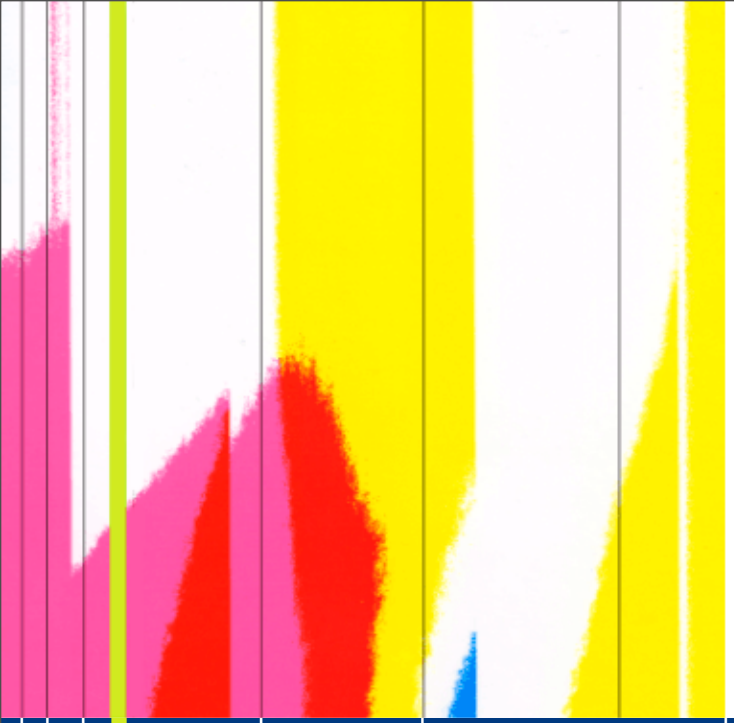


Les Codes Porteurs de Preuve

David Pichardie
équipe-projet Lande

16 Décembre 2008

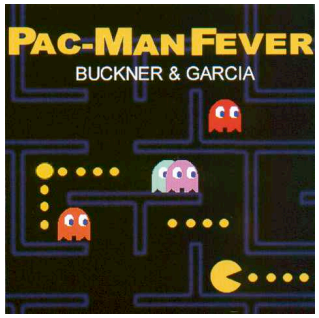
INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

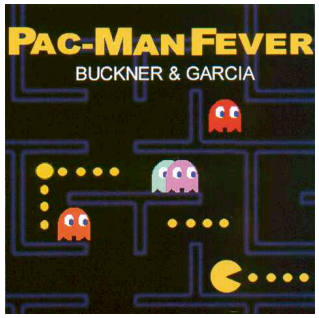
centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Introduction aux codes porteurs de preuve

Les dilemmes du code mobile



Les dilemmes du code mobile



Aucun risque ?

Code inconnu

Système hôte

Le code inconnu pourrait nuire au système hôte

- corruption des structures internes

Le code inconnu pourrait utiliser beaucoup de ressources

- calcul, mémoire, SMS...

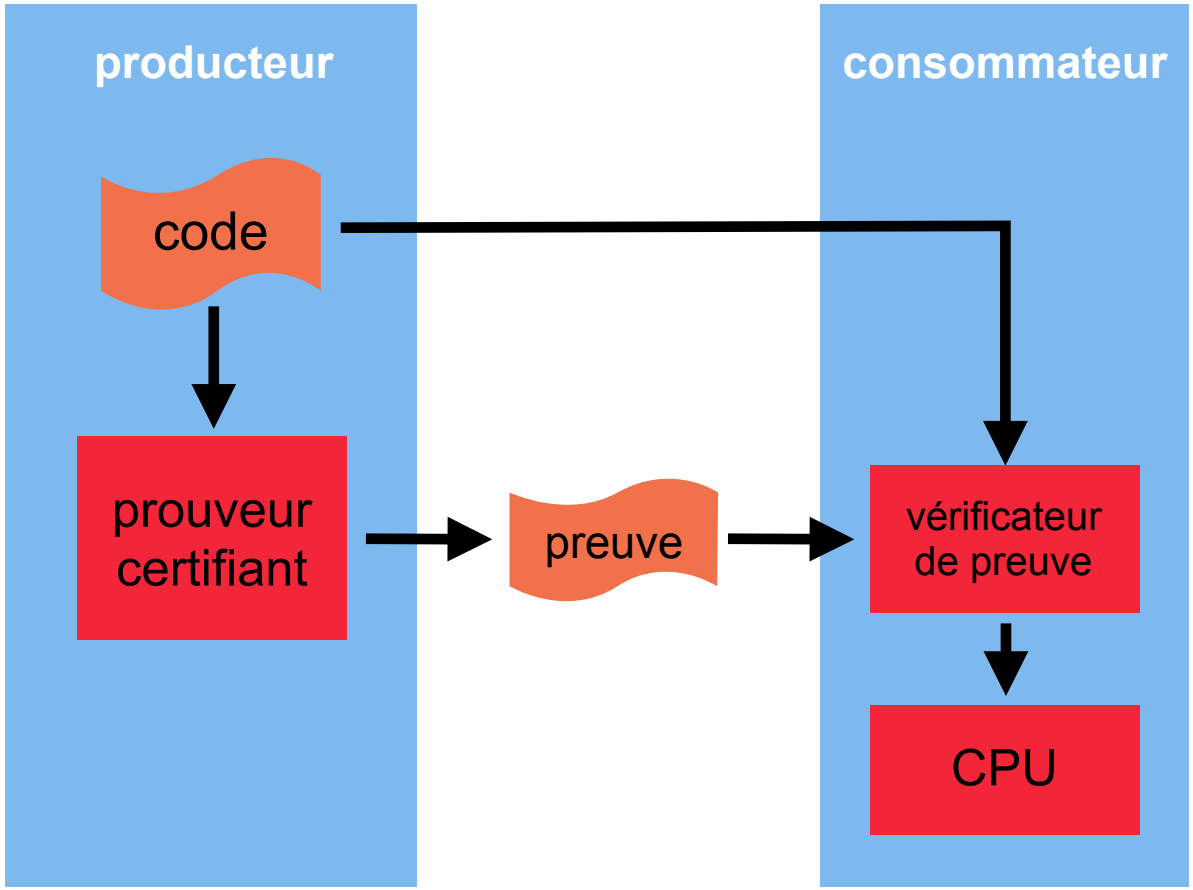
Le code inconnu pourrait révéler des données confidentielles

- listes des contacts, agenda, géo-localisation, caméra, micro...

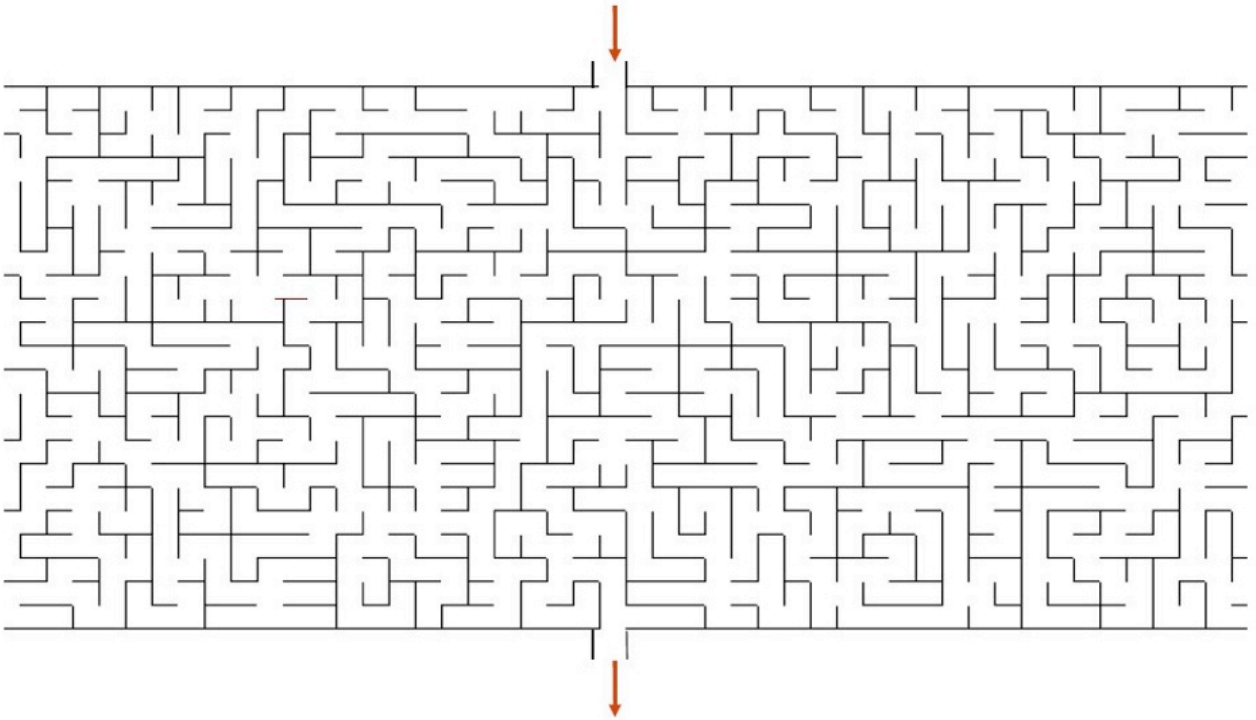
Le code porteur de preuve

Proof Carrying Code (PCC)

- Le code est transmis avec un certificat (preuve) de sûreté.
- Le certificat se suffit à lui même et est non falsifiable.
- Vérifier le certificat est beaucoup plus facile que le créer.



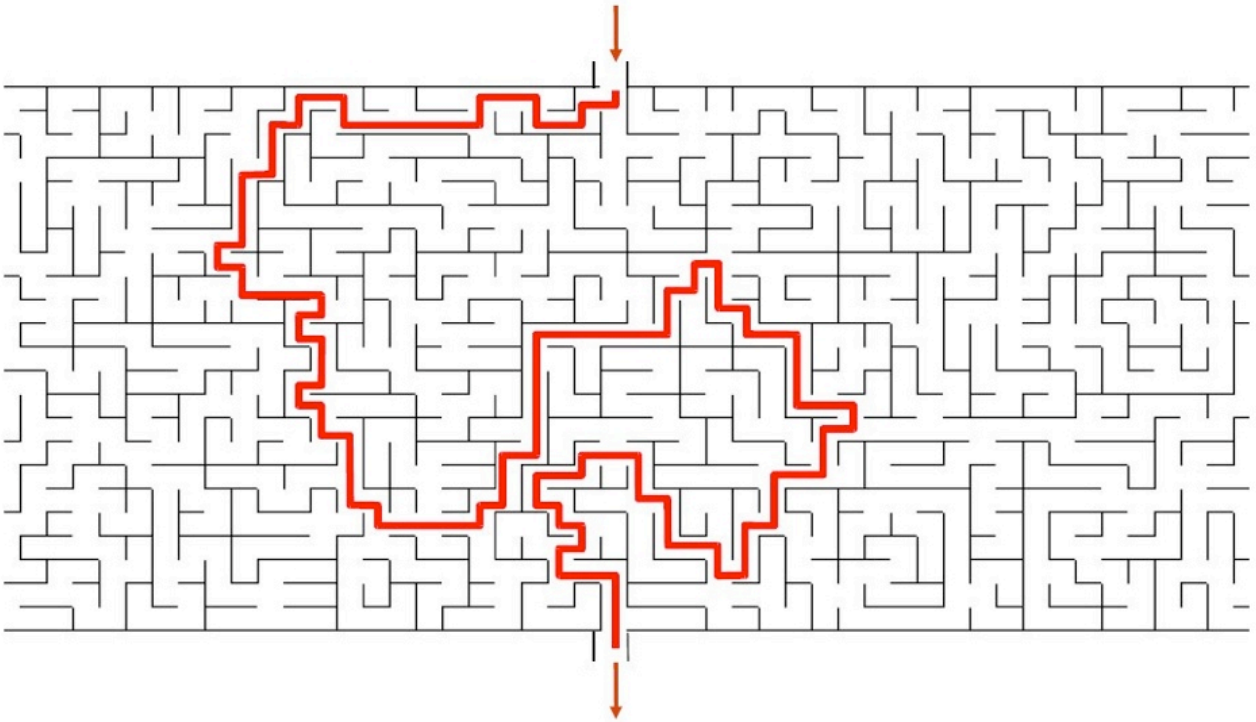
La métaphore du labyrinthe



programme = labyrinthe

©G. Necula

La métaphore du labyrinthe

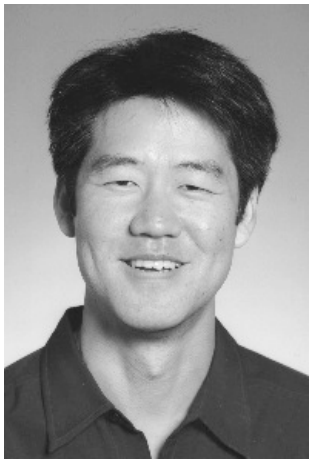


programme = labyrinthe

preuve = chemin rouge

©G. Necula

Les pionniers du PCC



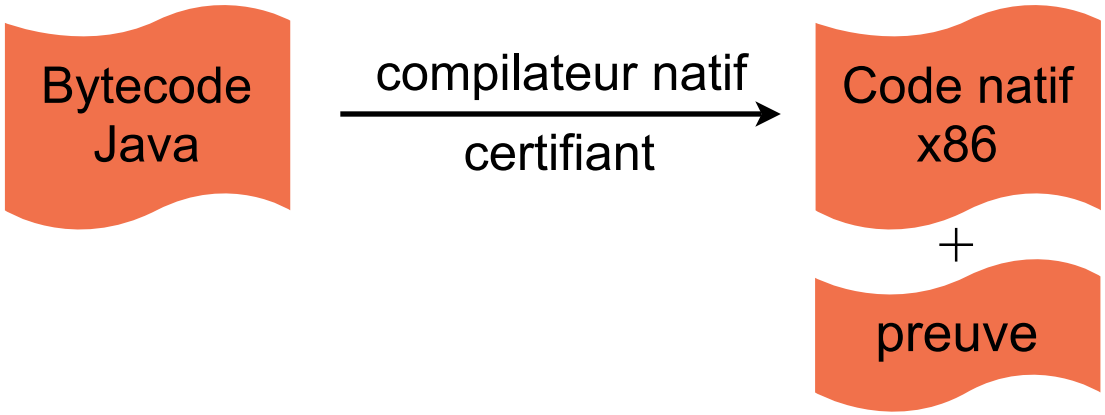
D’abord proposé par Georges Necula (Berkley) et Peter Lee (CMU).

- Necula & Lee, *Safe Kernel Extensions Without Run-Time Checking*, OSDI'96
- Necula, *Proof-Carrying Code*, POPL'97
- Necula & Lee, *The Design and Implementation of a Certifying Compiler*, PLDI'98
- Necula, *Compiling with Proofs*, Phd thesis, 1998

Un exemple de succès du PCC : *SpecialJ*

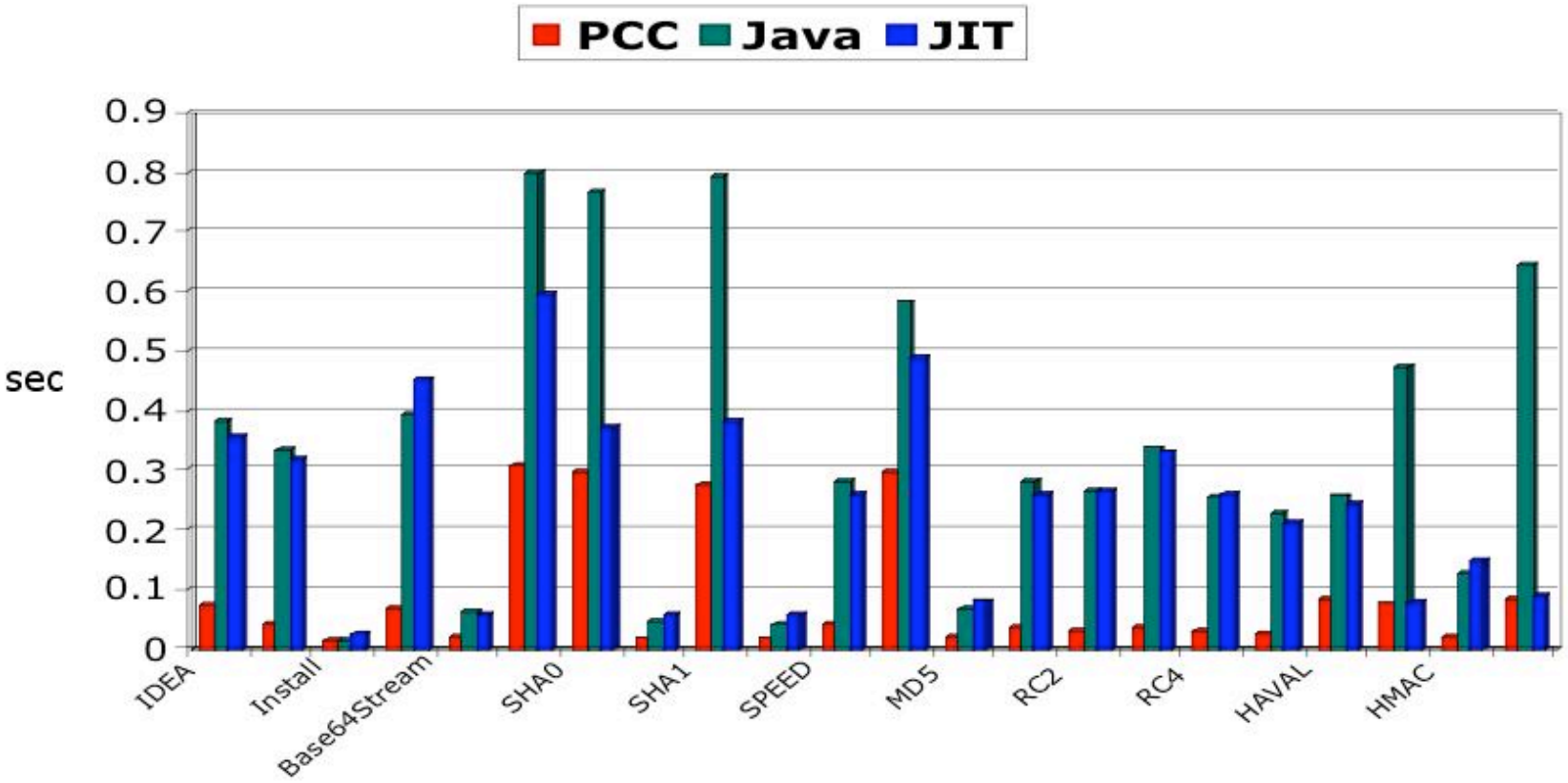
C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko and K. Cline, *A certifying compiler for Java*,PLDI'00.

Un exemple de succès du PCC : *SpecialJ*



C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko and K. Cline, *A certifying compiler for Java*,PLDI'00.

Un exemple de succès du PCC : *SpecialJ*

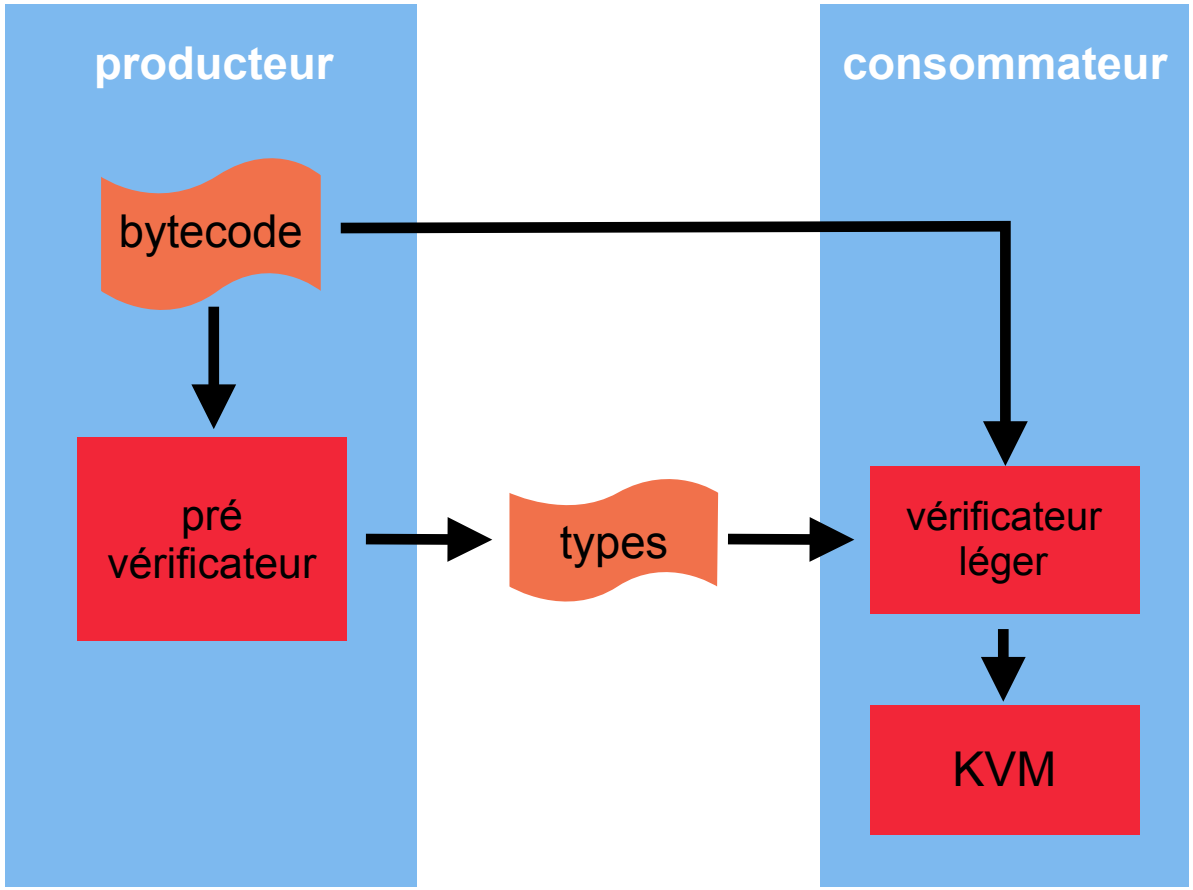


C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko and K. Cline, *A certifying compiler for Java*,PLDI'00.

Le PCC dans la vraie vie

Les machines virtuelles sur téléphone (KVM) suivent déjà cette approche

- La politique de sécurité est la sûreté mémoire standard de Java.
- Les certificats sont des annotations de type.
- La vérification de type est rapide, simple, et consomme peu de mémoire.
- Même approche pour Java SE 1.6.



Conclusions intermédiaires sur le PCC standard

- Le PCC propose un mélangeant étonnant de logique mathématique, vérification de programmes et des problèmes de sécurité concrets.
- Le PCC a surtout été utilisé pour des politiques de sécurité de type *sûreté mémoire*.
- Le PCC doit encore démontrer sa capacité à assurer des politiques de sécurité plus complexes, tout en conciliant de nombreuses qualités :
  - des petits certificats,
  - un vérificateur efficace et léger,
  - un vérificateur correct,
  - des outils effectifs pour générer des certificats,
  - une véritable intégration dans l'informatique ubiquitaire de demain.

PCC : recherches en cours

Le projet européen MOBIUS

http://mobius.inria.fr

- PCC pour le code mobile Java,
- Des politiques de sécurité au-delà de la sûreté mémoire
  - Analyse des flux d'information: les sorties publiques ne doivent pas dépendre des données confidentielles.
  - Analyse des usage de ressource : mémoire, temps, actions payantes...
  - Propriétés fonctionnelles: les compilateurs transformateurs de preuve.
- PCC certifié
  - La correction du vérificateur PCC est prouvée en Coq.

Le projet européen MOBIUS

http://mobius.inria.fr

- PCC pour le code mobile Java,
- Des politiques de sécurité au-delà de la sûreté mémoire
  - Analyse des flux d'information: les sorties publiques ne doivent pas dépendre des données confidentielles.
  - Analyse des usage de ressource : mémoire, temps, actions payantes...
  - Propriétés fonctionnelles: les compilateurs transformateurs de preuve.
- PCC certifié
  - La correction du vérificateur PCC est prouvée en Coq.

Coq : un animal à deux visages

http://coq.inria.fr

Coq : un animal à deux visages

http://coq.inria.fr

Premier visage :

- un assistant de preuve qui permet de construire, de manière interactive, une preuve formelle en logique constructive

Coq : un animal à deux visages

http://coq.inria.fr

Premier visage :

- un assistant de preuve qui permet de construire, de manière interactive, une preuve formelle en logique constructive

Second visage :

- un langage de programmation fonctionnelle muni d'un système de type très expressif
  exemple :

$$\texttt{sort} : \forall l : \texttt{list int}, \{\, l' : \texttt{list int} \mid \texttt{Sorted}(l') \wedge \texttt{Permutation}(l, l') \,\}$$

Coq : un animal à deux visages

http://coq.inria.fr

Premier visage :

- un assistant de preuve qui permet de construire, de manière interactive, une preuve formelle en logique constructive

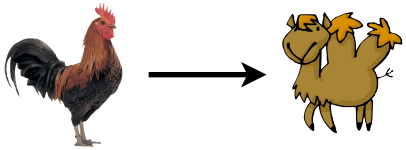
Second visage :

- un langage de programmation fonctionnelle muni d'un système de type très expressif
  exemple :

  $\texttt{sort} : \forall l : \texttt{list int}, \{ l' : \texttt{list int} \mid \texttt{Sorted}(l') \wedge \texttt{Permutation}(l, l') \}$

- et muni d'un mécanisme d'extraction vers Ocaml

  $\texttt{sort} : \texttt{list int} \to \texttt{list int}$



Coq : un animal à deux visages

http://coq.inria.fr

Premier visage :

- un assistant de preuve qui permet de construire, de manière interactive, une preuve formelle en logique constructive

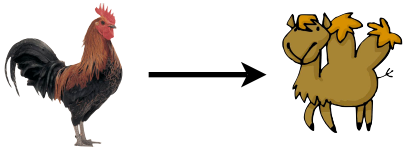
Second visage :

- un langage de programmation fonctionnelle muni d'un système de type très expressif
  exemple :

  $\texttt{sort} : \forall l : \texttt{list int}, \{ l' : \texttt{list int} \mid \texttt{Sorted}(l') \wedge \texttt{Permutation}(l, l') \}$

- et muni d'un mécanisme d'extraction vers Ocaml

  $\texttt{sort} : \texttt{list int} \to \texttt{list int}$

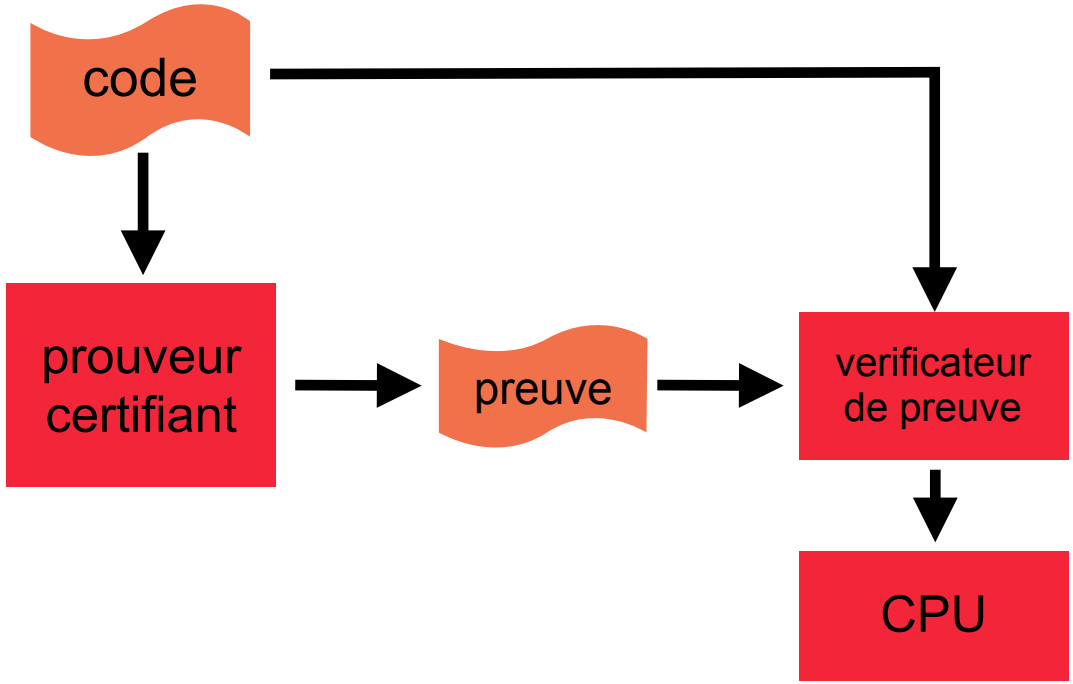


Ainsi, Coq permet à la fois de programmer, prouver correct et exécuter des programmes fonctionnels efficaces.

La base de confiance

Trusted Computing Base (TCB)

La base de confiance d'un système est composée de l'ensemble minimal des éléments dont la fiabilité est suffisante pour assurer la fiabilité globale du système.

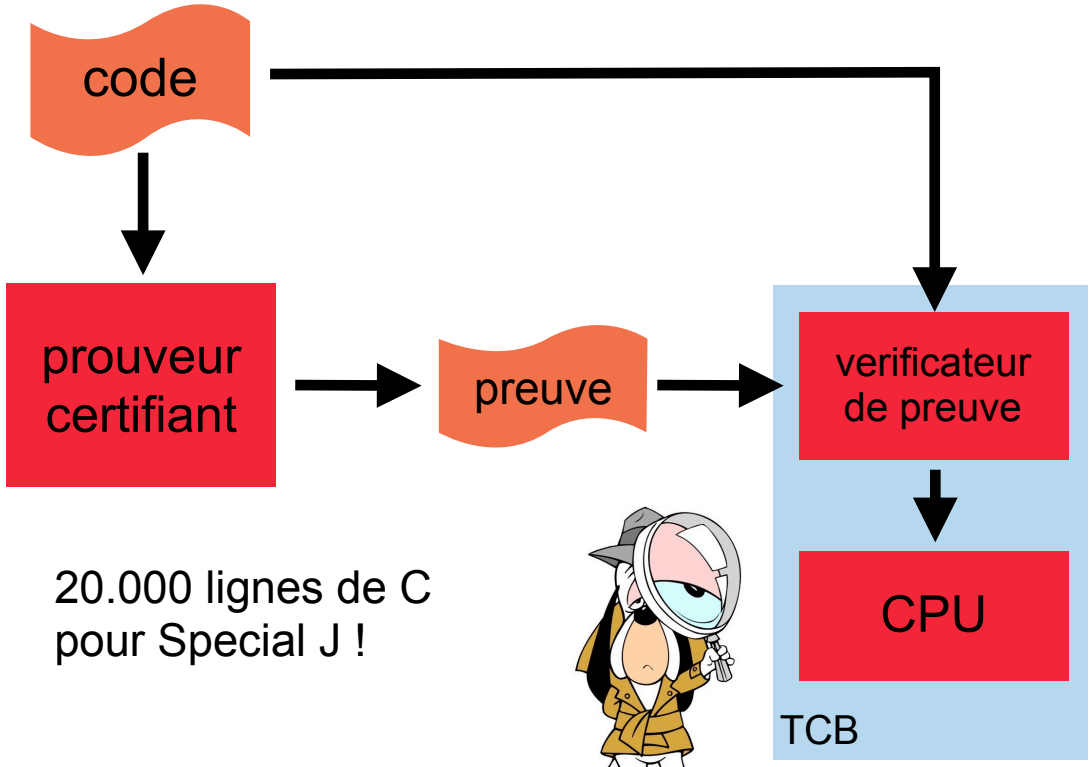


La base de confiance

Trusted Computing Base (TCB)

La base de confiance d'un système est composée de l'ensemble minimal des éléments dont la fiabilité est suffisante pour assurer la fiabilité globale du système.

- En PCC standard, la TCB comprend : le vérificateur de preuve + le système de calcul lui même.



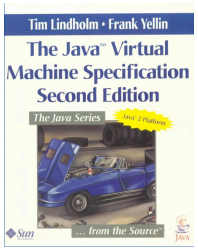
20.000 lignes de C pour Special J !

La base de confiance

Trusted Computing Base (TCB)

La base de confiance d'un système est composée de l'ensemble minimal des éléments dont la fiabilité est suffisante pour assurer la fiabilité globale du système.

- En PCC standard, la TCB comprend : le vérificateur de preuve + le système de calcul lui même.
- En PCC certifié, la TCB comprend : Coq + une spécification formelle de la correction du vérificateur + le système de calcul.



5.000 lignes de Coq pour la JVM

PCC certifié

PCC certifié

vérificateur de preuve

Contient une preuve de correction Coq du vérificateur à installer

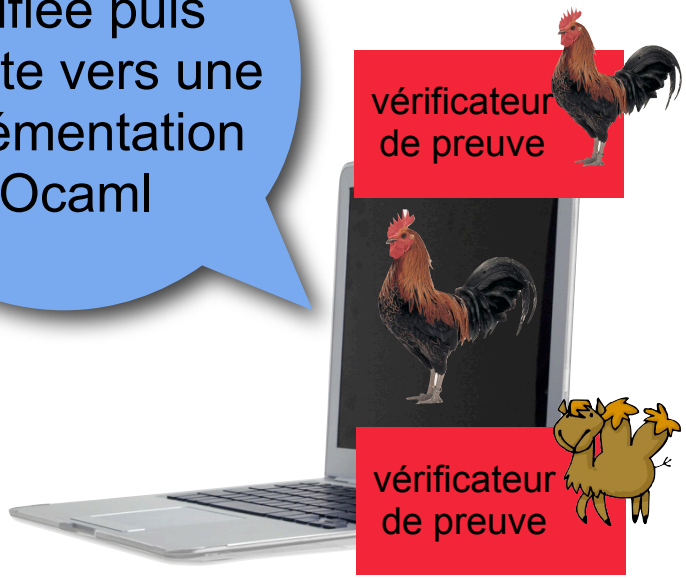
PCC certifié

PCC certifié

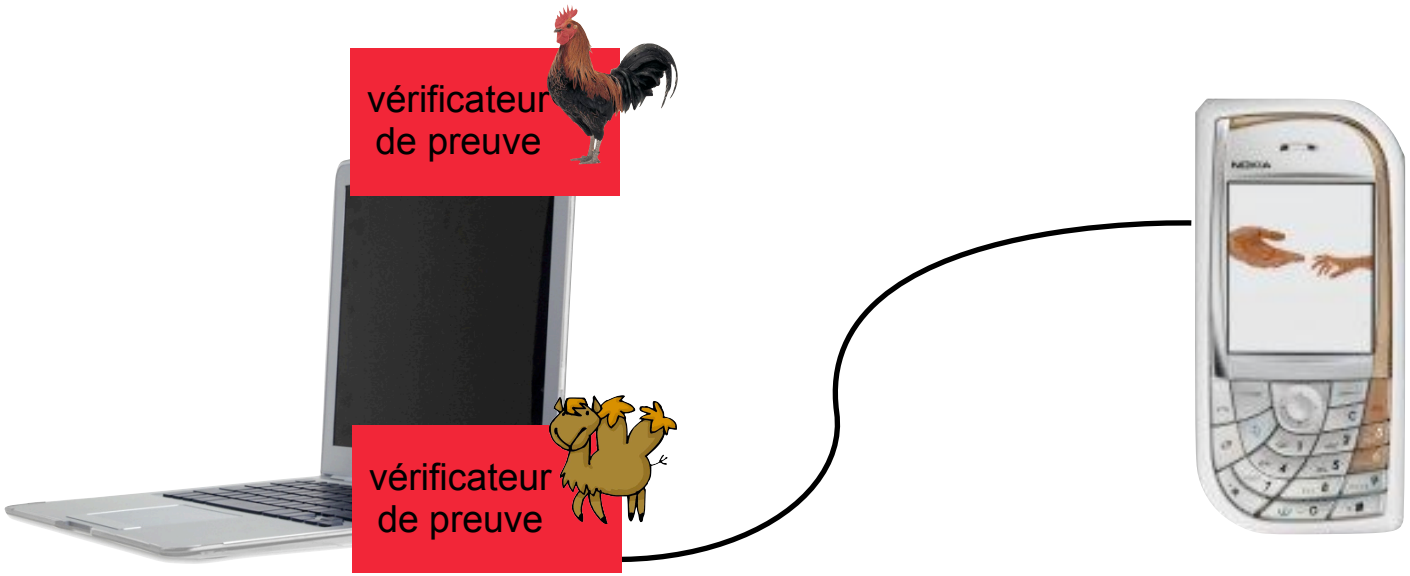
La preuve est vérifiée puis extraite vers une implémentation Ocaml

PCC certifié

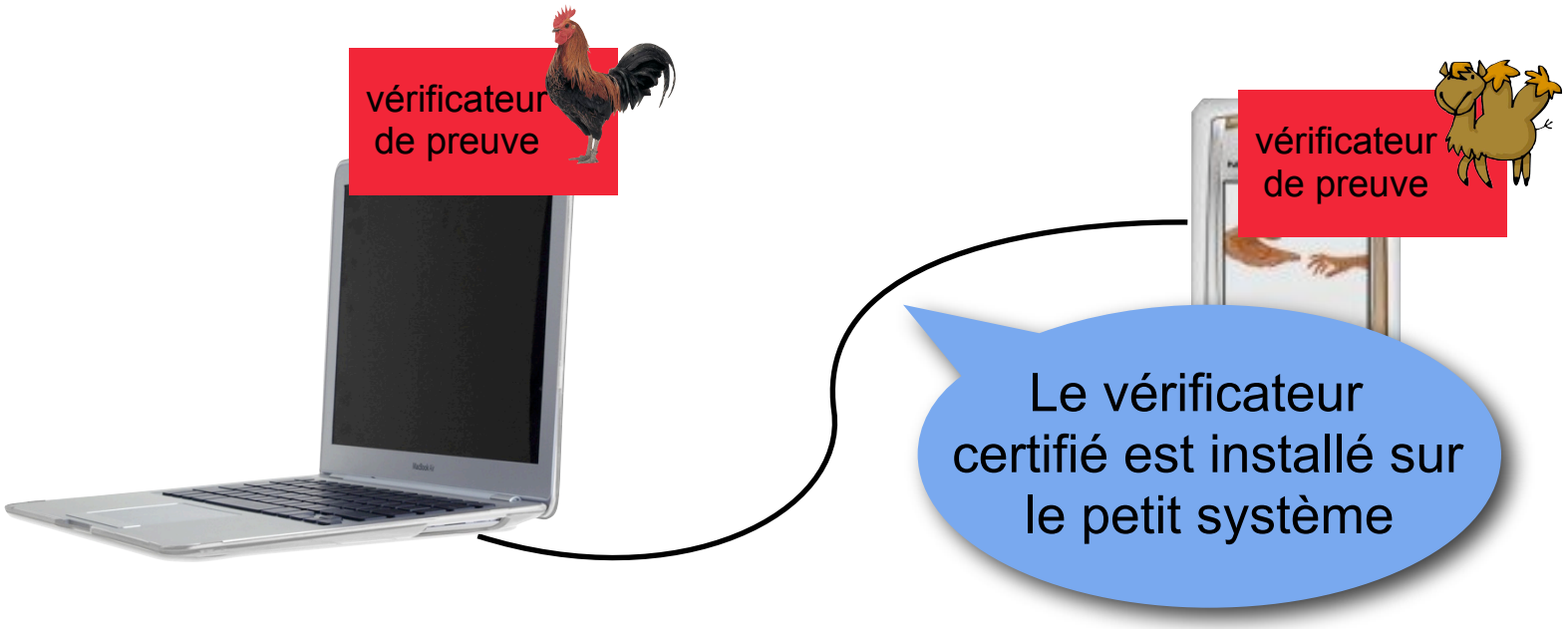
La preuve est vérifiée puis extraite vers une implémentation Ocaml



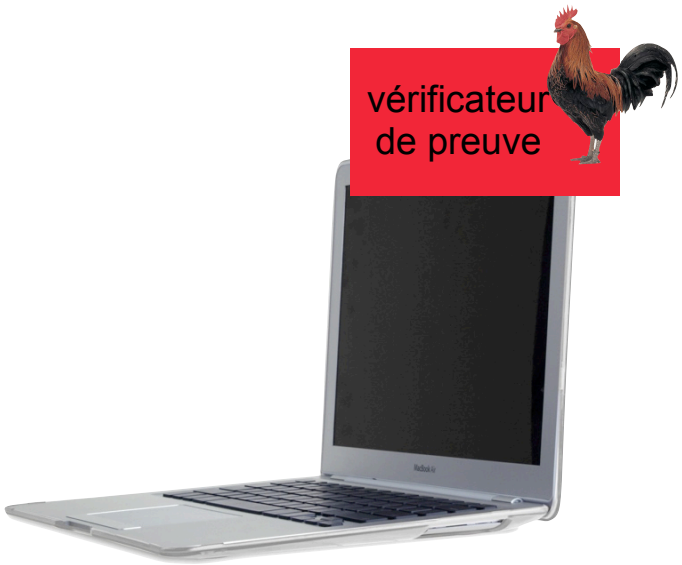
PCC certifié



PCC certifié



PCC certifié



PCC certifié

PCC certifié

PCC certifié

PCC certifié

PCC certifié

PCC certifié

Un seul vérificateur pour plusieurs codes mobiles...

Motivations pour le PCC certifié

Une analyse statique est un outil complexe

- le vérifieur standard du bytecode Java a nécessité plusieurs travaux de formalisation pointues
- les derniers travaux utilisaient des assistants de preuve (comme Coq)

Un vérificateur d'analyse statique est plus simple

- mais sa preuve de correction reste une activité difficile, réservée aux experts
- avec une preuve Coq,
  - seul le théorème démontré a besoin "d'être compris"
  - on certifie directement l'implémentation

Chaque nouvelle politique de sécurité demande

- un nouveau vérificateur
- une nouvelle preuve de correction
- une nouvelle expertise de la preuve de correction...etc

Des politiques de sécurité au-delà de la sûreté mémoire

- Analyse des flux d'informations

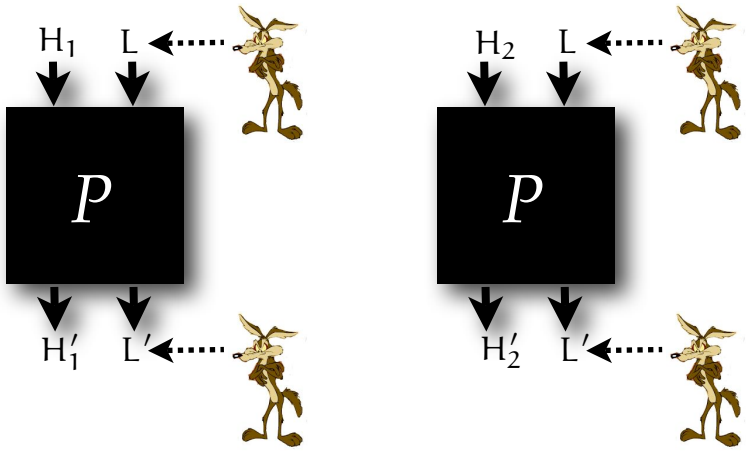
  G. Barthe, D. Pichardie and T. Rezk, A Certified Lightweight Non-Interference Java Bytecode Verifier, ESOP'07

Analyse des flux d'informations

Le programmeur marque les données avec des niveaux de confidentialité

high = confidentiel *low* = public

Un système de type assure qu'aucune sortie publique ne dépend d'une entrée confidentielle.



Le vérificateur de type est entièrement formalisé et prouvé correct en Coq, pour une JVM séquentielle avec classes, objets, tableaux, exceptions et appels virtuels.

Exemple de flux d'information

Flux explicite :

```
public int{L} foo(int{L} l; int{H} h) {
  return h;
}
```

Flux implicite :

```
public int{L} foo(int{L} l1; int{L} l2; int{H} h) {
  if (h==0) {return l1;} else {return l2;};
}
```

L'analyse des flux implicites nécessite une bonne compréhension des branchements d'un programme

- Les exceptions natives Java sont des sources (nombreuses) de branchement qu'il faut réussir à analyser statiquement.

Des politiques de sécurité au-delà de la sûreté mémoire

- Analyse des flux d'informations

Des politiques de sécurité au-delà de la sûreté mémoire

- Analyse des flux d'informations

  G. Barthe, D. Pichardie and T. Rezk, A Certified Lightweight Non-Interference Java Bytecode Verifier, ESOP'07

Des politiques de sécurité au-delà de la sûreté mémoire

- Analyse des flux d'informations

  G. Barthe, D. Pichardie and T. Rezk, A Certified Lightweight Non-Interference Java Bytecode Verifier, ESOP'07

- Analyse des pointeurs nulles

  L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. FMOODS'08.
  L. Hubert. A Non-Null annotation inferencer for Java bytecode. PASTE'08.
  http://nit.gforge.inria.fr

Des politiques de sécurité au-delà de la sûreté mémoire

- Analyse des flux d'informations

  G. Barthe, D. Pichardie and T. Rezk, A Certified Lightweight Non-Interference Java Bytecode Verifier, ESOP'07

- Analyse des pointeurs nulles

  L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. FMOODS'08.
  L. Hubert. A Non-Null annotation inferencer for Java bytecode. PASTE'08.
  http://nit.gforge.inria.fr

- Analyse des accès tableaux/ressources

  F. Besson, T. Jensen, D. Pichardie,T. Turpin

  - Nous développons une analyse polyédrique pour le bytecode Java.
  - Pour réduire la tailles des preuves, nous compressons les invariants.
  - Nous développons un format de preuve dédié aux polyèdres permettant une vérification rapide et certifiée.

```
static int bsearch(int key, int[] vec) { //PRE:
    //
    int low = 0, high = vec.length - 1;
    //
    while (low < high) {
        //
        int mid = low + (high - low) / 2;
        //
        if (key == vec[mid]) return mid;
        else if (key < vec[mid]) high = mid - 1;
        else low = mid + 1;
        //
        //
    }
    return -1; } // POST:
```

```
static int bsearch(int key, int[] vec) { //PRE: 0 ≤ |vec0|
    // key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ |vec0|
    int low = 0, high = vec.length - 1;
    // key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec0|
    while (low < high) {
        // key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low < high < |vec0|
        int mid = low + (high - low) / 2;
        // key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low
        // low + high − 1 ≤ 2 · mid ≤ low + hig
        if (key == vec[mid]) return mid;
        else if (key < vec[mid]) high = mid
        else low = mid + 1;
        // key0 = key ∧ |vec0| = |vec| ∧ −2 + 3 · low ≤ 2 · high + mid∧
        // −1 + 2 · low ≤ high + 2 · mid ∧ −1 + low ≤ mid ≤ 1 + high∧
        // high ≤ low + mid ∧ 1 + high ≤ 2 · low + mid ∧ 1 + low + mid ≤ |vec0| + high∧
        // 2 ≤ |vec0| ∧ 2 + high + mid ≤ |vec0| + low
    }
    return -1; } // POST: −1 ≤ res < |vec0|
```

L'analyse calcul des invariants linéaires entre les variables du programme

```
static int bsearch(int key, int[] vec) { //PRE: True
    // |vec0| = |vec| ∧ 0 ≤ |vec0|
    int low = 0, high = vec.length - 1;
    // |vec0| = |vec| ∧ 0 ≤ low ∧ high < |vec0| ∧ 0 ≤ |vec0|
    while (low < high) {
        // |vec0| = |vec| ∧ 0 ≤ low < high < |vec0|
        int mid = low + (high - low) / 2;
        // |vec| − |vec0| = 0 ∧ low ≥ 0 ∧ mid − low ≥ 0∧
        // 2·high − 2·mid − 1 ≥ 0 ∧ |vec0| − high − 1 ≥ 0
        if (key == vec[mid]) return mid;
        else if (key < vec[mid]) high = mid
        else low = mid + 1;
        // |vec0| = |vec| ∧ 0 ≤ low∧
        // 1 + high ≤ |vec0| ∧ 2 ≤ |vec0|
    }
    return -1; } // POST: res < |vec0|
```

Les invariants sont compressés

Le vérificateur polyédrique est rapide

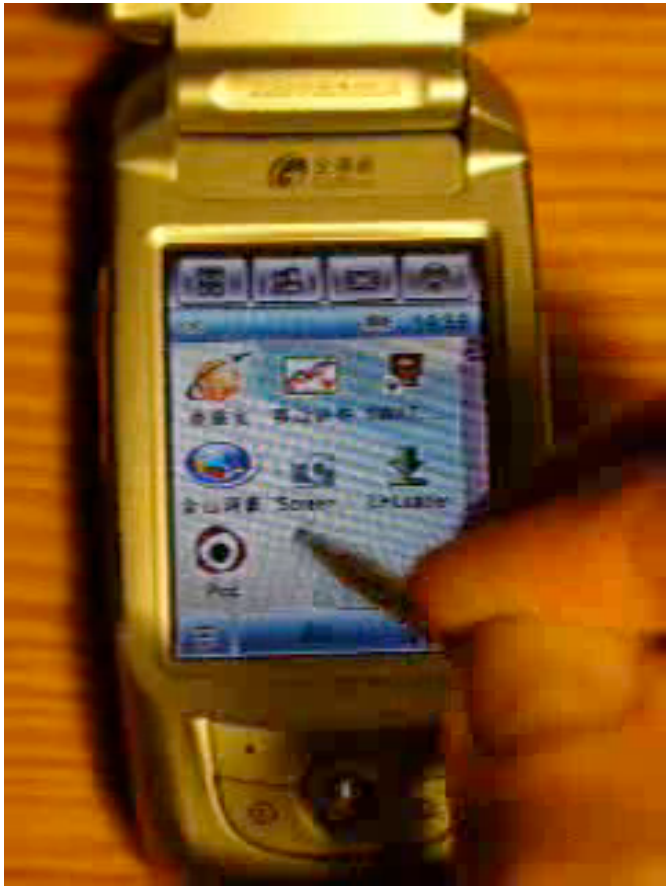
Program	size	Caml checker	Coq checker
BSearch	80	1.4	11.6
HeapSort	143	3.7	35.5
QuickSort	276	128.7	974.0
Random	883	8.0	44.3
Jacobi	135	1.7	9.2
LU	559	17.4	91.5
SparseCompRow	90	1.1	6.1
FFT	591	22.7	193.8

Temps en ms, tailles en nombre d'instructions.

Le vérificateur certifié tourne sur un téléphone !

G. Barthe, P. Crégut, B. Grégoire, T. Jensen, D. Pichardie. The MOBIUS Proof Carrying Code infrastructure. FMCO'07

Le vérificateur certifié tourne sur un téléphone !



G. Barthe, P. Crégut, B. Grégoire, T. Jensen, D. Pichardie. The MOBIUS Proof Carrying Code infrastructure. FMCO'07

Conclusions

Conclusions

Les analyses de flux d'informations et de ressource

- sont des outils particulièrement utiles pour le code mobile,
- mais elles nécessitent des techniques coûteuses et complexes.

Le code porteur de preuve permet

- de séparer clairement la recherche de preuve, de sa vérification,
- de rendre la vérification embarquée praticable.

Le code porteur de preuve certifié

- donne les plus hautes garanties sur la correction du vérificateur de preuve,
- mais nécessite encore un effort de formalisation nettement plus important que pour développer une approche PCC standard.

Merci