Mechanized Program Verification on a Capability Machine in Presence of Untrusted Code

Aïna Linn Georges [1] Armaël Guéneau [1] Thomas Van Strydonck [2] Amin Timany [1] Alix Trieu [1] Sander Huyghebaert [3] Dominique Devriese [3] Lars Birkedal [1]

[1] Aarhus University

[2] KU Leuven

[3] Vrije Universiteit Brussel

November 30, 2020

Overview

Iris Primer

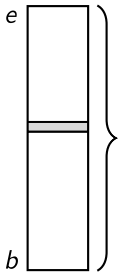
Iris is a concurrent separation logic framework.

- Separation logic constructs: $*, -\!\!*, \{P\}\ e\ \{Q\}$
- Two modalities: $\triangleright, \square$
- Invariants: $\boxed{\mathrm{P}}$
- Resource algebras: $\mapsto, \cdots$

Capability Machines

Machine Capabilities

An unforgeable token of authority



(p, b, e, a)

- Permission: ro, rx, rw, rwx, e, o
- Bounds of authority: $[b, e)$
- Current address: a

- Load and Store instructions dynamically check that the address is within bounds, and that the permission permits the action
- If all conditions are met, the instruction succeeds and executes
- Otherwise, the instruction **fails**

Capability Machine Language

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

$$([\mathrm{pc} := (\mathrm{rx}, b, e, a)]reg, [a := i]mem) \rightarrow ([\mathrm{pc} := (\mathrm{rx}, b, e, a{+}1)]reg', mem')$$

$$\begin{array}{lll}
\rho & \in & \mathbb{Z} + \mathrm{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ \rho \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
 & & \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \\
 & & \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \texttt{getp}\ r\ r \mid \\
 & & \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}$$

A Simple Example

Passing a ro Capability to Unknown Code

$$\begin{aligned} \mathit{prog} = {} & \text{let } x = \text{alloc } 1 \text{ in} \\ & \text{let } y = \text{restrict } x \text{ ro in} \\ & \text{unknown\_code}(y); \\ & \text{assert } (!x = 1); \\ & \text{halt}() \end{aligned}$$

$$\mathit{if} \quad ([\mathrm{pc} := (\mathrm{rx}, a_1, a_n, a_1)]\mathit{reg}, [a_1 - a_n := \mathit{prog}]\mathit{mem}) \longrightarrow^* (\mathit{reg}', \mathit{mem}')$$
$$\mathit{then} \quad \mathit{mem}'(a_{\mathit{flag}}) = 0$$

The assertion will depend on two properties:

1. capability pattern property of ro
2. local state encapsulation

Passing a ro Capability to Unknown Code

$$\begin{aligned} \mathit{prog} = {} & \text{let } x = \text{alloc } 1 \text{ in} \\ & \text{let } y = \text{restrict } x \text{ ro in} \\ & \text{unknown\_code}(y); \\ & \text{assert } (!x = 1); \\ & \text{halt}() \end{aligned}$$

$$\begin{aligned} \mathit{if} \;\; & ([\mathrm{pc} := (\mathrm{rx}, a_1, a_n, a_1)]\mathit{reg}, [a_1 - a_n := \mathit{prog}]\mathit{mem}) \longrightarrow^* (\mathit{reg}', \mathit{mem}') \\ \mathit{then} \;\; & \mathit{mem}'(a_{\mathit{flag}}) = 0 \end{aligned}$$

The assertion will depend on two properties:

1. capability pattern property of ro
2. local state encapsulation

Proving Such a Specification

The outline of our methodology:

1. A program logic to describe such a spec, and step through the known part of the code
    - Adequacy of the logic
2. A logical relation defining a notion of capability safety for reasoning about the execution of unknown code
    - Fundamental theorem of logical relations

All in the Iris framework.

A Program Logic for Capability Machines

Resources

Registers	$r \Mapsto w$
Memory	$a \mapsto w$

What should a Hoare triple look like?

$$decode(w) = instr \rightarrow$$

$$\{\ \mathrm{pc} \Mapsto (p, b, e, a) * a \mapsto w * \cdots\ \}$$
$$\ldots$$
$$\{\ \cdots\ \}$$

An Atomic Instrucion

Executing a single instruction

$$\begin{array}{l} \mathit{decode}(w) = \mathit{instr} \to \\ \mathrm{ValidPC}(p, b, e, a) \to \end{array}$$

$$\{\ \mathsf{pc} \Mapsto (p, b, e, a) * a \mapsto w * \cdots\ \}$$

SingleStep

$$\{\ \mathsf{pc} \Mapsto (p, b, e, a + 1) * a \mapsto w * \cdots\ \}$$

A Full Program

Executing a sequence of instructions

$$\text{map decode } l = prog \to$$
$$\text{ValidPCRange}(p, b, e, -)(a_1, a_n) \to$$

$$\{ \text{ pc} \Mapsto (p, b, e, a_1) * [a_1 - a_n] \mapsto l * \cdots \}$$
Repeat SingleStep
$$\{ \text{ pc} \Mapsto (p, b, e, a_n) * [a_1 - a_n] \mapsto l * \cdots \}$$

Repeat SingleStep $\to$ Repeat Done Standby $\to \cdots$
$\to$ Repeat Done Halted $\to$ Done Halted

A Full Program

Executing a sequence of instructions

$$\text{map decode } l = prog \rightarrow$$
$$\text{ValidPCRange}(p, b, e, -)(a_1, a_n) \rightarrow$$

$$\{\ \mathsf{pc} \Mapsto (p, b, e, a_1) * [a_1 - a_n] \mapsto l * \cdots\ \}$$
Repeat SingleStep
$$\{\ \mathsf{pc} \Mapsto (p, b, e, a_n) * [a_1 - a_n] \mapsto l * \cdots\ \}$$

Repeat SingleStep $\rightarrow$ Repeat Done Standby $\rightarrow \cdots$
$\rightarrow$ Repeat Done Halted $\rightarrow$ Done Halted

A Program Fragment (or Macro)

Executing a macro, or a sequence of instructions within a program

$$\text{map decode } l = \mathit{prog} \rightarrow$$
$$\text{ValidPCRange}(p, b, e, -)(a_1, a_n) \rightarrow$$

$$\{\ \mathsf{pc} \Mapsto (p, b, e, a_1) * [a_1 - a_n] \mapsto l\ \ * \cdots *$$
$$\triangleright (\mathsf{pc} \Mapsto (p, b, e, a_n) * [a_1 - a_n] \mapsto l * \cdots \twoheadrightarrow \Phi)\ \}$$

Repeat SingleStep

$$\{\ \Phi\ \}$$

Some Syntactic Sugar

A single instruction:

$\langle w_0; P\rangle \rightarrow \langle w_1; Q\rangle \quad \triangleq \quad \mathrm{pc} \Mapsto w_0 * P \mathrel{-\!\!*} \textbf{wp}\ \mathrm{SingleStep}\ \{\mathrm{pc} \Mapsto w_1 * Q\}$

A full program:

$\{w; P\} \rightsquigarrow \bullet \quad \triangleq \quad \mathrm{pc} \Mapsto w * P \mathrel{-\!\!*} \textbf{wp}\ \mathrm{Repeat\ SingleStep}\ \{\mathrm{True}\}$

A program fragment:

$\{w_0; P\} \rightsquigarrow \{w_1; Q\} \quad \triangleq \quad \{w_0; P * \{w_1; Q\} \rightsquigarrow \bullet\} \rightsquigarrow \bullet$

The Logical Relation

Goal

We want to reason about unknown code. We need to define what it means for arbitrary code to *behave well*.

A capability machine program *behaves well* if it does not violate capability safety.

The logical relation defines a contract that well behaved capability machine programs must follow. We use this contract as the interface between known secure code, and unknown arbitrary code, when reasoning about the full program.

The Logical Relation

- Expression relation
  - The execution does not get stuck: validity of the registers is sufficient for executing the program
  - All declared invariants hold at every step of execution
- Value relation

$$\boxed{\mathcal{E}(w)} \quad \triangleq \forall reg, \left\{ w; \mathop{\ast}_{(r,v)\in reg, r\neq \mathrm{pc}} r \Mapsto v \ast \mathcal{V}(v) \right\} \rightsquigarrow \bullet$$

$$\boxed{\mathcal{V}(w)} \quad \begin{cases} \mathcal{V}(z) & \triangleq \text{True} \\ \mathcal{V}(\mathrm{e}, b, e, a) & \triangleq \triangleright \square \mathcal{E}(\mathrm{rx}, b, e, a) \\ \mathcal{V}(\mathrm{ro}/\mathrm{rx}, b, e, -) & \triangleq \mathop{\ast}_{a\in[b,e[} \exists P, \boxed{\exists w,\ a \mapsto w \ast P(w)} \ast \\ & \qquad \triangleright \square \forall w,\ P(w) \twoheadrightarrow \mathcal{V}(w) \\ \mathcal{V}(\mathrm{rw}/\mathrm{rwx}, b, e, -) & \triangleq \mathop{\ast}_{a\in[b,e[} \boxed{\exists w,\ a \mapsto w \ast \mathcal{V}(w)} \end{cases}$$

The Fundamental Theorem of the Logical Relation

$$\boxed{\mathcal{E}(w)} \quad \triangleq \forall reg,\ \left\{w;\ \mathop{\ast}_{(r,v)\in reg, r\neq \mathrm{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

$$\boxed{\mathcal{V}(w)} \quad \begin{cases} \mathcal{V}(z) & \triangleq \text{True} \\ \mathcal{V}(\mathrm{e}, b, e, a) & \triangleq \triangleright \Box\, \mathcal{E}(\mathrm{rx}, b, e, a) \\ \mathcal{V}(\mathrm{ro}/\mathrm{rx}, b, e, -) & \triangleq \mathop{\ast}_{a\in[b,e[} \exists P, \boxed{\exists w,\ a \mapsto w * P(w)} * \\ & \qquad\qquad \triangleright \Box \forall w,\ P(w) \twoheadrightarrow \mathcal{V}(w) \\ \mathcal{V}(\mathrm{rw}/\mathrm{rwx}, b, e, -) & \triangleq \mathop{\ast}_{a\in[b,e[} \boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)} \end{cases}$$

Theorem (FTLR)

Let $p \in \mathrm{Perm}$, $b, e, a \in \mathrm{Addr}$. If $\mathcal{V}(p, b, e, a)$, then $\mathcal{E}(p, b, e, a)$.

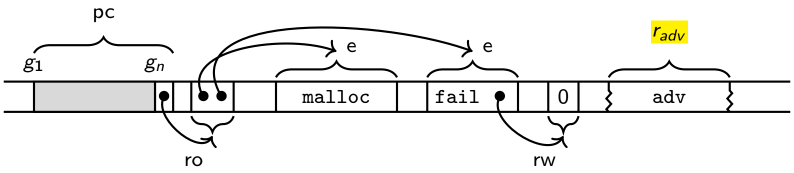
Proving an Example Specification

The Assembly Code of our Example Program

```
g1: malloc $f_m$ 1
    move $r_{env}$ $r_1$
    move $r_7$ $r_1$
    store $r_{env}$ 1
    restrict $r_7$ ($encodePerm$(RO))
    call $f_m$ ⋯ $r_{adv}$ [$r_{env}$] [$r_7$] ;; where ⋯ is the offset to
    restore_locals $r_2$ [$r_{env}$]
    load r_t0 $r_{env}$
    assert $f_a$ $r_0$ 1
    halt
gn:
```

1. Define a memory layout for the specification of g
2. Define and prove specifications for the macros: `malloc`, `call`, `restore_locals`, `assert`

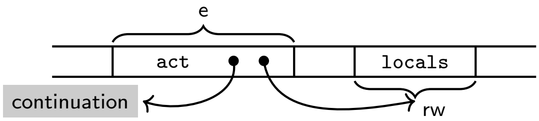
Memory Layout for the Specification of g



- Specifications for `malloc`, `call`, `restore_locals` and `assert` will assert locally compatible layouts
- For instance: the specification for `malloc` asserts full ownership over the addresses it may hand out

A Heap Based Calling Convention

1. Dynamically allocates heap space to store the activation record
2. Dynamically allocates heap space to store the current local state
3. Sets up the continuation, and stores the continuation and the capability to the local state into the activation record
4. Clears all registers except parameters and an E capability pointing to the activation record
5. Jumps to the call destination



The calling convention enforces local state encapsulation only if `malloc` is correct!

The Full Specification

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (p, g_1, g_n, g_1); \begin{array}{l} \bigast_{(r,v)\in reg, r \notin \{\mathrm{pc}, r_{adv}\}} r \Mapsto v \\ * \; r_{adv} \Mapsto w_{adv} \\ * \; \mathcal{V}(w_{adv}) \\ * \; [g_1, g_n) \mapsto g_{instrs} \end{array} \right\} \rightsquigarrow \bullet$$

By adequacy of weakest preconditions, if we can prove the above specification, we can prove that a_{flag} points to 0 at every step of execution. In other words, the assertion will not fail.

The Full Specification

$\rightarrow$ $\mathtt{g}_1$: $\underline{\mathtt{malloc}}$ f_m $\mathbf{1}$
    move r_{env} r_1
    move r_7 r_1
    store r_{env} 1
    restrict r_7 $(encodePerm(\text{RO}))$
    $\underline{\mathtt{call}}$ f_m $\cdots$ r_{adv} [r_{env}] [r_7] ;; where $\cdots$ is the offset to next instruction
    $\underline{\mathtt{restore\_locals}}$ r_2 [r_{env}]
    load r_t0 r_{env}
    $\underline{\mathtt{assert}}$ f_a r_0 $\mathbf{1}$
    halt
$\mathtt{g}_n$:

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (p, \mathtt{g}_1, \mathtt{g}_n, \mathtt{g}_1); \begin{array}{l} \bigast_{(r,v)\in reg, r\notin\{\text{pc}, r_{adv}\}} r \Mapsto v \\ * \ r_{adv} \Mapsto w_{adv} \\ * \ \mathcal{V}(w_{adv}) \\ * \ [\mathtt{g}_1, \mathtt{g}_n) \mapsto g_{instrs} \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g1 :  malloc  f_m  1
      move  r_env  r_1
      move  r_7  r_1
      store  r_env  1
      restrict  r_7  (encodePerm(RO))
      call  f_m  ···  r_adv  [r_env]  [r_7]  ;; where ··· is the offset to next instruction
      restore_locals  r_2  [r_env]
      load  r_t0  r_env
      assert  f_a  r_0  1
      halt
gn :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (p, g_1, g_n, g_2); \begin{array}{l} \bigast_{(r,v)\in reg, r\notin\{pc, r_{adv}, r_1\}} r \Mapsto v \\ * \; r_{adv} \Mapsto w_{adv} \\ * \; \mathcal{V}(w_{adv}) \\ * \; [g_1, g_n) \mapsto g_{instrs} \\ * \; r_1, \Mapsto (\text{rwx}, b, b+1, b) \\ * \; b \mapsto 0 \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g_1 :  malloc  $f_m$  1
       move  $r_{env}$  $r_1$
       move  $r_7$  $r_1$
       store  $r_{env}$  1
       restrict  $r_7$  ($encodePerm(\text{RO})$)
       call  $f_m$  ···  $r_{adv}$  [$r_{env}$]  [$r_7$]  ;; where ··· is the offset to next instruction
       restore_locals  $r_2$  [$r_{env}$]
       load  r_t0  $r_{env}$
       assert  $f_a$  $r_0$  1
       halt
g_n :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (p, g_1, g_n, g_3); \begin{array}{l} \bigast_{(r,v)\in reg, r \notin \{\mathrm{pc}, r_{adv}, r_1, r_{env}\}} r \Mapsto v \\ * \; r_{adv} \Mapsto w_{adv} \\ * \; \mathcal{V}(w_{adv}) \\ * \; [g_1, g_n) \mapsto g_{instrs} \\ * \; r_1, r_{env} \Mapsto (\mathrm{rwx}, b, b+1, b) \\ * \; b \mapsto 0 \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g₁ : malloc $f_m$ 1
     move $r_{env}$ $r_1$
     move $r_7$ $r_1$
     store $r_{env}$ 1
     restore $r_7$ ($encodePerm(\text{RO})$)
     call $f_m$ ⋯ $r_{adv}$ [$r_{env}$] [$r_7$] ;; where ⋯ is the offset to next instruction
     restore_locals $r_2$ [$r_{env}$]
     load r_t0 $r_{env}$
     assert $f_a$ $r_0$ 1
     halt
gₙ :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (p, \mathrm{g}_1, \mathrm{g}_n, \mathrm{g}_4); \begin{array}{l} \ast_{(r,v) \in reg, r \notin \{\mathrm{pc}, r_{adv}, r_1, r_{env}, r_7\}} \; r \Mapsto v \\ \ast \; r_{adv} \Mapsto w_{adv} \\ \ast \; \mathcal{V}(w_{adv}) \\ \ast \; [\mathrm{g}_1, \mathrm{g}_n) \mapsto g_{instrs} \\ \ast \; r_1, r_{env} \Mapsto (\mathrm{rwx}, b, b+1, b) \\ \ast \; b \mapsto 0 \\ \ast \; r_7 \Mapsto (\mathrm{rwx}, b, b+1, b) \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g1 : malloc f_m 1
     move r_env r1
     move r7 r1
     store r_env 1
     restrict r7 (encodePerm(RO))
     call f_m ··· r_adv [r_env] [r7] ;; where ··· is the offset to next instruction
     restore_locals r2 [r_env]
     load r_t0 r_env
     assert f_a r0 1
     halt
gn :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (p, g_1, g_n, g_5); \begin{array}{l} \mathop{\ast}_{(r,v)\in reg, r\notin\{pc, r_{adv}, r_1, r_{env}, r_7\}} r \Mapsto v \\ \ast\ r_{adv} \Mapsto w_{adv} \\ \ast\ \mathcal{V}(w_{adv}) \\ \ast\ [g_1, g_n) \mapsto g_{instrs} \\ \ast\ r_1, r_{env} \Mapsto (\mathrm{rwx}, b, b+1, b) \\ \ast\ b \mapsto 1 \\ \ast\ r_7 \Mapsto (\mathrm{rwx}, b, b+1, b) \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g1 : malloc f_m 1
     move r_env r_1
     move r_7 r_1
     store r_env 1
     restrict r_7 (encodePerm(RO))
     call f_m ··· r_adv [r_env] [r_7]  ;; where ··· is the offset to next instruction
     restore_locals r_2 [r_env]
     load r_t0 r_env
     assert f_a r_0 1
     halt
gn :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (p, g_1, g_n, g_6); \begin{array}{l} \bigstar_{(r,v)\in reg, r\notin\{pc, r_{adv}, r_1, r_{env}, r_7\}} r \Mapsto v \\ * \; r_{adv} \Mapsto w_{adv} \\ * \; \mathcal{V}(w_{adv}) \\ * \; [g_1, g_n) \mapsto g_{instrs} \\ * \; r_1, r_{env} \Mapsto (\mathrm{rwx}, b, b+1, b) \\ * \; b \mapsto 1 \\ * \; r_7 \Mapsto (\mathrm{ro}, b, b+1, b) \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g_1 :  malloc $f_m$ 1
       move $r_{env}$ $r_1$
       move $r_7$ $r_1$
       store $r_{env}$ 1
       restrict $r_7$ ($encodePerm$(RO))
       call $f_m$ ··· $r_{adv}$ [$r_{env}$] [$r_7$] ;; where ··· is the offset to next instruction
  →    restore_locals $r_2$ [$r_{env}$]
       load r_t0 $r_{env}$
       assert $f_a$ $r_0$ 1
       halt
g_n :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}, \boxed{a[g_1, g_n) \mapsto g_{instrs}}$$

$$\boxed{act * a_l \mapsto (\text{rwx}, b, b+1, b)}$$

$$\vdash \left\{ w_{adv}; \begin{array}{l} \ast_{(r,v)\in reg, r\notin\{\text{pc}, r_7, r_0\}} r \Mapsto 0 \\ * \ \mathcal{V}(w_{adv}) \\ * \ b \mapsto 1 \\ * \ r_7 \Mapsto (\text{ro}, b, b+1, b) \\ * \ r_0 \Mapsto (\text{e}, act_1, act_m, act_1) \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g₁ : malloc f_m 1
     move r_env r_1
     move r_7 r_1
     store r_env 1
     restrict r_7 (encodePerm(RO))
     call f_m ··· r_adv [r_env] [r_7] ;; where ··· is the offset to next instruction
→    restore_locals r_2 [r_env]
     load r_t0 r_env
     assert f_a r_0 1
     halt
g_n :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}, \boxed{a[g_1, g_n) \mapsto g_{instrs}}$$

$$\boxed{act * a_l \mapsto (\mathrm{rwx}, b, b+1, b)}, \boxed{\exists w, b \mapsto w * \lceil w = 1 \rceil}$$

$$\vdash \left\{ w_{adv};\ \begin{array}{l} \bigstar_{(r,v)\in reg, r \notin \{\mathrm{pc}, r_7, r_0\}} r \Mapsto 0 \\ * \ \mathcal{V}(w_{adv}) \\ * \ r_7 \Mapsto (\mathrm{ro}, b, b+1, b) \\ * \ r_0 \Mapsto (\mathrm{e}, act_1, act_m, act_1) \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g1 : malloc f_m 1
     move r_env r1
     move r7 r1
     store r_env 1
     restrict r7 (encodePerm(RO))
     call f_m ··· r_adv [r_env] [r7] ;; where ··· is the offset to next instruction
     restore_locals r2 [r_env]
     load r_t0 r_env
     assert f_a r0 1
     halt
gn :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}, \boxed{a[g_1, g_n) \mapsto g_{instrs}}$$

$$\boxed{act * a_l \mapsto (\text{rwx}, b, b+1, b)}, \boxed{\exists w, b \mapsto w * \lceil w = 1 \rceil}$$

$$\vdash \left\{ w_{adv};\ \begin{array}{l} \bigstar_{(r,v)\in reg, r \notin \{\text{pc}, r_7, r_0\}}\ r \Mapsto 0 \\ * \ \mathcal{V}(w_{adv}) \\ * \ r_7 \Mapsto (\text{ro}, b, b+1, b) \\ * \ r_0 \Mapsto (\text{e}, act_1, act_m, act_1) \\ * \ \mathcal{E}(w_{adv}) \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g1 : malloc f_m 1
     move r_env r_1
     move r_7 r_1
     store r_env 1
     restrict r_7 (encodePerm(RO))
     call f_m ··· r_adv [r_env] [r_7] ;; where ··· is the offset to next instruction
→    restore_locals r_2 [r_env]
     load r_t0 r_env
     assert f_a r_0 1
     halt
gn :
```

$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}, \boxed{a[g_1, g_n) \mapsto g_{instrs}}$

$\boxed{act * a_l \mapsto (\text{rwx}, b, b+1, b)}, \boxed{\exists w, b \mapsto w * \lceil w = 1 \rceil}$

$$\vdash \left\{ w_{adv}; \begin{array}{l} \mathop{\ast}_{(r,v)\in reg, r \notin \{\text{pc}, r_7, r_0\}} r \Mapsto 0 \\ * \ \mathcal{V}(w_{adv}) \\ * \ r_7 \Mapsto (\text{ro}, b, b+1, b) \\ * \ r_0 \Mapsto (\text{e}, act_1, act_m, act_1) \\ * \ \forall reg, \left\{ w_{adv}; \mathop{\ast}_{(r,v)\in reg, r \neq \text{pc}} r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g1 : malloc f_m 1
     move r_env r_1
     move r_7 r_1
     store r_env 1
     restrict r_7 (encodePerm(RO))
     call f_m ··· r_adv [r_env] [r_7] ;; where ··· is the offset to next instruction
     restore_locals r_2 [r_env]
→    load r_t0 r_env
     assert f_a r_0 1
     halt
gn :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}, \boxed{a[g_1, g_n) \mapsto g_{instrs}}$$

$$\boxed{act * a_l \mapsto (\text{rwx}, b, b+1, b)}, \boxed{\exists w, b \mapsto w * \lceil w = 1 \rceil}$$

$$\vdash \left\{ (p, g_1, g_n, g_8); \begin{array}{l} \mathop{\Large *}_{(r,v)\in reg', r\notin\{\text{pc}, r_{env}\}} r \Mapsto v \\ * \ \mathcal{V}(w_{adv}) \\ * \ r_{env} \Mapsto (\text{rwx}, b, b+1, b) \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g1 : malloc f_m 1
     move r_env r_1
     move r_7 r_1
     store r_env 1
     restrict r_7 (encodePerm(RO))
     call f_m ··· r_adv [r_env] [r_7] ;; where ··· is the offset to next instruction
     restore_locals r_2 [r_env]
     load r_t0 r_env
→    assert f_a r_0 1
     halt
g_n :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}, \boxed{a[g_1, g_n) \mapsto g_{instrs}}$$

$$\boxed{act * a_l \mapsto (\mathrm{rwx}, b, b+1, b)}, \boxed{\exists w, b \mapsto w * \lceil w = 1 \rceil}$$

$$\vdash \left\{ (p, g_1, g_n, g_9);\ \begin{array}{l} \bigstar_{(r,v)\in reg', r \notin \{\mathrm{pc}, r_{env}, r_0\}}\ r \Mapsto v \\ * \ \mathcal{V}(w_{adv}) \\ * \ r_{env} \Mapsto (\mathrm{rwx}, b, b+1, b) \\ * \ r_0 \Mapsto w * \lceil w = 1 \rceil \end{array} \right\} \rightsquigarrow \bullet$$

The Full Specification

```
g1 : malloc fm 1
     move renv r1
     move r7 r1
     store renv 1
     restrict r7 (encodePerm(RO))
     call fm ··· radv [renv] [r7] ;; where ··· is the offset to next instruction
     restore_locals r2 [renv]
     load r_t0 renv
     assert fa r0 1
  →  halt
gn :
```

$$\boxed{inv_{malloc}}, \boxed{inv_{envTable}}, \boxed{a_{flag} \mapsto 0}, \boxed{a[g_1, g_n) \mapsto g_{instrs}}$$

$$\boxed{act * a_l \mapsto (\mathrm{rwx}, b, b+1, b)}, \boxed{\exists w, b \mapsto w * \lceil w = 1 \rceil}$$

$$\vdash \left\{ (p, g_1, g_n, g_{10});\ \begin{array}{l} \mathop{\huge *}_{(r,v)\in reg', r\notin\{\mathrm{pc}, r_{env}, r_0\}} r \Mapsto v \\ * \ \mathcal{V}(w_{adv}) \\ * \ r_{env} \Mapsto (\mathrm{rwx}, b, b+1, b) \\ * \ r_0 \Mapsto w * \lceil w = 1 \rceil \end{array} \right\} \rightsquigarrow \bullet$$

Key Takeaways from the Proof

- We relied on the local state encapsulation of (rwx, b, $b+1$, b)
- We used the definition of $\mathcal{V}$ for ro capabilities to allocate an invariant which let us successfully step through the assertion
- We used the FTLR to step through the unknown code pointed to by w_{adv}
- Last step; apply adequacy of weakest precondition to finally show the lemma we started out with
    - Mostly straightforward, with the exception of establishing $\mathcal{V}(w_{adv})$! We do this by restricting the arbitrary words w_{adv} point to to be *integers only* (a bit like only considering a closed program)

Final Remarks

What did I not cover in this presentation? A stack! [1]

- well bracketed control flow
- A more sophisticated machine, and calling convention

Robust and Compositional Verification of Object Capability Patterns (David Swasey, Deepak Garg, Derek Dreyer) [2]

[1] A. L. Georges, A. Guéneau, T. Van Strydonck, A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, and L. Birkedal, "Efficient and Provable Local Capability Revocation using Uninitialized Capabilities," in *POPL*, 2021. [Online]. Available: https://iris-project.org/pdfs/2021-popl-ucaps-final.pdf.

[2] D. Swasey, D. Garg, and D. Dreyer, "Robust and Compositional Verification of Object Capability Patterns," in *OOPSLA*, ACM, 2017. [Online]. Available: https://people.mpi-sws.org/~swasey/papers/ocpl/ocpl-20170418.pdf (visited on 09/21/2017).