Implementing a Capability Machine model into Iris

Aïna Linn Georges Alix Trieu Lars Birkedal

Aarhus University

ageorges@cs.au.dk

October 27, 2019

Introduction

High-level Programming Language

Assembly

- Local state encapsulation
- Well bracketed control flow

Introduction

High-level Programming Language

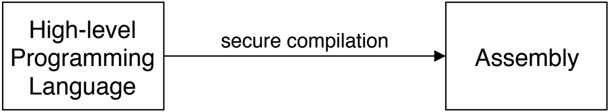
Assembly

- Local state encapsulation
- Well bracketed control flow

Introduction

- Local state encapsulation
- Well bracketed control flow

Introduction



- Local state encapsulation
- Well bracketed control flow

Layers of Abstraction

Programming Languages

- Local State Encapsulation
- Well Bracketed Control Flow

Assembly Language

- Programs lie in Memory, Program Counter, ...
- Arbitrary Pointer Manipulation
- Arbitrary Jumps

Machine Code

- Instruction Decoding, Cache, etc.

...

Layers of Abstraction

Programming Languages

- Local State Encapsulation
- Well Bracketed Control Flow

Assembly Language

- Programs lie in Memory, Program Counter, ...
- Arbitrary Pointer Manipulation
- Arbitrary Jumps

Machine Code

- Instruction Decoding, Cache, etc.

...

Layers of Abstraction

Programming Languages

- Local State Encapsulation
- Well Bracketed Control Flow

Assembly Language

- Programs lie in Memory, Program Counter, ...
- Arbitrary Pointer Manipulation
- Arbitrary Jumps

Machine Code

- Instruction Decoding, Cache, etc.

...

Layers of Abstraction

Programming Languages

- Local State Encapsulation
- Well Bracketed Control Flow

Assembly Language

- Programs lie in Memory, Program Counter, ...
- Arbitrary Pointer Manipulation
- Arbitrary Jumps

Machine Code

- Instruction Decoding, Cache, etc.

...

Layers of Abstraction

Programming Languages

- Local State Encapsulation
- Well Bracketed Control Flow

Assembly Language

- Programs lie in Memory, Program Counter, ...
- Arbitrary Pointer Manipulation
- Arbitrary Jumps

Machine Code

- Instruction Decoding, Cache, etc.

...

Layers of Abstraction

Programming Languages

- Local State Encapsulation
- Well Bracketed Control Flow

Assembly Language

- Programs lie in Memory, Program Counter, ...
- Arbitrary Pointer Manipulation
- Arbitrary Jumps

Machine Code

- Instruction Decoding, Cache, etc.

...

Layers of Abstraction

Programming Languages

- Local State Encapsulation
- Well Bracketed Control Flow

Assembly Language

- Programs lie in Memory, Program Counter, ...
- Arbitrary Pointer Manipulation
- Arbitrary Jumps

Machine Code

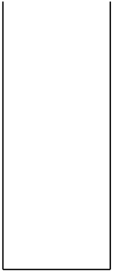
- Instruction Decoding, Cache, etc.

...

Capability Machines

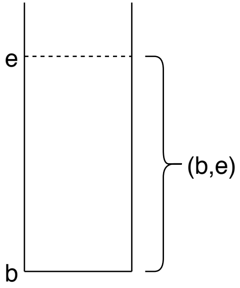
Capability Machine

Capability: An unforgeable token of authority



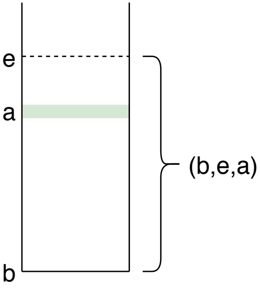
Capability Machine

Capability: An unforgeable token of authority

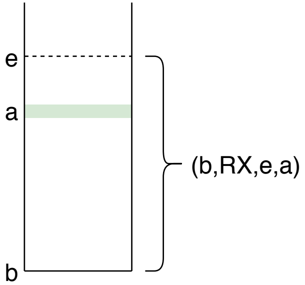


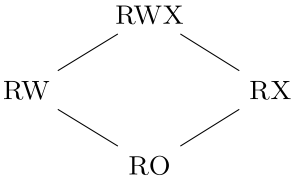
Capability Machine

Capability: An unforgeable token of authority

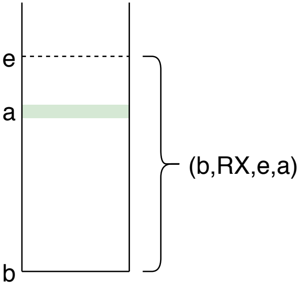


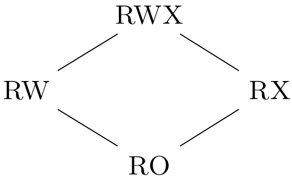
Capability: An unforgeable token of authority





Capability: An unforgeable token of authority

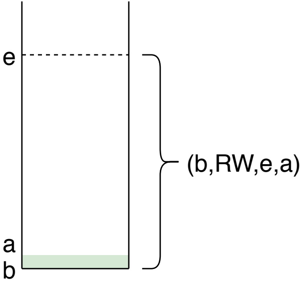




Enforcing Local Stack Encapsulation using Capabilities

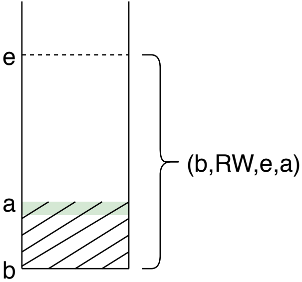
Capability Safety

Local State Encapsulation



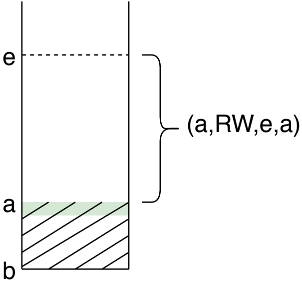

```
push r_stk 1
scall r
pop r_stk r_1
assert r_1 1
halt
```

Local State Encapsulation




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

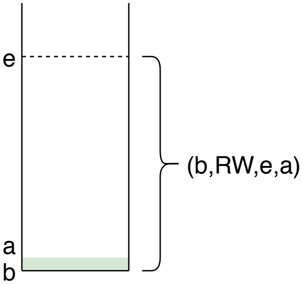
Local State Encapsulation




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

Enforcing Well Bracketed Control Flow using Capabilities

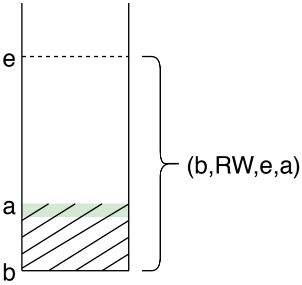
Well Bracketed Control Flow




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

Capability Safety

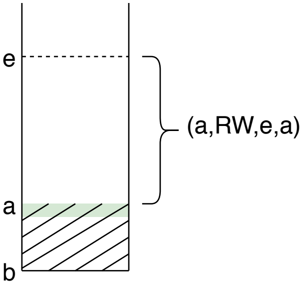
Well Bracketed Control Flow




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

Capability Safety

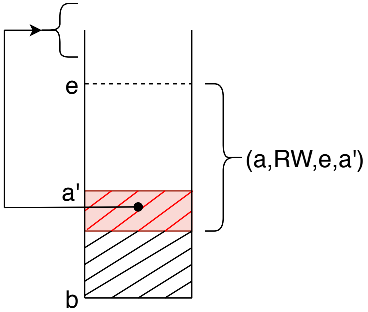
Well Bracketed Control Flow




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

Capability Safety

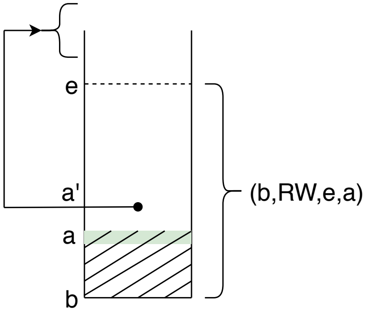
Well Bracketed Control Flow




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

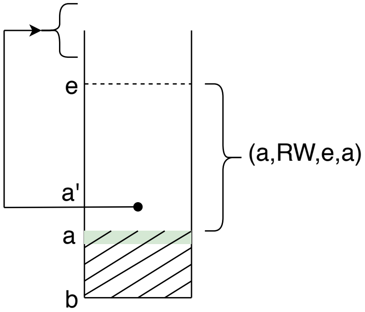
Capability Safety

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
```

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)

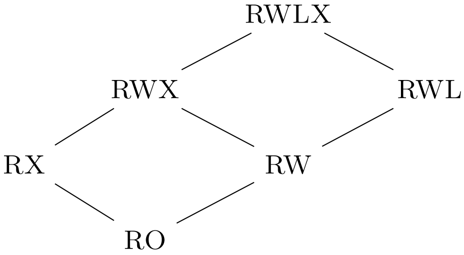
well-bracketed

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

not well-bracketed

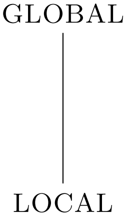
Local Capabilities

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



well-bracketed

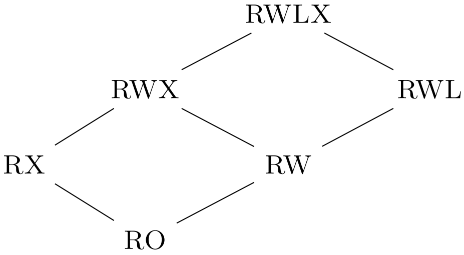
(p,**Global**,b,e,a)



not well-bracketed

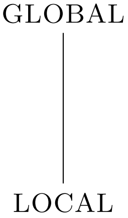
Local Capabilities

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



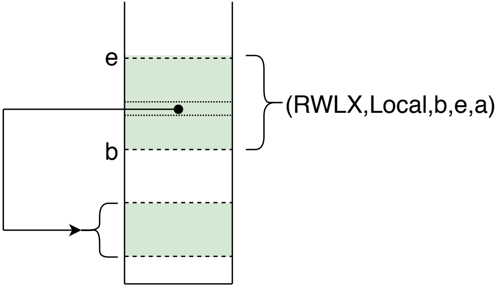
well-bracketed

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



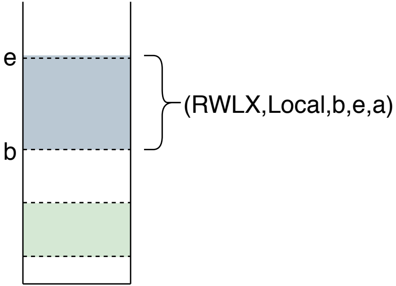
not well-bracketed

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**

Expressing Capability Safety

- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**
    1. embed the language into Iris
    2. define a program logic by proving Hoare Triples
    3. define the logical relation
    4. prove the fundamental theorem of logical relations
    5. 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

- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**
  1. embed the language into Iris
  2. define a program logic by proving Hoare Triples
  3. define the logical relation
  4. prove the fundamental theorem of logical relations
  5. 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

- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**
  1. embed the language into Iris
  2. define a program logic by proving Hoare Triples
  3. define the logical relation
  4. prove the fundamental theorem of logical relations
  5. 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

- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**
  1. embed the language into Iris
  2. define a program logic by proving Hoare Triples
  3. define the logical relation
  4. prove the fundamental theorem of logical relations
  5. 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

- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**
  1. embed the language into Iris
  2. define a program logic by proving Hoare Triples
  3. define the logical relation
  4. prove the fundamental theorem of logical relations
  5. 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

- **using a logical relation on an untyped (or uni-typed) language to capture semantic properties of the language**
  1. embed the language into Iris
  2. define a program logic by proving Hoare Triples
  3. define the logical relation
  4. prove the fundamental theorem of logical relations
  5. 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

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

- 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

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

- 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

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

Points-to Predicate with Permissions

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

Proving Hoare Triples

Successful Execution

Hoare Triples of the Program Logic: Success

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

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

Hoare Triples of the Program Logic: Success

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

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

Hoare Triples of the Program Logic: Success

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

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

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

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

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

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

- World: A collection of state transition systems to reason about *local state*

$$\mathcal{V}(W)(z) \triangleq \exists z' \in \mathbb{Z}.z = z'$$

$$\mathcal{V}(W)(((\textsc{ro}, g), b, e, a)) \triangleq \text{read\_write\_cond(RO,b,e)}$$

$$\mathcal{V}(W)(((\textsc{rx}, g), b, e, a)) \triangleq \text{read\_write\_cond(RX,b,e)}$$

$$* \ \square\ \text{exec\_cond(W)(RX,g,b,e)}$$

A unary logical relation of an un-typed language

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

- World: A collection of state transition systems to reason about *local state*

$$\mathcal{V}(W)(z) \triangleq \exists z' \in \mathbb{Z}. z = z'$$

$$\mathcal{V}(W)(((\text{RO}, g), b, e, a)) \triangleq \text{read\_write\_cond(RO,b,e)}$$

$$\mathcal{V}(W)(((\text{RX}, g), b, e, a)) \triangleq \text{read\_write\_cond(RX,b,e)}$$

$$* \ \square \ \text{exec\_cond(W)(RX,g,b,e)}$$

The Value Relation

A unary logical relation of an un-typed language

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

- World: A collection of state transition systems to reason about *local state*

$$\mathcal{V}(W)(z) \triangleq \exists z' \in \mathbb{Z}. z = z'$$
$$\mathcal{V}(W)(((\text{RO}, g), b, e, a)) \triangleq \text{read\_write\_cond(RO,b,e)}$$
$$\mathcal{V}(W)(((\text{RX}, g), b, e, a)) \triangleq \text{read\_write\_cond(RX,b,e)}$$
$$\quad * \ \square \ \text{exec\_cond(W)(RX,g,b,e)}$$

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

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

The Expression Relation

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

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

The Expression Relation

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

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

The Expression Relation

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

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

The Expression Relation

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

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

The Expression Relation

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

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

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

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

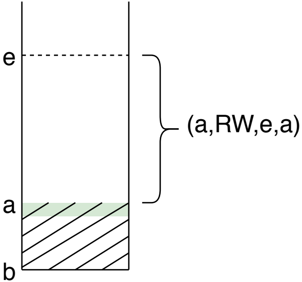
- "If we can read a region" : $p = \mathrm{RX} \lor p = \mathrm{RWX} \lor p = \mathrm{RWLX}$
- "and every word in that region is safe": read_write_cond (p, b, e)
- "then we can safely execute it": $\mathcal{E}(W)(((p, g), b, e, a))$

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

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
```

Conclusion

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

John Smith (2012)
Title of the publication
Journal Name 12(3), 45 – 678.