Cap' ou pas cap' ?

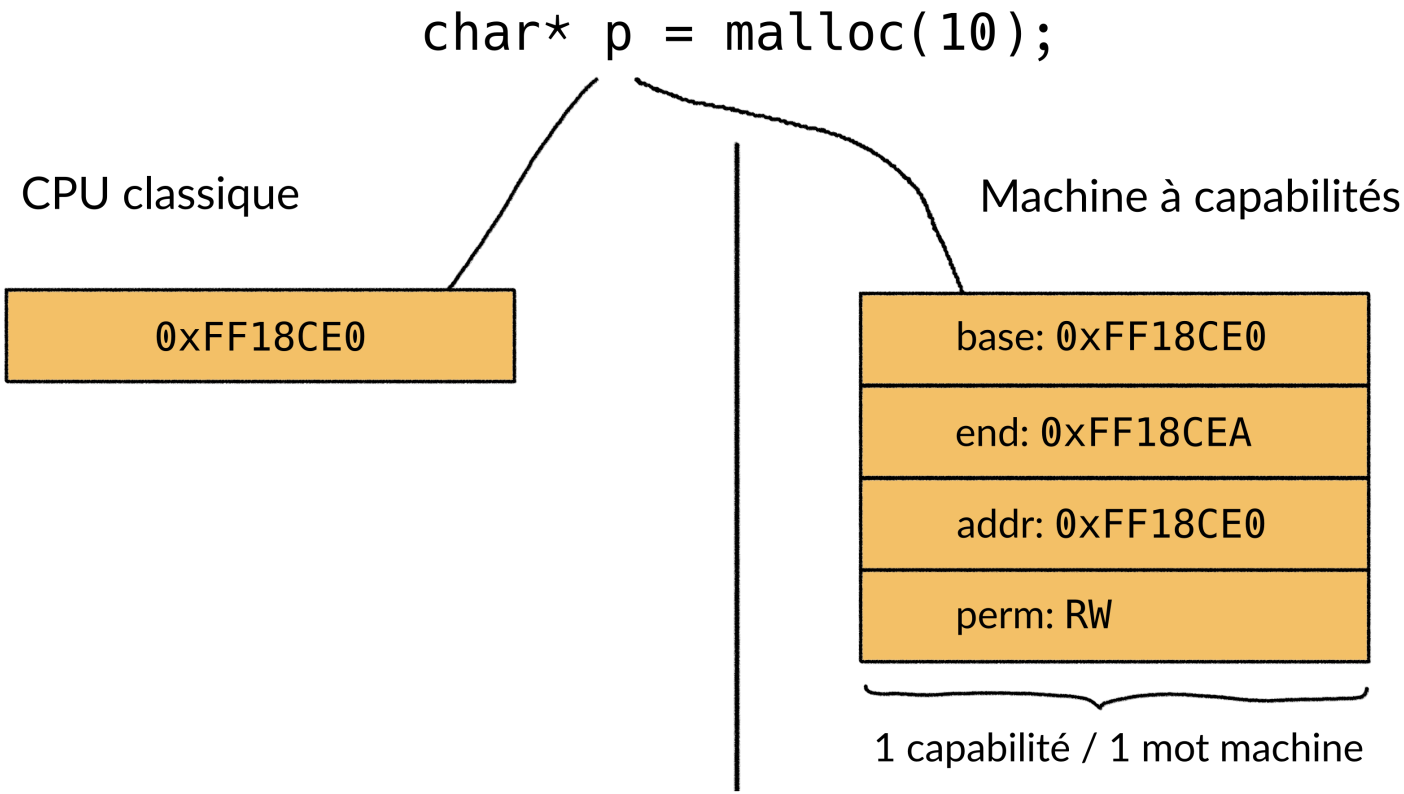
Preuve de programmes pour une machine à capabilités en présence de code inconnu

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

[1]Aarhus University, Danemark [2]KU Leuven, Belgique [3]Vrije Universiteit Brussel, Belgique

Les capabilités : des primitives de sûreté flexibles

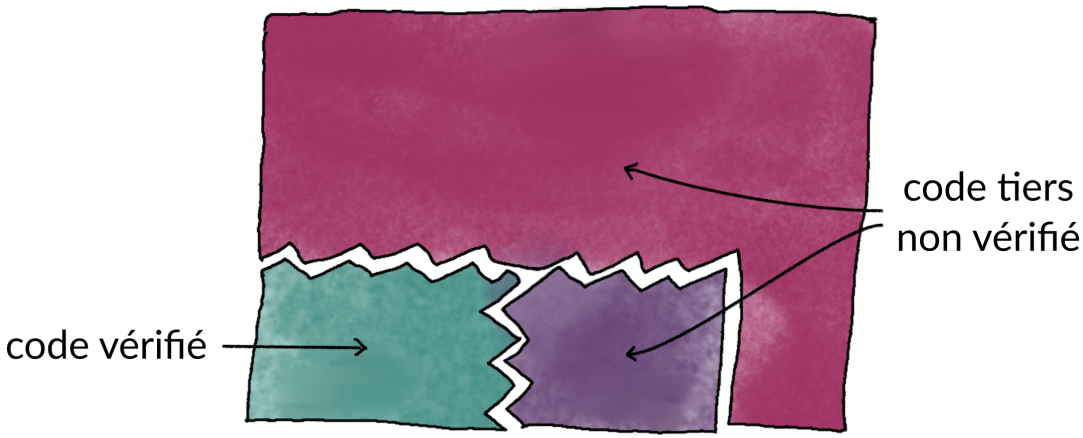
Capabilité = pointeur + métadonnées



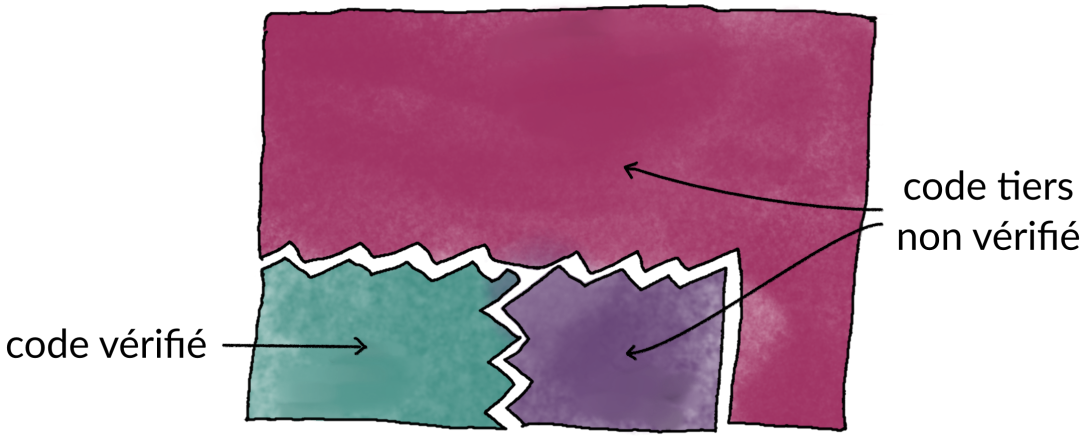
Les bornes et permissions sont **vérifiées par la machine**.

(pour en savoir plus sur les machines à capabilités CHERI : `cheri-cpu.org`)

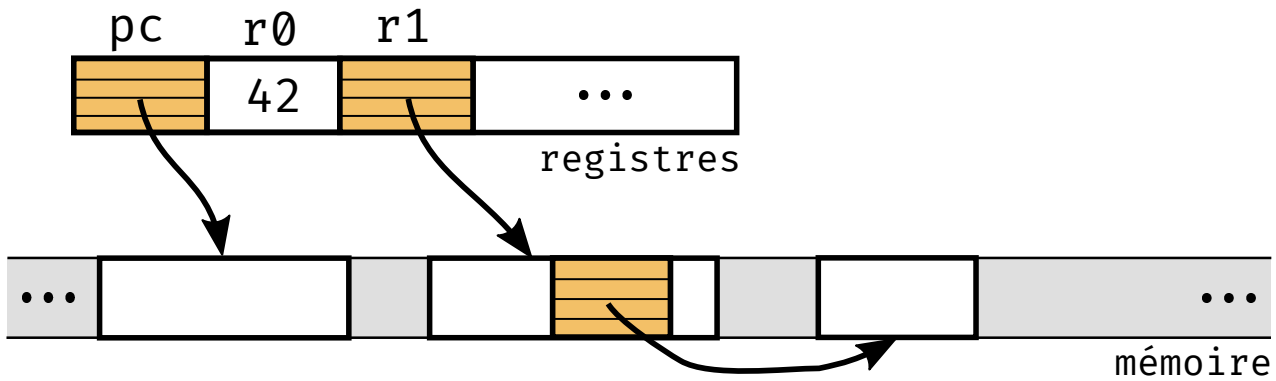
Sécuriser les interactions avec du code non vérifié



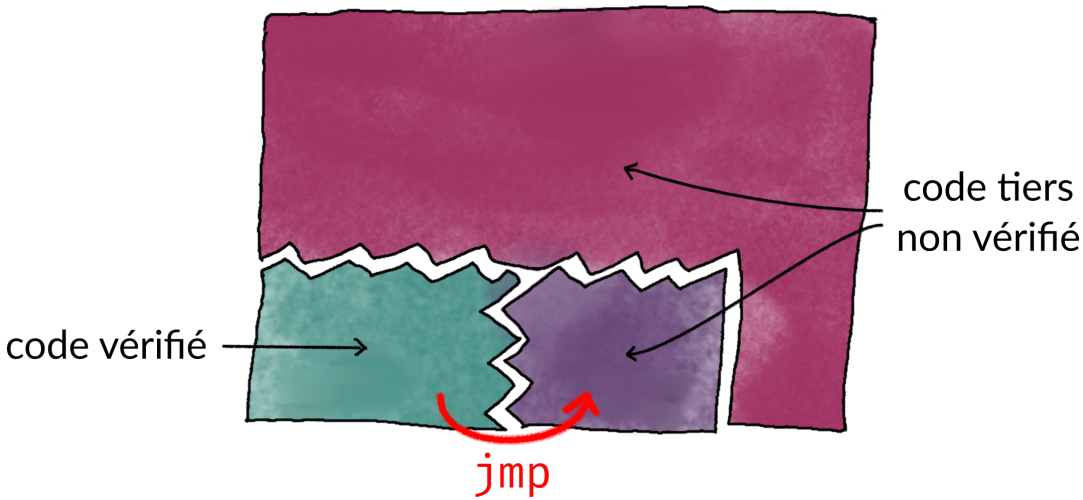
Sécuriser les interactions avec du code non vérifié



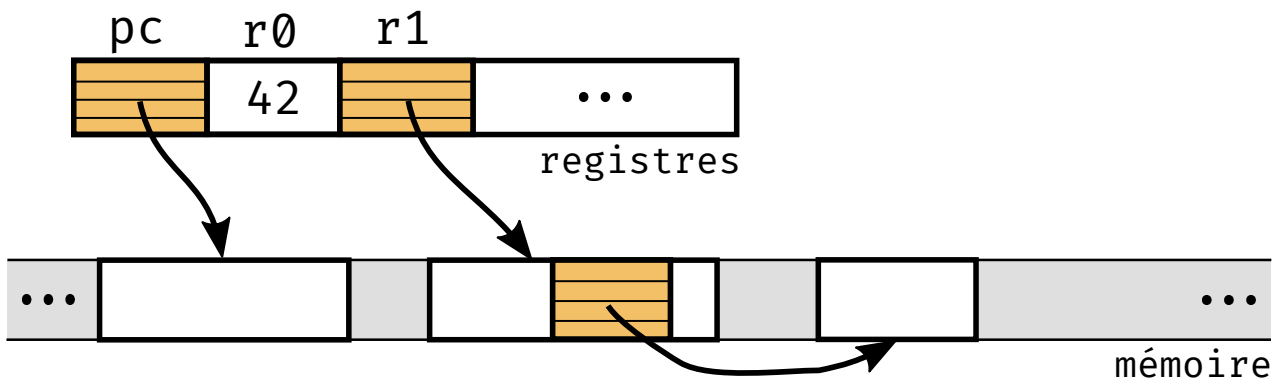
Les capabilités sont le seul moyen d'accéder à la mémoire :



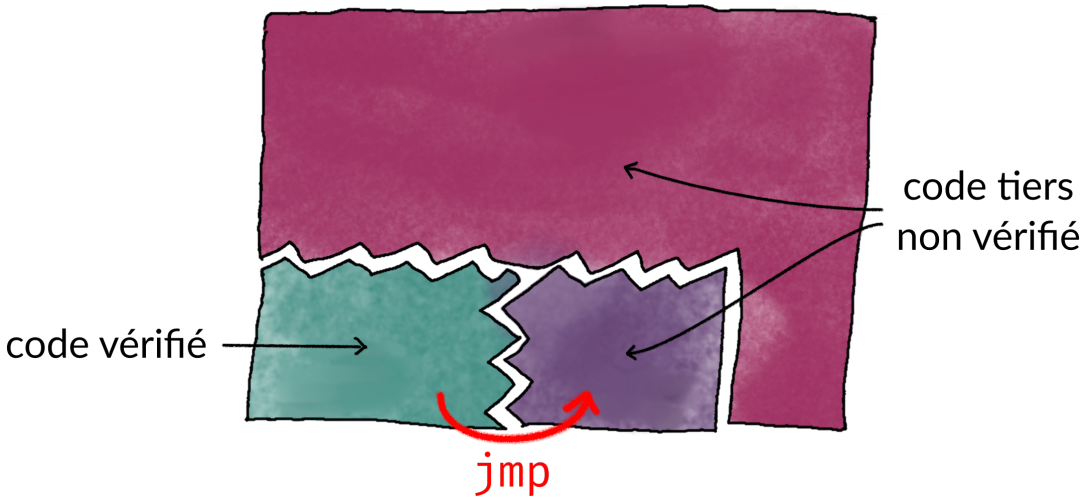
Sécuriser les interactions avec du code non vérifié



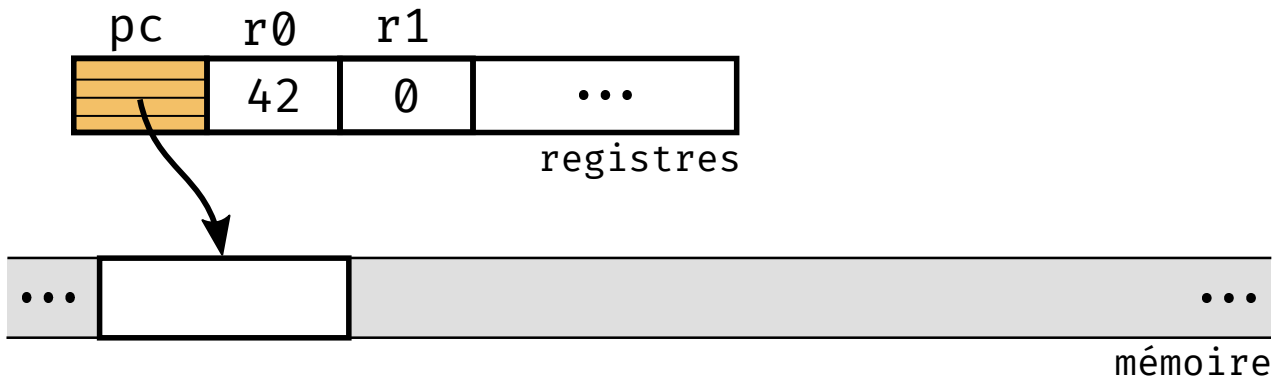
Les capabilités sont le seul moyen d'accéder à la mémoire :



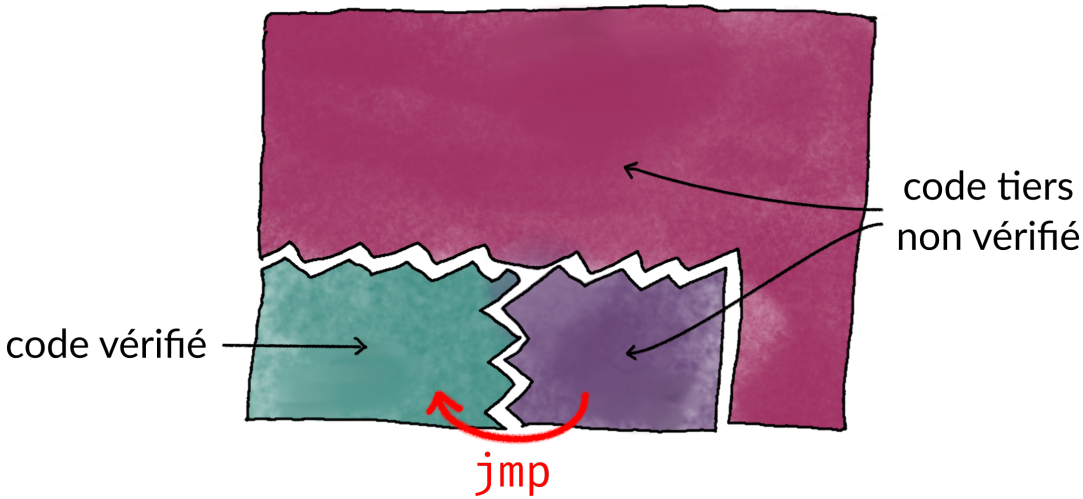
Sécuriser les interactions avec du code non vérifié



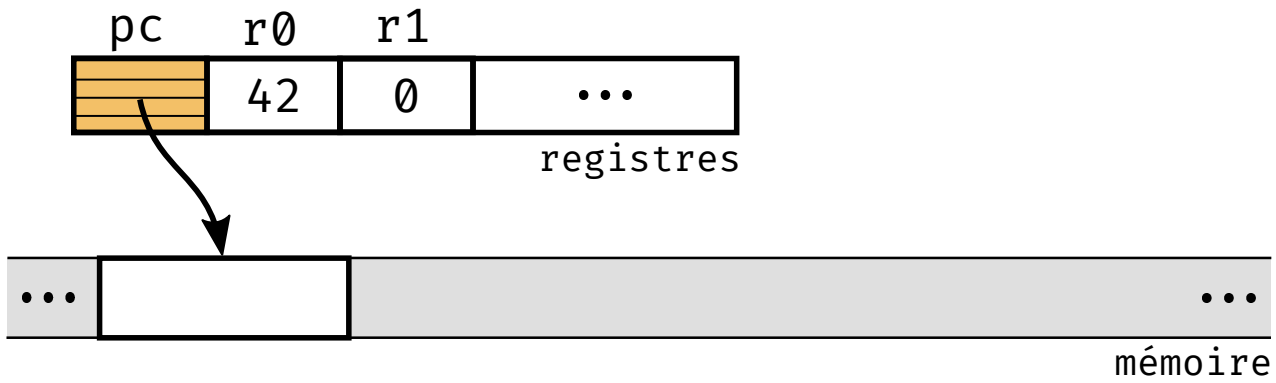
Les capabilités sont le seul moyen d'accéder à la mémoire :



Sécuriser les interactions avec du code non vérifié

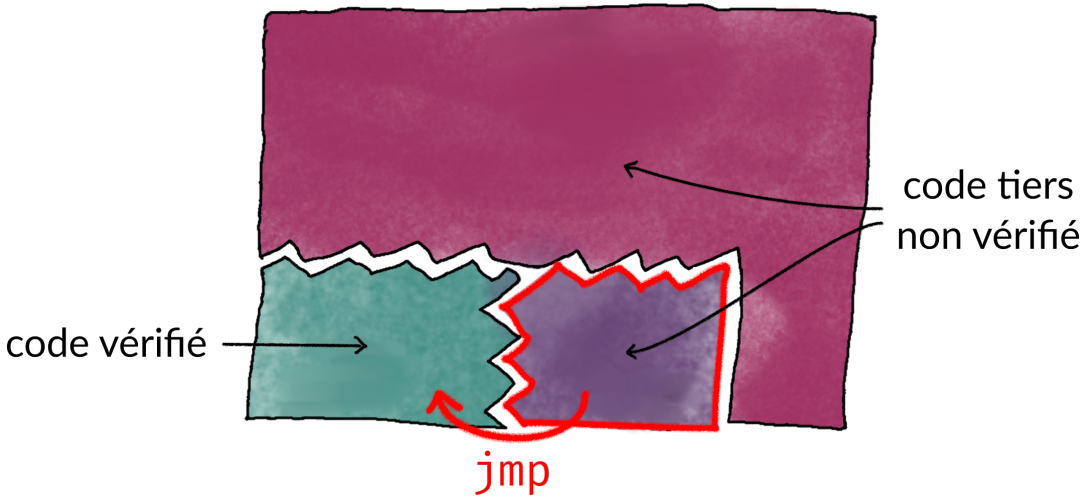


Les capabilités sont le seul moyen d'accéder à la mémoire :

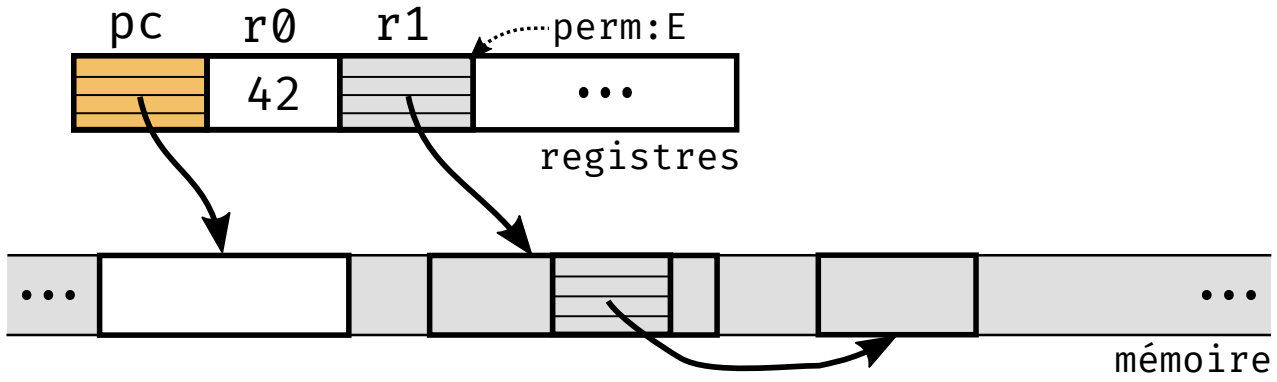


Et dans l'autre sens ?

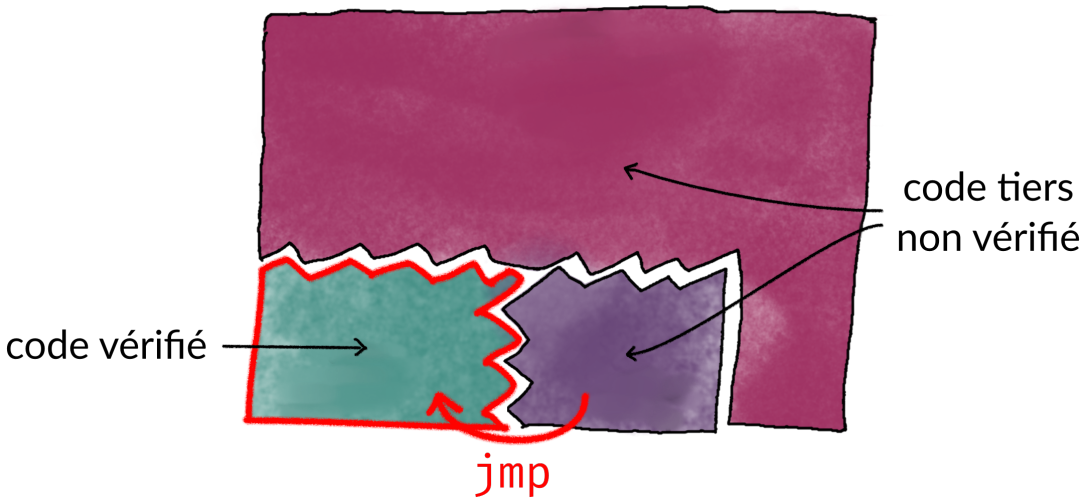
Capabilités sentinelle (perm=E) : “clôtures” bas niveau



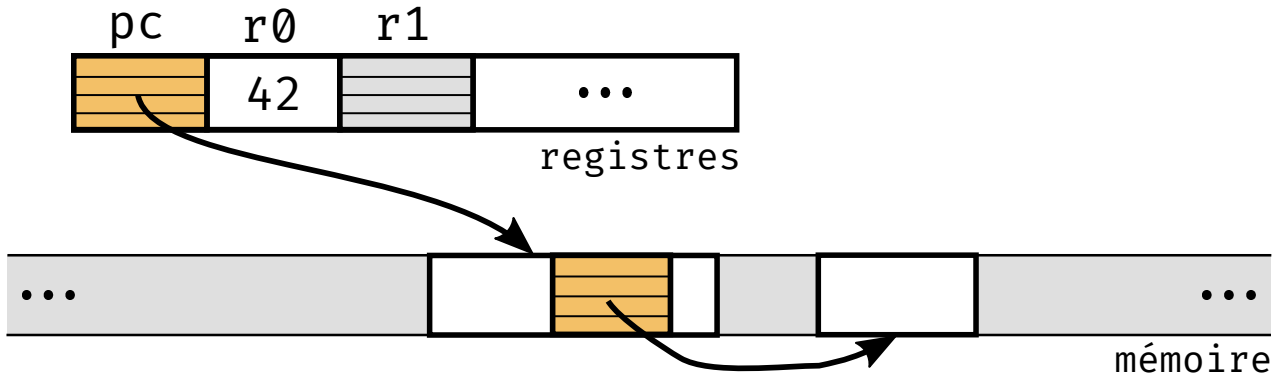
On donne au code non vérifié une capabilité `E` opaque, qui devient active après un `jmp`. Avant `jmp r1` :



Capabilités sentinelle (perm=E) : “clôtures” bas niveau



On donne au code non vérifié une capabilité `E` opaque, qui devient active après un `jmp`. Après `jmp r1` :



Exemple : un compteur sécurisé

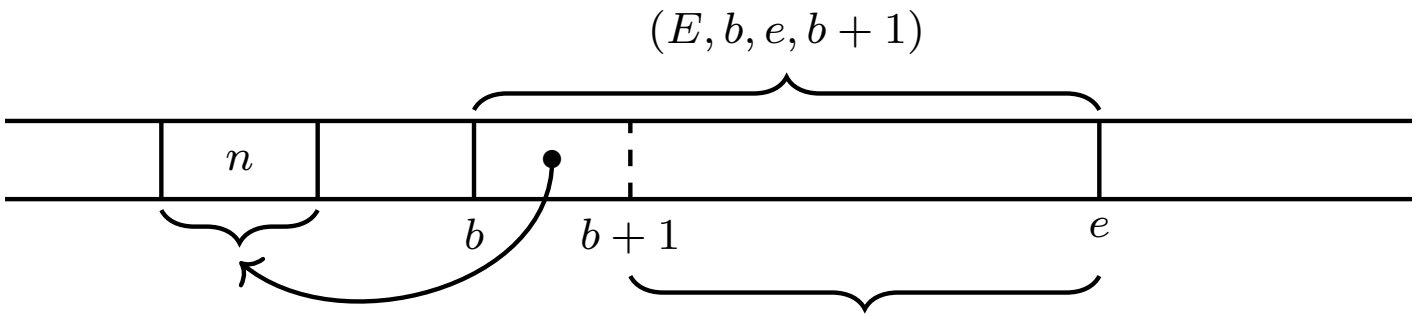
```
let x = ref 0 in
(fun () -> x := !x + 1; !x)
```

Montrer que : à tout instant, `x` contient un entier positif, quel que soit le code utilisant le compteur.

Exemple : un compteur sécurisé

```
let x = ref 0 in
(fun () -> x := !x + 1; !x)
```

Montrer que : à tout instant, `x` contient un entier positif, quel que soit le code utilisant le compteur.

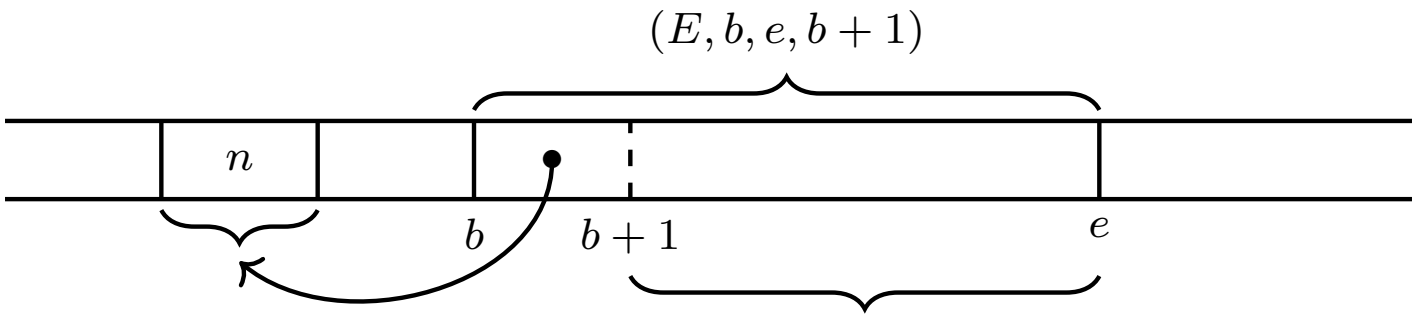



```
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
mov r2 0
jmp rret
```

Exemple : un compteur sécurisé

```
let x = ref 0 in
(fun () -> x := !x + 1; !x)
```

Montrer que : à tout instant, `x` contient un entier positif, quel que soit le code utilisant le compteur.




```
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
mov r2 0
jmp rret
```

Note : on finit par un `jmp` sur une capabilité inconnue !

Comment raisonner sur ce programme ?

On utilise la logique de séparation Iris embarquée dans Coq, pour définir :

1. Une logique de programme permettant la vérification déductive de programmes concrets en code machine;

2. Une “spécification universelle” pour du code non vérifié inconnu.

Logique de programme

La spécification prouvée pour notre compteur sécurisé est de la forme :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0} \vdash \{w_{unk};\ r_1 \Mapsto (E, b, b+1, e) * \ldots\} \rightsquigarrow \bullet$$

Où :

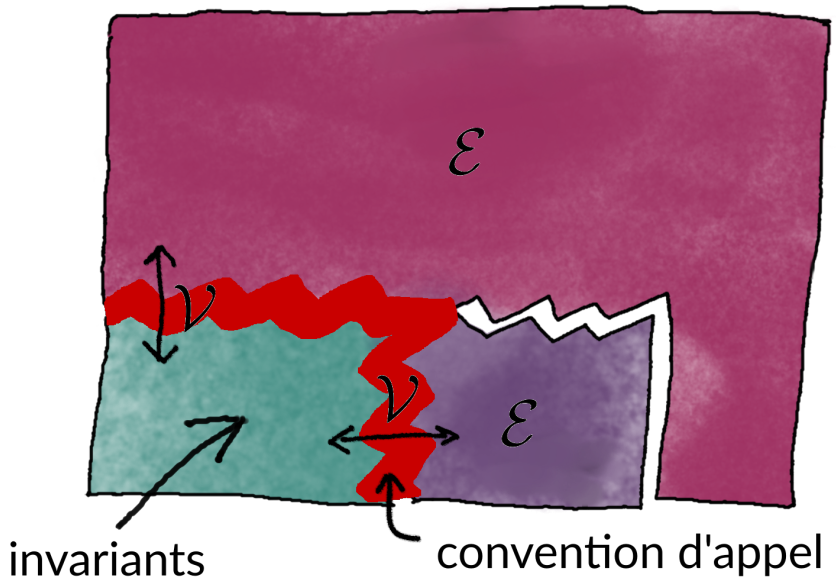
- $\boxed{P}$ décrit que "P est vrai à tout moment de l'exécution"
- $a \mapsto w$ décrit que "la case mémoire d'adresse a contient w"
- $r \Mapsto w$ décrit que "le registre r contient w"
- $\{w; P\} \rightsquigarrow \bullet$ décrit que "la machine peut s'exécuter en partant d'un état satisfaisant pc $\Mapsto w * P$ (tout en préservant les invariants)"

Logique de programme (2)

Sert également à spécifier les instructions machine individuelles, par exemple pour mov :

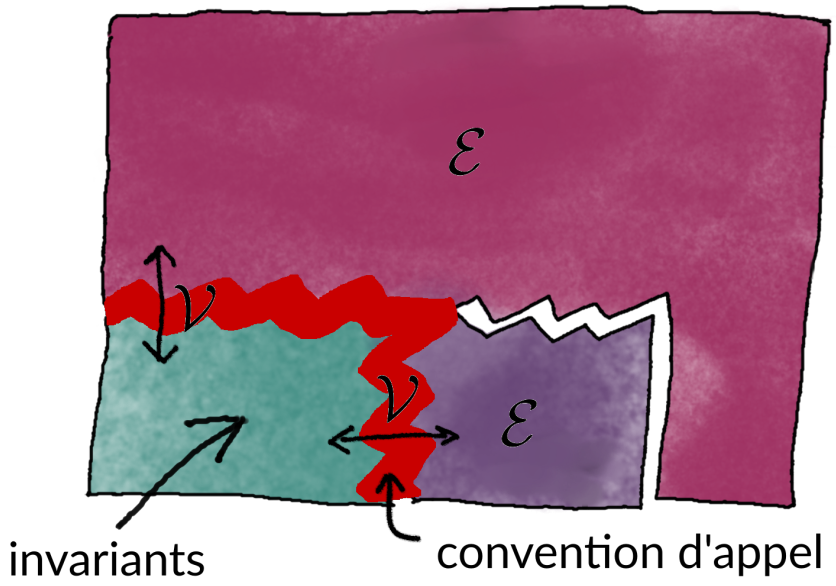
$$decode(w) = \mathsf{mov}\ r_1\ r_2 \rightarrow$$
$$b \leq a < e\ \wedge\ p \in \{\mathrm{RX}, \mathrm{RWX}\} \rightarrow$$
$$\{\ \mathsf{pc} \Mapsto (p, b, e, a) \qquad * a \mapsto w * r_1 \Mapsto x * r_2 \Mapsto y\ \}$$
$$\mathsf{SingleStep}$$
$$\{\ \mathsf{pc} \Mapsto (p, b, e, a + 1)\ * a \mapsto w * r_1 \Mapsto y * r_2 \Mapsto y\ \}$$

Raisonner en présence de code non vérifié



- Invariants sur **l'état local** du code vérifié
- $\mathcal{E}(c)$: c pointe vers du code sûr à exécuter
- $\mathcal{V}(c)$: c pointe vers des données sûres à partager

Raisonner en présence de code non vérifié



- Invariants sur **l'état local** du code vérifié
- $\mathcal{E}(c)$: c pointe vers du code sûr à exécuter
- $\mathcal{V}(c)$: c pointe vers des données sûres à partager

Théorème Fondamental : $\forall c.\, \mathcal{V}(c) \Longrightarrow \mathcal{E}(c)$.

Corollaire : pour tout c pointant vers du code inconnu, on a $\mathcal{E}(c)$.

$\Longrightarrow \mathcal{E}$ donne une **spécification universelle** du code inconnu.

Preuve : inspecter la sémantique de chaque instruction machine possible

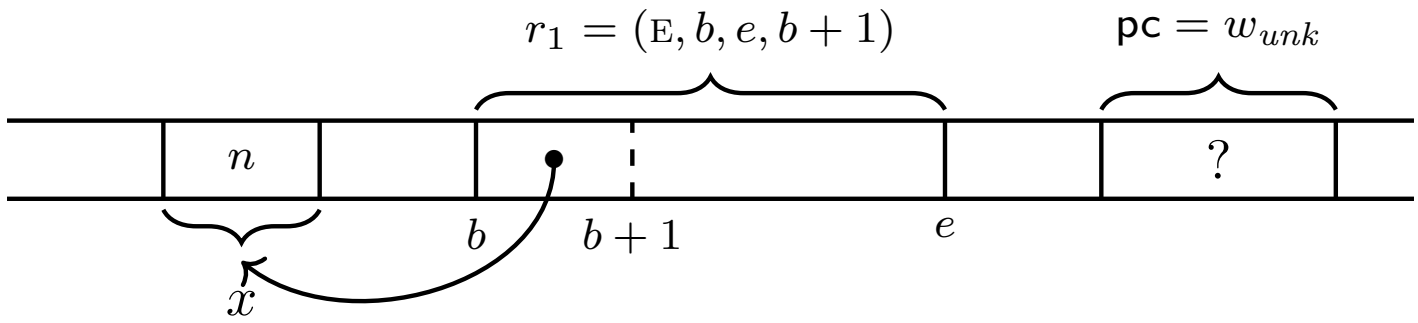
$\mathcal{E}/\mathcal{V}$: définition

Rappel : $\mathcal{E}$ = “code sûr à exécuter”, $\mathcal{V}$ = “données sûres à partager”

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

$$\mathcal{V}(w) \quad \begin{cases} \mathcal{V}(z) & \triangleq \ \mathsf{True} \\ \mathcal{V}(\mathrm{E}, b, e, a) & \triangleq \ \triangleright\, \square\, \mathcal{E}(\mathrm{RX}, b, e, a) \\ \mathcal{V}(\mathrm{RW}/\mathrm{RWX}, b, e, -) & \triangleq \ \mathbin{*}_{a\in[b,e)} \boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)} \\ \mathcal{V}(\mathrm{RO}/\mathrm{RX}, b, e, -) & \triangleq \ \ldots \end{cases}$$

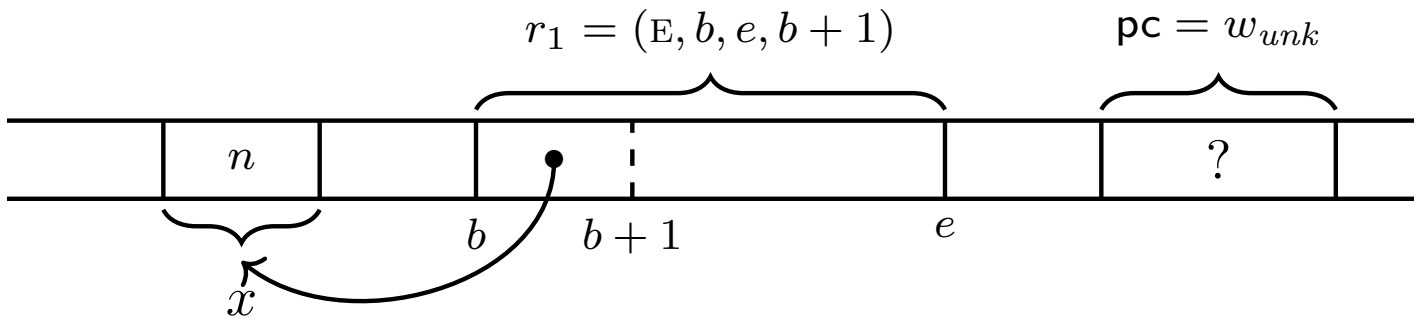
De retour à l'exemple



Spécification pour le scénario entier :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b, e) \mapsto (\mathrm{RW}, x, x+1, x) :: \mathit{instrs}}, \mathcal{V}(w_{unk})$$
$$\vdash \left\{ w_{unk}; \begin{array}{l} *_{r \notin \{\mathsf{pc}, r_1\}} (\exists z, r \Mapsto z) \\ r_1 \Mapsto (\mathrm{E}, b, e, b+1) \end{array} \right\} \rightsquigarrow \bullet$$

De retour à l'exemple

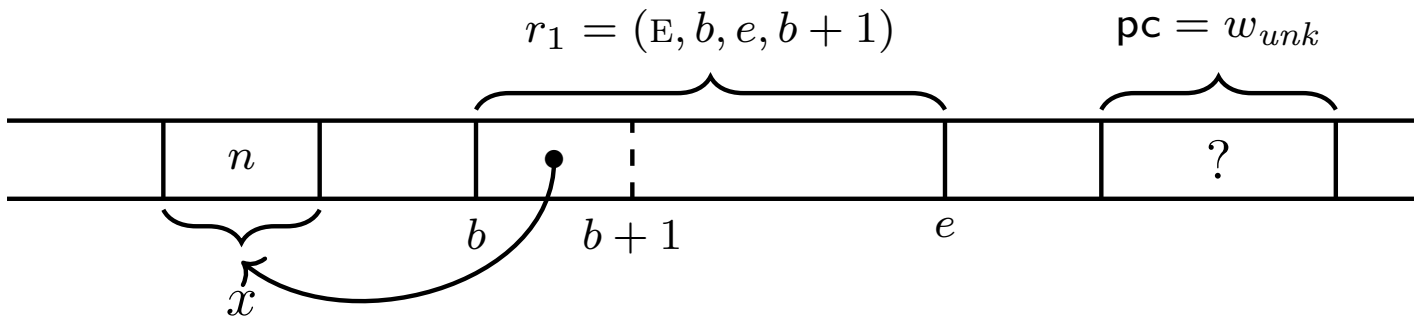


Spécification pour le scénario entier :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b, e) \mapsto (\text{RW}, x, x+1, x) :: \mathit{instrs}}, \mathcal{V}(w_{unk})$$
$$\vdash \left\{ w_{unk};\ \begin{array}{l} *_{r \notin \{\text{pc}, r_1\}} (\exists z, r \Mapsto z) \\ r_1 \Mapsto (\text{E}, b, e, b+1) \end{array} \right\} \rightsquigarrow \bullet$$

- On suppose $\mathcal{V}(w_{unk})$: par ex. w_{unk} ne pointe que vers du code

De retour à l'exemple

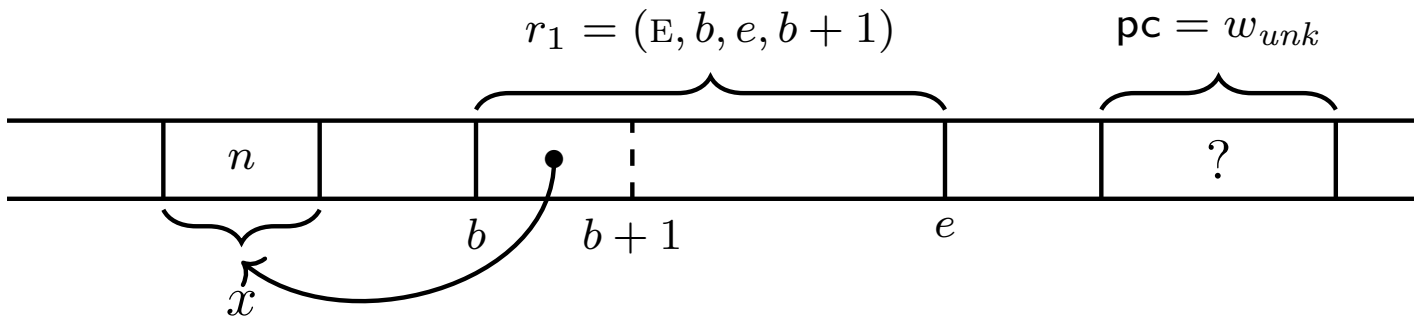


Spécification pour le scénario entier :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b, e) \mapsto (\text{RW}, x, x+1, x) :: instrs}, \mathcal{V}(w_{unk})$$
$$\vdash \left\{ w_{unk}; \begin{array}{l} *_{r \notin \{\text{pc}, r_1\}} (\exists z, r \Mapsto z) \\ r_1 \Mapsto (\text{E}, b, e, b+1) \end{array} \right\} \rightsquigarrow \bullet$$

- On suppose $\mathcal{V}(w_{unk})$: par ex. w_{unk} ne pointe que vers du code
- On utilise le **Théorème Fondamental** : $\mathcal{V}(w_{unk}) \Longrightarrow \mathcal{E}(w_{unk})$

De retour à l'exemple

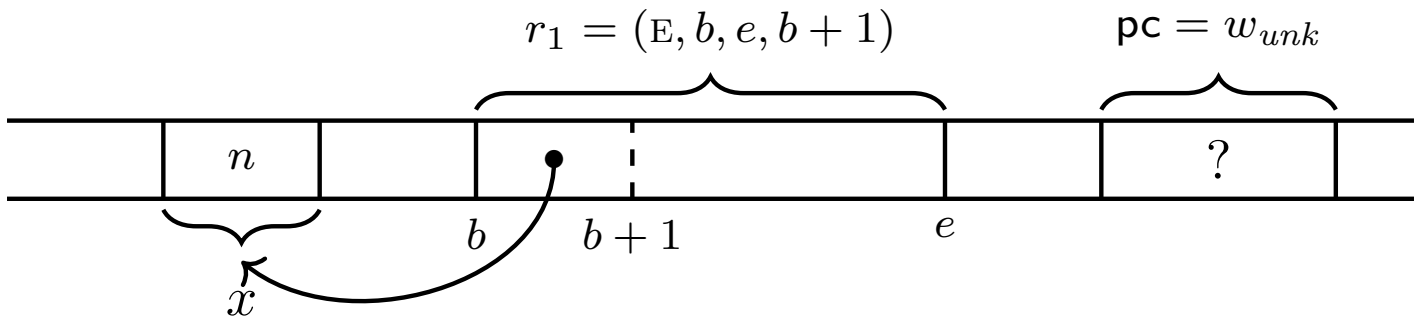


Spécification pour le scénario entier :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b, e) \mapsto (\text{RW}, x, x+1, x) :: \mathit{instrs}}, \mathcal{V}(w_{unk})$$
$$\vdash \left\{ w_{unk};\ \begin{array}{l} \ast_{r \notin \{\mathsf{pc}, r_1\}} (\exists z, r \Mapsto z) \\ r_1 \Mapsto (\text{E}, b, e, b+1) \end{array} \right\} \rightsquigarrow \bullet$$

- On suppose $\mathcal{V}(w_{unk})$: par ex. w_{unk} ne pointe que vers du code
- On utilise le **Théorème Fondamental** : $\mathcal{V}(w_{unk}) \Longrightarrow \mathcal{E}(w_{unk})$
- $\mathcal{E}(w_{unk}) \triangleq \{w_{unk}; \ast\, r \Mapsto v * \mathcal{V}(v)\} \rightsquigarrow \bullet$

De retour à l'exemple

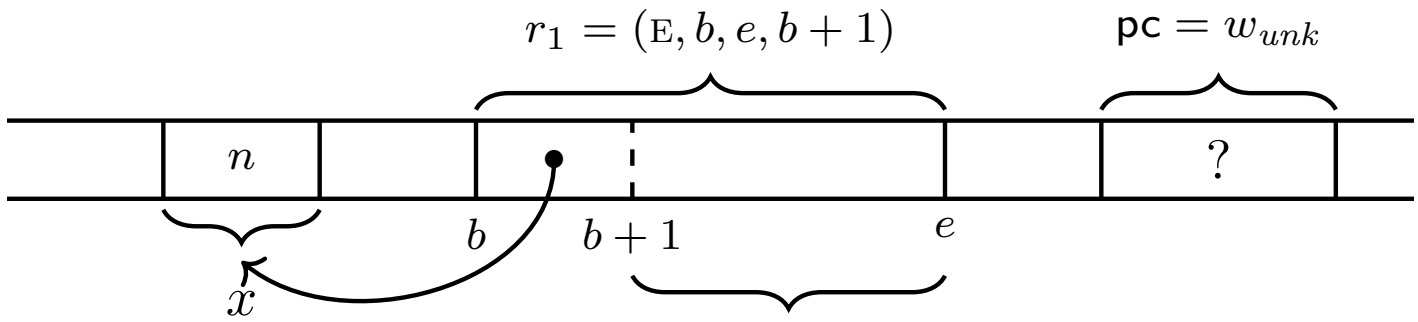


Spécification pour le scénario entier :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b, e) \mapsto (\text{RW}, x, x+1, x) :: \mathit{instrs}}, \mathcal{V}(w_{unk})$$
$$\vdash \left\{ w_{unk}; \begin{array}{l} \ast_{r \notin \{\text{pc}, r_1\}} (\exists z, r \Mapsto z) \\ r_1 \Mapsto (\text{E}, b, e, b+1) \end{array} \right\} \rightsquigarrow \bullet$$

- On suppose $\mathcal{V}(w_{unk})$: par ex. w_{unk} ne pointe que vers du code
- On utilise le **Théorème Fondamental** : $\mathcal{V}(w_{unk}) \Longrightarrow \mathcal{E}(w_{unk})$
- $\mathcal{E}(w_{unk}) \triangleq \{w_{unk}; \ast\, r \Mapsto v * \mathcal{V}(v)\} \rightsquigarrow \bullet$
- Il faut prouver $\mathcal{V}(\text{E}, b, e, b+1)$: sûreté de la clôture

De retour à l'exemple : sûreté de la clôture

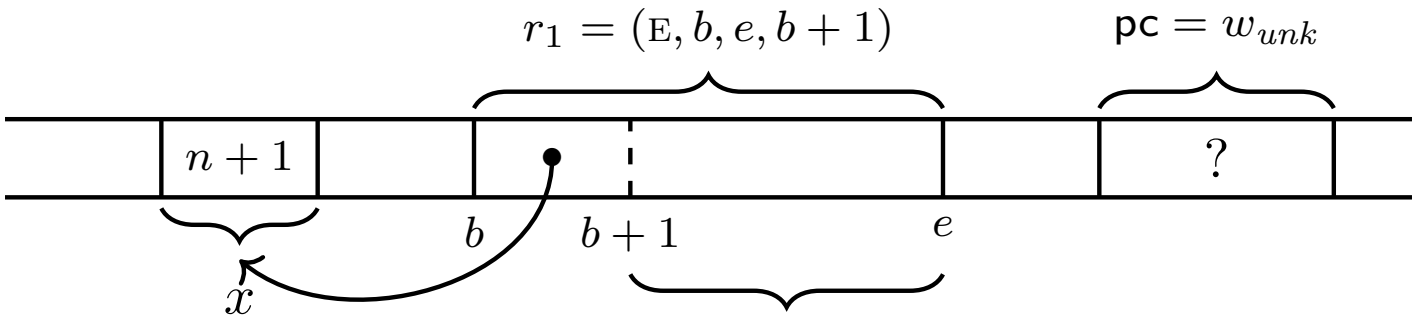



```
-> pc est ici
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
mov r2 0
jmp rret
```

Spécification de la clôture :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b, e) \mapsto (\text{RW}, x, x+1, x) :: instrs}$$
$$\vdash \left\{ (\text{RX}, b, e, b+1);\ \ *_{r \notin \{\mathsf{pc}\}} (\exists v, r \Mapsto v * \mathcal{V}(v)) \ \right\} \rightsquigarrow \bullet$$

De retour à l'exemple : sûreté de la clôture (2)



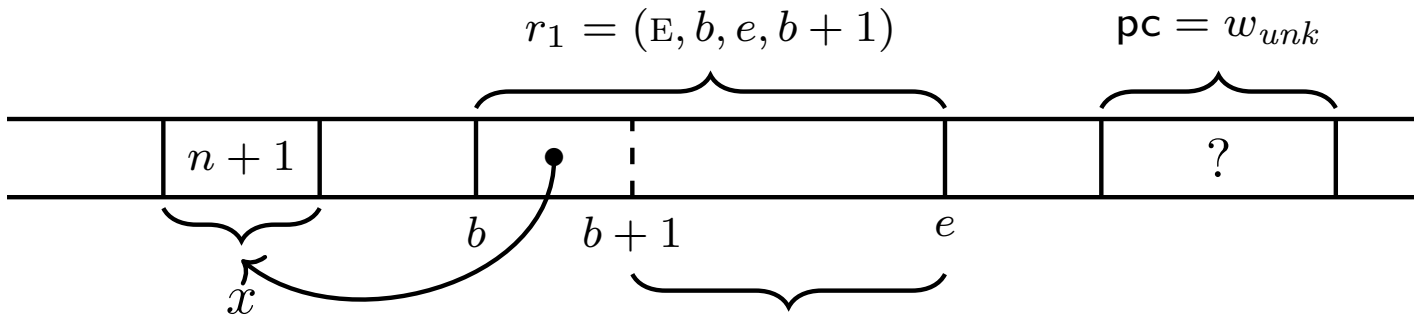

```
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
-> pc est ici
mov r2 0
jmp rret
```

Après avoir vérifié le code d'incrémentation :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b,e) \mapsto (\text{RW}, x, x+1, x) :: \mathit{instrs}}$$

$$\vdash \left\{ (\text{RX}, b, e, b+7); \begin{array}{l} *_{r \notin \{\mathsf{pc}, r_1, r_2\}} (\exists v, r \Mapsto v * \mathcal{V}(v)) * \\ r_1 \Mapsto (n+1) * r_2 \Mapsto (\text{RW}, x, x+1, x) \end{array} \right\} \rightsquigarrow \bullet$$

De retour à l'exemple : sûreté de la clôture (3)

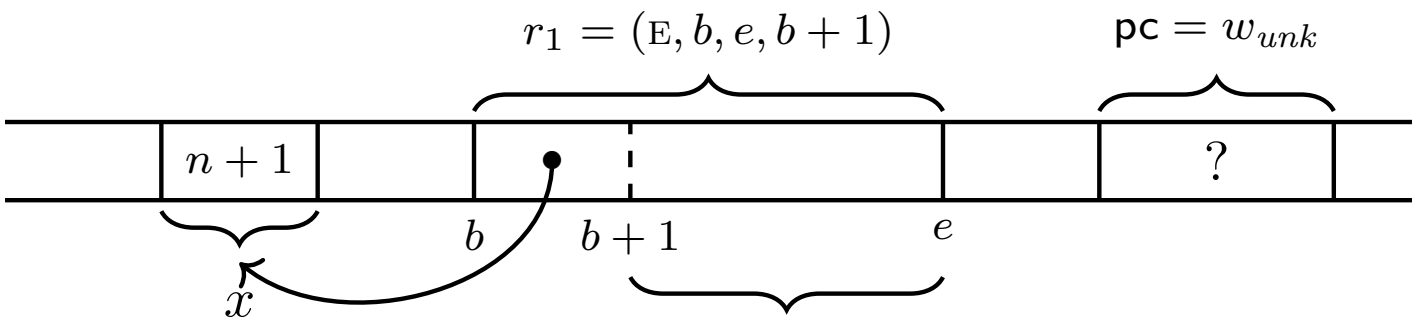



```
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
mov r2 0
-> pc est ici
jmp rret
```

Après avoir vérifié le code d'incrémentation et de nettoyage :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b, e) \mapsto (\text{RW}, x, x+1, x) :: \mathit{instrs}}$$
$$\vdash \left\{ (\text{RX}, b, e, b+8); \begin{array}{l} *_{r \notin \{\mathsf{pc}, r_1, r_2\}} (\exists v, r \Rightarrow v * \mathcal{V}(v)) * \\ r_1 \Rightarrow (n+1) * r_2 \Rightarrow 0 \end{array} \right\} \rightsquigarrow \bullet$$

De retour à l'exemple : sûreté de la clôture (4)

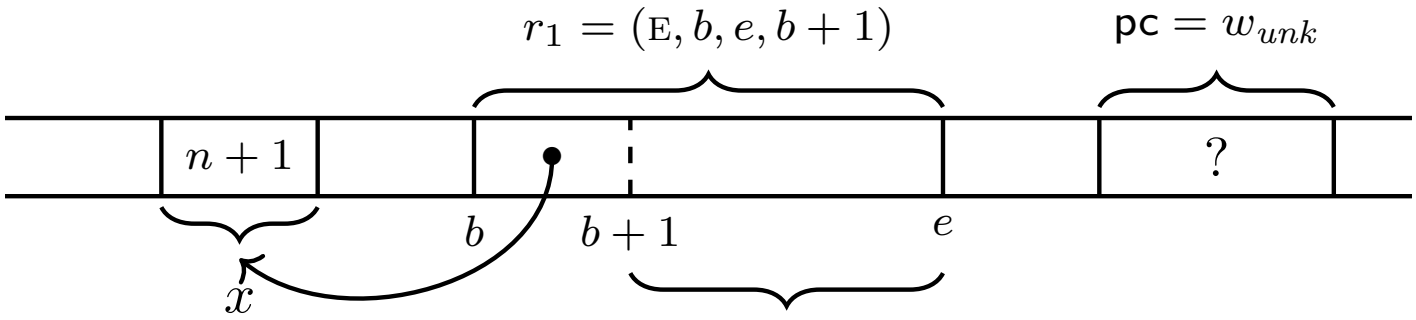



```
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
mov r2 0
jmp rret
-> pc = wret
```

Après avoir `jmp` sur le pointeur de retour w_{ret} (inconnu mais sûr) :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b,e) \mapsto (\text{RW}, x, x+1, x) :: \mathit{instrs}}, \mathcal{V}(w_{ret})$$
$$\vdash \left\{ w_{ret}; \begin{array}{l} *_{r \notin \{\text{pc}, r_1, r_2\}} (\exists v, r \Rightarrow v * \mathcal{V}(v)) * \\ r_1 \Rightarrow (n+1) * r_2 \Rightarrow 0 \end{array} \right\} \rightsquigarrow \bullet$$

De retour à l'exemple : sûreté de la clôture (4)




```
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
mov r2 0
jmp rret
-> pc = wret
```

Après avoir `jmp` sur le pointeur de retour w_{ret} (inconnu mais sûr) :

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b,e) \mapsto (\text{RW}, x, x+1, x) :: \mathit{instrs}}, \mathcal{V}(w_{ret})$$
$$\vdash \left\{ w_{ret}; \begin{array}{l} *_{r \notin \{\text{pc}, r_1, r_2\}} (\exists v, r \Mapsto v * \mathcal{V}(v)) * \\ r_1 \Mapsto (n+1) * r_2 \Mapsto 0 \end{array} \right\} \rightsquigarrow \bullet$$

Qed en utilisant le **Théorème Fondamental** ! $\mathcal{V}(w_{ret}) \Longrightarrow \mathcal{E}(w_{ret})$

Preuve de l'exemple : en résumé

$$\boxed{\exists n, x \mapsto n \wedge n \geq 0}, \boxed{[b, e) \mapsto (\mathrm{RW}, x, x+1, x) :: \mathit{instrs}}, \mathcal{V}(w_{unk})$$
$$\vdash \left\{ w_{unk};\ \begin{array}{l} \ast_{r \notin \{\mathsf{pc}, r_1\}} (\exists z, r \Mapsto z) \\ r_1 \Mapsto (\mathrm{E}, b, e, b+1) \end{array} \right\} \rightsquigarrow \bullet$$

On a prouvé cette spécification en utilisant le Théorème Fondamendal deux fois :

- Initialement, pour "exécuter" le code inconnu
- En redonnant le contrôle au code inconnu à la fin de la clôture

Conclusion

- Les capabilités sont puissantes… à condition de bien les utiliser : d'où l'intérêt des preuves formelles
- Notre approche se généralise pour raisonner sur des interactions code vérifié–non vérifié plus complexes (cf notre article POPL'21)
- On est juché sur les épaules de géants, notamment Iris

https://github.com/logsem/cerise