Sécurité des logiciels et analyse statique

David Pichardie

Projet Lande, INRIA Rennes - Bretagne Atlantique

Analyse de programme

Objet : déduire mécaniquement des propriétés sur le comportement d'un programme sans l'exécuter.

Domaines d'application :

- initialement : la compilation (optimisation de code)
- de nos jours : la vérification et toujours la compilation

Statique *vs.* dynamique

Vérification statique :

- réalisée au moment de la compilation ou de l'installation
- caractérise toutes les exécutions possibles
- approximative : considère plus d'exécutions que nécessaire
- peut prédire l'arrêt...

Vérification dynamique :

- occasionne un coût à l'exécution
- ne caractérise qu'une exécution (ou un sous ensemble)
- précise : connaît l'état courant du programme
- ne peut pas regarder dans le futur

Approximative... pourquoi ?

La mauvaise nouvelle

Le théorème de Rice

Dans un langage de programmation Turing-complet, pour toute propriété non-triviale, le problème
«Est-ce qu'un programme donné satisfait cette propriété»
ne peut pas être résolu automatiquement pour tout programme, par un même outil.

Les solutions trouvées par les méthodes formelles

- Vérifier un modèle plutôt qu'un programme (*model checking*)
- Vérifier le programme de manière interactive, avec l'aide de l'utilisateur (méthodes déductive : Coq, méthode B)
- Calculer une approximation du comportement du programme (analyse statique)

Le théorème de Rice vue par l'analyse statique

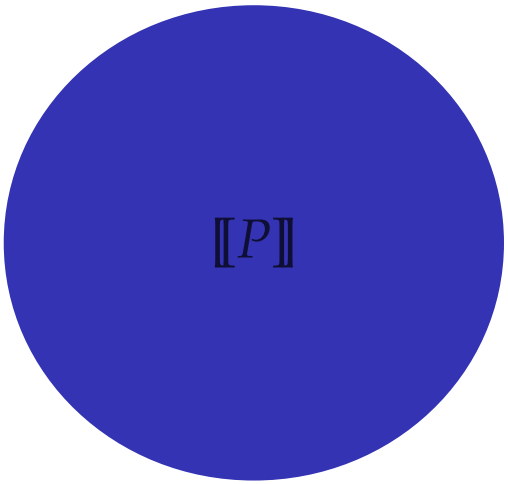
> *Aucune analyse statique ne peut démontrer ou invalider une propriété non-triviale pour tout programme, en un temps fini.*

Mais cela ne signifie pas qu'elle n'est pas capable de le faire pour au moins un !

- ASTRÉE[1] a analysé avec succès les commandes de vol des Airbus (~ 1 M lignes de C !)

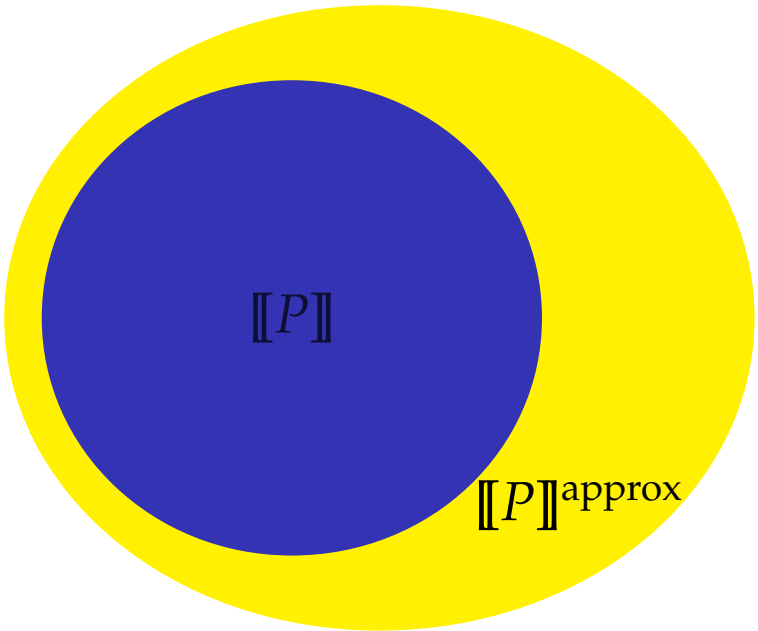
[1] `http://www.astree.ens.fr/`

Une analyse statique calcule une approximation



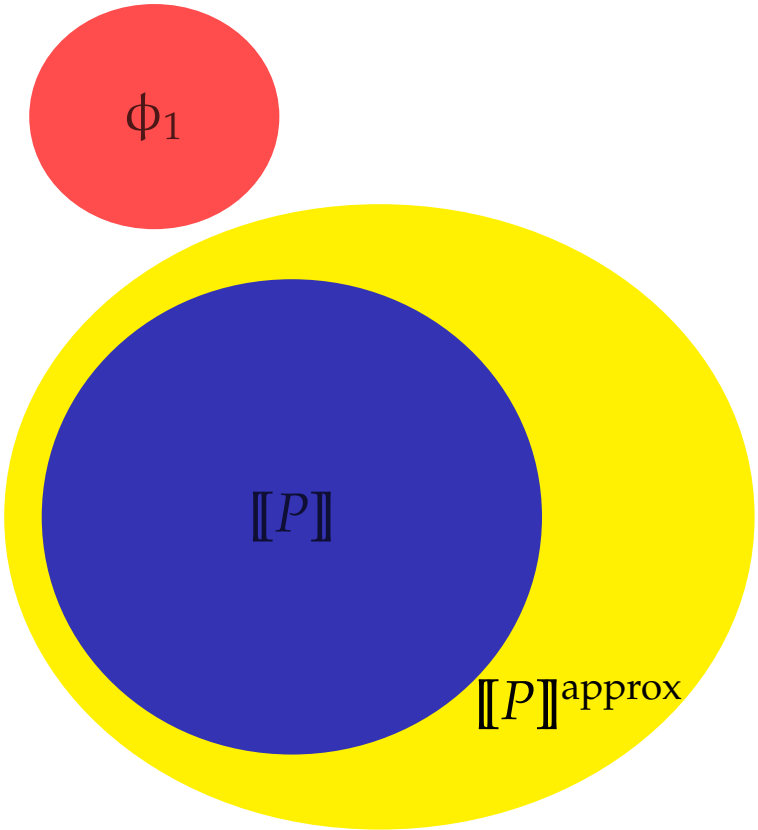
$[\![P]\!]$: ensemble des états atteignables par P (non calculable)

Une analyse statique calcule une approximation



$[\![P]\!]$:	ensemble des états atteignables par P	(non calculable)
$[\![P]\!]^{\text{approx}}$:	approximation calculée par l'analyse	(computable)

Une analyse statique calcule une approximation

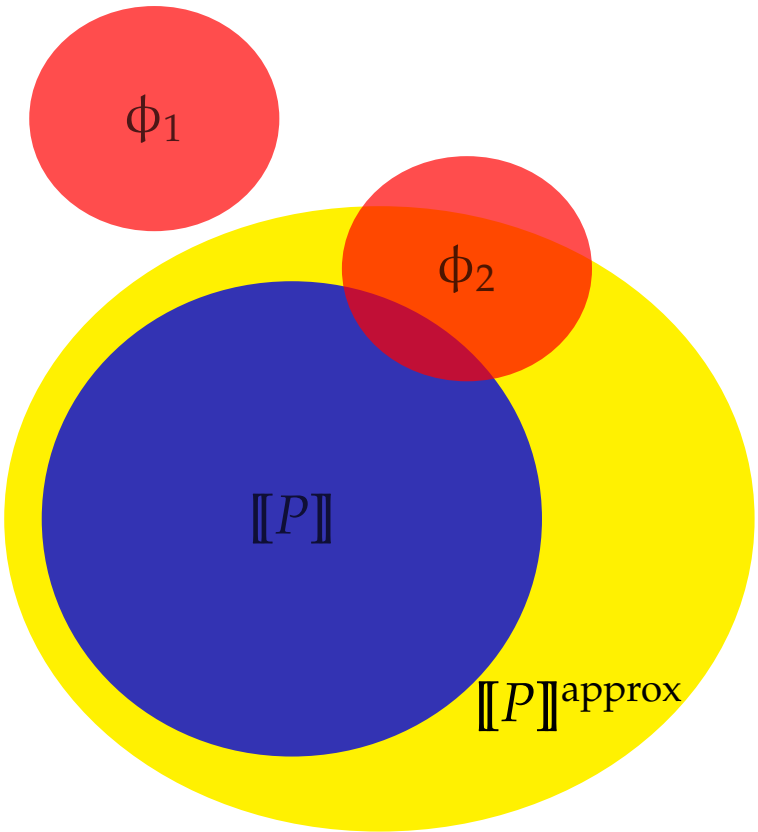


Cas 1 : P est sûre vis-à-vis de ϕ_1 et l'analyseur le prouve

$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

$[\![P]\!]$:	ensemble des états atteignables par P	(non calculable)
$[\![P]\!]^{\text{approx}}$:	approximation calculée par l'analyse	(computable)
ϕ_1 :	états dangereux/erronés	(calculable)

Une analyse statique calcule une approximation



Cas 1 : P est sûre vis-à-vis de ϕ_1 et l'analyseur le prouve

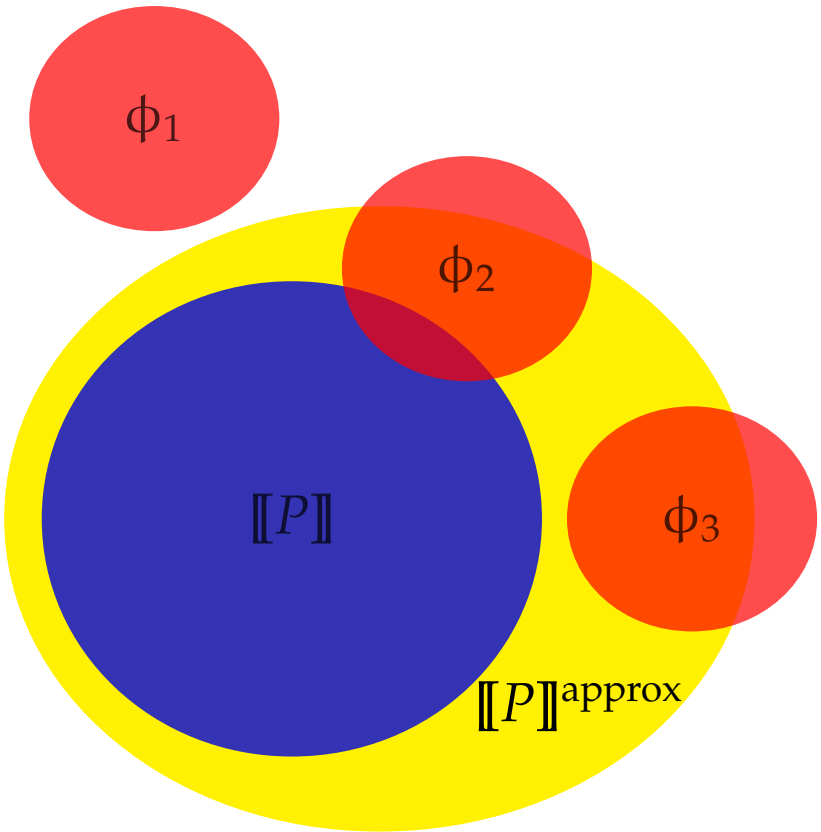
$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

Cas 2 : P n'est pas sûre vis-à-vis de ϕ_2 et l'analyseur le *dit* (*vraie alarme*)

$$[\![P]\!] \cap \phi_2 \neq \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_2 \neq \emptyset$$

$[\![P]\!]$:	ensemble des états atteignables par P	(non calculable)
$[\![P]\!]^{\text{approx}}$:	approximation calculée par l'analyse	(computable)
ϕ_1 :	états dangereux/erronés	(calculable)

Une analyse statique calcule une approximation



Cas 1 : P est sûre vis-à-vis de ϕ_1 et l'analyseur le prouve

$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

Cas 2 : P n'est pas sûre vis-à-vis de ϕ_2 et l'analyseur le *dit* (*vraie alarme*)

$$[\![P]\!] \cap \phi_2 \neq \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_2 \neq \emptyset$$

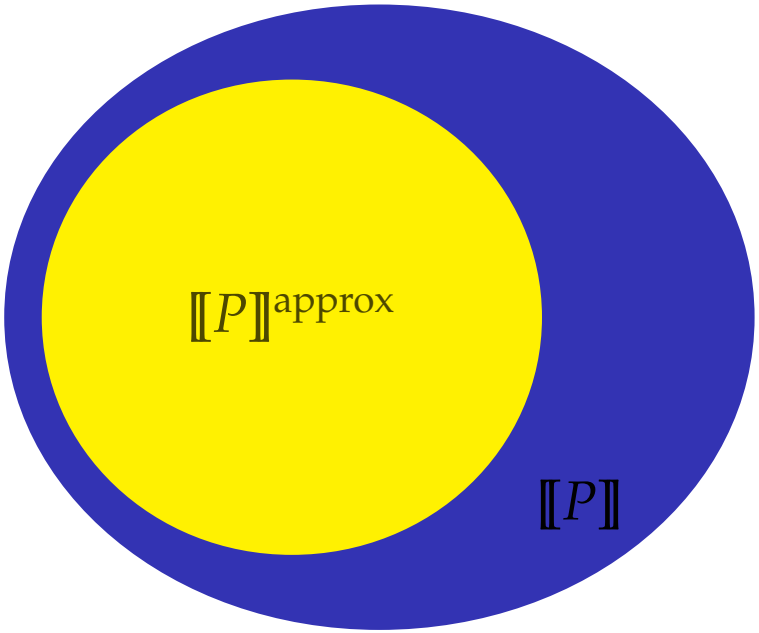
Cas 3 : P est sûre vis-à-vis de ϕ_3 mais l'analyseur ne peut pas le prouver (*fausse alarme*) !

$$[\![P]\!] \cap \phi_3 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_3 \neq \emptyset$$

$[\![P]\!]$:	ensemble des états atteignables par P	(non calculable)
$[\![P]\!]^{\text{approx}}$:	approximation calculée par l'analyse	(computable)
ϕ_1, ϕ_2, ϕ_3 :	états dangereux/erronés	(calculable)

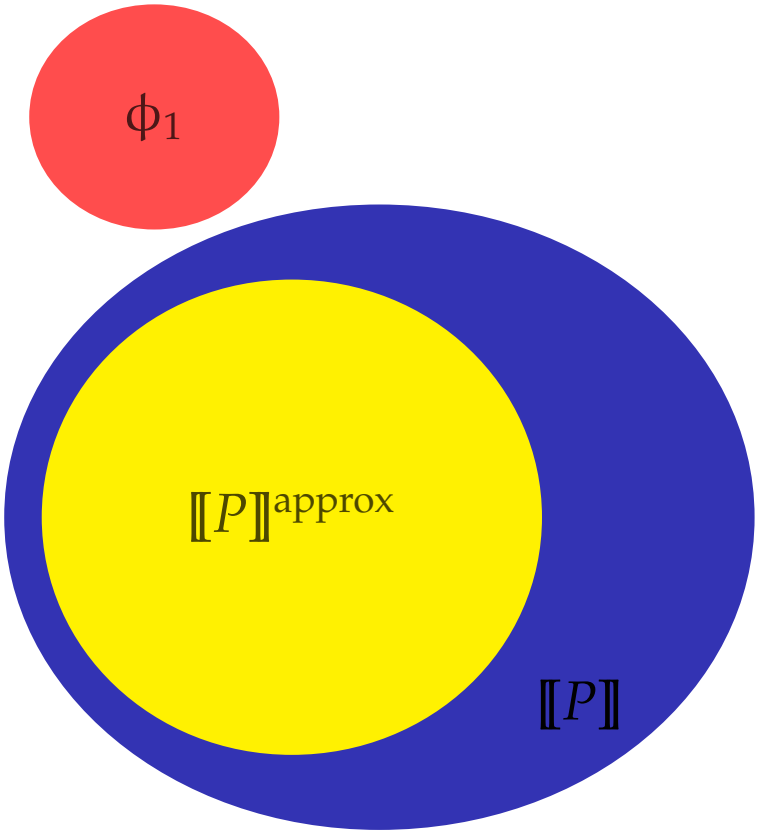
Preuve/Test

Pour prouver un programme, on sur-approxime.
Pour tester un programme, on sous-approxime.



Preuve/Test

Pour prouver un programme, on sur-approxime.
Pour tester un programme, on sous-approxime.

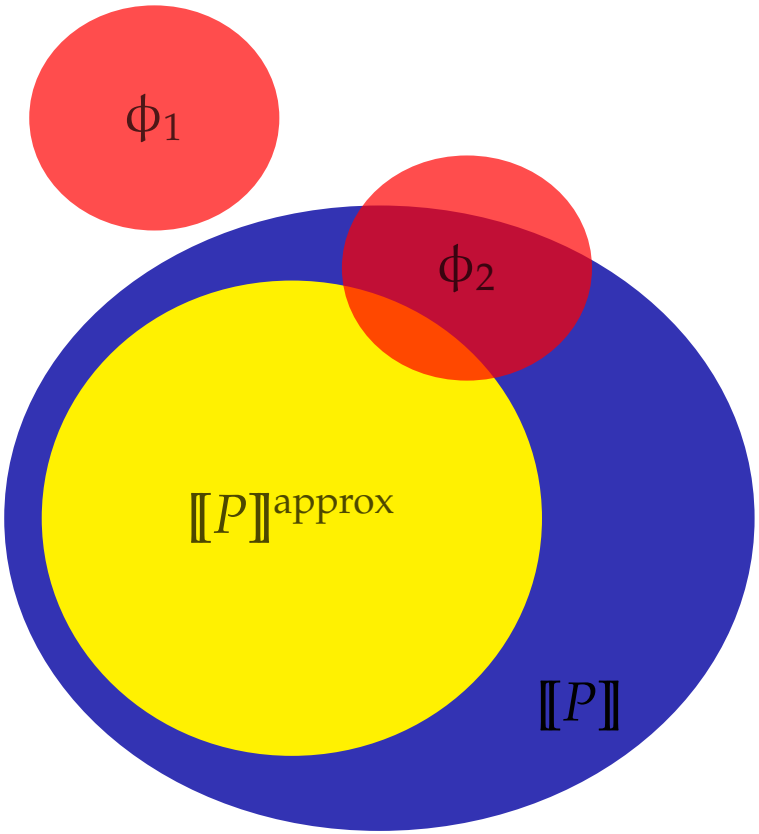


Cas 1 : P est sûre vis-à-vis de ϕ_1 et le test ne le démontre pas

$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

Preuve/Test

Pour prouver un programme, on sur-approxime.
Pour tester un programme, on sous-approxime.



Cas 1 : P est sûre vis-à-vis de ϕ_1 et le test ne le démontre pas

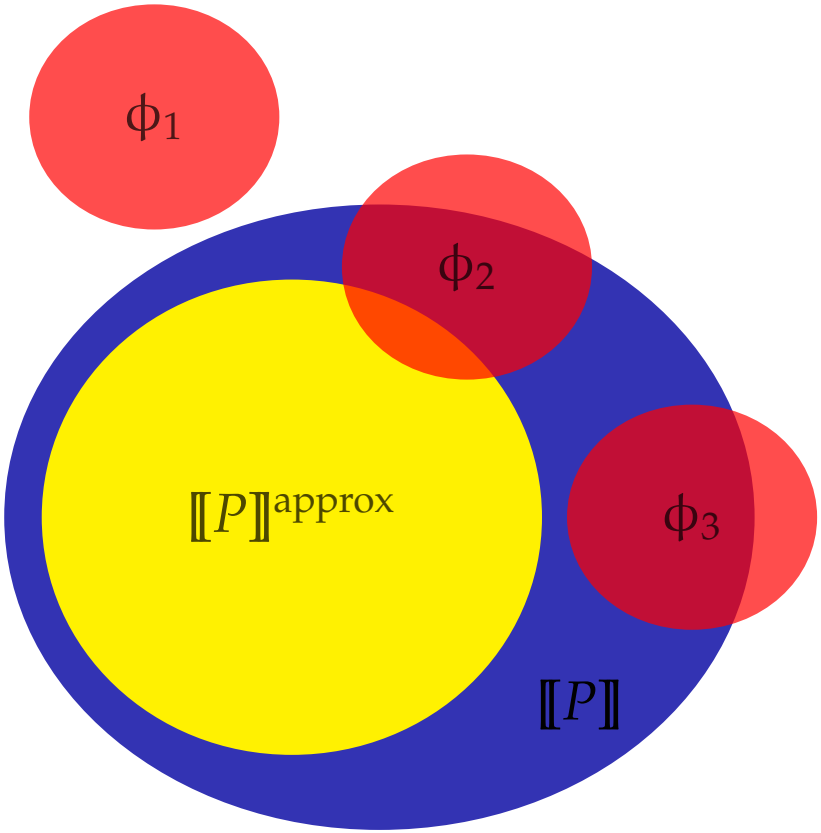
$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

Cas 2 : P n'est pas sûre vis-à-vis de ϕ_2 et le test le prouve

$$[\![P]\!] \cap \phi_2 \neq \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_2 \neq \emptyset$$

Preuve/Test

Pour prouver un programme, on sur-approxime.
Pour tester un programme, on sous-approxime.



Cas 1 : P est sûre vis-à-vis de ϕ_1 et le test ne le démontre pas

$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

Cas 2 : P n'est pas sûre vis-à-vis de ϕ_2 et le test le prouve

$$[\![P]\!] \cap \phi_2 \neq \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_2 \neq \emptyset$$

Cas 3 : P n'est pas sûre vis-à-vis de ϕ_3 mais le test ne le dit pas !

$$[\![P]\!] \cap \phi_3 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_3 \neq \emptyset$$

Plan

Plan

Plan

Plan

Plan

Analyse de propriétés de sécurité

L'analyse statique peut assurer statiquement certaines propriétés de sécurité

- l'utilisation «légale »d'une ressource,
- l'absence de fuite d'information confidentielle vers des canaux publiques.

Ce type d'analyse s'appuie généralement sur des pré-analyses qui aident à comprendre le comportement d'un programme.

- analyse de flot de contrôle,
- analyse relationnelles numériques.
- analyse des exceptions sortantes (déréférencement de pointeur nul, accès hors des bornes de tableau, ...),

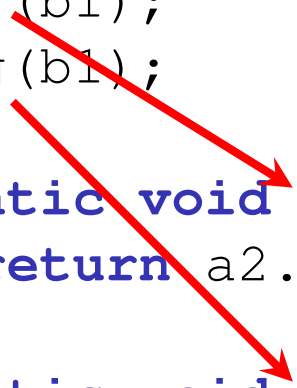
Analyse de flot de contrôle

```
static int main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  return a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  return b3.foo();
}
```

```
class A {
  void foo(){ ... }
}
class B extends A{
  void foo(){ ... }
}
class C extends B{
  void foo() { ... }
}
```

Analyse de flot de contrôle

```
static int main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  return a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  return b3.foo();
}
```



```
class A {
  void foo(){ ... }
}
class B extends A{
  void foo(){ ... }
}
class C extends B{
  void foo() { ... }
}
```

Les appels statiques sont facile à «comprendre ».
Les appels virtuels sont plus incertains...

Analyse de flot de contrôle

```
static int main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  return a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  return b3.foo();
}
```

```
class A {
  void foo(){ ... }
}
class B extends A{
  void foo(){ ... }
}
class C extends B{
  void foo() { ... }
}
```

Avec une simple information de typage

- `a2` est de type `A`
- `b3` est de type `B`

Analyse de flot de contrôle

```
static int main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  return a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  return b3.foo();
}
```

```
class A {
  void foo(){ ... }
}
class B extends A{
  void foo(){ ... }
}
class C extends B{
  void foo() { ... }
}
```

Avec une approximation des objets par leur classes

- class(`a2`) = `B`
- class(`b3`) = `C`

Analyse de flot de contrôle

Il est souvent plus précis de tenir compte des sites de créations des objets

- analyses de *points-to*

... et souvent encore plus précis de considérer différentes instances d'une même méthode en fonction de son contexte d'appel

- analyses sensibles au contexte

Ce type d'analyse peut rapidement devenir coûteux mais beaucoup de progrès ont eu lieu ces dernières années (BDD).

Analyses relationnelles numériques

La théorie de l'interprétation abstraite[Cousot&Cousot77]

- a fournit un cadre théorique à l'analyse statique,
- mais a aussi donné naissance à des analyses extrêmement puissantes.

Exemple : l'analyse polyhédrique [Cousot&Halbwachs78]

- but : inférer des invariants numériques linéaires entre les variables d'un programme

Analyse polyhédrique : exemple de résultat

```
assert(T.length>=1); i=1;
```
$\{1 \leqslant i \leqslant T.length\}$

```
while i<T.length {
```
$\{1 \leqslant i \leqslant T.length - 1\}$

```
  p = T[i]; j = i-1;
```
$\{1 \leqslant i \leqslant T.length - 1 \ \wedge \ -1 \leqslant j \leqslant i - 1\}$

```
  while (0<=j and T[j]>p) {
```
$\{1 \leqslant i \leqslant T.length - 1 \ \wedge \ 0 \leqslant j \leqslant i - 1\}$

```
    T[j]=T[j+1]; j = j-1;\\
```
$\{1 \leqslant i \leqslant T.length - 1 \ \wedge \ -1 \leqslant j \leqslant i - 2\}$

```
  };
```
$\{1 \leqslant i \leqslant T.length - 1 \ \wedge \ -1 \leqslant j \leqslant i - 1\}$

```
  T[j+1]=p; i = i+1;\\
```
$\{2 \leqslant i \leqslant T.length + 1 \ \wedge \ -1 \leqslant j \leqslant i - 2\}$

```
};
```
$\{i = T.length\}$

Utile, par exemple, pour vérifier les accès aux tableaux.

Analyse des exceptions sortantes

- exception `NulPointer`
  - analyse plus ou moins facile selon que le programmeur « protège » ou non ses déréférencements,

```
if (x!=null) { ...; x.f; ...};
```

  - sinon, il faut être capable d'inférer des invariants (ou annotations) sur les champs des objets[2].
- exceptions `ArrayOutOfBounds`, `Arithmetic`
  - nécessite une analyse (relationelle) numérique.
- exceptions créées par l'utilisateur
  - nécessite une analyse de classe (ou *points-to*).

[2] Null-ability Inference Tool : `http://nit.gforge.inria.fr`

Analyse de ressources

Un exemple de ressource : le SMS.

En mode *one-shot*, le profile MIDP de Java impose un écran de confirmation avant chaque envoie de SMS.

Généralisation[3] :

- `grant?(n);` : demande de permission avec multiplicité,
- `consume(); ...; consume(); ...` : consommation d'une permission

Politique de sécurité : *un programme n'accède jamais à une ressource dont il n'a pas la permission*

[3] F. Besson, G. Dufay and T. Jensen, *A Formal Model of Access Control for Mobile Interactive Devices*, ESORICS'06

Analyse de ressources

Cette politique de sécurité peut être assurée dynamiquement

- mais un *mauvais* programme provoquera alors une exception de sécurité.

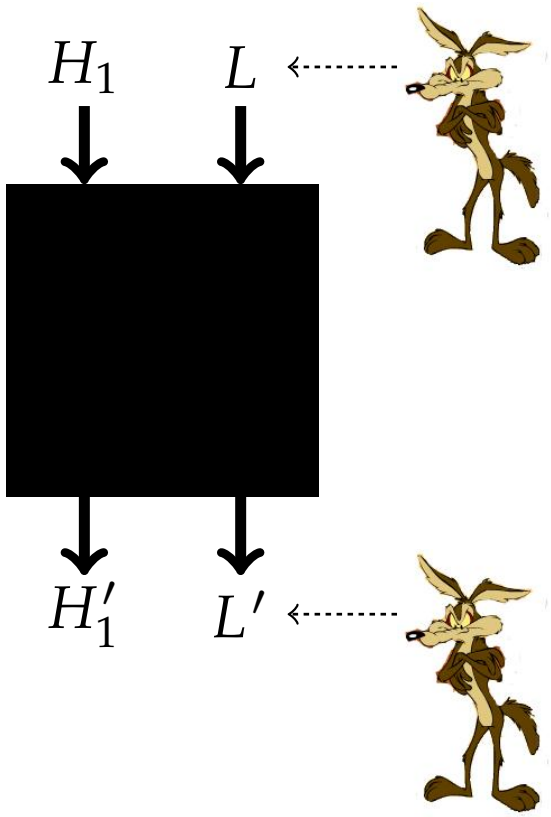
Cette politique de sécurité peut être assurée statiquement

- un *mauvais* programme sera rejeté avant l'exécution,
- l'analyse peut s'aider d'une analyse relationelle numérique.

```
grant?(sendSMS(*),addr_book.length) ;
  // 0 ⩽ addr_book.length = #SMSpermissions
for (i = 0 , i <  addr_book.length, i++)
  // 0 < addr_book.length − i ⩽ #SMSpermissions
  if (*) consume(SMS(addr_book[i].no),send);
  // 0  ⩽  #SMSpermissions
```

Non-interference

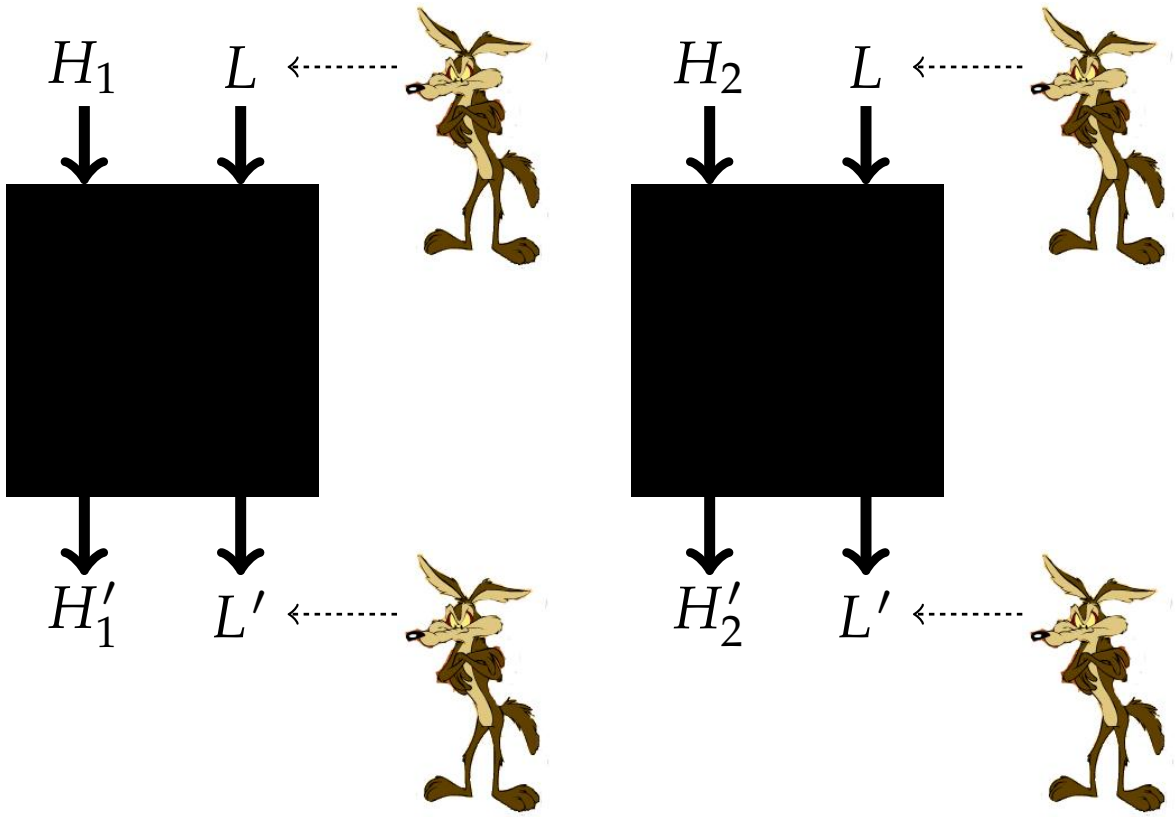
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidentiel Low = publique

Non-interference

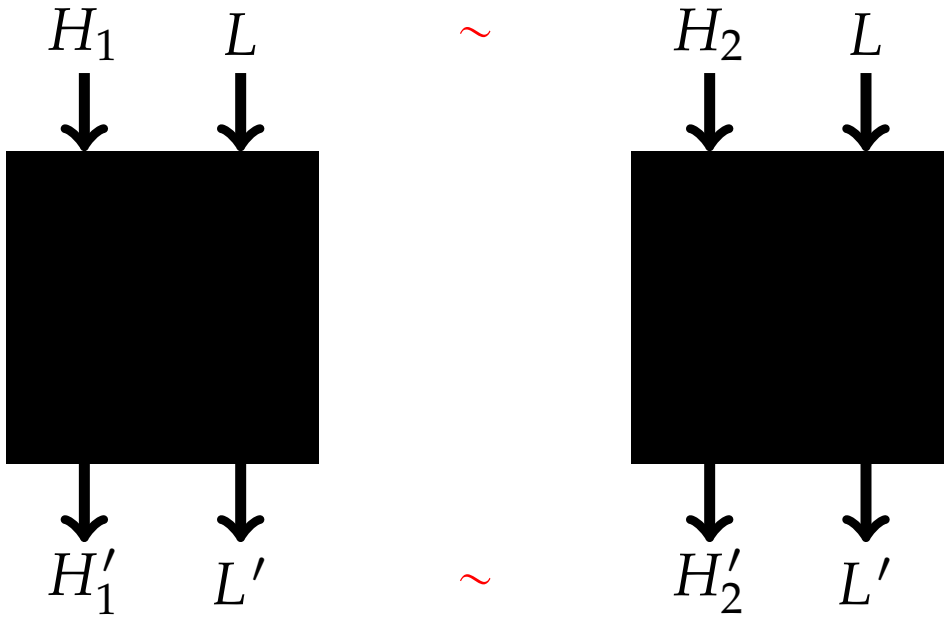
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidentiel Low = publique

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



$$\forall s_1, s_2, s_1', s_2', \quad s_1 \sim s_2 \wedge (P, s_1) \Downarrow s_1' \wedge (P, s_2) \Downarrow s_2' \implies s_1' \sim s_2'$$

High = confidentiel Low = publique

Exemples

1. `h := l`
2. `l := h`
3. `if (h1>h2) then l := 1 else l := 2`
4. `while (h) do l := l+1 ; l := 0`

Démontrer la non-interférence

La non-interférence d'un programme peut être assurer par typage

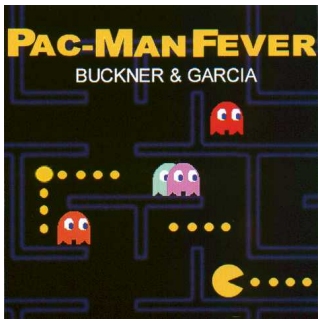
- A. Myers, *Jflow : Practical mostly-static information flow control*, POPL'99
- G. Barthe, D. Pichardie and T. Rezk, *A Certified Lightweight Non-Interference Java Bytecode Verifier*, ESOP'07

Mais la non-interférence apparaît parfois comme une propriété trop forte

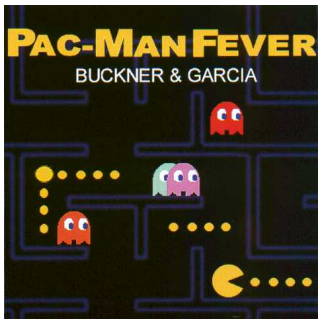
- exemple : vérification d'un mot de passe
- solution : *déclassification*
  - A. Sabelfeld and D. Sands, *Declassification : Dimensions and Principles*, JCS'07

Plan

Les dilemmes des codes mobiles...



Les dilemmes des codes mobiles...



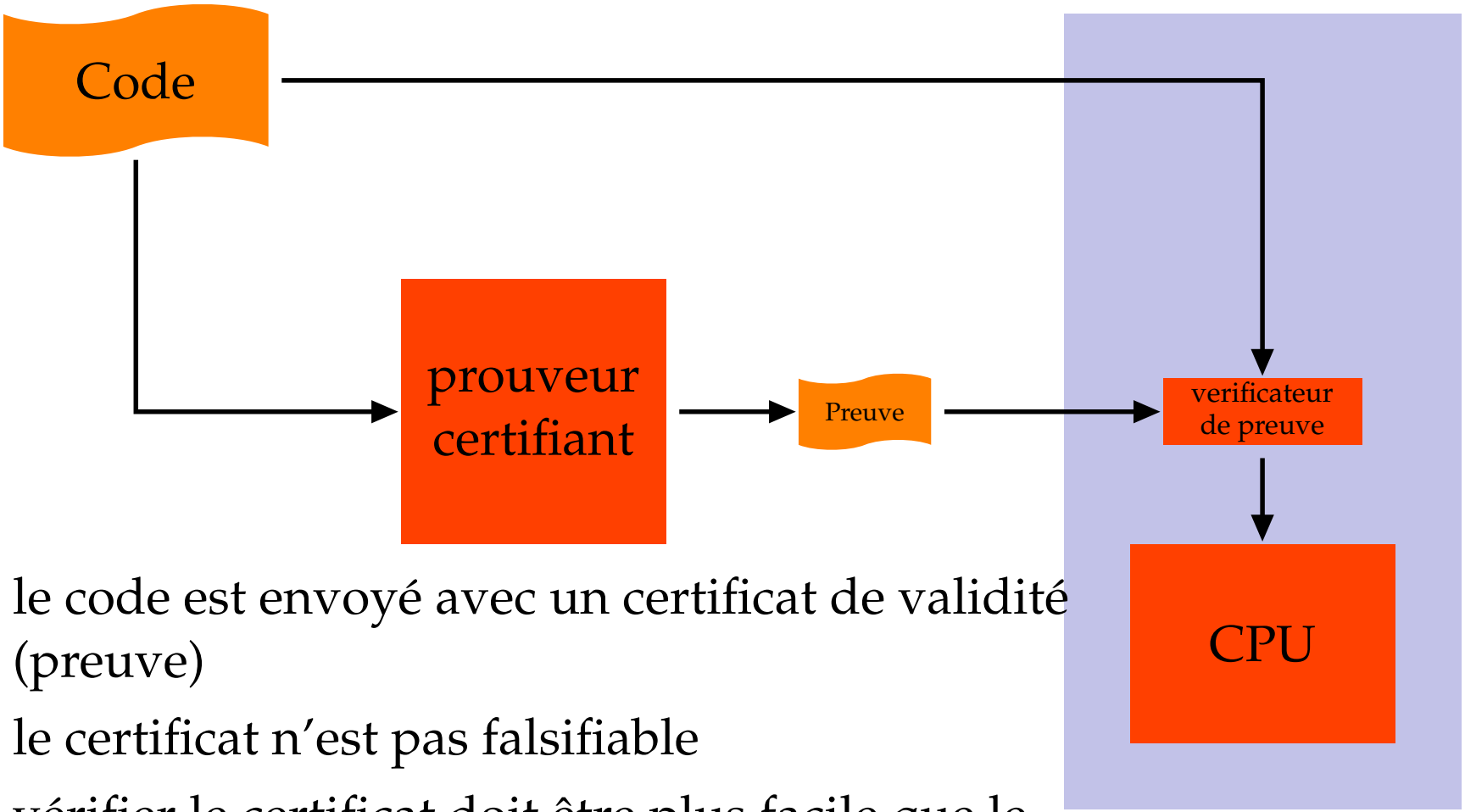
Sûr ?

Solutions

- Authentification cryptographie : un tiers de confiance signe le code
  - on fait confiance au tiers pas au programme
  - certificat cryptographique
- Code porteur de preuve (PCC : Proof-Carrying Code[4])
  - le programme est accompgné d'un certificat *sémantique*
  - on vérifie le comportement, pas la provenance

[4] G. Necula, *Proof-Carrying Code*, POPL'97

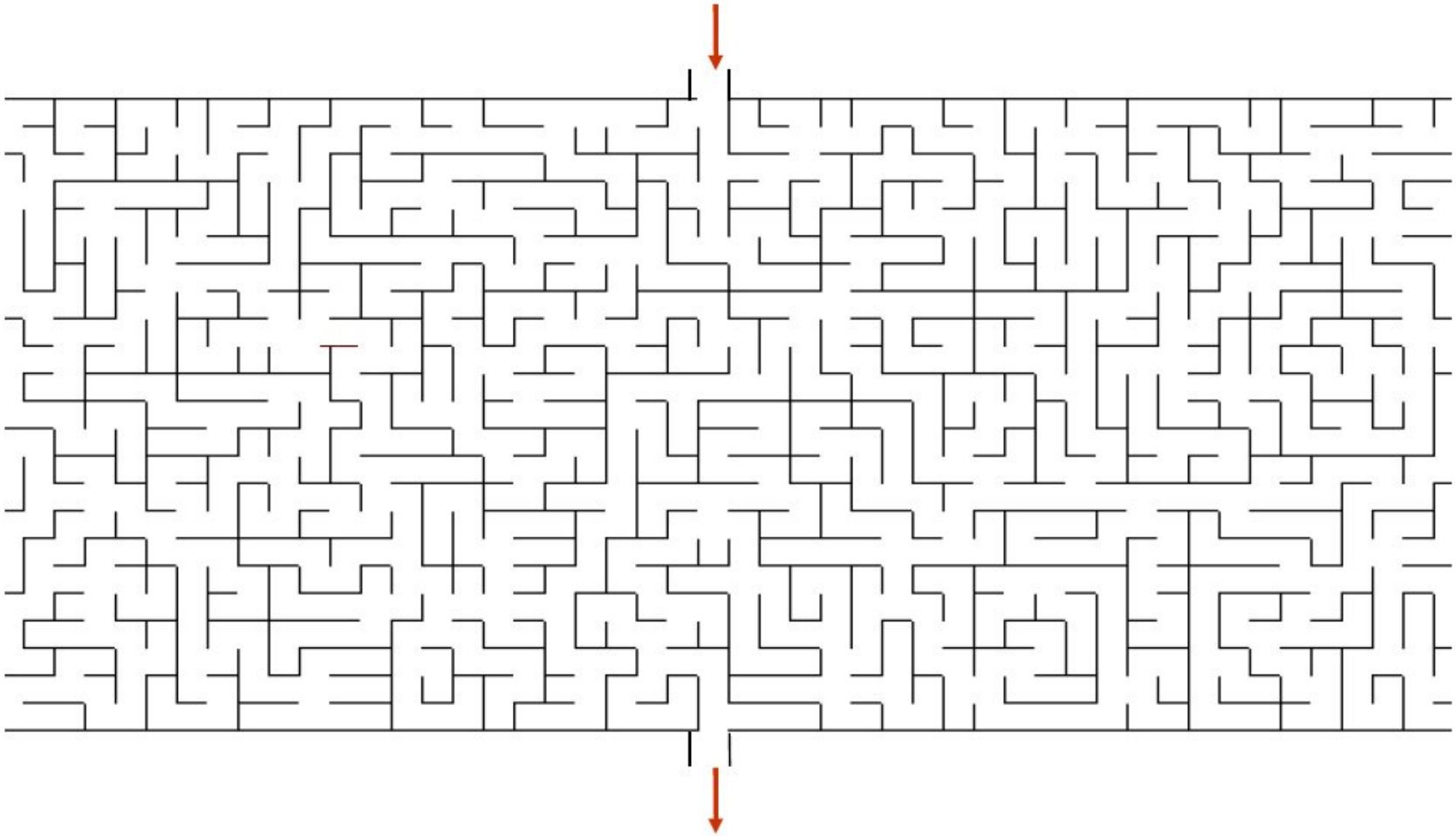
Code porteur de preuve : principes



- le code est envoyé avec un certificat de validité (preuve)
- le certificat n'est pas falsifiable
- vérifier le certificat doit être plus facile que le produire

La métaphore du labyrinthe

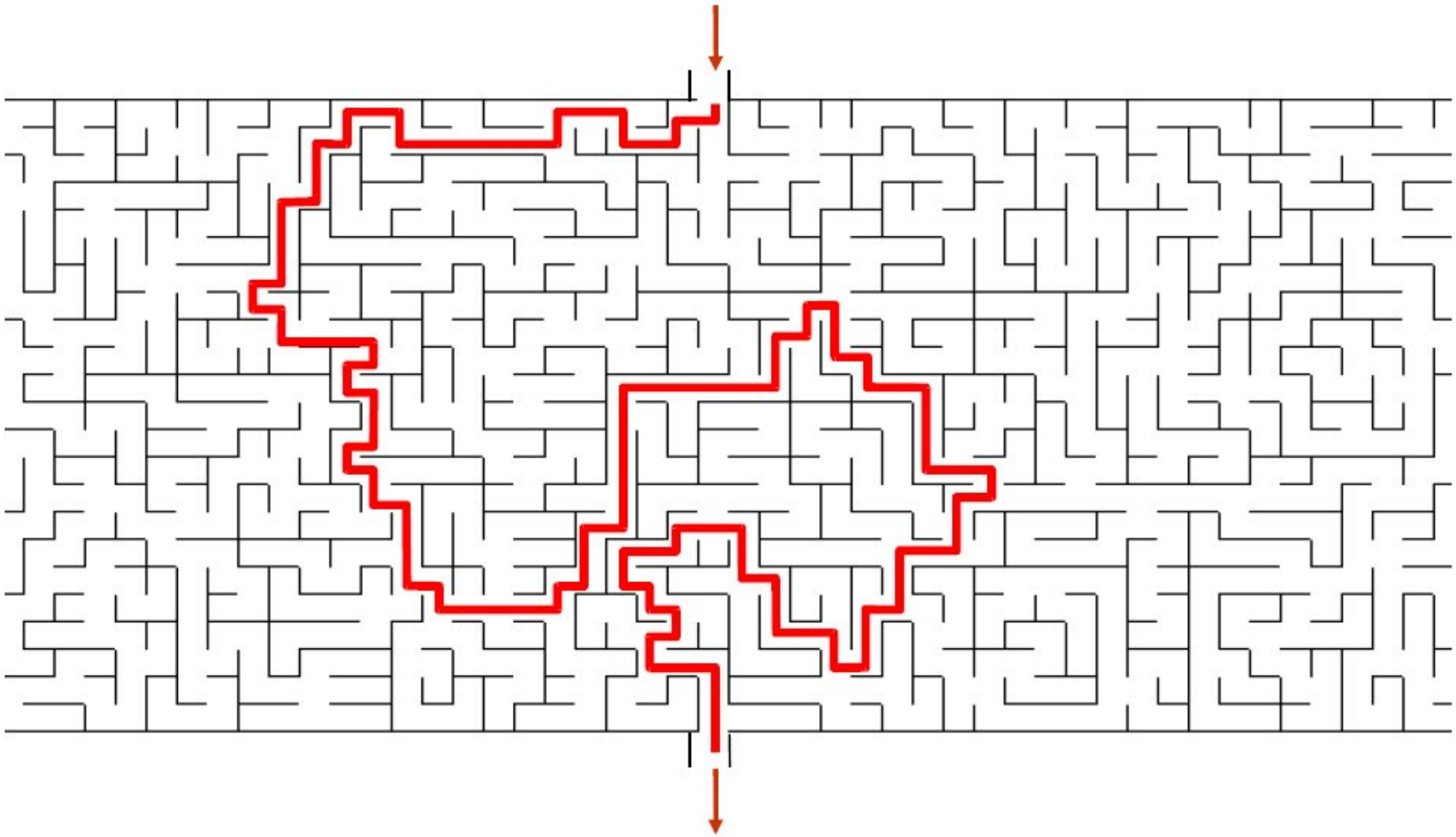
©G. Necula



programme = labyrinthe

La métaphore du labyrinthe

©G. Necula

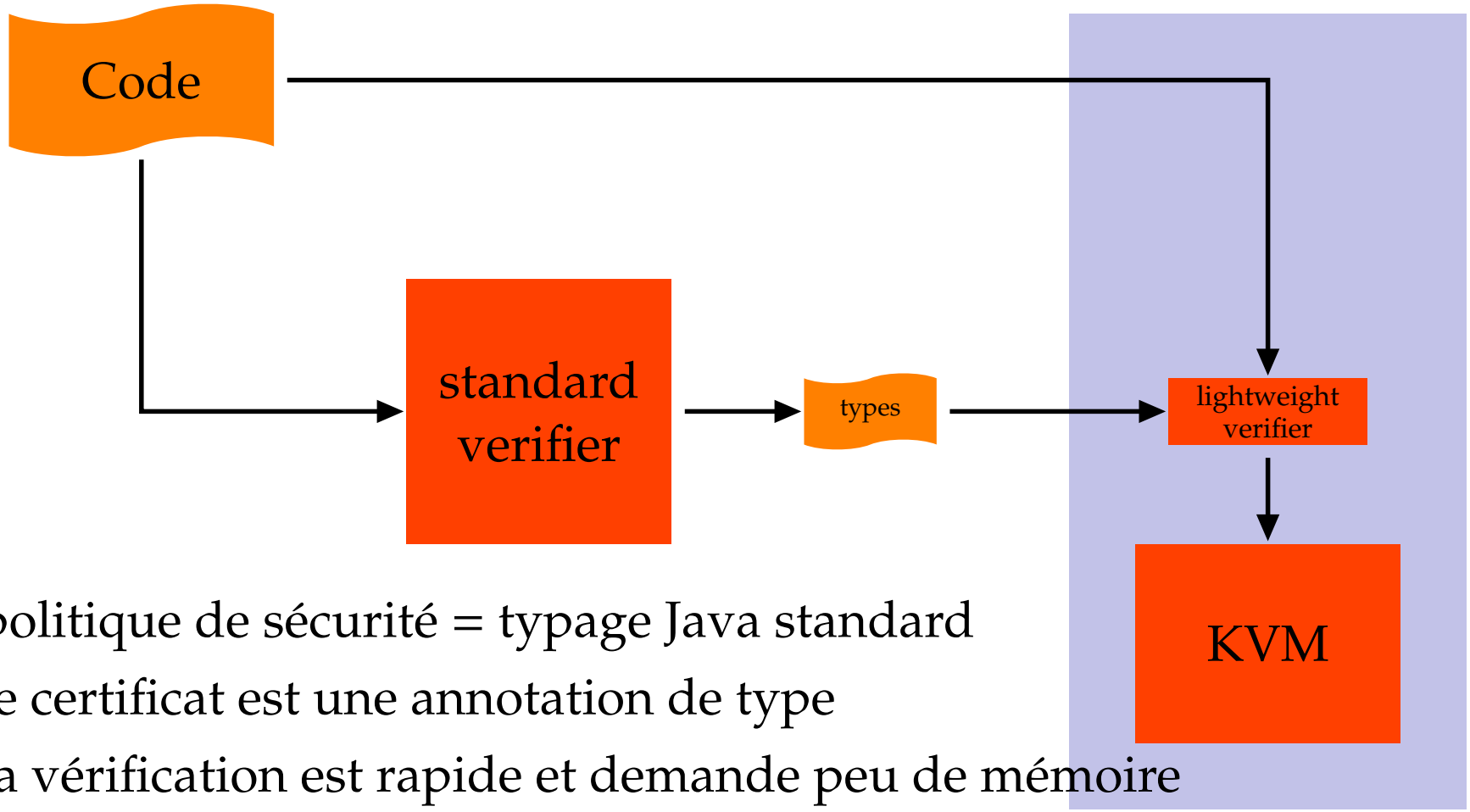


programme = labyrinthe

preuve = chemin rouge

Code porteur de preuve par analyse statique

Les JVM pour téléphones mobiles (KVM) suivent cette approche :
Lightweight Bytecode Verifier



- politique de sécurité = typage Java standard
- le certificat est une annotation de type
- la vérification est rapide et demande peu de mémoire

Code porteur de preuve par interprétation abstraite

Une approche[5] basée sur le cadre théorique de l'interprétation abstraite

- une analyse statique est spécifiée comme un post-point fixe $[\![p]\!]^\sharp$ d'une fonctionnelle $F^\sharp$ dans un treillis

$$F^\sharp\left([\![p]\!]^\sharp\right) \sqsubseteq [\![p]\!]^\sharp$$

- $[\![p]\!]^\sharp$ est calculée par des méthodes itératives complexes (avec des heuristiques)
- mais vérifier un post-point fixe donné est beaucoup plus simple et rapide !
- pour garantir la validité de l'approche : le vérificateur de post-point fixe est programmé et certifié en Coq !

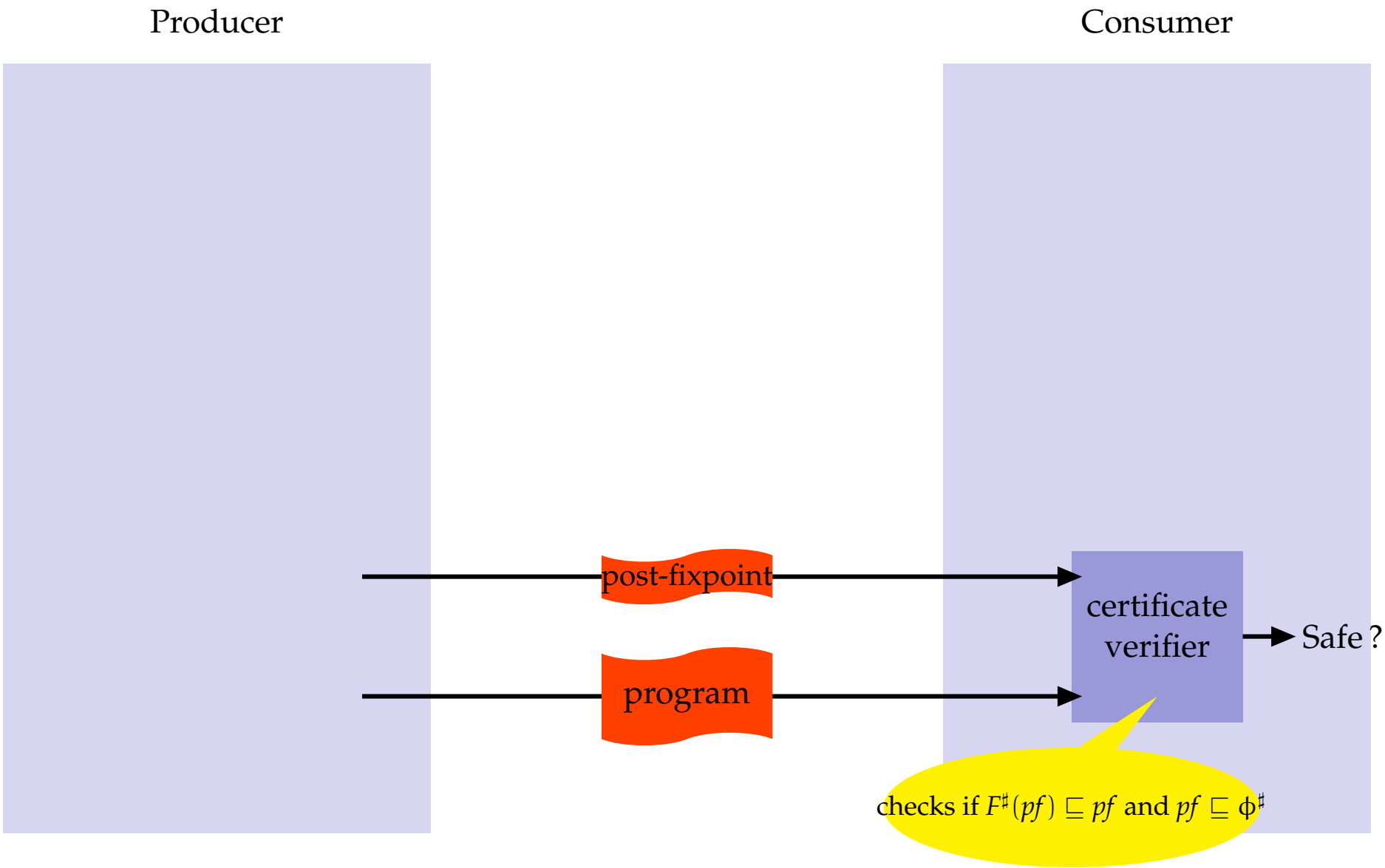
[5] F. Besson, T. Jensen and D. Pichardie, *Proof-carrying code from certified abstract interpretation to fixpoint compression*, TCS, 2006

PCC by abstract interpretation

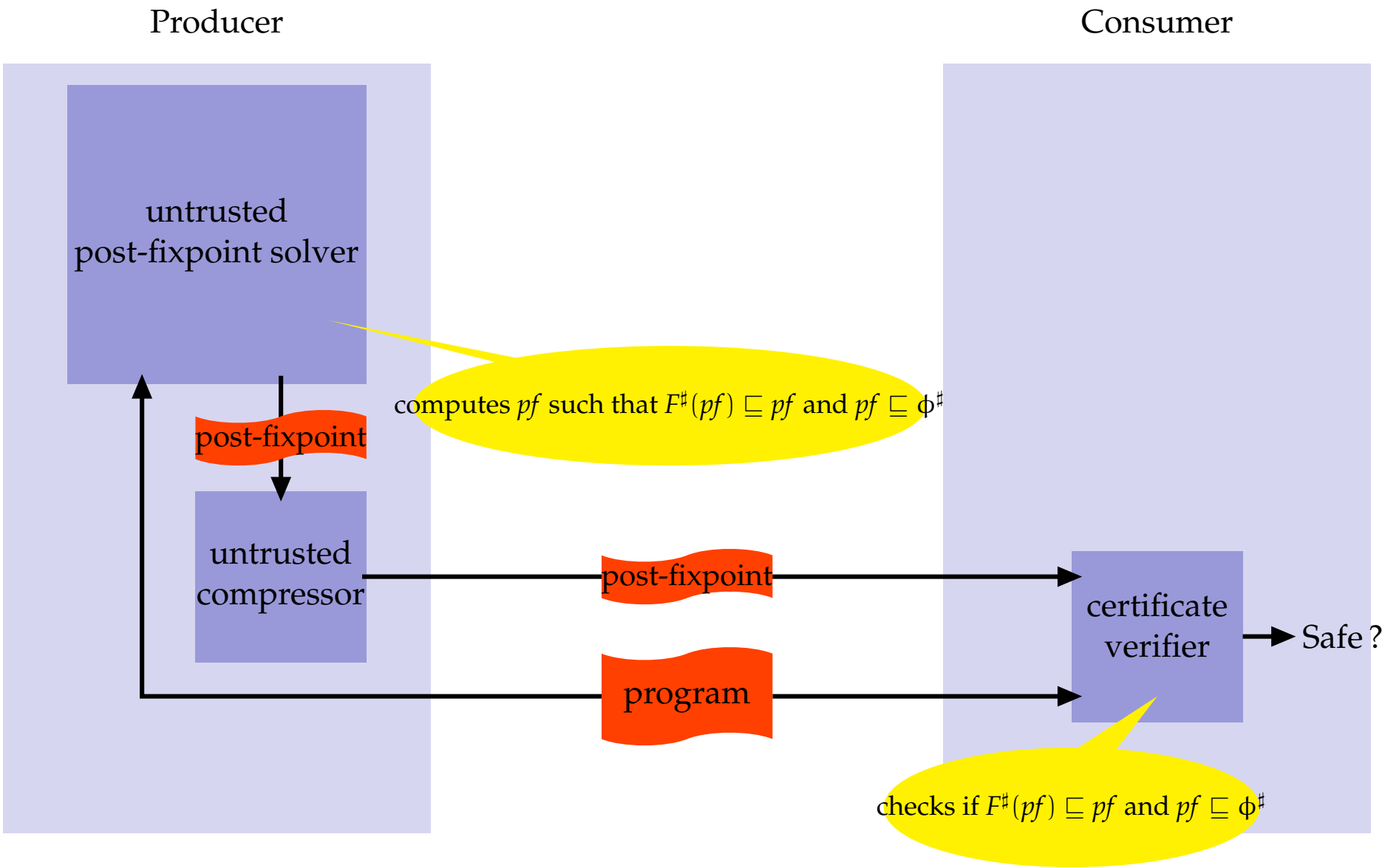
Producer

Consumer

PCC by abstract interpretation



PCC by abstract interpretation



Certified PCC by abstract interpretation

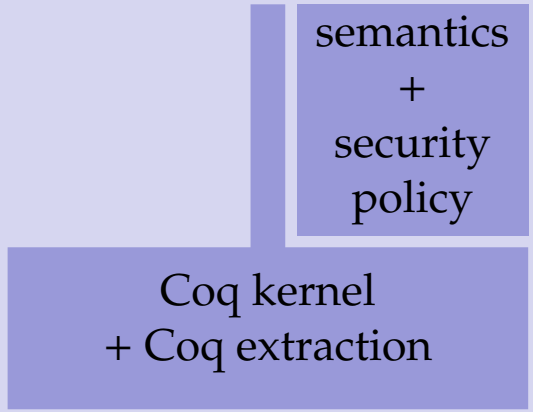
Producer

Consumer

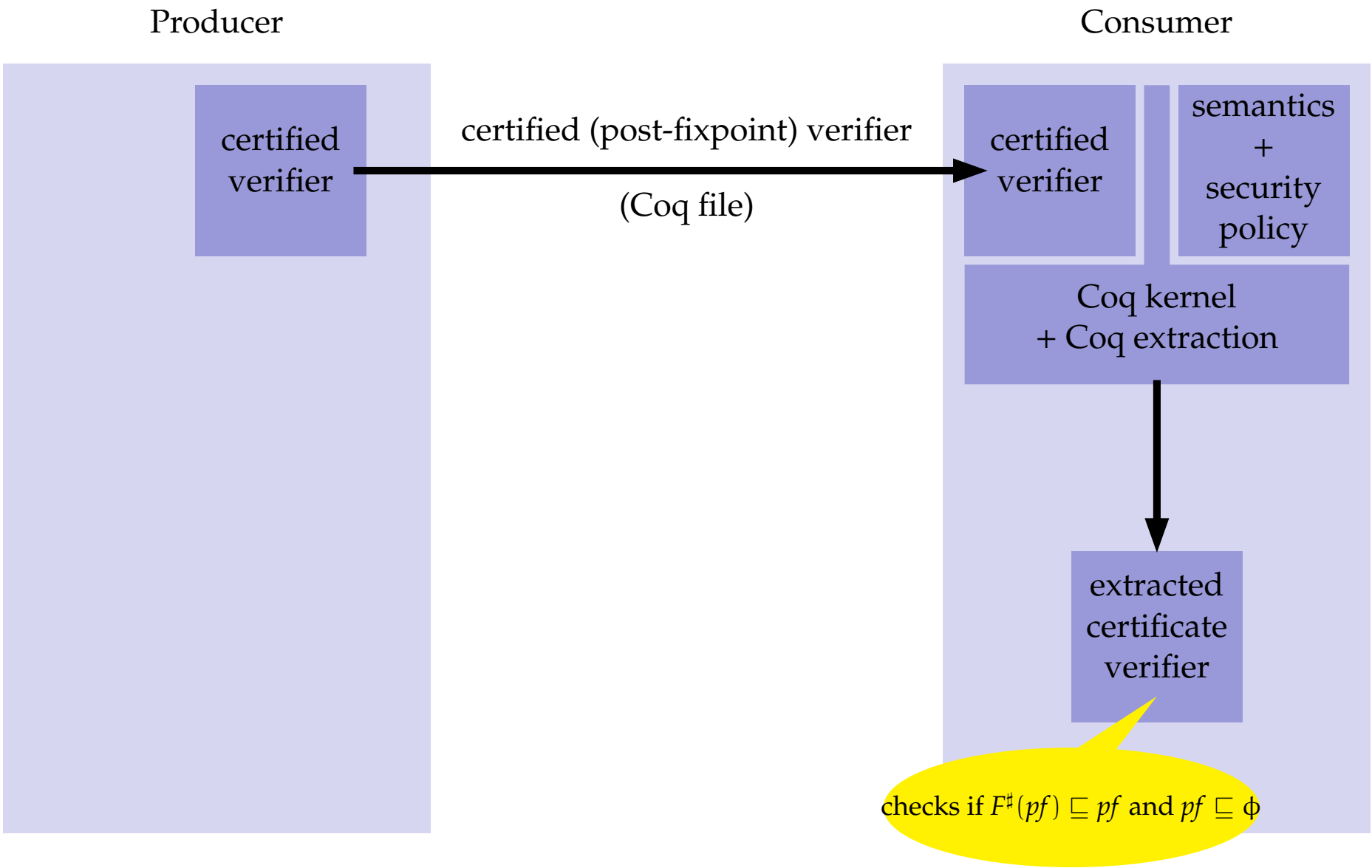
Certified PCC by abstract interpretation

Producer

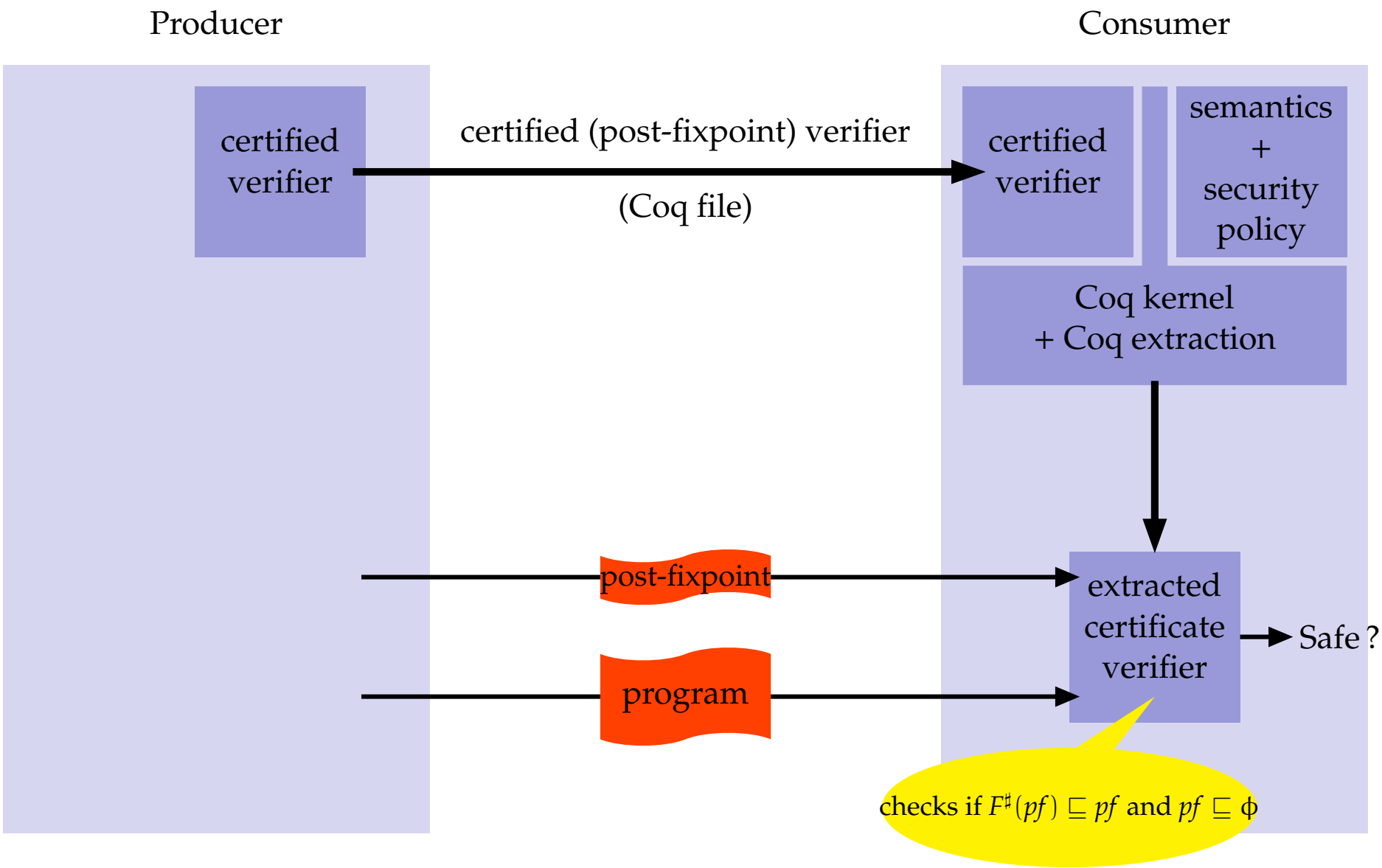
Consumer



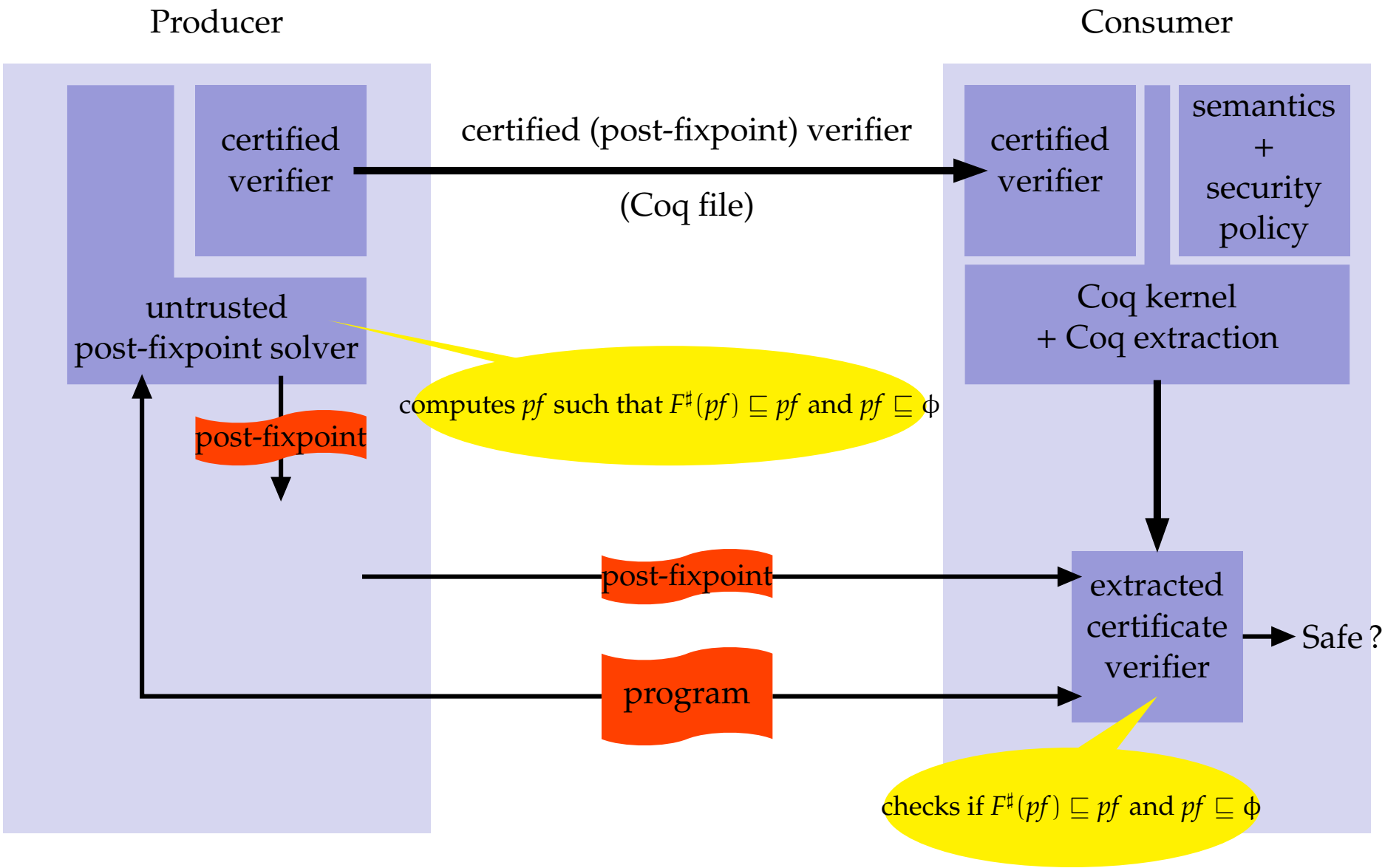
Certified PCC by abstract interpretation



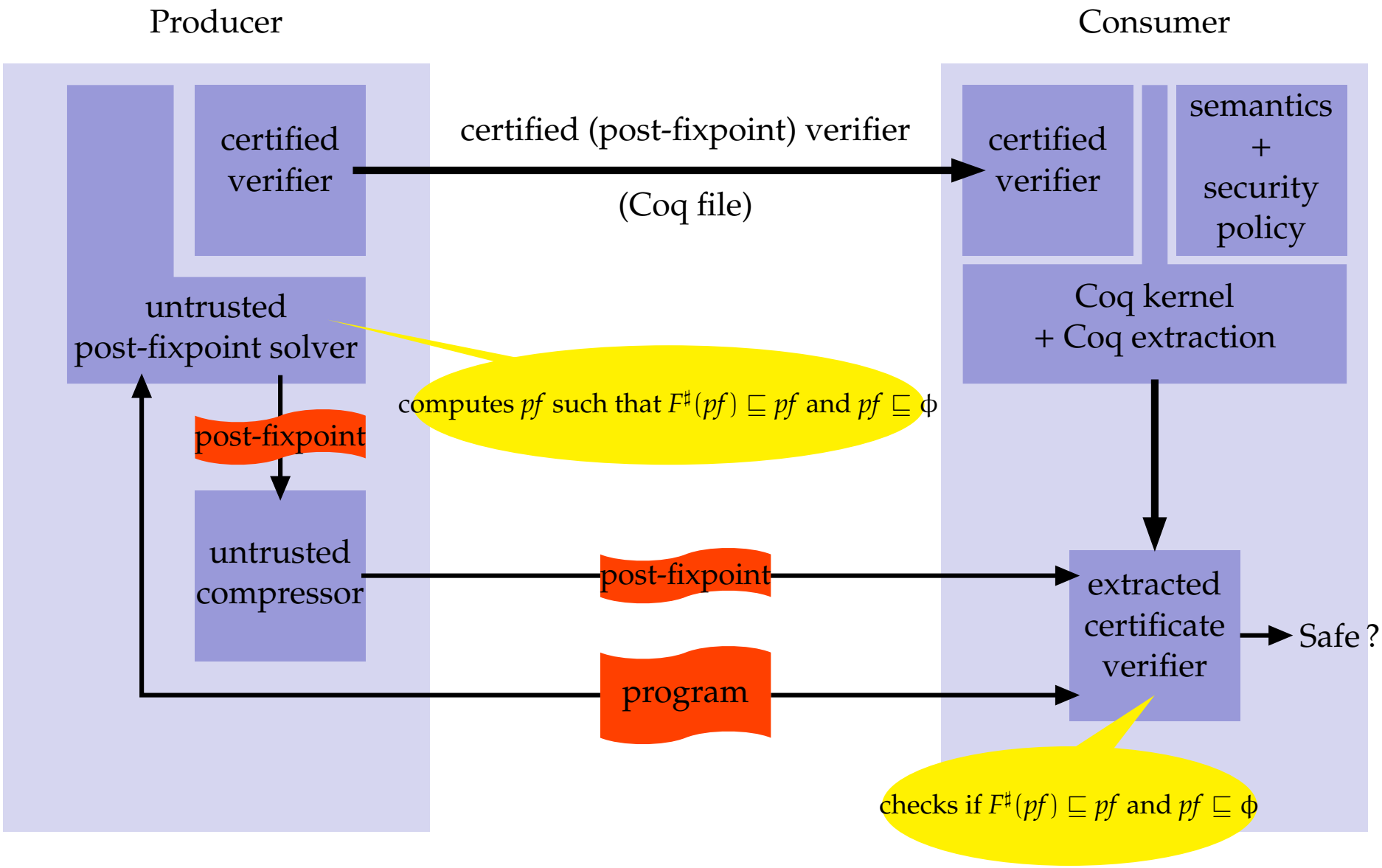
Certified PCC by abstract interpretation



Certified PCC by abstract interpretation



Certified PCC by abstract interpretation



Plan

Conclusions

L'analyse statique est une technique de vérification prometteuse

- a besoin de propriétés de sécurité bien ciblées
- mais ne peut pas marcher pour tout *style* de programmation

Les certificats *sémantiques* permettent

- de *démocratiser* les méthodes formelles en séparant clairement
  - recherche de preuve (par le programmeur ou l'expert),
  - et vérification de preuve (par *Mr Tout-Le-Monde*)
- et de certifier la validité sémantique de la vérification.