Preuves de programmes : au-delà de la correction fonctionnelle

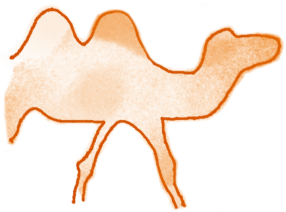
Armaël Guéneau

- 2020– : **postdoc** avec **Lars Birkedal** à l'université d'Aarhus (Danemark)
- 2016–2019 : **thèse** avec **François Pottier** et **Arthur Charguéraud** à Inria
- 2012–2016 : ENS de Lyon

Équipe d'accueil : Celtique

Mon domaine de recherche

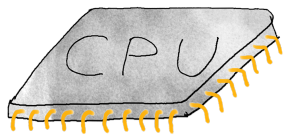
langages de programmation ∩ méthodes formelles



OCaml

CakeML

logique de séparation



langage machine

preuves mécanisées : **Coq**, HOL4

But : établir formellement des propriétés pour des programmes concrets

Pourquoi vérifier formellement des logiciels ?

Une quantité de code de plus en plus importante contrôle des systèmes critiques divers : il est important d'éliminer les *bugs* et *failles de sécurité*.

Pourquoi vérifier formellement des logiciels ?

Une quantité de code de plus en plus importante contrôle des systèmes critiques divers : il est important d'éliminer les *bugs* et *failles de sécurité*.

Irréaliste de **tout** vérifier. Il semble préférable de :

1) **vérifier le mieux possible les composants les plus critiques** ;
2) **garantir leur bon fonctionnement au sein du système complet**.

Vérification formelle de la complexité asymptotique de programmes (thèse)

Un bon programme :

calcule le bon résultat, en consommant des **ressources raisonnables**

$\rightarrow$ sinon : attaques par déni de service

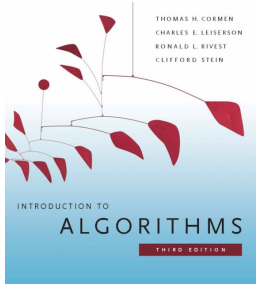
Dans ma thèse :

Vérifier formellement la correction fonctionnelle et la **complexité asymptotique** de programmes implémentant des algorithmes non triviaux

Vérification formelle de la complexité asymptotique de programmes (thèse) (2)

Travaux existants :

Démonstrations papier :
bornes en O(),
raisonnements informels, pseudo-code idéalisé



Méthodes formelles :
analyses automatiques restreintes, ou
comptage manuel ("$25n + 78$ appels de fonction")

RaML

CFML+$

Ce travail :

Preuves de complexité avec O() vérifiées en Coq

- pour des programmes impératifs, d'ordre supérieur
- avec bornes de complexité amortie
- pouvant dépendre des arguments de correction fonctionnelle

Thèse : contributions

Formalisation de O() :

- O() à une et plusieurs variables
- lemmes adaptés à l'analyse de programmes

Méthodologie pour la preuve formelle de bornes de complexité :

- **mécanisme semi-automatique d'inférence de coût**
- implémenté comme une extension de CFML
  (*framework* de logique de séparation en Coq)

Étude de cas significative :
vérification d'un algorithme de l'état de l'art

Publications : **ESOP'18**, **ITP'19**, Coq Workshop'18

Thèse : étude de cas

Vérification d'un algorithme de détection incrémentale de cycles dans un graphe par Bender, Fineman, Gilbert, Tarjan (2015).

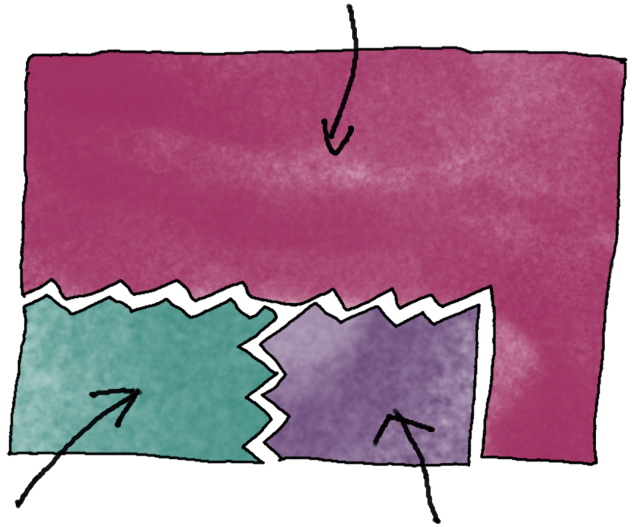
```
Theorem add_edge_spec : ∀g G v w,
  let m := card (edges G) in let n := card (vertices G) in
  v ∈ vertices G ∧ w ∈ vertices G ∧ ¬ has_edge G v w →
  Spec add_edge_or_detect_cycle [g v w]
    PRE (IsGraph g G ∗ $(ψ (m+1) n − ψ m n))
    POST (fun res ⇒ match res with
          | EdgeAdded ⇒ IsGraph g (G ⊎ (v, w))
          | EdgeCreatesCycle ⇒ [rtclosure (has_edge G) w v ]
          end).

(*  complexité amortie ψ ∈ O(m min(√m, n^{2/3}) + n)  *)
Theorem bound_ψ :
  dominated _ ψ (fun m n ⇒ m ∗ min (sqrt_up m) (Int_part (IZR n ^ (2/3))) + n).
```

- Meilleure borne de complexité connue pour des graphes épars
- Mon implémentation vérifiée est utilisée dans le *build system* dune
  → élimine les bugs de l'implémentation précédente
  → jusqu'à 7x plus rapide

- Utilisation potentielle dans le noyau de Coq (travail en cours)

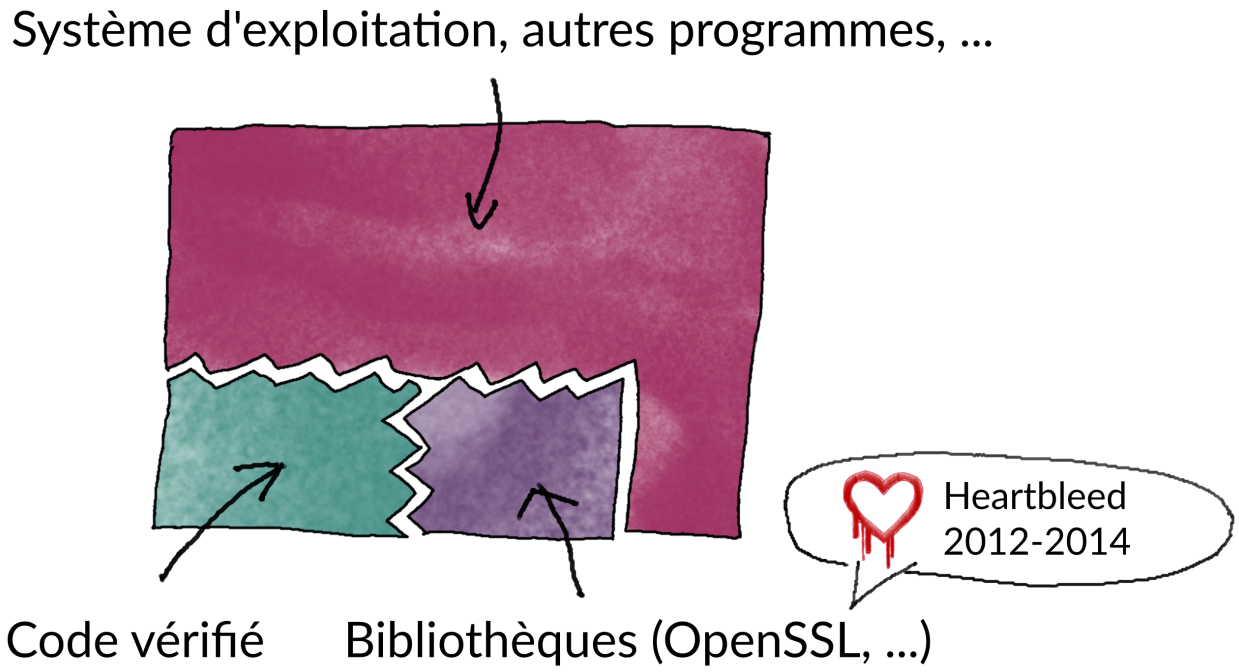
Sécuriser les interactions entre code connu et inconnu à l'aide de capabilités (postdoc)

Système d'exploitation, autres programmes, ...

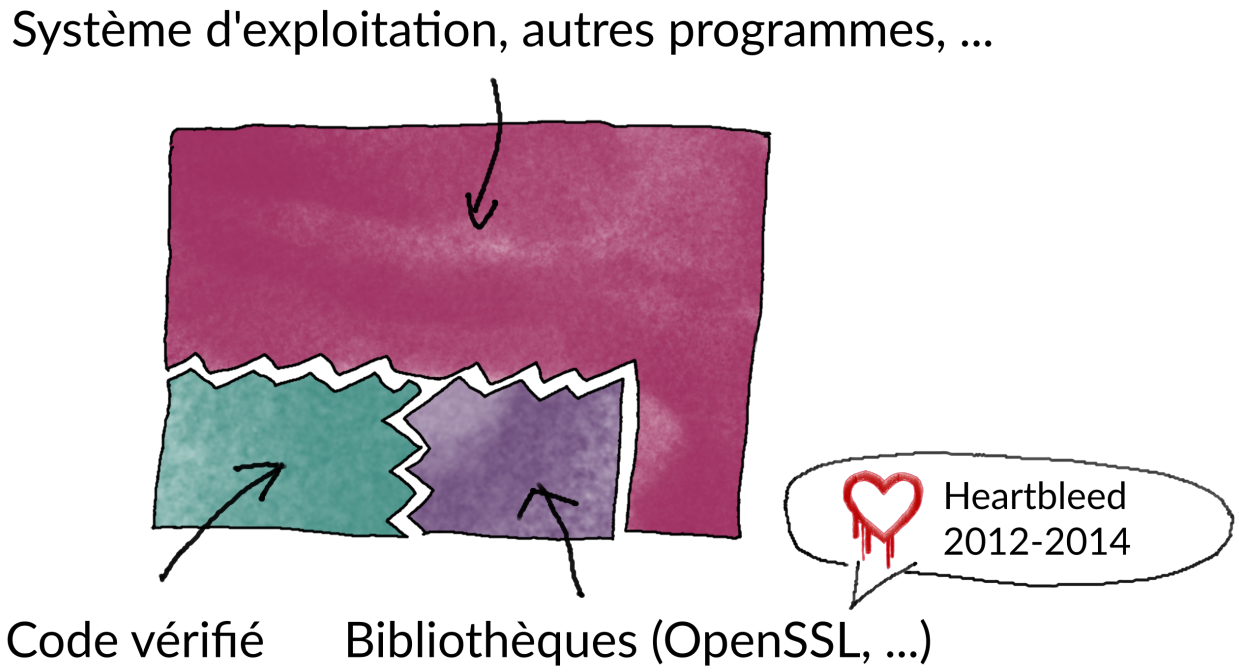


Code vérifié Bibliothèques (OpenSSL, ...)

Sécuriser les interactions entre code connu et inconnu à l'aide de capabilités (postdoc)

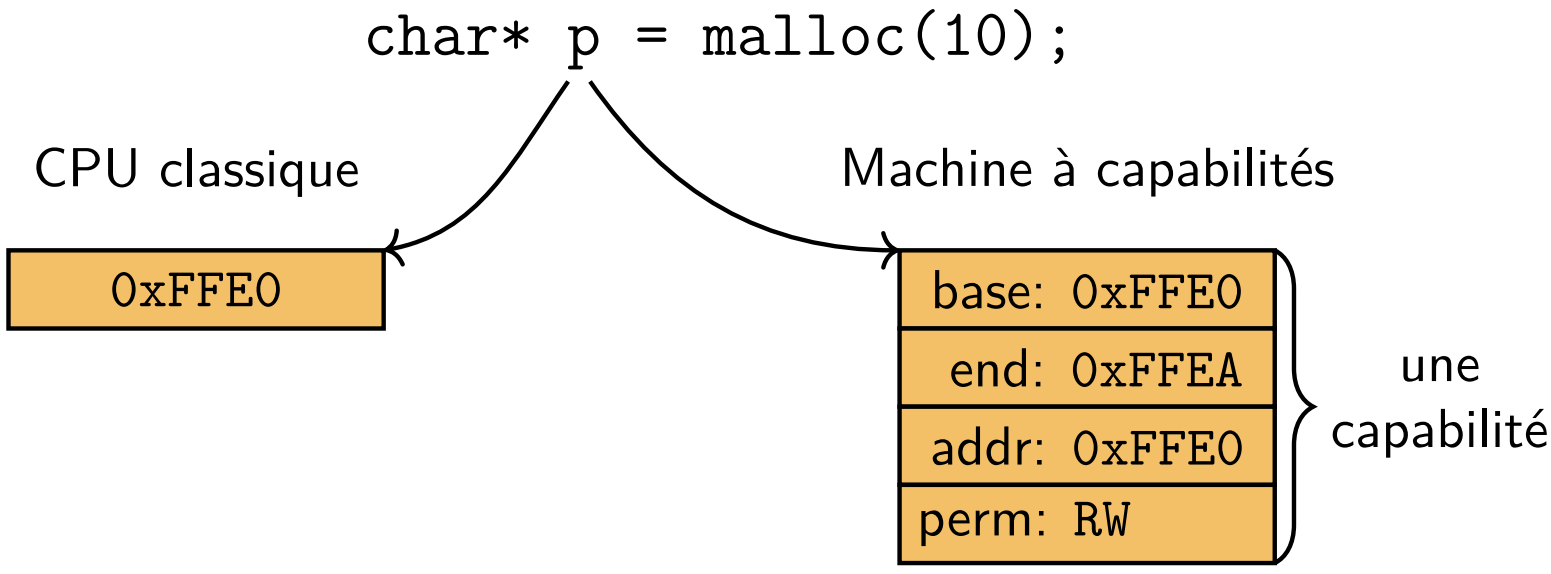


Sécuriser les interactions entre code connu et inconnu à l'aide de capabilités (postdoc)



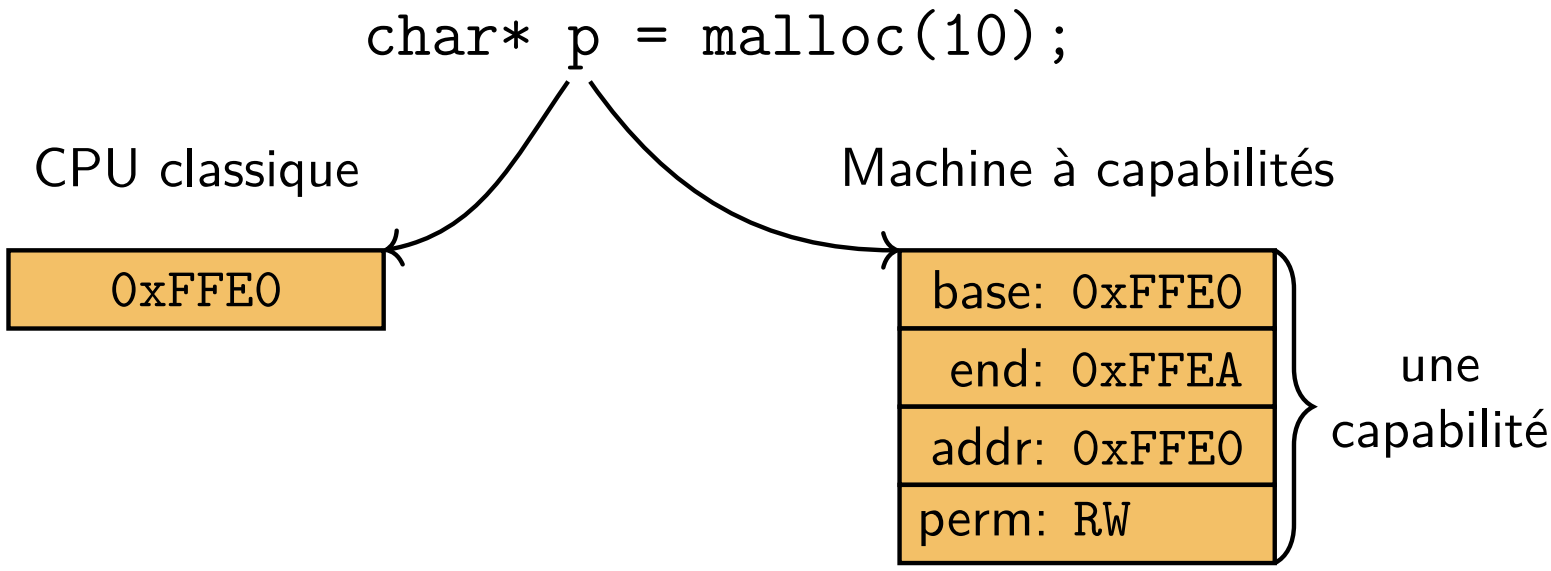
- Quels mécanismes pour sécuriser les interactions ?
  → différentes approches (sandboxing, software fault isolation, ...)
  → on s'intéresse ici à l'utilisation de *capabilités matérielles*

Capabilité = pointeur + métadonnées



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

Capabilité = pointeur + métadonnées

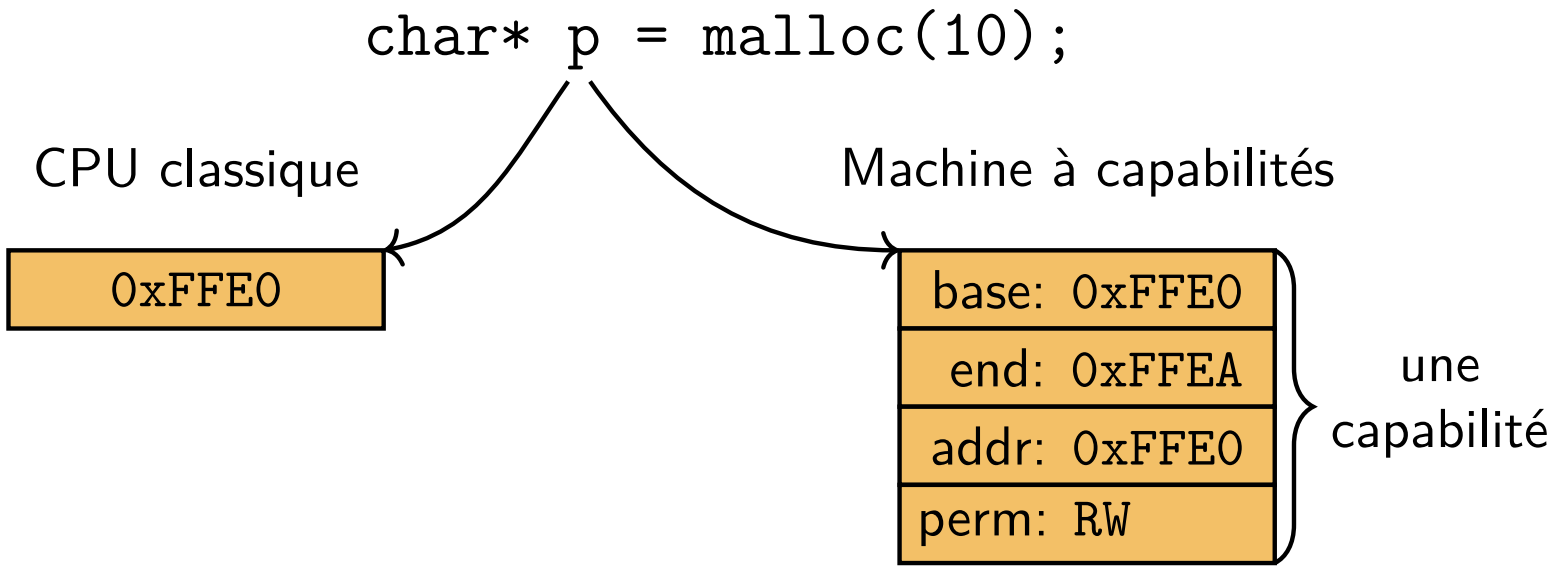


Les bornes et permissions sont **vérifiées par la machine**.

⇒ Comment **utiliser** les capabilités pour fournir des garanties de sécurité ?
Comment **vérifier formellement** les propriétés de sécurité obtenues ?

Publications : POPL'21, JFLA'21, PriSC'21, (soumission à CSF'21)

Capabilité = pointeur + métadonnées



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

⇒ Comment **utiliser** les capabilités pour fournir des garanties de sécurité ?
Comment **vérifier formellement** les propriétés de sécurité obtenues ?

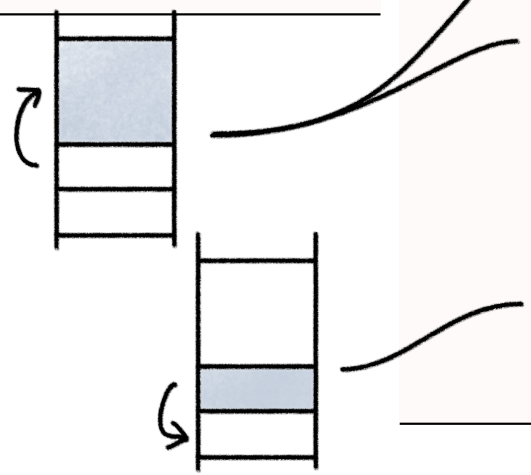
Publications : **POPL'21**, JFLA'21, PriSC'21, (soumission à CSF'21)
POPL'21: intégrité du flot de contrôle + protection des données locales

Garanties de sécurité obtenues : exemple (POPL'21)

```
void unknown(void);
void f(void) {
  static int x = 0;
  x = 0;
  unknown();
  x = 1;
  unknown();
  assert (x == 1);
}
```

Garanties de sécurité obtenues : exemple (POPL'21)

```
void unknown(void);
void f(void) {
  static int x = 0;
  x = 0;
  unknown();
  x = 1;
  unknown();
  assert (x == 1);
}
```



```
g1: malloc r2 1
    store r2 0
    move r3 pc
    lea r3 offset
    crtcls [(x, r2)] r3
    jmp r0
f1: reqglob r_adv
    prepstack r_stk
    store r_env 0
    scallU r_adv ([], [r0, r_adv, r_env])
    store r_env 1
    scallU r_adv ([], [r0, r_env])
    load r_adv r_env
    assert r_adv 1
    getb r1 r_stk
    add r2 r1 10
    subseg r_stk r1 r2
    mclear r_stk
    rclear Regs\{pc, r0}
    jmp r0
```

400 instructions après dépliage des macros

Garanties de sécurité obtenues : exemple (POPL'21)

```
void unknown(void);
void f(void) {
  static int x = 0;
  x = 0;
  unknown();
  x = 1;
  unknown();
  assert (x == 1);
}
```

```
g1: malloc r2 1
    store r2 0
    move r3 pc
    lea r3 offset
    crtcls [(x, r2)] r3
    jmp r0
f1: reqglob r_adv
    prepstack r_stk
    store r_env 0
    scallU r_adv ([], [r0, r_adv, r_env])
    store r_env 1
    scallU r_adv ([], [r0, r_env])
    load r_adv r_env
    assert r_adv 1
    getb r1 r_stk
    add r2 r1 10
    subseg r_stk r1 r2
    mclear r_stk
    rclear Regs\{pc, r0}
    jmp r0
```

400 instructions après dépliage des macros

Théorème : l'assertion réussit, quel que soit le code inconnu

Preuve : 5kLOC (preuve de l'exemple), 5kLOC (convention d'appel)

Garanties de sécurité obtenues : exemple (POPL'21)

```
void unknown(void);
void f(void) {
  static int x = 0;
  x = 0;
  unknown();
  x = 1;
  unknown();
  assert (x == 1);
}
```

```
g1: malloc r2 1
    store r2 0
    move r3 pc
    lea r3 offset
    crtcls [(x, r2)] r3
    jmp r0
f1: reqglob r_adv
    prepstack r_stk
    store r_env 0
    scallU r_adv ([], [r0, r_adv, r_env])
    store r_env 1
    scallU r_adv ([], [r0, r_env])
    load r_adv r_env
    assert r_adv 1
    getb r1 r_stk
    add r2 r1 10
    subseg r_stk r1 r2
    mclear r_stk
    rclear Regs\{pc, r0}
    jmp r0
```

400 instructions après dépliage des macros

Théorème : spécification universelle pour le code inconnu

Preuve :

12kLOC (formalisation de la machine)

10kLOC (preuve du théorème)

$\simeq$ 1,5 personne.an de travail

Théorème : l'assertion réussit, quel que soit le code inconnu

Preuve : 5kLOC (preuve de l'exemple), 5kLOC (convention d'appel)

Projet de recherche

Preuve de programmes en contexte

Objectif : faciliter l'adoption de code formellement vérifié au sein de systèmes logiciels réalistes

Comment ? des techniques pour vérifier des composants logiciels, *et* prouver qu'ils peuvent être utilisés dans un contexte non vérifié

Axe 1 : capabilités

Vérification de composants système sûrs

- allocateur mémoire (attaques "*use after free*"), ordonnanceur

Compilation vérifiée vers machines à capabilités

- compilateur vérifié pour CHERI-C (extension de CompCert ?)
- FFI sûre pour CakeML

$\rightarrow$ théorèmes de "*secure compilation*" : préservation de propriétés de sécurité entre programmes source et cible

Axe 2 : analyse de ressources

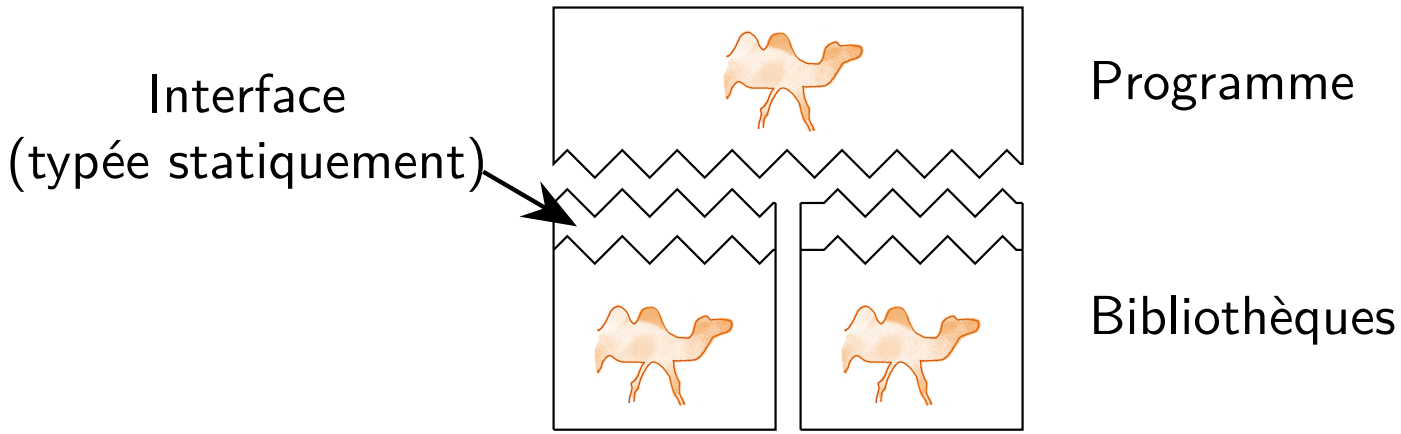
Intégration avec des outils automatisés d'analyse de ressources

- par exemple : outils de Hoffman et al. (RAML), Carbonneaux et al.

Preuve de complexité en espace

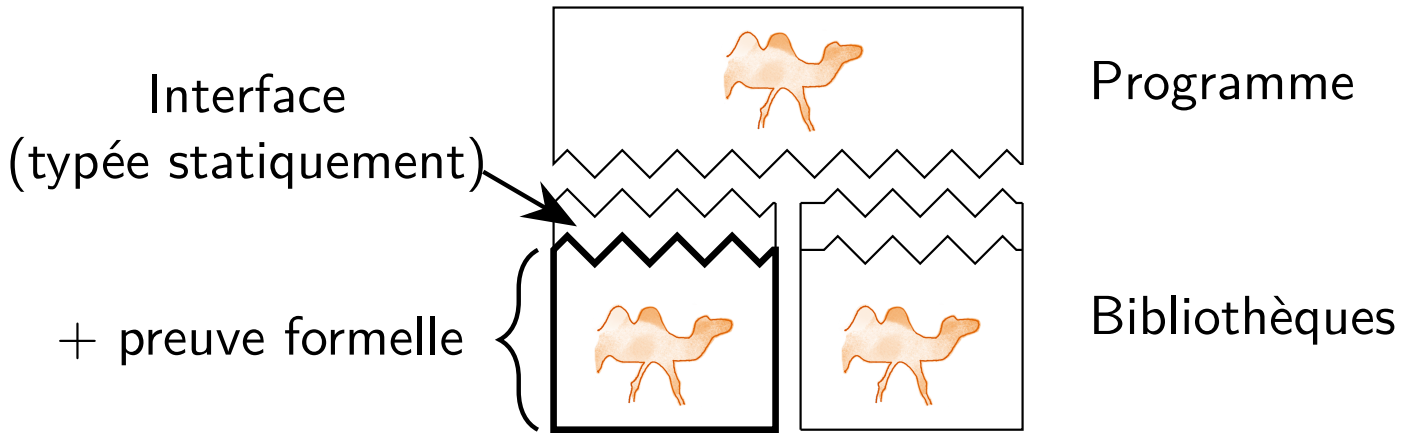
Axe 3 : vérification par composants dans un langage statiquement typé

Le typage statique : une force pour la programmation modulaire



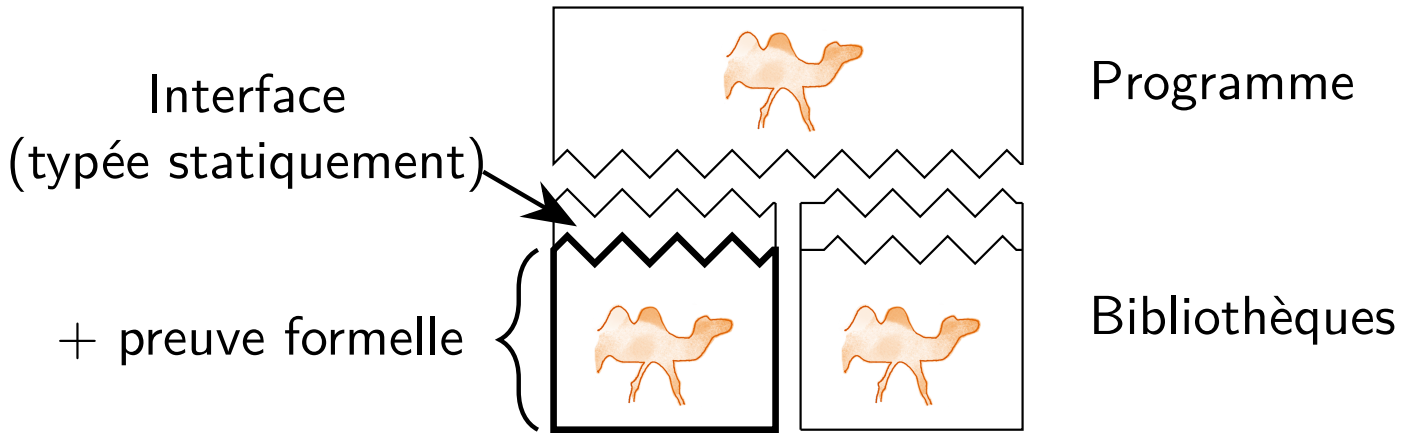
Axe 3 : vérification par composants dans un langage statiquement typé

Le typage statique : une force pour la programmation modulaire



Axe 3 : vérification par composants dans un langage statiquement typé

Le typage statique : une force pour la programmation modulaire



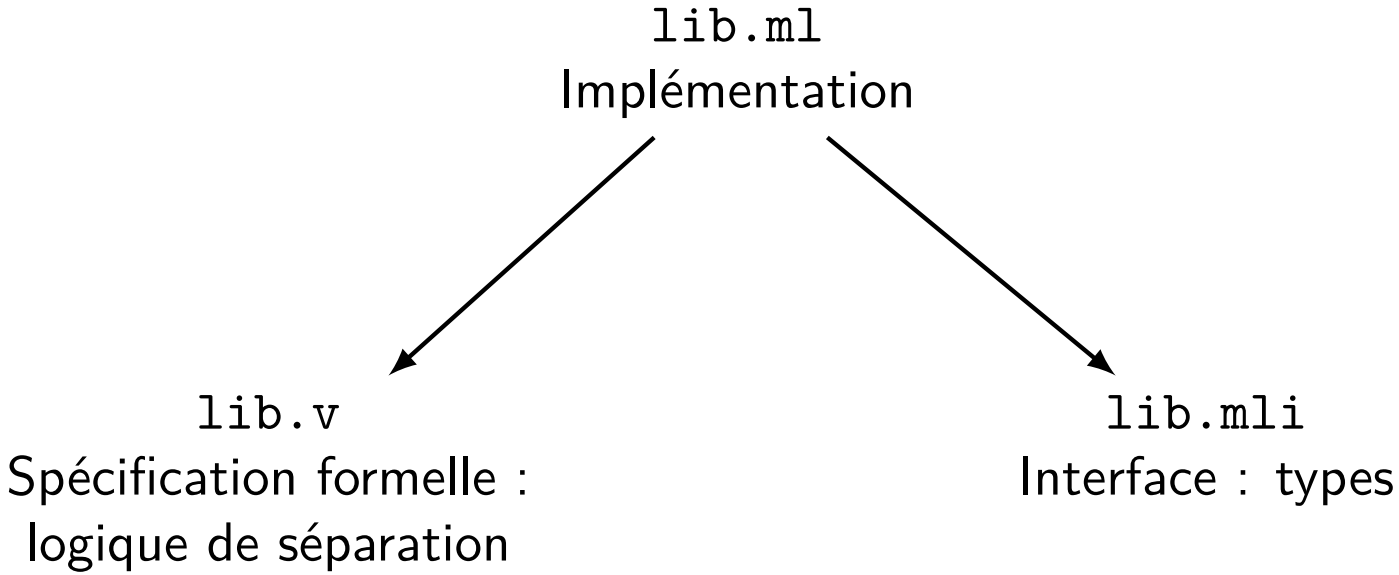
Idée : relier des spécifications en logique de séparation avec des interfaces exprimées dans un système de type à la ML

Axe 3a : interaction entre code vérifié et code non vérifié mais bien typé

$\rightarrow$ Un outil de vérification permettant de démontrer que des **garanties établies en logique de séparation** pour une bibliothèque sont **préservées par tout contexte non vérifié mais bien typé**.

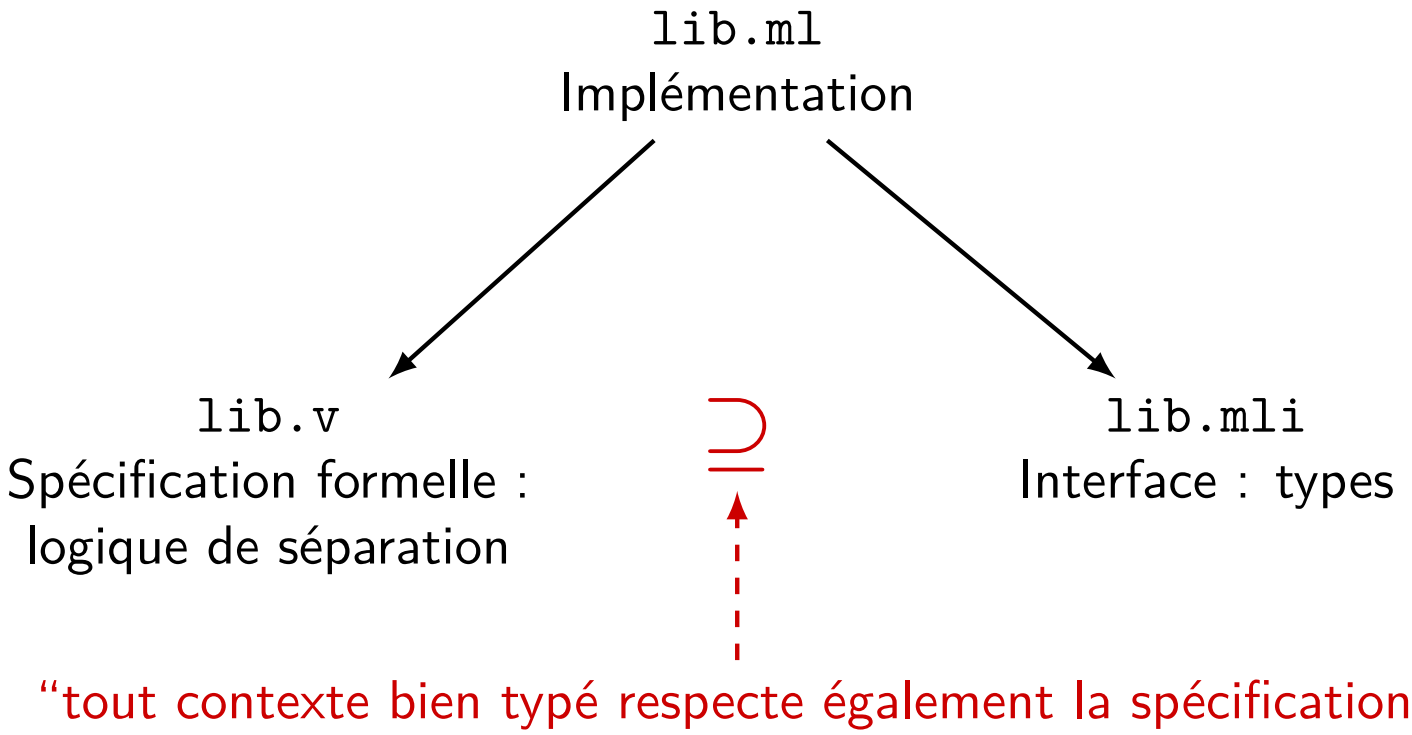
Axe 3a : interaction entre code vérifié et code non vérifié mais bien typé

$\rightarrow$ Un outil de vérification permettant de démontrer que des **garanties établies en logique de séparation** pour une bibliothèque sont **préservées par tout contexte non vérifié mais bien typé**.



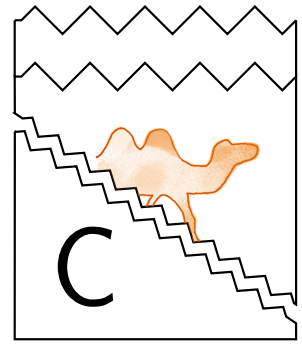
Axe 3a : interaction entre code vérifié et code non vérifié mais bien typé

→ Un outil de vérification permettant de démontrer que des **garanties établies en logique de séparation** pour une bibliothèque sont **préservées par tout contexte non vérifié mais bien typé**.



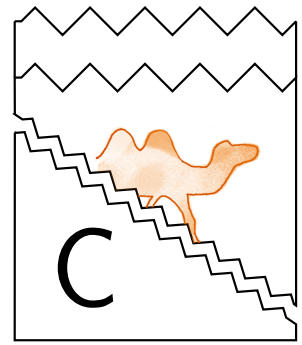
Axe 3b : vérification de composants multi-langages haut niveau/bas niveau

Cadre :
bibliothèques d'un langage haut niveau avec GC (OCaml)
partiellement implémentées avec du code bas niveau (C)



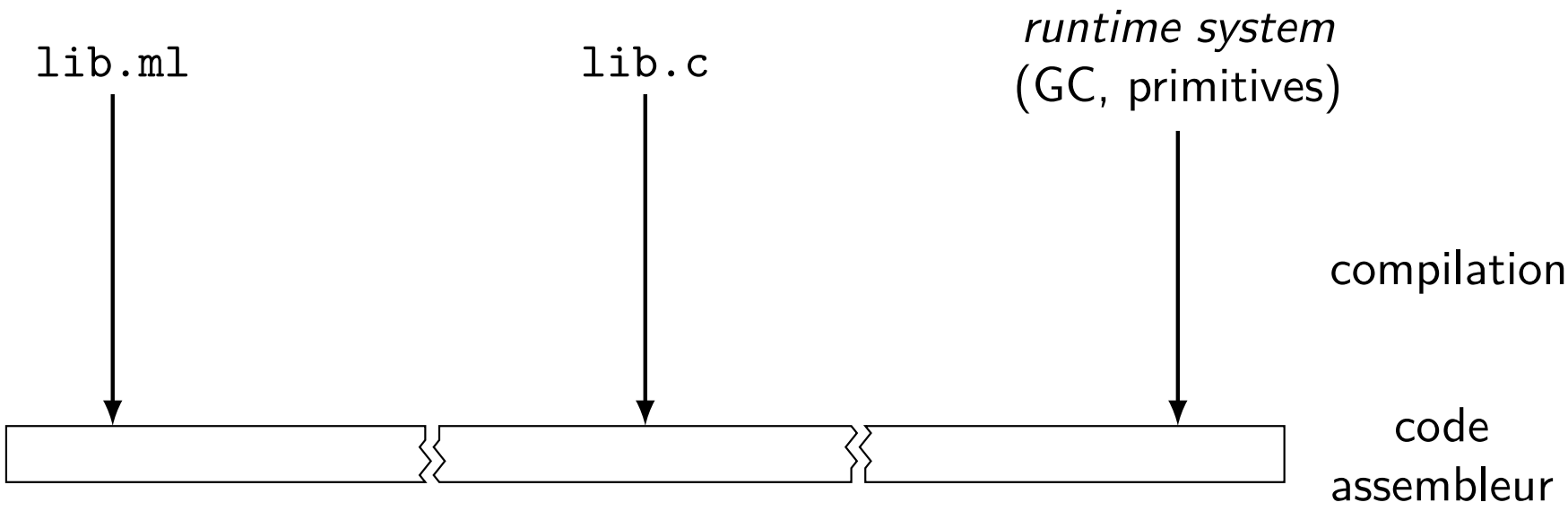
Axe 3b : vérification de composants multi-langages haut niveau/bas niveau

Cadre :
bibliothèques d'un langage haut niveau avec GC (OCaml)
partiellement implémentées avec du code bas niveau (C)

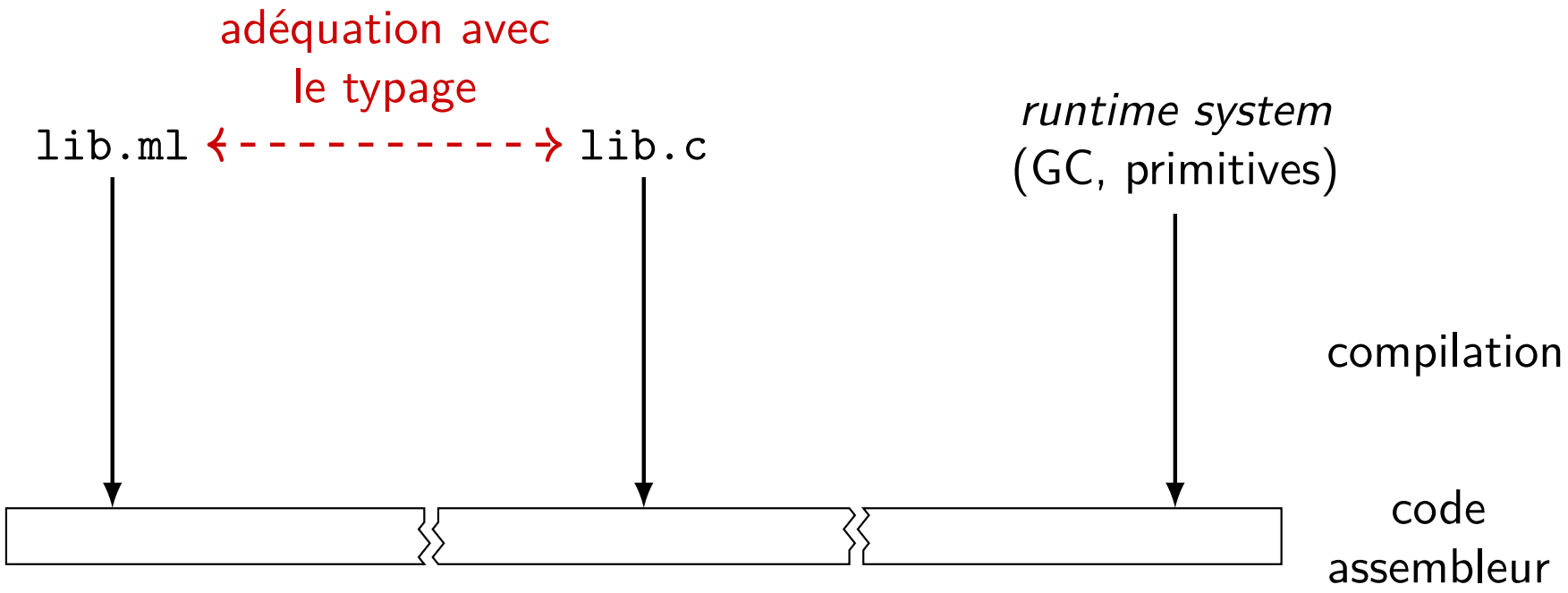


$\rightarrow$ Une approche permettant de **vérifier de bout en bout la correction d'un composant multi-langage**, et garantir que celui-ci préserve les bonnes propriétés du langage environnant.

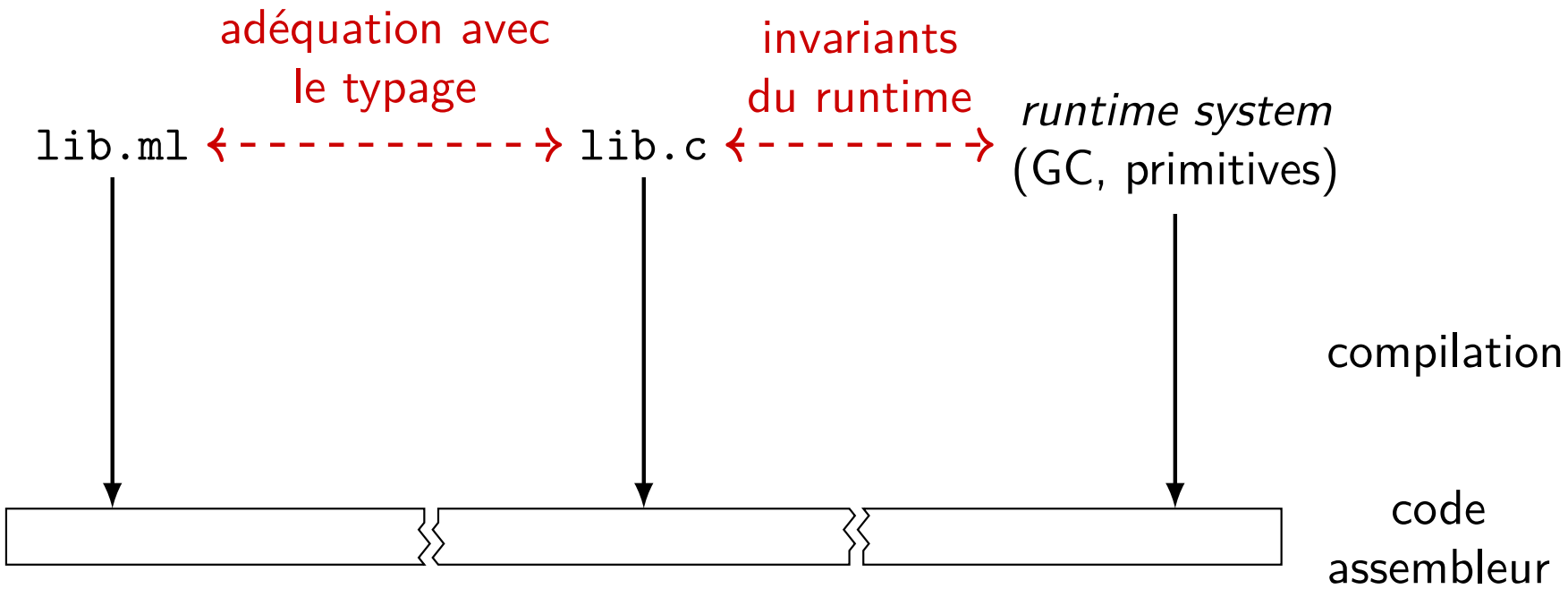
Axe 3b : vérification de composants multi-langages haut niveau/bas niveau (2)



Axe 3b : vérification de composants multi-langages haut niveau/bas niveau (2)



Axe 3b : vérification de composants multi-langages haut niveau/bas niveau (2)



Intégration dans Celtique

Compilation vérifiée et sécurité (CompCert+SFI/non-interférence...) :
F. Besson, S. Blazy, D. Demange, T. Jensen, D. Pichardie
$\rightarrow$ compilation vers capabilités

Compilation "*just-in-time*" vérifiée : S. Blazy, D. Pichardie
$\rightarrow$ interoperabilité entre code haut et bas niveau (FFI)

Procédures de décision intégrées à Coq : F. Besson
$\rightarrow$ procédures vérifiées d'analyse automatique de complexité

Sémantiques mécanisées passant à l'échelle : T. Jensen, A. Schmitt
$\rightarrow$ sémantiques pour machines à capabilités réalistes adaptées à la preuve de programmes

Intégration dans Celtique

Compilation vérifiée et sécurité (CompCert+SFI/non-interférence...) :
F. Besson, S. Blazy, D. Demange, T. Jensen, D. Pichardie
$\rightarrow$ compilation vers capabilités

Compilation "*just-in-time*" vérifiée : S. Blazy, D. Pichardie
$\rightarrow$ interoperabilité entre code haut et bas niveau (FFI)

Procédures de décision intégrées à Coq : F. Besson
$\rightarrow$ procédures vérifiées d'analyse automatique de complexité

Sémantiques mécanisées passant à l'échelle : T. Jensen, A. Schmitt
$\rightarrow$ sémantiques pour machines à capabilités réalistes adaptées à la preuve de programmes

J'apporte mes compétences en :

- logique de séparation
- preuve de complexité
- machines à capabilités

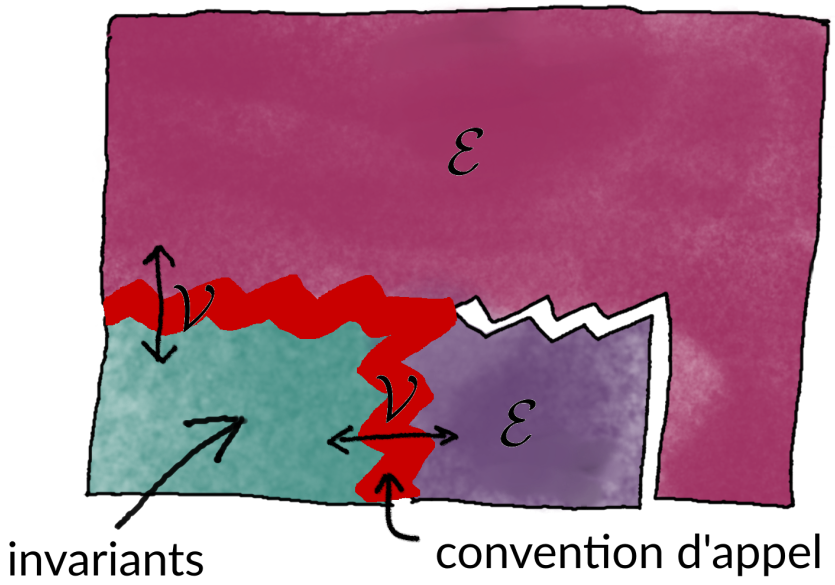
Publications en conférences

- L. Birkedal, T. Dinsdale-Young, A. Guéneau, G. Jaber, K. Svendsen, N. Tzevelekos : *Theorems for Free from Separation Logic Specifications*. **ICFP**, 2021. Nouveau !
- A. L. Georges, A. Guéneau, T. Van Strydonck, A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, L. Birkedal : *Efficient and provable local capability local capability revocation using uninitialized capabilities*. **POPL**, 2021.
- A. L. Georges, A. Guéneau, T. Van Strydonck, A. Timany, A. Trieu, D. Devriese, L. Birkedal : *Cap' ou pas cap'? Preuve de programmes pour une machine à capabilitus en présence de code inconnu*. JFLA, 2021.
- A. Guéneau, J.-H. Jourdan, A. Charguéraud, F. Pottier : *Formal proof and analysis of an incremental cycle detection algorithm*. **ITP**, 2019.
- A. Guéneau, A. Charguéraud, F. Pottier : *A fistful of dollars : formalizing asymptotic complexity claims via deductive program verification*. **ESOP**, 2018.
- A. Guéneau, M. Myreen, R. Kumar, M. Norrish : *Verified characteristic formulae for CakeML*. **ESOP**, 2017.

Productions logicielles

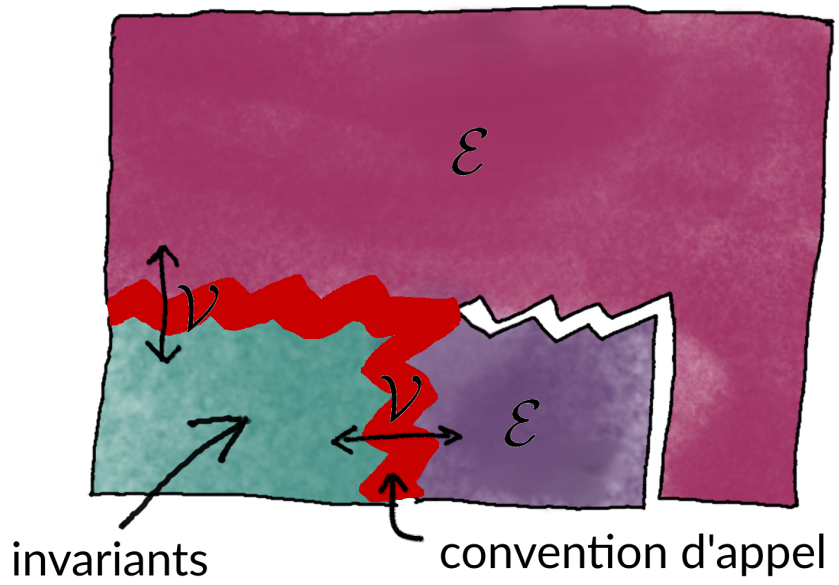
- *Bibliothèques Coq pour la vérification de complexité*. **Auteur principal.**
- *Algorithme formellement vérifié de détection incrémentale de cycles*. **Auteur principal.**
- *Outil pour la vérification de programmes CakeML en logique de séparation*. **Auteur principal**
- *Formalisation de propriétés de sécurité pour machines à capabilités*.
- *Contributions au compilateur OCaml*

Principes de raisonnement (POPL'21)



- Invariants 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

Principes de raisonnement (POPL'21)



- Invariants 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 : on a $\mathcal{E}(c)$ pour tout programme inconnu pointé par c.

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

Preuve : **12kLOC** (formalisation de la machine à capabilités) +
10kLOC (preuve du théorème)
$\simeq$ 1,5 personne.an de travail