Implementing a Capability Machine model into Iris

Aïna Linn Georges Alix Trieu Lars Birkedal

Aarhus University

ageorges@cs.au.dk

January 15, 2020

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

Reasoning about a Machine with Local Capabilities: [Skorstensgaard, 2018]

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

Reasoning about a Machine with Local Capabilities: [Skorstensgaard, 2018]

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

Reasoning about a Machine with Local Capabilities: [Skorstensgaard, 2018]

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

Reasoning about a Machine with Local Capabilities:
[Skorstensgaard, 2018]

Introduction

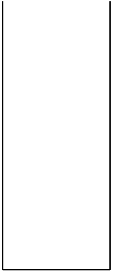
- Capability machines allow for fine grained control over pointer permissions
- Good target for secure compilation
- In particular: we are interested in enforcing certain higher level abstractions such as *local state encapsulation* as *well-bracketed control flow* at the lowest level of the machine
- 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

Reasoning about a Machine with Local Capabilities: [Skorstensgaard, 2018]

Capability Machines

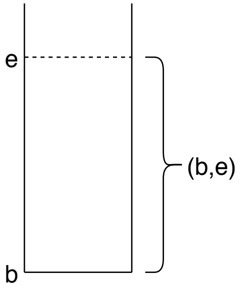
Capability Machine

Capability: An unforgeable token of authority



Capability Machine

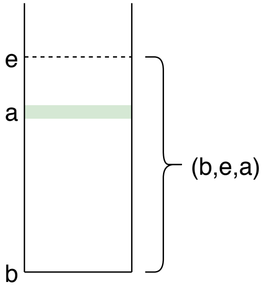
Capability: An unforgeable token of authority



- Range

Capability Machine

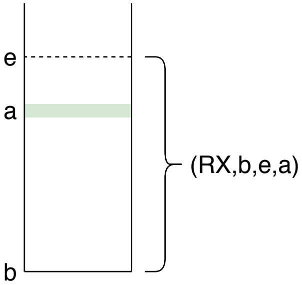
Capability: An unforgeable token of authority

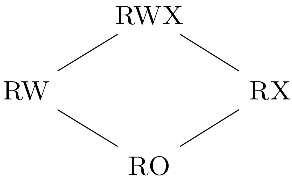


- Address

Capability Machine

Capability: An unforgeable token of authority

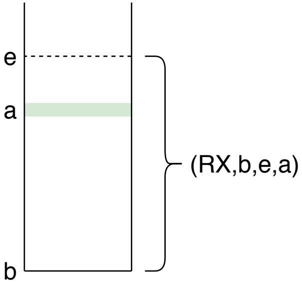


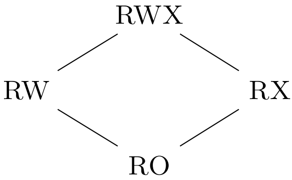


- Permission

Capability Machine

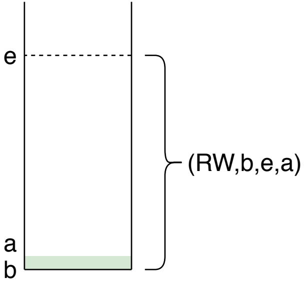
Capability: An unforgeable token of authority





Enforcing Well-Bracketed Control Flow using Capabilities

Well-Bracketed Control Flow

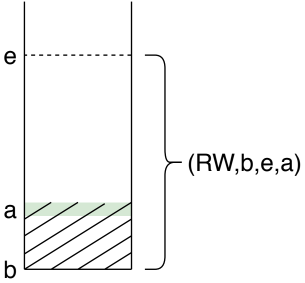



```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

- We start with a stack with range b to e

Capability Safety

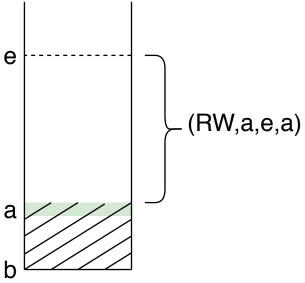
Well-Bracketed Control Flow




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

- Push some local state

Well-Bracketed Control Flow

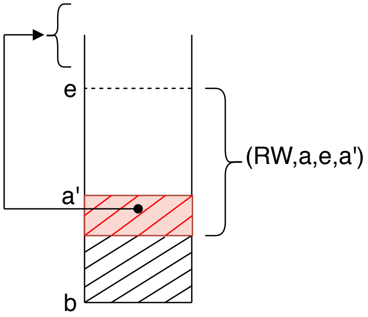



```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

- Prepare adversary stack

Capability Safety

Well-Bracketed Control Flow

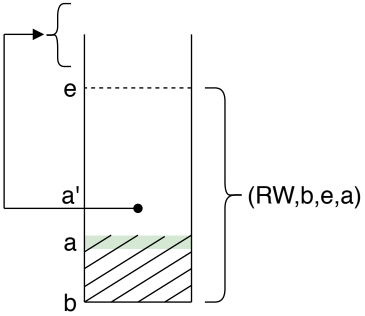



```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
push r_stk 2
scall r
halt
```

- Adversary possesses a return capability

Capability Safety

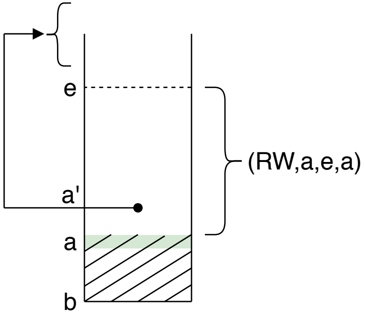
Well-Bracketed Control Flow




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

- Once jumped to we get back original stack - we pop the stack and assert that local state did not change

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
```

- Prepare the adversary stack for second call

Local Capabilities

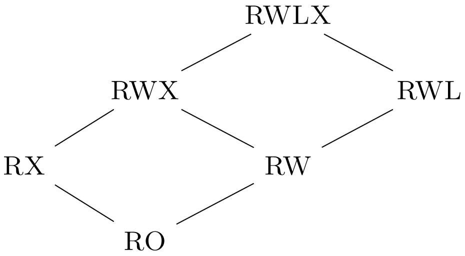
Local Capabilities

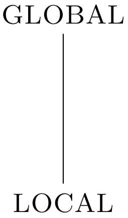
(p,**Local**,b,e,a) (p,**Global**,b,e,a)

- Local capabilities can only be stored where you have write local permission

Local Capabilities

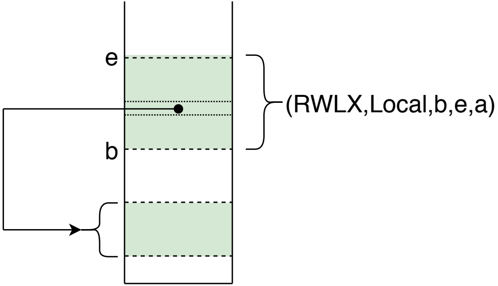
(p,**Local**,b,e,a) (p,**Global**,b,e,a)





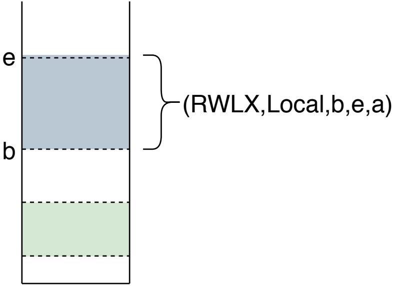
- Local capabilities can only be stored where you have write local permission

Calling Convention



- We want the adversary to lose any temporary capabilities (such as return capabilities) upon return of a function call

Calling Convention



- We want the adversary to lose any temporary capabilities (such as return capabilities) upon return of a function call

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:*

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

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:*

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

Step-indexed Kripke Logical Relation

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

- 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

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

- 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*:

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

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
  - Always and Later Modalities

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
  - Always and Later Modalities

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
  - Always and Later Modalities

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
  - Always and Later Modalities

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
  - Always and Later Modalities

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
  - Always and Later Modalities

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
  - Always and Later Modalities

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
  - Always and Later Modalities

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 - Challenges

- Region invariants: Iris invariants
- Future world relation: frame preserving updates and world satisfaction
- Step indexing: later modality

Challenges

- Iris was designed with more high level languages in mind, how do we embed a low level machine language into Iris
- Iris abstracts away certain details we want to reason about directly
- There is only one frame preserving update, we need to distinguish between two future world relations

Expressing Capability Safety in Iris - Challenges

- Region invariants: Iris invariants
- Future world relation: frame preserving updates and world satisfaction
- Step indexing: later modality

Challenges

- Iris was designed with more high level languages in mind, how do we embed a low level machine language into Iris
- Iris abstracts away certain details we want to reason about directly
- There is only one frame preserving update, we need to distinguish between two future world relations

Expressing Capability Safety in Iris - Challenges

- Region invariants: Iris invariants
- Future world relation: frame preserving updates and world satisfaction
- Step indexing: later modality

Challenges

- Iris was designed with more high level languages in mind, how do we embed a low level machine language into Iris
- Iris abstracts away certain details we want to reason about directly
- There is only one frame preserving update, we need to distinguish between two future world relations

Expressing Capability Safety in Iris - Challenges

- Region invariants: Iris invariants
- Future world relation: frame preserving updates and world satisfaction
- Step indexing: later modality

Challenges

- Iris was designed with more high level languages in mind, how do we embed a low level machine language into Iris
- Iris abstracts away certain details we want to reason about directly
- There is only one frame preserving update, we need to distinguish between two future world relations

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

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

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

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

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

The Value Relation

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

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

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

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

A unary logical relation of an un-typed language

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

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

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

A unary logical relation of an un-typed language

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

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

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

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

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

The Value Relation

A unary logical relation of an un-typed language

$$\mathcal{V} : STS \to \; Word \to iProp\ \Sigma$$

Challenge: distinguish between Local and Global capabilities:

- At the level of the value relation
- Model revocation

STS: A collection of state transition systems

$$\mathcal{V}(\Sigma)((\mathrm{RW}, g), b, e, a) \triangleq \mathop{\huge *}_{a \in [b,e]} \boxed{\exists w, a \mapsto_a [RW]w * \mathcal{V}(\Sigma)(w)}$$

From World to state transition system collection

On paper:

$$\begin{aligned}
\text{Region} = \; & \{\mathit{Revoked}\} \uplus \\
& \{\mathit{Temporary}\} \times \color{red}{\text{State} \times \text{Rels}} \\
& \times (\text{State} \to (\mathit{Wor} \xrightarrow[\sqsupseteq_{pub}]{mon,ne} \text{UPred}(\text{MemSeg}))) \uplus \\
& \{\mathit{Permanent}\} \times \color{red}{\text{State} \times \text{Rels}} \\
& \times (\text{State} \to (\mathit{Wor} \xrightarrow[\sqsupseteq_{priv}]{mon,ne} \text{UPred}(\text{MemSeg}))) \\
\text{World} = \; & \mathbb{N} \rightharpoonup \text{Region}
\end{aligned}$$

In the Iris mechanization, we use a collection of state transition systems:

$$\Sigma : \mathbb{N} \rightharpoonup \mathit{States} \times \mathbb{N} \rightharpoonup \mathit{Rels}$$

The world circularity problem is now handled using Iris invariants and saved predicates.

From World to state transition system collection

On paper:

$$
\begin{aligned}
\text{Region} = \; & \{\mathit{Revoked}\} \uplus \\
& \{\mathit{Temporary}\} \times \text{State} \times \text{Rels} \\
& \times (\text{State} \to (\mathit{Wor} \xrightarrow[\sqsupseteq_{pub}]{mon,ne} \text{UPred}(\text{MemSeg}))) \uplus \\
& \{\mathit{Permanent}\} \times \text{State} \times \text{Rels} \\
& \times (\text{State} \to (\mathit{Wor} \xrightarrow[\sqsupseteq_{priv}]{mon,ne} \text{UPred}(\text{MemSeg}))) \\
\text{World} = \; & \mathbb{N} \rightharpoonup \text{Region}
\end{aligned}
$$

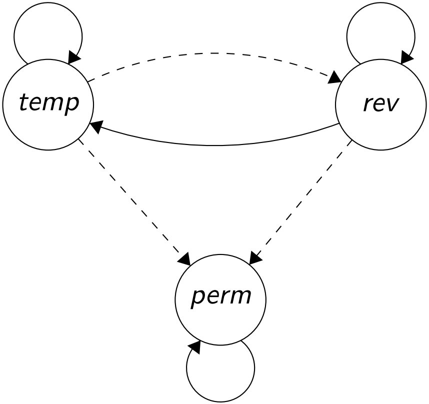
In the Iris mechanization, we use a collection of state transition systems:

$$\Sigma : \mathbb{N} \rightharpoonup \mathit{States} \times \mathbb{N} \rightharpoonup \mathit{Rels}$$

The world circularity problem is now handled using Iris invariants and saved predicates.

Standard STS

$$\Sigma : \mathbb{N} \rightharpoonup \mathit{States} \times \mathbb{N} \rightharpoonup \mathit{Rels}$$



- Dotted lines: private transitions
- Continuous lines: public transitions

What's new

What's new: capability machine viewpoint

- Mechanized formalization: currently $\sim$ 25000 lines of Iris code
- At a higher level of abstraction
  - Step index $\rightarrow$ later modality
  - World $\rightarrow$ collection of state transition systems

What's new: Iris formalization viewpoint

- Formalization of a machine language, with no distinction between program and memory
- Distinction between well-bracketed and non well-bracketed calls: using public/private transitions

What's new

What's new: capability machine viewpoint

- Mechanized formalization: currently $\sim$ 25000 lines of Iris code
- At a higher level of abstraction
  - Step index $\rightarrow$ later modality
  - World $\rightarrow$ collection of state transition systems

What's new: Iris formalization viewpoint

- Formalization of a machine language, with no distinction between program and memory
- Distinction between well-bracketed and non well-bracketed calls: using public/private transitions

What's new

What's new: capability machine viewpoint

- Mechanized formalization: currently $\sim$ 25000 lines of Iris code
- At a higher level of abstraction
  - Step index $\rightarrow$ later modality
  - World $\rightarrow$ collection of state transition systems

What's new: Iris formalization viewpoint

- Formalization of a machine language, with no distinction between program and memory
- Distinction between well-bracketed and non well-bracketed calls: using public/private transitions

Conclusion

Contributions

- First mechanization of a core model of a capability machine. The mechanization includes:
  - Embedding of a capability machine language into Iris (first embedding of a machine language into Iris)
  - Mechanized proof of the fundamental theorem of logical relations
  - Mechanized proof of capability safety of two non-trivial example programs

Future work

- Mechanize a proof of capability safety of the awkward example
- Expand the mechanization with new capabilities and calling conventions that solve current shortcomings of local capabilities

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).

Program Logic

Abstract Instructions

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

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

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

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

- 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

A Capability Points-to Predicate

Points-to Predicate with Permissions

$$a \mapsto_a [RWL]w$$

Points-to Predicate with Permissions

$$a \mapsto_a [RWL]w \Rrightarrow a \mapsto_a [RWL]((p, Local), b, e, l)$$

Points-to Predicate with Permissions

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

Points-to Predicate with Permissions

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

Proving Hoare Triples

Successful Execution

Hoare Triples of the Program Logic: Success

$$
\begin{aligned}
& decode(w) = \text{Load dst src} \\
& \wedge \ \text{isCorrectPC}\ ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) \\
& \wedge \ \text{readAllowed}\ p_{src} \ \wedge \ \text{withinBounds}\ (b_{src}, e_{src}, a_{src})
\end{aligned}
$$

$$
\begin{aligned}
\{\{\{ & \ \text{PC} \mapsto_r ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) * a_{pc} \mapsto_a [p_{pc}] w \\
& * dst \mapsto_r w_{dst} * src \mapsto_r ((p_{src}, g_{src}), b_{src}, e_{src}, a_{src}) \\
& * a_{src} \mapsto_a [p_{src}] w_{src} \}\}\} \\
& \text{Instr Executable} \\
\{\{\{ & \ \text{PC} \mapsto_r ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc} + 1) * a_{pc} \mapsto_a [p_{pc}] w \\
& * dst \mapsto_r w_{src} * src \mapsto_r ((p_{src}, g_{src}), b_{src}, e_{src}, a_{src}) \\
& * a_{src} \mapsto_a [p_{src}] w_{src} \}\}\}
\end{aligned}
$$

Hoare Triples of the Program Logic: Success

$$
\begin{aligned}
& \mathit{decode}(w) = \text{Load dst src} \\
\wedge\ & \text{isCorrectPC } ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) \\
\wedge\ & \text{readAllowed } p_{src} \ \wedge\ \text{withinBounds } (b_{src}, e_{src}, a_{src})
\end{aligned}
$$

$$
\begin{aligned}
\{\{\{\ & \text{PC} \mapsto_r ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) * a_{pc} \mapsto_a [p_{pc}]w \\
& * \mathit{dst} \mapsto_r w_{dst} * \mathit{src} \mapsto_r ((p_{src}, g_{src}), b_{src}, e_{src}, a_{src}) \\
& * a_{src} \mapsto_a [p_{src}]w_{src}\}\}\} \\
& \text{Instr Executable} \\
\{\{\{\ & \text{PC} \mapsto_r ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc} + 1) * a_{pc} \mapsto_a [p_{pc}]w \\
& * \mathit{dst} \mapsto_r w_{src} * \mathit{src} \mapsto_r ((p_{src}, g_{src}), b_{src}, e_{src}, a_{src}) \\
& * a_{src} \mapsto_a [p_{src}]w_{src}\}\}\}
\end{aligned}
$$

Hoare Triples of the Program Logic: Success

$$
\begin{aligned}
& decode(w) = \text{Load dst src} \\
\wedge\ & \text{isCorrectPC}\ ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) \\
\wedge\ & \text{readAllowed}\ p_{src}\ \wedge\ \text{withinBounds}\ (b_{src}, e_{src}, a_{src})
\end{aligned}
$$

$$
\begin{aligned}
\{\{\{\ & \mathrm{PC} \mapsto_r ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) * a_{pc} \mapsto_a [p_{pc}]w \\
& * dst \mapsto_r w_{dst} * src \mapsto_r ((p_{src}, g_{src}), b_{src}, e_{src}, a_{src}) \\
& * a_{src} \mapsto_a [p_{src}]w_{src} \}\}\} \\
& \text{Instr Executable} \\
\{\{\{\ & \mathrm{PC} \mapsto_r ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc} + 1) * a_{pc} \mapsto_a [p_{pc}]w \\
& * dst \mapsto_r w_{src} * src \mapsto_r ((p_{src}, g_{src}), b_{src}, e_{src}, a_{src}) \\
& * a_{src} \mapsto_a [p_{src}]w_{src} \}\}\}
\end{aligned}
$$

Failed Execution

Hoare Triples of the Program Logic: Failure

$$
\begin{aligned}
& \quad decode(w) = \text{Load dst src} \\
& \wedge \ \text{isCorrectPC} \ ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) \\
& \wedge \ \neg\text{readAllowed} \ p_{src} \ \vee \ \neg\text{withinBounds} \ (b_{src}, e_{src}, a_{src}) \\
& \{\{\{ \ \text{PC} \mapsto_r ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) * a_{pc} \mapsto_a [p_{pc}]w \\
& \quad * \ src \mapsto_r ((p_{src}, g_{src}), b_{src}, e_{src}, a_{src})\}\}\} \\
& \quad \text{Instr Executable} \\
& \{\{\{ \ \text{FailedV}, \top\}\}\}
\end{aligned}
$$

Hoare Triples of the Program Logic: Failure

$$
\begin{array}{l}
\quad decode(w) = \text{Load dst src} \\
\wedge \text{ isCorrectPC } ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) \\
\wedge \neg\text{readAllowed } p_{src} \;\vee\; \neg\text{withinBounds } (b_{src}, e_{src}, a_{src}) \\
\{\{\{ \text{ PC} \mapsto_r ((p_{pc}, g_{pc}), b_{pc}, e_{pc}, a_{pc}) * a_{pc} \mapsto_a [p_{pc}]w \\
\quad * \, src \mapsto_r ((p_{src}, g_{src}), b_{src}, e_{src}, a_{src})\}\}\} \\
\quad \text{Instr Executable} \\
\{\{\{ \text{ FailedV}, \top\}\}\}
\end{array}
$$

The Execute Condition

The Execute Condition

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

The Expression Relation

The Expression Relation

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

The Expression Relation

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

$$\text{context}(\Sigma)(r) = \;?$$

The Expression Relation

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

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

The Expression Relation

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

$$\text{context}(\Sigma)(r) = \Big(\mathop{\Huge *}_{r_i \mapsto w \in r} r_i \mapsto_r w \Big) \wedge \mathsf{full\_map}\ r$$
$$* \ \mathsf{na\_inv}\ \gamma_{na} \top$$

The Expression Relation

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

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

The Expression Relation

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

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

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": $\mathcal{E}(\Sigma)(((p, g), b, e, a))$

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

- "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": $\mathcal{E}(\Sigma)(((p, g), b, e, a))$

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

- "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": $\mathcal{E}(\Sigma)(((p, g), b, e, a))$

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

- "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": $\mathcal{E}(\Sigma)(((p, g), b, e, a))$

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

Proving the Fundamental Theorem

$$ \tag{1} $$

$$\mathop{\ast}_{a\in[b,e]} \text{read\_write\_cond } a \tag{2}$$

$$\ast\, \mathcal{R}(r) \tag{3}$$

- -

$$\mathop{\ast}_{reg\hookrightarrow w\in r} reg \mapsto_r w \tag{4}$$

$$\mathrm{WP}\ \text{Seq (Instr Executable)}\ \{v, v = HaltedV \implies \exists \Sigma' r', \Sigma' \sqsupseteq_{priv} \Sigma * \text{context}(\Sigma')(r')\}$$

Proving the Fundamental Theorem

$$ \qquad (1) $$

$$ \mathop{\ast}_{a\in[b,e]} \text{read\_write\_cond } a \qquad (2) $$

$$ \ast\, \mathcal{R}(r) \qquad (3) $$

- -

$$ \mathop{\ast}_{reg \hookrightarrow w \in r \setminus \mathrm{PC}} reg \mapsto_r w \qquad (4) $$

$$ \ast\, \mathrm{PC} \mapsto_r (pc_g, pc_p, pc_b, pc_e, pc_a) \qquad (5) $$

$$ \mathrm{WP} \text{ Seq (Instr Executable) } \{v, v = HaltedV \implies \exists \Sigma' r', \Sigma' \sqsupseteq_{priv} \Sigma \ast \text{context}(\Sigma')(r')\} $$

Proving the Fundamental Theorem

$$ \tag{1} $$

$$\mathop{\Large *}_{a \in [b,e]} \text{read\_write\_cond } a \tag{2}$$

$$\ast\, \mathcal{R}(r) \tag{3}$$

- -

$$\mathop{\Large *}_{reg \hookrightarrow w \in r \setminus \mathrm{PC}} reg \mapsto_r w \tag{4}$$

$$\ast\, \mathrm{PC} \mapsto_r (pc_g, pc_p, pc_b, pc_e, pc_a) \tag{5}$$

$$\ast\, pc_a \mapsto_a [pc_p]w \tag{6}$$

$$\mathrm{WP}\ \text{Seq (Instr Executable)}\ \{v, v = HaltedV \implies$$
$$\exists \Sigma' r', \Sigma' \sqsupseteq_{priv} \Sigma * \text{context}(\Sigma')(r')\}$$

Proving the Fundamental Theorem

$$decode(w) = \text{Load } dst\ src \tag{1}$$

$$\ast\ \mathcal{R}(r) \tag{2}$$

- -

$$\mathop{\ast}_{reg \hookrightarrow w \in r \setminus \mathrm{PC}} reg \mapsto_r w \tag{3}$$

$$\ast\ \mathrm{PC} \mapsto_r (pc_g, pc_p, pc_b, pc_e, pc_a) \tag{4}$$

$$\ast\ pc_a \mapsto_a [pc_p]w \tag{5}$$

$$\mathrm{WP}\ \text{Seq (Instr Executable)}\ \{v, v = HaltedV \implies \exists \Sigma' r', \Sigma' \sqsupseteq_{priv} \Sigma * \text{context}(\Sigma')(r')\}$$

Proving the Fundamental Theorem

$$decode(w) = \text{Load } dst\ src \qquad (1)$$
$$\ast\ \mathcal{V}(p, g, b, e, a) \qquad (2)$$

- -

$$\ast\ dst \mapsto_r w_{dst} \qquad (3)$$
$$\ast\ src \mapsto_r (p, g, b, e, a) \qquad (4)$$
$$\ast\ \mathrm{PC} \mapsto_r (pc_g, pc_p, pc_b, pc_e, pc_a) \qquad (5)$$
$$\ast\ pc_a \mapsto_a [pc_p]w \qquad (6)$$

$$\mathrm{WP}\ \text{Seq (Instr Executable)}\ \{v, v = HaltedV \implies \exists \Sigma' r', \Sigma' \sqsupseteq_{priv} \Sigma \ast \text{context}(\Sigma')(r')\}$$

Proving the Fundamental Theorem

$$decode(w) = \text{Load } dst\ src \quad (1)$$
$$\ast\ \mathcal{V}(p, g, b, e, a) \quad (2)$$
$$\ast\ \mathcal{V}(w_{src}) \quad (3)$$

- -

$$\ast\ dst \mapsto_r w_{dst} \quad (4)$$
$$\ast\ src \mapsto_r (p, g, b, e, a) \quad (5)$$
$$\ast\ \mathrm{PC} \mapsto_r (pc_g, pc_p, pc_b, pc_e, pc_a) \quad (6)$$
$$\ast\ pc_a \mapsto_a [pc_p]w \quad (7)$$
$$\ast\ a \mapsto_a [p]w_{src} \quad (8)$$

$$\mathrm{WP}\ \text{Seq (Instr Executable)}\ \{v, v = HaltedV \implies \exists \Sigma' r', \Sigma' \sqsupseteq_{priv} \Sigma \ast \text{context}(\Sigma')(r')\}$$

Proving the Fundamental Theorem

$$
\begin{array}{ll}
decode(w) = \text{Load } dst\ src & (1)\\
\ast\ \mathcal{V}(p, g, b, e, a) & (2)\\
\ast\ \mathcal{V}(w_{src}) & (3)
\end{array}
$$

- -

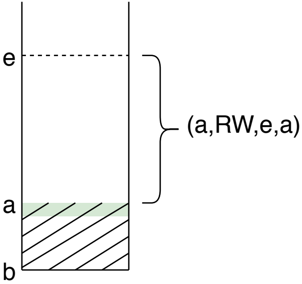
$$
\begin{array}{ll}
\ast\ dst \mapsto_r {\color{red} w_{src}} & (4)\\
\ast\ src \mapsto_r (p, g, b, e, a) & (5)\\
\ast\ \mathrm{PC} \mapsto_r (pc_g, pc_p, pc_b, pc_e, pc_a) & (6)\\
\ast\ pc_a \mapsto_a [pc_p]w & (7)\\
\ast\ a \mapsto_a [p]w_{src} & (8)
\end{array}
$$

$$
\mathrm{WP}\ \text{Seq (Instr Executable)}\ \{v, v = HaltedV \implies \exists \Sigma' r', \Sigma' \sqsupseteq_{priv} \Sigma * \text{context}(\Sigma')(r')\}
$$

Reasoning about Unknown Code

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
```

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