

# 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  $\cap$  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**

→ sinon : attaques par déni de service

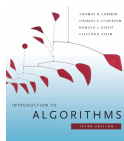
**Dans ma thèse :**

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”)



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 :  $\forall$ g G v w,  
  let m := card (edges G) in let n := card (vertices G) in  
  v  $\in$  vertices G  $\wedge$  w  $\in$  vertices G  $\wedge$   $\neg$  has_edge G v w  $\rightarrow$   
  Spec add_edge_or_detect_cycle [g v w]  
  PRE (IsGraph g G *  $\$(\psi (m+1) n - \psi m n)$ )  
  POST (fun res  $\Rightarrow$  match res with  
    | EdgeAdded  $\Rightarrow$  IsGraph g (G  $\uplus$  (v, w))  
    | EdgeCreatesCycle  $\Rightarrow$  [rtclosure (has_edge G) w v ]  
  end).
```

(\* complexité amortie  $\psi \in O(m \min(\sqrt{m}, n^{2/3}) + n)$  \*)

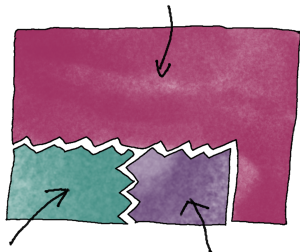
```
Theorem bound_ψ :  
  dominated _ ψ (fun m n  $\Rightarrow$  m * min (sqrt_up m) (Int_part (IZR n ^ (2/3)))) + n).
```

- Meilleure borne de complexité connue pour des graphes épars
- Mon implémentation vérifiée est utilisée dans le *build system* d'une  
→ 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 capacités (postdoc)

Système d'exploitation, autres programmes, ...

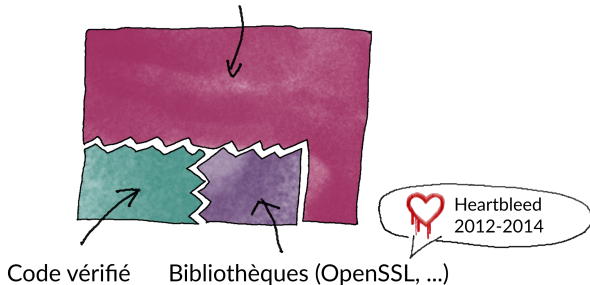


Code vérifié

Bibliothèques (OpenSSL, ...)

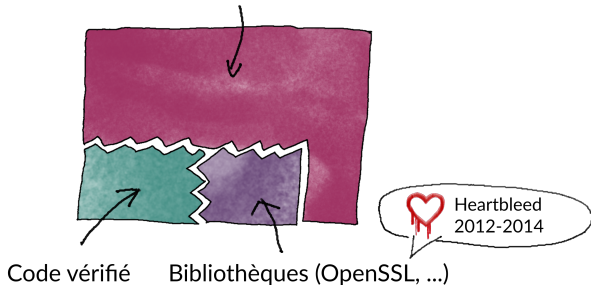
# Sécuriser les interactions entre code connu et inconnu à l'aide de capacités (postdoc)

Système d'exploitation, autres programmes, ...



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

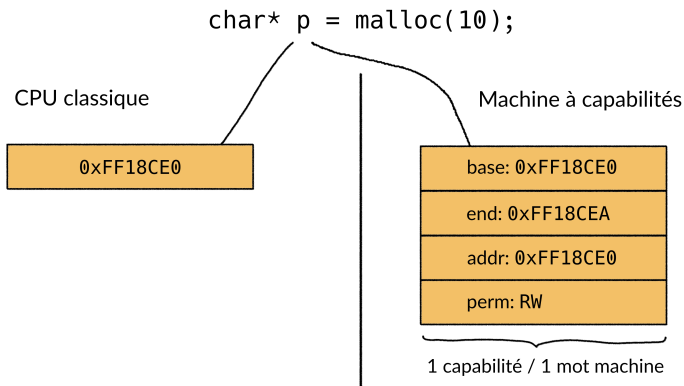
Système d'exploitation, autres programmes, ...



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

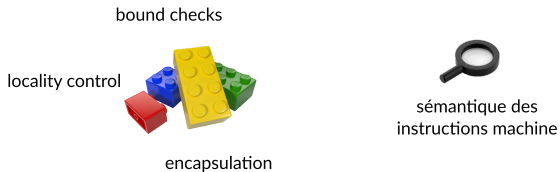
# Les machines à capacités à la rescousse

Capabilité = pointeur + métadonnées



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

# Des capacités aux garanties de sécurité



Quelles garanties de sécurité obtient-on ? Ça dépend !

# Des capacités aux garanties de sécurité

bound checks

locality control



encapsulation



sémantique des  
instructions machine

Quelles garanties de sécurité obtient-on ? Ça dépend !

**Dans ce travail, développement de :**



propriétés de sécurité  
haut niveau



principes de preuve de programmes  
en présence de code inconnu

# Contributions

POPL'21

JFLA'21

intégrité du flot de  
contrôle

protection des données  
locales

confidentialité

PriSC'21 + travail en cours

propriétés de trace sur  
les entrées/sorties

soumission à CSF'21

# Contributions

POPL'21

JFLA'21

intégrité du flot de  
contrôle

protection des données  
locales

confidentialité

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);
}
```

---

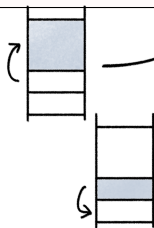
## Garanties de sécurité obtenues : exemple (POPL'21)

```
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
```

400 instructions après  
dépliage des macros

```
f1: reqglob radv  
    prepstack rstk  
    store renv 0  
    scallU radv ([], [r0, radv, renv])  
    store renv 1  
    scallU radv ([], [r0, renv])  
    load radv renv  
    assert radv 1  
    getb r1 rstk  
    add r2 r1 10  
    subseg rstk r1 r2  
    mclear rstk  
    rclear Regs\{pc, r0}  
    jmp r0
```



## 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);
}
```

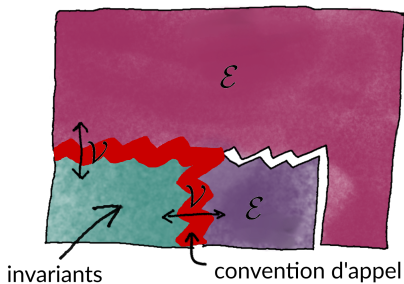
```
g1: malloc r2 1
    store r2 0
    move r3 pc
    lea r3 offset
    crtcls [(x, r2)] r3
    jmp r0
f1: reqglob radv
    prepstack rstk
    store renv 0
    scallU radv ([], [r0, radv, renv])
    store renv 1
    scallU radv ([], [r0, renv])
    load radv renv
    assert radv 1
    getb r1 rstk
 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

400 instructions après  
dépliage des macros

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

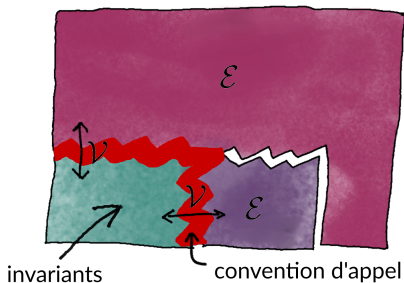
Preuve : 5kLOC (convention d'appel), 5kLOC (preuve de l'exemple)

## 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  $c$ .

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

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

# Projet : preuves de programmes en contexte (1)

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

Axe 1 : capacités

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

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 3 : vérification par composants dans un langage statiquement typé  
(par exemple : OCaml)

## Projet : preuves de programmes en contexte (2)

Axe 3 : vérification par composants dans un langage statiquement typé  
(par exemple : OCaml)

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**.



## 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)

→ 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)

- Preuves et langages :
  - 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

## Publications en conférences

- 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 capabilités 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*