

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

$$\mathcal{V}((\text{RW}, g), b, e, a) \triangleq \mathop{\ast}_{a \in [b,e]} \boxed{\exists w, a \mapsto_a [RW]w \twoheadrightarrow \mathcal{V}(W)(w)}$$

The Value Relation

A unary logical relation of an un-typed language

$$\mathcal{V} : \mathit{STS} \rightarrow \ \mathit{Word} \rightarrow \mathit{iProp}\ \Sigma$$

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

$$\mathcal{V}(W)((\textsc{rw}, g), b, e, a) \triangleq \mathop{\Huge *}_{a \in [b,e]} \boxed{\exists w, a \mapsto_a [RW] w \mathrel{-\!\!*} \mathcal{V}(W)(w)}$$

The Execute Condition

The Execute Condition

$$\text{exec\_cond(W)(p,g,b,e)} \triangleq \begin{cases} \forall a \in [b\ e], W' \sqsupseteq_{pub} W. & \\ \quad \triangleright\ \mathcal{E}(W')((((p,g),b,e,a)) & g = Local \\ & \\ \forall a \in [b\ e], W' \sqsupseteq_{priv} W. & \\ \quad \triangleright\ \mathcal{E}(W')((((p,g),b,e,a)) & g = Global \end{cases}$$

The Expression Relation

The Expression Relation

$$
\begin{aligned}
\mathcal{E}(W)(pc) \triangleq \forall r, \mathcal{R}(W)(r) \;*\; & \text{context}(W)(r[\text{PC} := pc]) \\
\mathrel{-\!\!*} \; & \mathrm{WP} \ \text{Seq (Instr Executable)} \\
& \{v, v = \mathit{HaltedV} \implies \exists W' r', W' \sqsubseteq_{priv} W \\
& * \ \text{context}(W')(r')\}
\end{aligned}
$$

The Expression Relation

$$\begin{aligned}\mathcal{E}(W)(pc) \triangleq \forall r, \mathcal{R}(W)(r) \ * \ & \text{context}(W)(r[\text{PC} := pc]) \\ \twoheadrightarrow \ & \mathrm{WP}\ \text{Seq (Instr Executable)} \\ & \quad \{v, v = \mathit{HaltedV} \implies \exists W' r', W' \sqsubseteq_{priv} W \\ & \quad * \ \text{context}(W')(r')\}\end{aligned}$$

$$\text{context}(W)(r) = \ ?$$

The Expression Relation

$$\mathcal{E}(W)(pc) \triangleq \forall r, \mathcal{R}(W)(r) \ * \ \text{context}(W)(r[\text{PC} := pc])$$
$$-\!\!* \ \mathrm{WP}\ \text{Seq (Instr Executable)}$$
$$\{v, v = HaltedV \implies \exists W' r', W' \sqsubseteq_{priv} W$$
$$* \ \text{context}(W')(r')\}$$

$$\text{context}(W)(r) = \Big(\mathop{\Huge *}_{r_i \mapsto w \in r} r_i \mapsto_r w \Big) \wedge \text{full\_map}\ r$$

The Expression Relation

$$\mathcal{E}(W)(pc) \triangleq \forall r, \mathcal{R}(W)(r) \ * \ \text{context}(W)(r[\text{PC} := pc])$$
$$\mathrel{-\!\!*} \ \mathrm{WP} \ \text{Seq (Instr Executable)}$$
$$\{v, v = HaltedV \implies \exists W' r', W' \sqsubseteq_{priv} W$$
$$* \ \text{context}(W')(r')\}$$

$$\text{context}(W)(r) = \Big(\mathop{\Large *}_{r_i \mapsto w \in r} r_i \mapsto_r w \Big) \land \text{full\_map } r$$
$$* \ \text{na\_inv } \gamma_{na} \top$$

The Expression Relation

$$
\begin{aligned}
\mathcal{E}(W)(pc) \triangleq \forall r, \mathcal{R}(W)(r) \; * \; & \text{context}(W)(r[\text{PC} := pc]) \\
-\!\!* \; & \mathrm{WP}\ \text{Seq (Instr Executable)} \\
& \{v, v = \mathit{HaltedV} \implies \exists W' r', W' \sqsubseteq_{priv} W \\
& * \, \text{context}(W')(r')\}
\end{aligned}
$$

$$
\begin{aligned}
\text{context}(W)(r) = \; & \Big(\mathop{\Huge *}_{r_i \mapsto w \in r} r_i \mapsto_r w \Big) \land \text{full\_map}\ r \\
& * \, \text{na\_inv}\ \gamma_{na} \top \\
& * \, \text{sts\_full}\ W
\end{aligned}
$$

The Expression Relation

$$\mathcal{E}(W)(pc) \triangleq \forall r, \mathcal{R}(W)(r) \ * \ \text{context}(W)(r[\text{PC} := pc])$$
$$\twoheadrightarrow \ \mathrm{WP}\ \text{Seq}\ (\text{Instr Executable})$$
$$\{v, v = HaltedV \implies \exists W' r', W' \sqsubseteq_{priv} W$$
$$* \ \text{context}(W')(r')\}$$

$$\text{context}(W)(r) = \Big(\mathop{\Huge *}_{r_i \mapsto w \in r} r_i \mapsto_r w\Big) \land \text{full\_map}\ r$$
$$* \ \text{na\_inv}\ \gamma_{na} \top$$
$$* \ \text{sts\_full}\ W$$
$$* \ \text{region}\ W$$

The Fundamental Theorem of Logical Relations

The Fundamental Theorem of logical relations

If we can read a region, and every word in that region is safe, then we can safely execute it

- "If we can read a region" : $p = \mathrm{RX} \vee p = \mathrm{RWX} \vee p = \mathrm{RWLX}$
- "and every word in that region is safe": read_write_cond (p, b, e)
- "then we can safely execute it": $\mathcal{E}(W)(((p, g), b, e, a))$

$$(p = \mathrm{RX} \vee p = \mathrm{RWX} \vee p = \mathrm{RWLX}) \implies$$
$$\text{read\_write\_cond } (p, b, e) \implies \mathcal{E}(W)(((p, g), b, e, a))$$

- "If we can read a region" : $p = \mathrm{RX} \vee p = \mathrm{RWX} \vee p = \mathrm{RWLX}$
- "and every word in that region is safe": read_write_cond (p, b, e)
- "then we can safely execute it": $\mathcal{E}(W)(((p, g), b, e, a))$

$$(p = \mathrm{RX} \vee p = \mathrm{RWX} \vee p = \mathrm{RWLX}) \implies$$
$$\text{read\_write\_cond } (p, b, e) \implies \mathcal{E}(W)(((p, g), b, e, a))$$

- "If we can read a region" : $p = \mathrm{RX} \lor p = \mathrm{RWX} \lor p = \mathrm{RWLX}$
- "and every word in that region is safe":
  read_write_cond (p, b, e)
- "then we can safely execute it": $\mathcal{E}(W)(((p, g), b, e, a))$

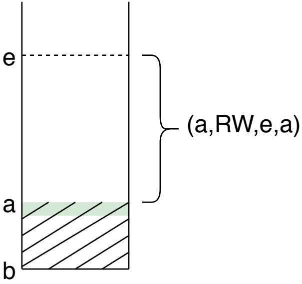
$$(p = \mathrm{RX} \lor p = \mathrm{RWX} \lor p = \mathrm{RWLX}) \implies$$
$$\text{read\_write\_cond}\ (p, b, e) \implies \mathcal{E}(W)(((p, g), b, e, a))$$

- "If we can read a region" : $p = \mathrm{RX} \vee p = \mathrm{RWX} \vee p = \mathrm{RWLX}$
- "and every word in that region is safe": read_write_cond (p, b, e)
- "then we can safely execute it": $\mathcal{E}(W)(((p, g), b, e, a))$

$$(p = \mathrm{RX} \vee p = \mathrm{RWX} \vee p = \mathrm{RWLX}) \implies$$
$$\text{read\_write\_cond } (p, b, e) \implies \mathcal{E}(W)(((p, g), b, e, a))$$

Reasoning about Unknown Code

We use the fundamental theorem to reason about calls to an unknown adversary




```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
halt
```

$$
\begin{aligned}
\mathcal{E}(W)(pc) \triangleq \forall r, \mathcal{R}(W)(r) \ * \ \text{context}(W)(r[\text{PC} := pc]) \\
\mathrel{-\!\!*} \ \mathrm{WP}\ \text{Seq (Instr Executable)} \\
\{v, v = \mathit{HaltedV} \implies \exists W' r', W' \sqsubseteq_{priv} W \\
* \, \text{context}(W')(r')\}
\end{aligned}
$$

Conclusion

- Embed a capability machine into Iris
- Define its program logic
- Mechanize a unary logical relation for an untyped capability machine language
- Prove the fundamental theorem of logical relations
- Reason about examples that rely on Local Stack Encapsulation and Well-Bracketed Control Flow with calls to an unknown adversary

References

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal (2018)
Reasoning About a Machine with Local Capabilities
ESOP *Programming Languages and Systems* 475–501.

Derek Dreyer, Georg Neis, Lars Birkedal (2012)
The impact of higher-order state and control effects on local relational reasoning
Journal of Functional Programming 22(4-5) 477–528.

Derek Dreyer, Amal Ahmed, Lars Birkedal (2011)
Logical Step-Indexed Logical Relations
LMCS 7(2:16).