Preuves de programmes : au-delà de la correction fonctionnelle

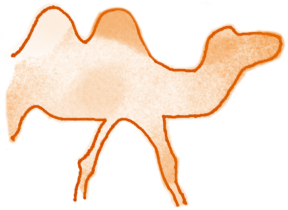
Armaël Guéneau

- 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

Équipes d'accueil :
Celtique (IRISA, Rennes), LMF (Orsay/Paris–Saclay), PPS (IRIF, Paris 7)

Mon domaine de recherche

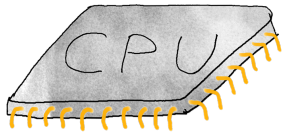
langages de programmation ∩ méthodes formelles



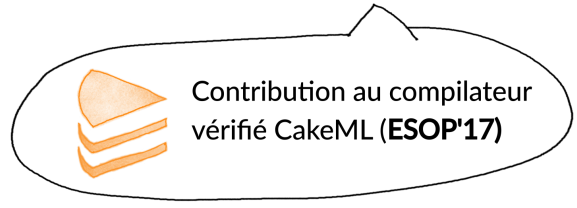
OCaml

membre de l'équipe de développement du compilateur

assistants de preuve: **Coq**, HOL4



langage machine



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)

Calculer le bon résultat, en consommant des **ressources raisonnables**
$\rightarrow$ attaques par déni de service
$\rightarrow$ consommation en "*gas*" dans les *smart contracts*

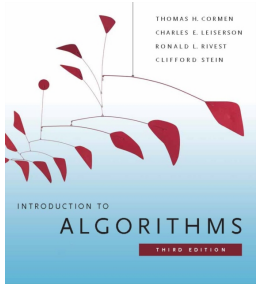
Dans ma thèse :

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 et al. (2015)

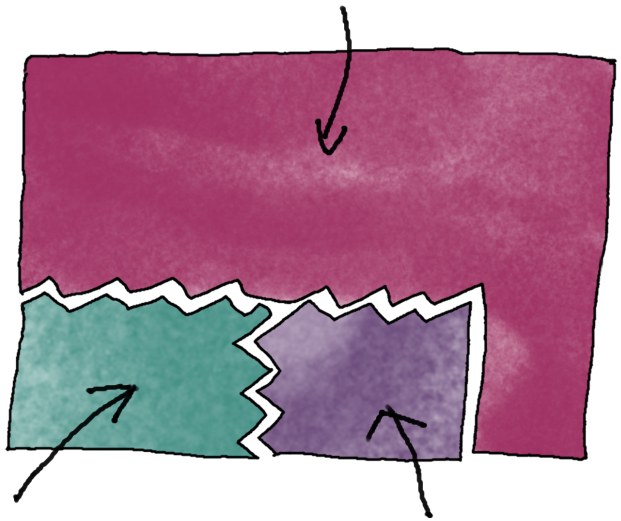
```
Theorem add_edge_spec : ∀g G v w,
  let m := card (edges G) in let n := card (vertices G) in
  v ∈ vertices G ∧ w ∈ vertices G ∧ ¬ has_edge G v w →
  Spec add_edge_or_detect_cycle [g v w]
    PRE (IsGraph g G ∗ $(ψ (m+1) n − ψ m n))
    POST (fun res ⇒ match res with
          | EdgeAdded ⇒ IsGraph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

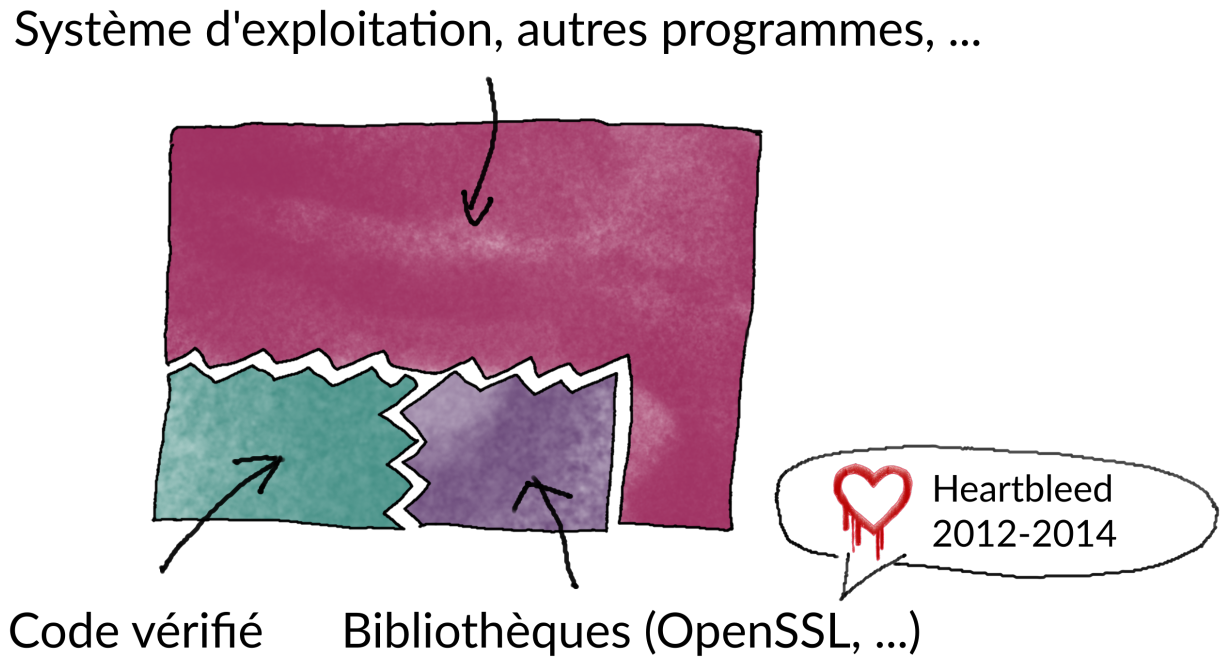
Sécuriser les interactions entre code connu et inconnu à l'aide de capabilités (postdoc)

Système d'exploitation, autres programmes, ...

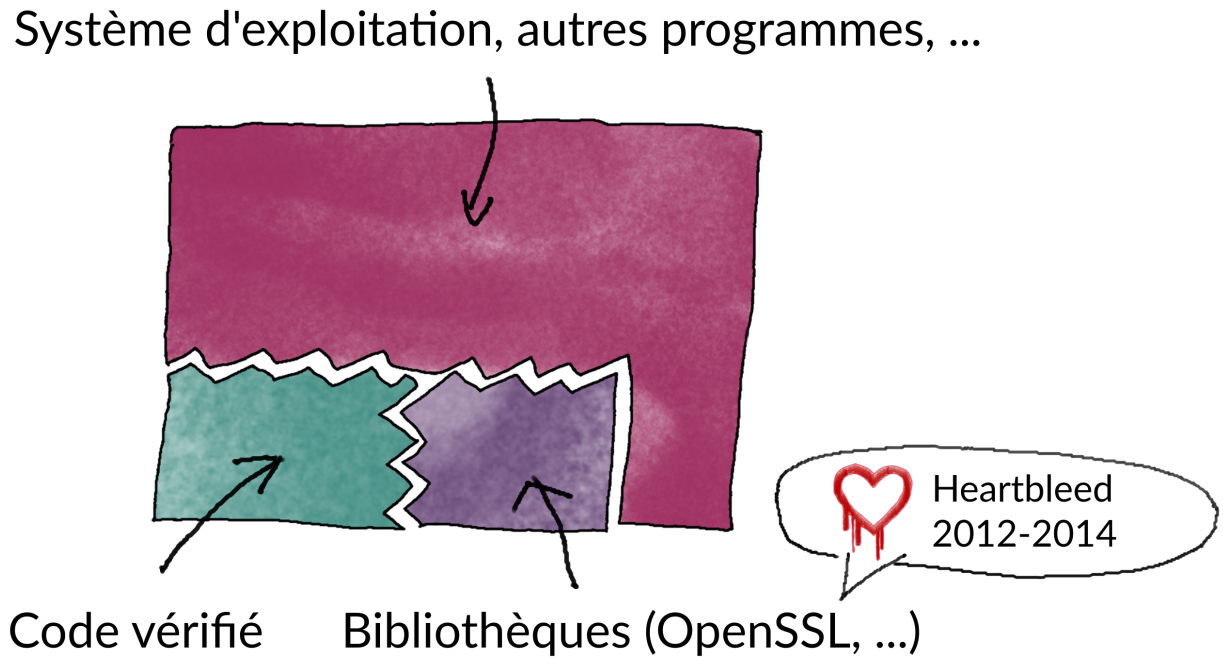


Code vérifié Bibliothèques (OpenSSL, ...)

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



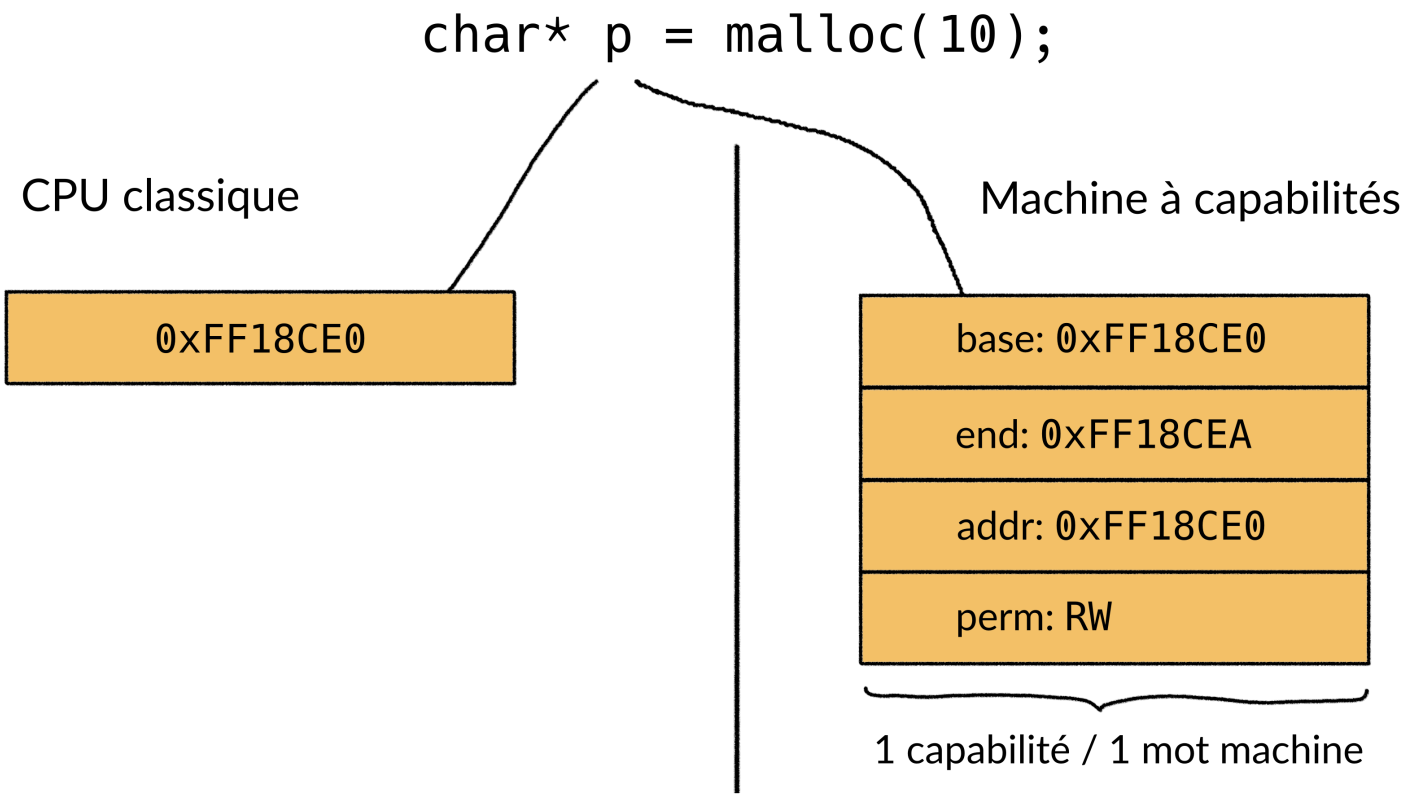
Sécuriser les interactions entre code connu et inconnu à l'aide de capabilités (postdoc)



- Quels mécanismes pour sécuriser les interactions ?
  $\rightarrow$ différentes approches (sandboxing, software fault isolation, ...)
  $\rightarrow$ on s'intéresse ici à l'utilisation de *capabilités matérielles*
- Comment raisonner sur du code interagissant avec du code inconnu ?
  $\rightarrow$ logique de séparation expressive embarquée dans Coq

Les machines à capabilités à la rescousse

Capabilité = pointeur + métadonnées

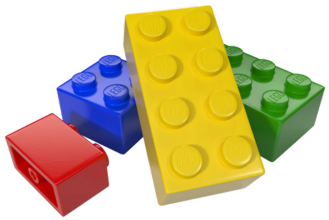


Les bornes et permissions sont **vérifiées par la machine**.

Des capabilités aux garanties de sécurité

bound checks

locality control



encapsulation

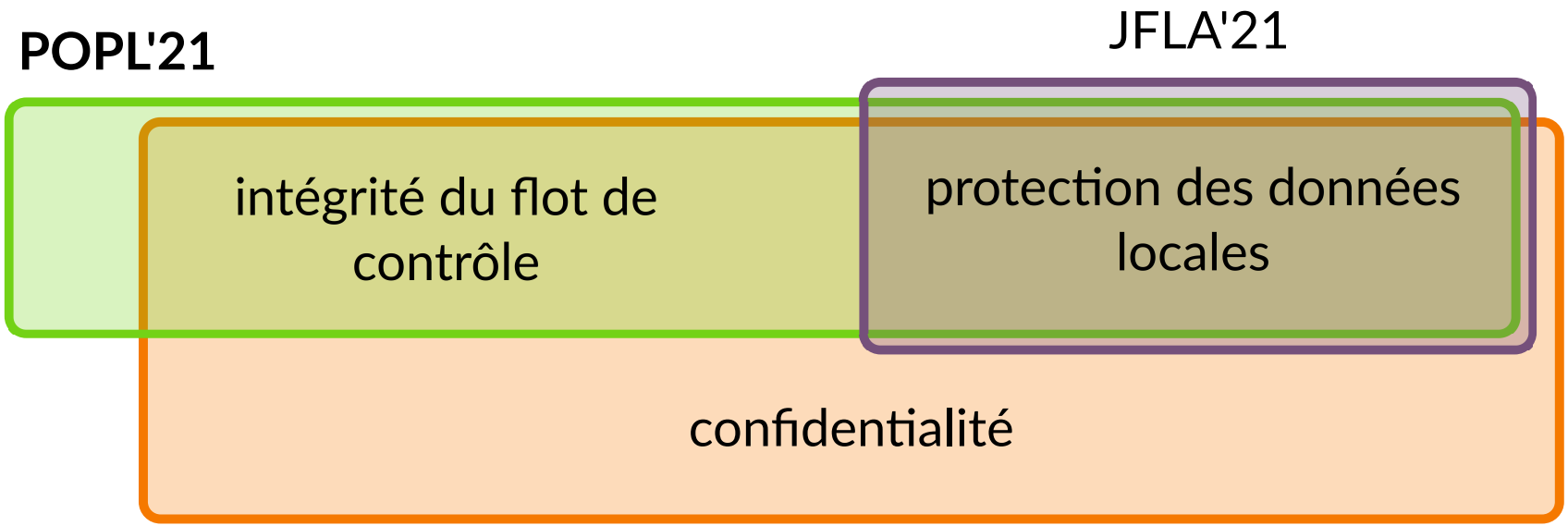
sémantique des
instructions machine

Ce travail

propriétés de sécurité
haut niveau

principes de preuve de programmes
en présence de code inconnu

Contributions

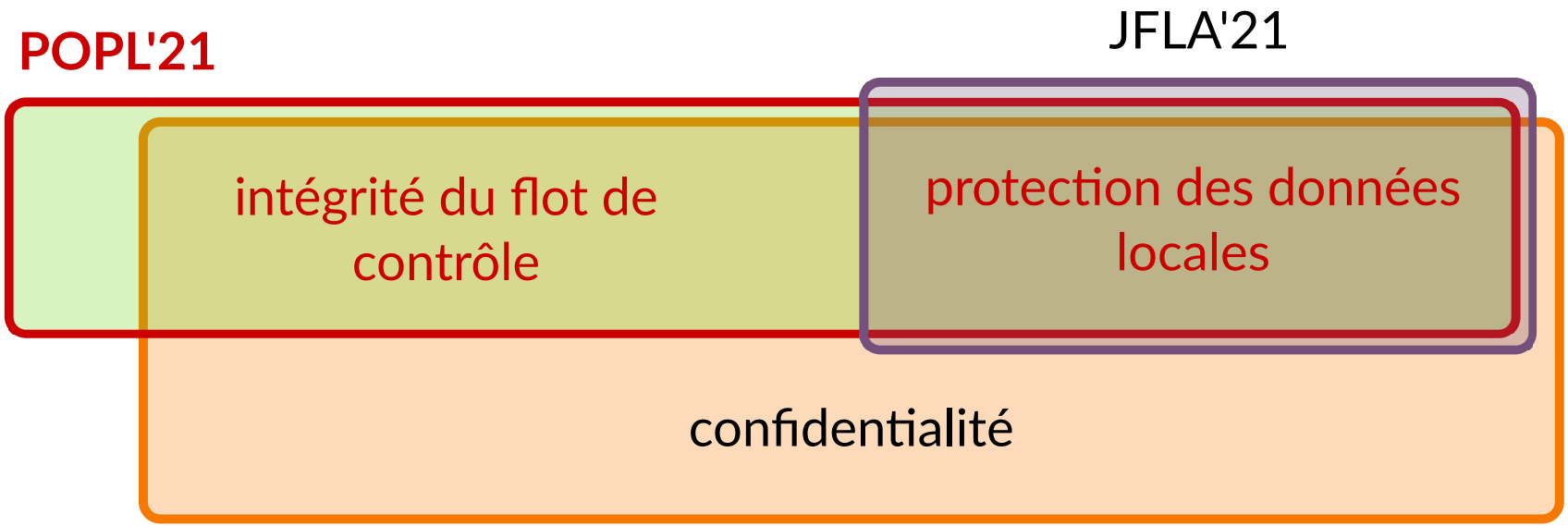


PriSC'21 + travail en cours

propriétés de trace sur les entrées/sorties

soumission à CSF'21

Contributions



PriSC'21 + travail en cours

propriétés de trace sur les entrées/sorties

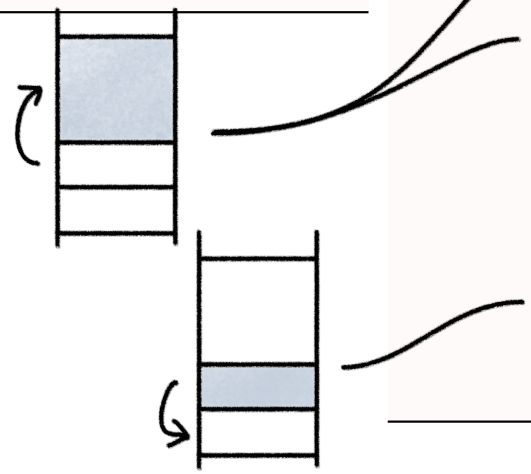
soumission à CSF'21

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);
}
```

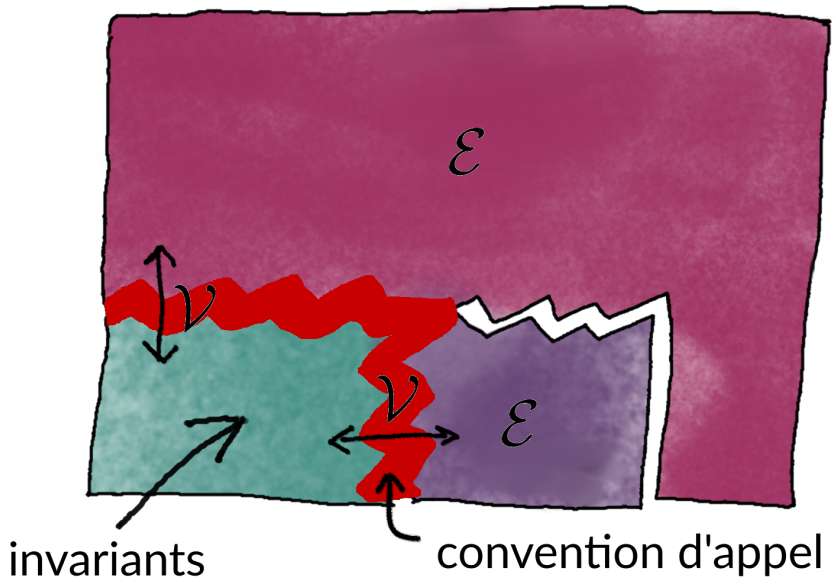
```
g1: malloc r2 1
    store r2 0
    move r3 pc
    lea r3 offset
    crtcls [(x, r2)] r3
    jmp r0
f1: reqglob r_adv
    prepstack r_stk
    store r_env 0
    scallU r_adv ([], [r0, r_adv, r_env])
    store r_env 1
    scallU r_adv ([], [r0, r_env])
    load r_adv r_env
    assert r_adv 1
    getb 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

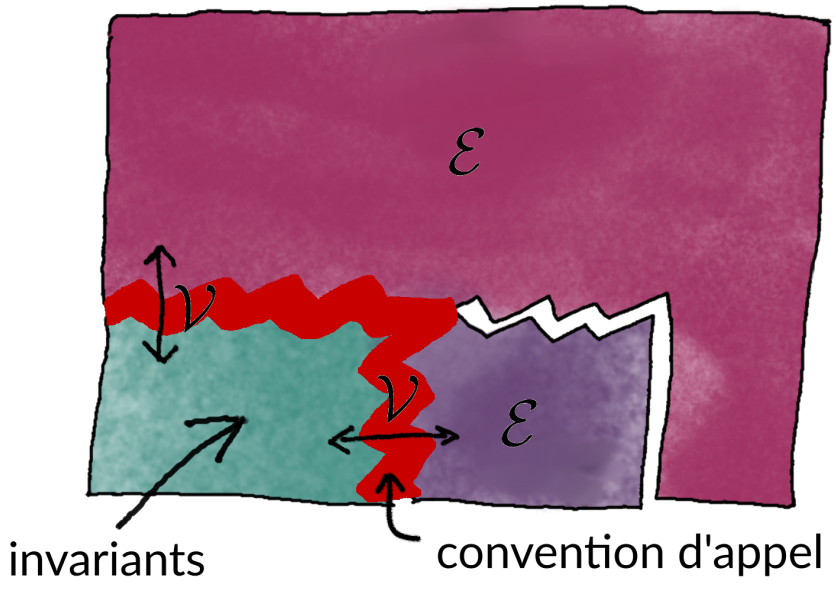
Preuve : 5kLOC (convention d'appel), 5kLOC (preuve de l'exemple)

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)
$\simeq$ 6 mois de travail pour 4 personnes

Projet : preuves de programmes en contexte (1)

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

Axe 1 : capabilités

- composants système sûrs vérifiés : allocateur mémoire, ordonnanceur
- **compilation vers machines à capabilités**

Axe 2 : analyse de ressources

- **intégration avec des outils automatisés** d'analyse de ressources
- preuve de complexité en espace

Projet : preuves de programmes en contexte (2)

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

Cadre : langage statiquement typé (par exemple : OCaml)

$\rightarrow$ 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**.

Projet : preuves de programmes en contexte (3)

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

Cadre : interopérabilité entre un langage haut niveau avec GC (OCaml) et un langage 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.

Intégration

Équipe Celtique, IRISA à Rennes (UMR 6074)

- Compilation vérifiée et sécurité (CompCert + SFI, non-interférence) :
  F. Besson, D. Demange, S. Blazy, T. Jensen, D. Pichardie
- Sémantiques mécanisées : T. Jensen, A. Schmitt

LMF à Orsay/Paris–Saclay (UMR 9021)

- Preuve de programmes :
  - Why3 : J.-C. Filliâtre, C. Marché, G. Melquiond, A. Paskevich
  - Coq : S. Boldo, E. Contejean, J.-H. Jourdan, G. Melquiond
- Intégration Coq et outils de preuve automatisés :
  G. Dowek (Dedukti), C. Keller (SMTCoq)

Équipe PPS, IRIF à Paris 7 (UMR 8243)

- Développement de Coq : H. Herbelin
- Évolution robuste de composants logiciels OCaml :
  R. Treinen, H. Férée