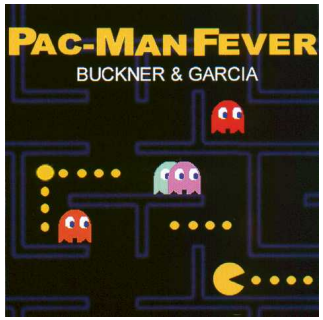
Proof Carrying Code : a quick tour

École chercheurs IRISA - Sécurité logicielle - Janvier 2008

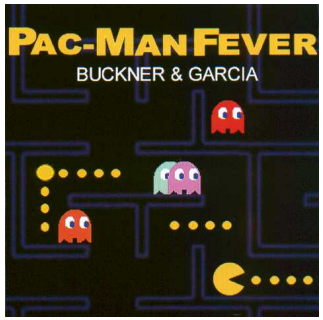
David Pichardie

INRIA Rennes - Bretagne Atlantique - projet Lande

Mobile code dilemmas...



Mobile code dilemmas...



Secure ?

Untrusted code

Host system

- The untrusted code may cause damages on the system
  - intern structure corruption
- The untrusted code may use too many resources
  - CPU, memory, SMS...
- The untrusted code may reveal confidential data to an attacker
  - phonebook, diary, geo-localisation, camera, audio-recorder...

Solutions

- Cryptographic authentication : a trusted source signs the code
  - we don't trust the code but its source (e.g. phone operator)
  - restricts the exchange possibilities : it's difficult to gain trust if you are not a big company

Solutions

- Cryptographic authentication : a trusted source signs the code
  - we don't trust the code but its source (e.g. phone operator)
  - restricts the exchange possibilities : it's difficult to gain trust if you are not a big company
- Dynamic checking (sand box, monitoring)
  - reduces the execution speed
  - programs may raise scaring security exceptions like :

    Your program as attempted a forbidden action !

    - annoying situation, specially when the program has been signed by a big company...
    - users could progressively loose confidence in mobile code security

Solutions

- Cryptographic authentication : a trusted source signs the code
  - we don't trust the code but its source (e.g. phone operator)
  - restricts the exchange possibilities : it's difficult to gain trust if you are not a big company
- Dynamic checking (sand box, monitoring)
  - reduces the execution speed
  - programs may raise scaring security exceptions like :

    Your program as attempted a forbidden action !

    - annoying situation, specially when the program has been signed by a big company...
    - users could progressively loose confidence in mobile code security
- Proof-Carrying Code (PCC)
  - no trust required in the code producer
  - no runtime overhead

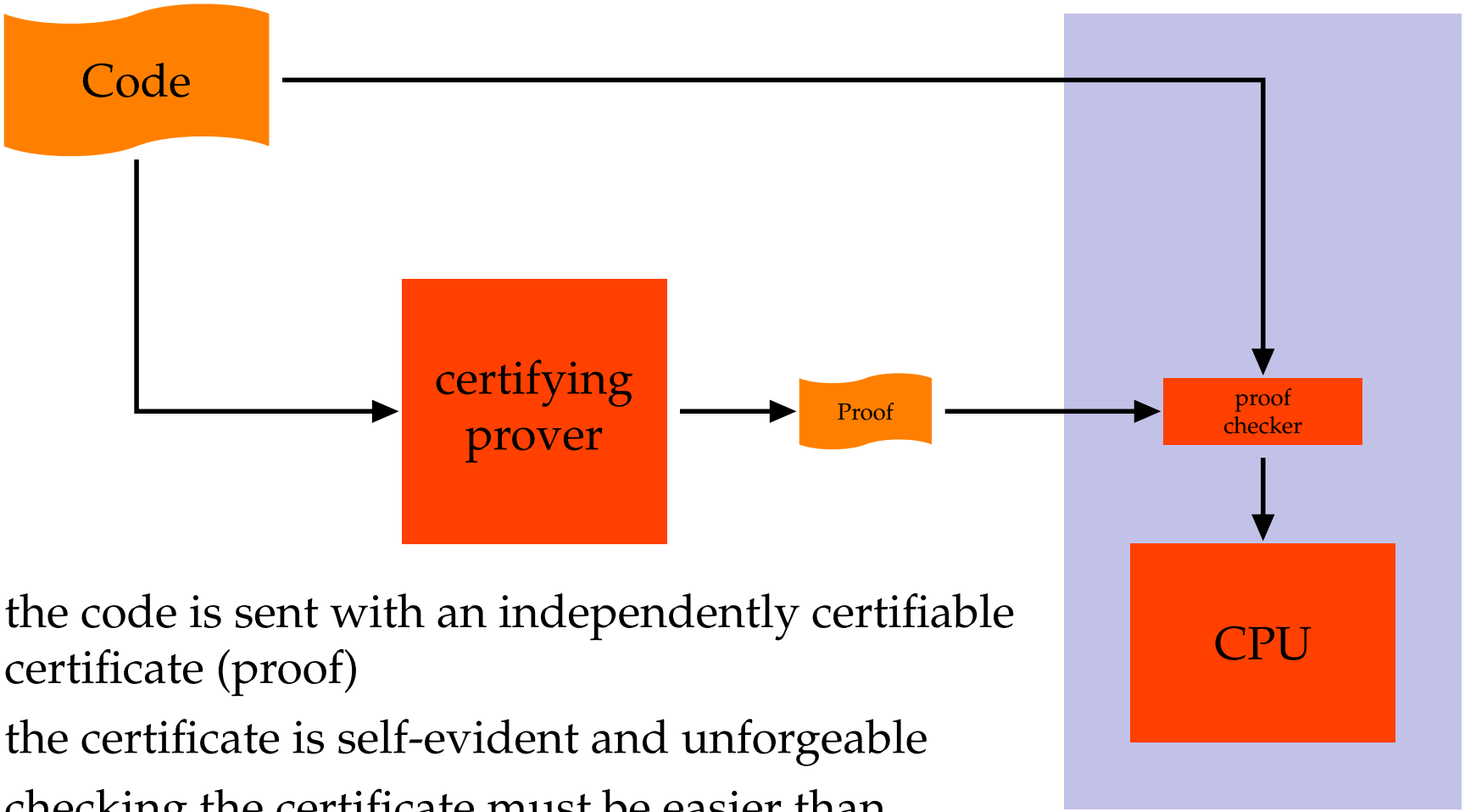
Solutions

- Cryptographic authentication : a trusted source signs the code
  - we don't trust the code but its source (e.g. phone operator)
  - restricts the exchange possibilities : it's difficult to gain trust if you are not a big company
- Dynamic checking (sand box, monitoring)
  - reduces the execution speed
  - programs may raise scaring security exceptions like :

    Your program as attempted a forbidden action !

    - annoying situation, specially when the program has been signed by a big company...
    - users could progressively loose confidence in mobile code security
- Proof-Carrying Code (PCC)
  - no trust required in the code producer
  - no runtime overhead
- The three approaches can be combined to take advantages of all

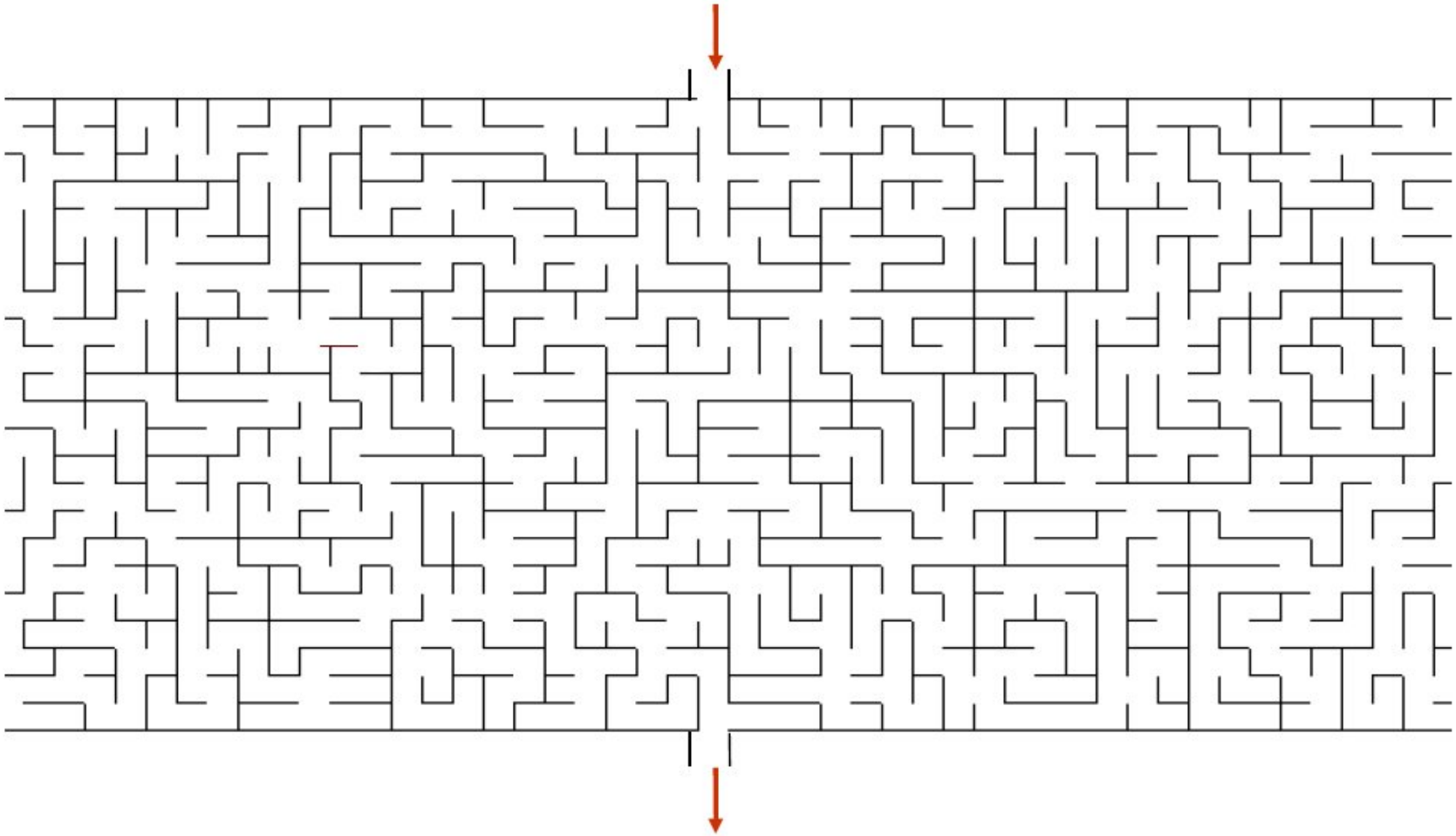
Proof carrying code : principles



- the code is sent with an independently certifiable certificate (proof)
- the certificate is self-evident and unforgeable
- checking the certificate must be easier than producing it

The maze metaphor

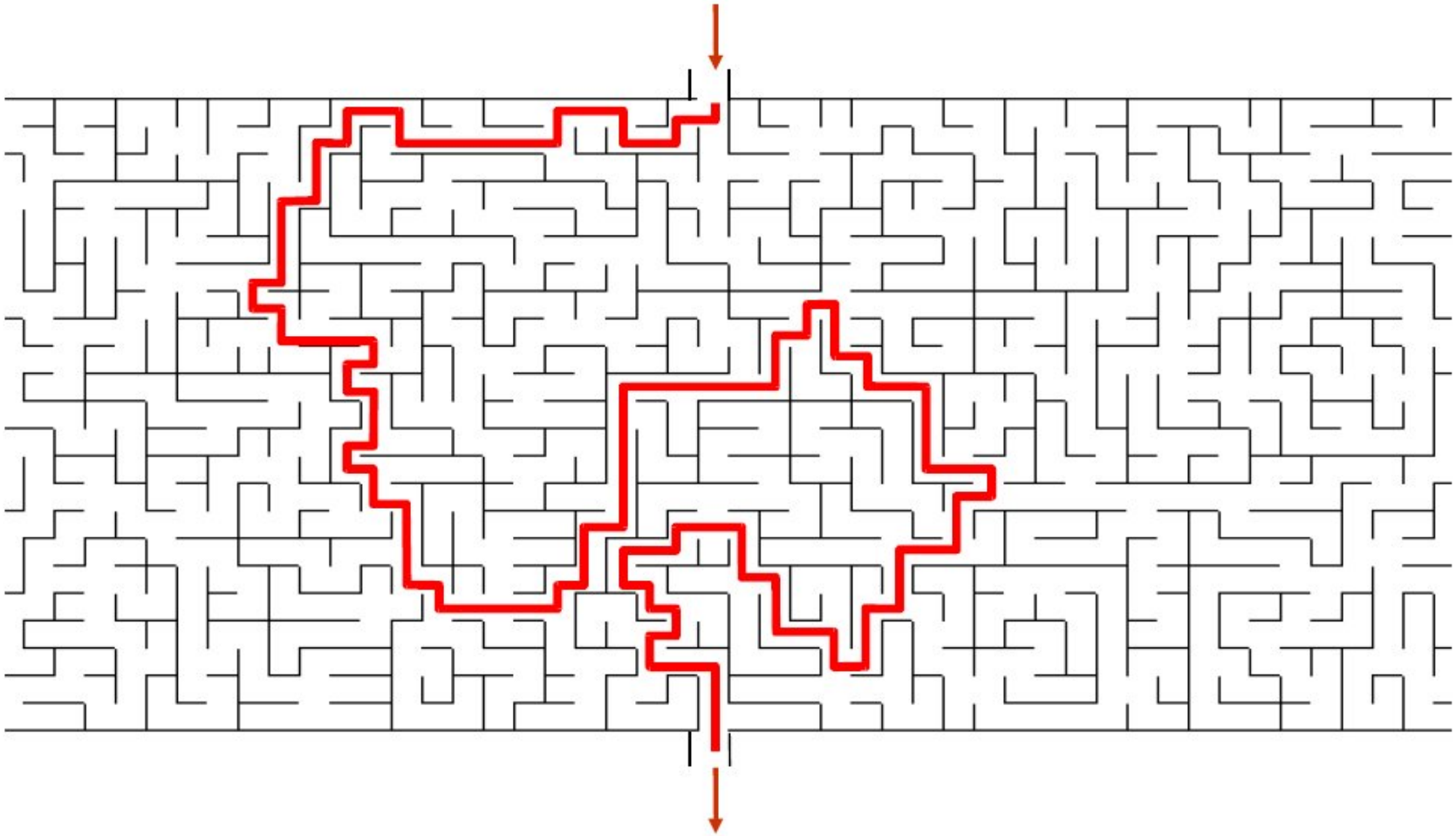
©G. Necula



program = maze

The maze metaphor

©G. Necula

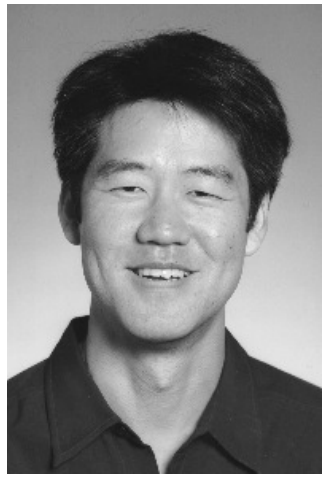


program = maze

proof = red path

Outline

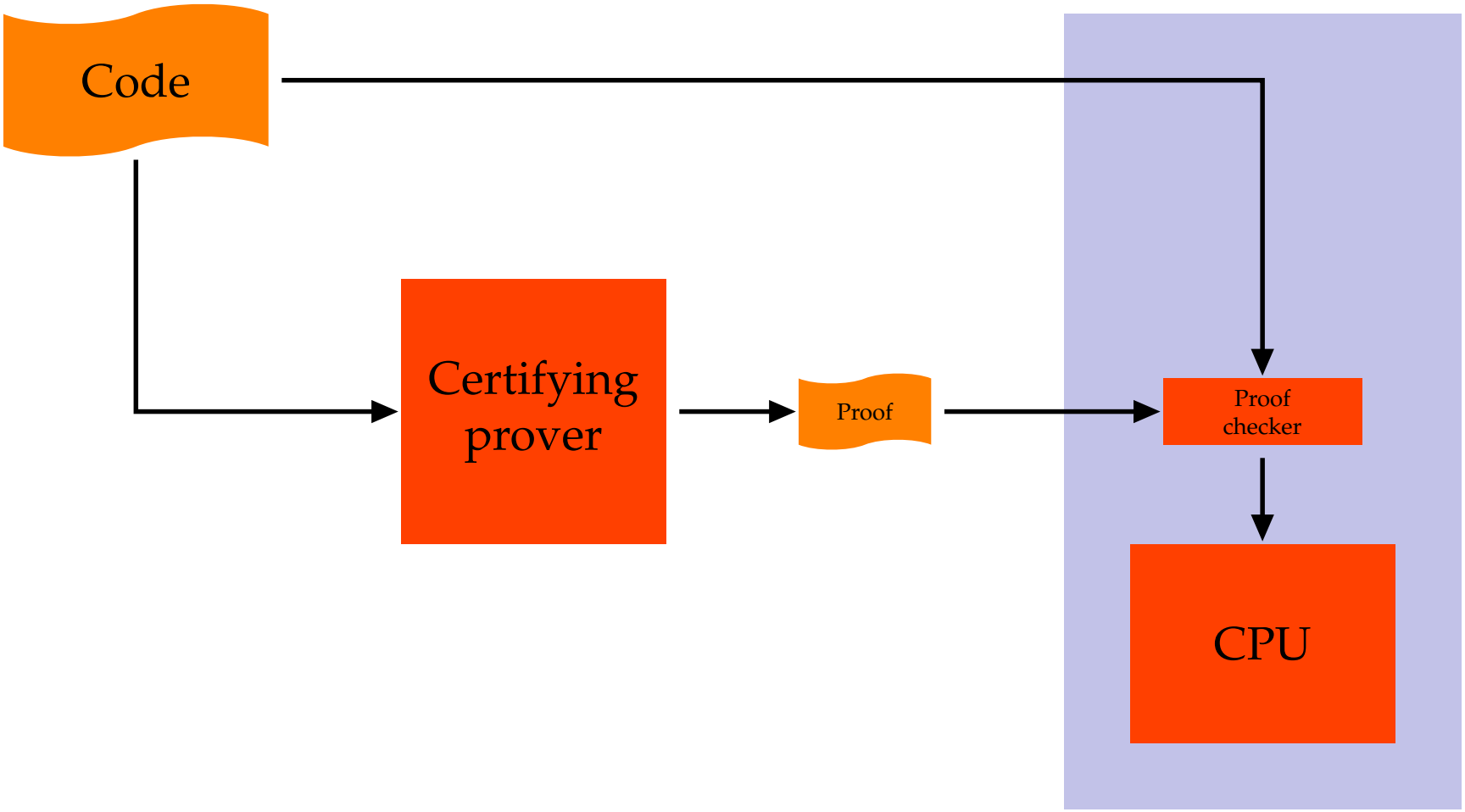
The Proof Carrying Code's pioneers



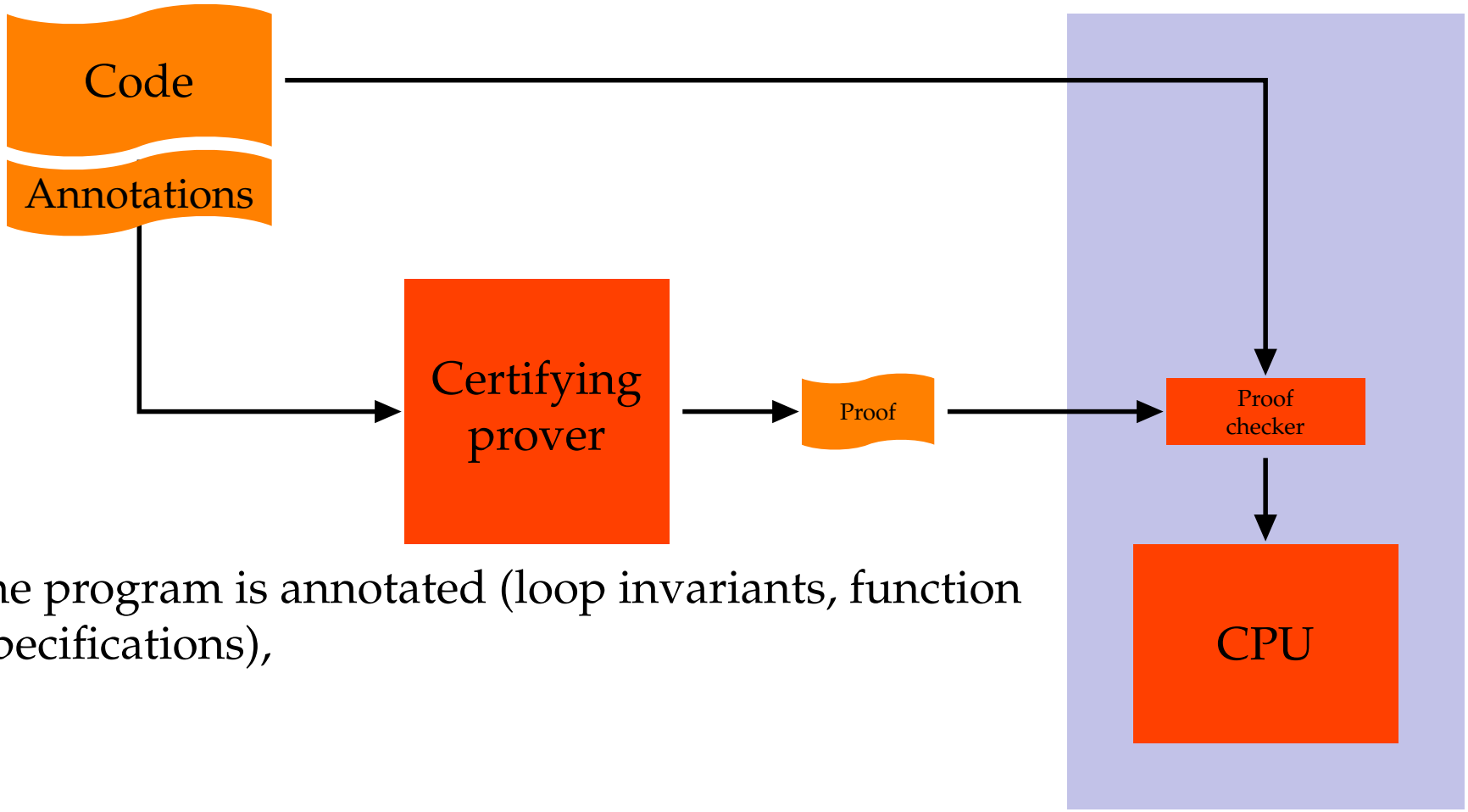
First proposed by Georges Necula (Berkley) and Peter Lee (CMU).

- 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

Proof carrying code : standard framework

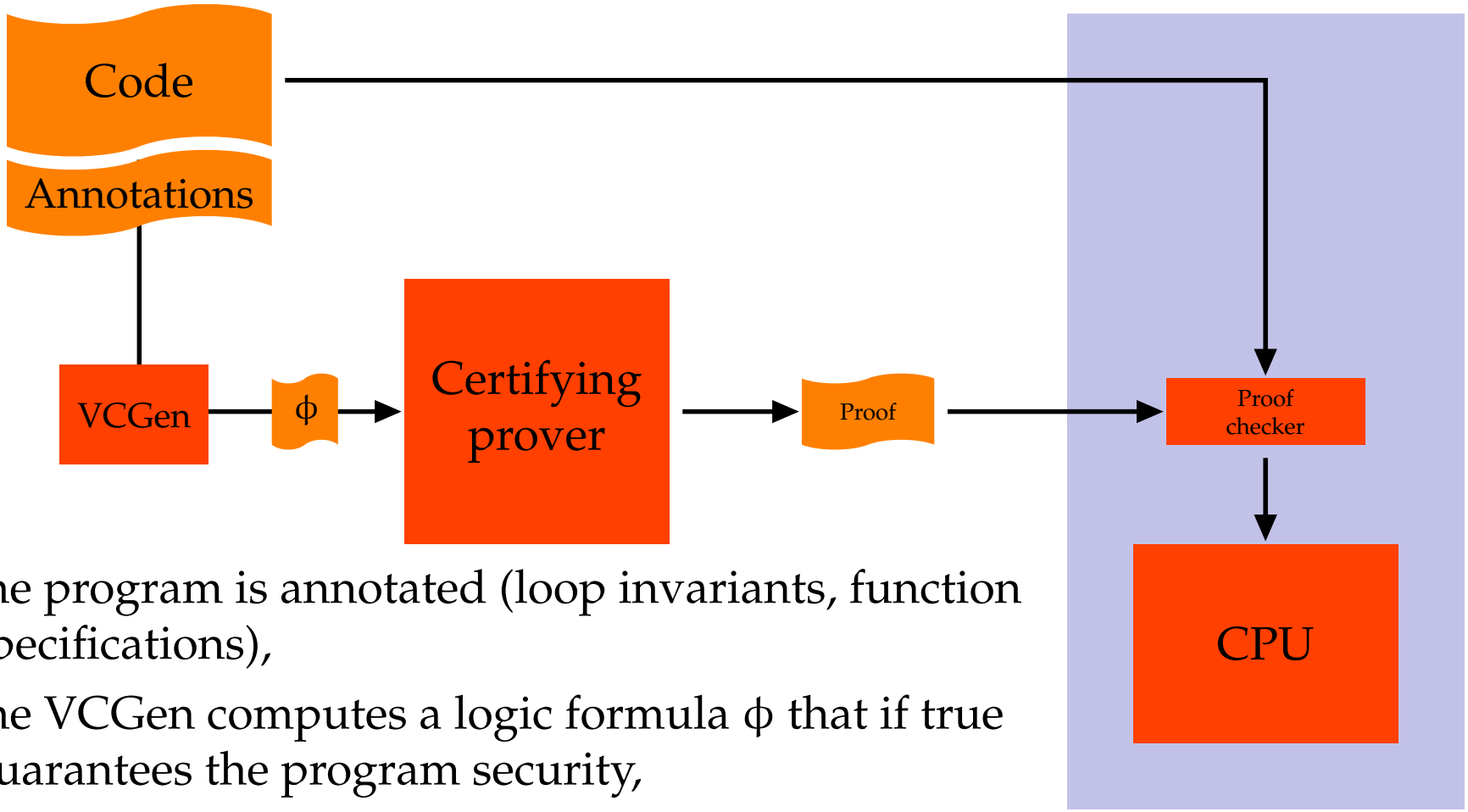


Proof carrying code : standard framework



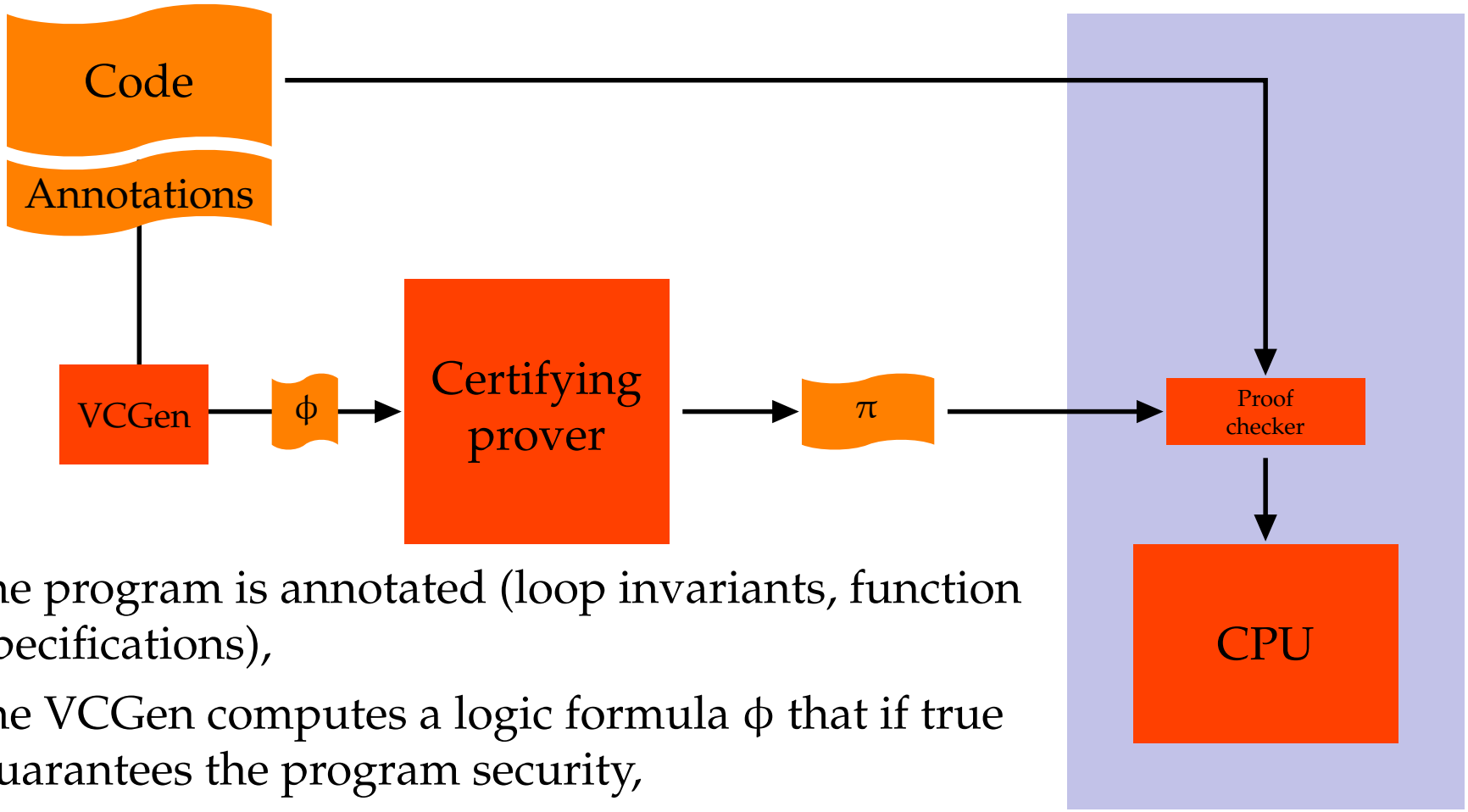
- the program is annotated (loop invariants, function specifications),

Proof carrying code : standard framework



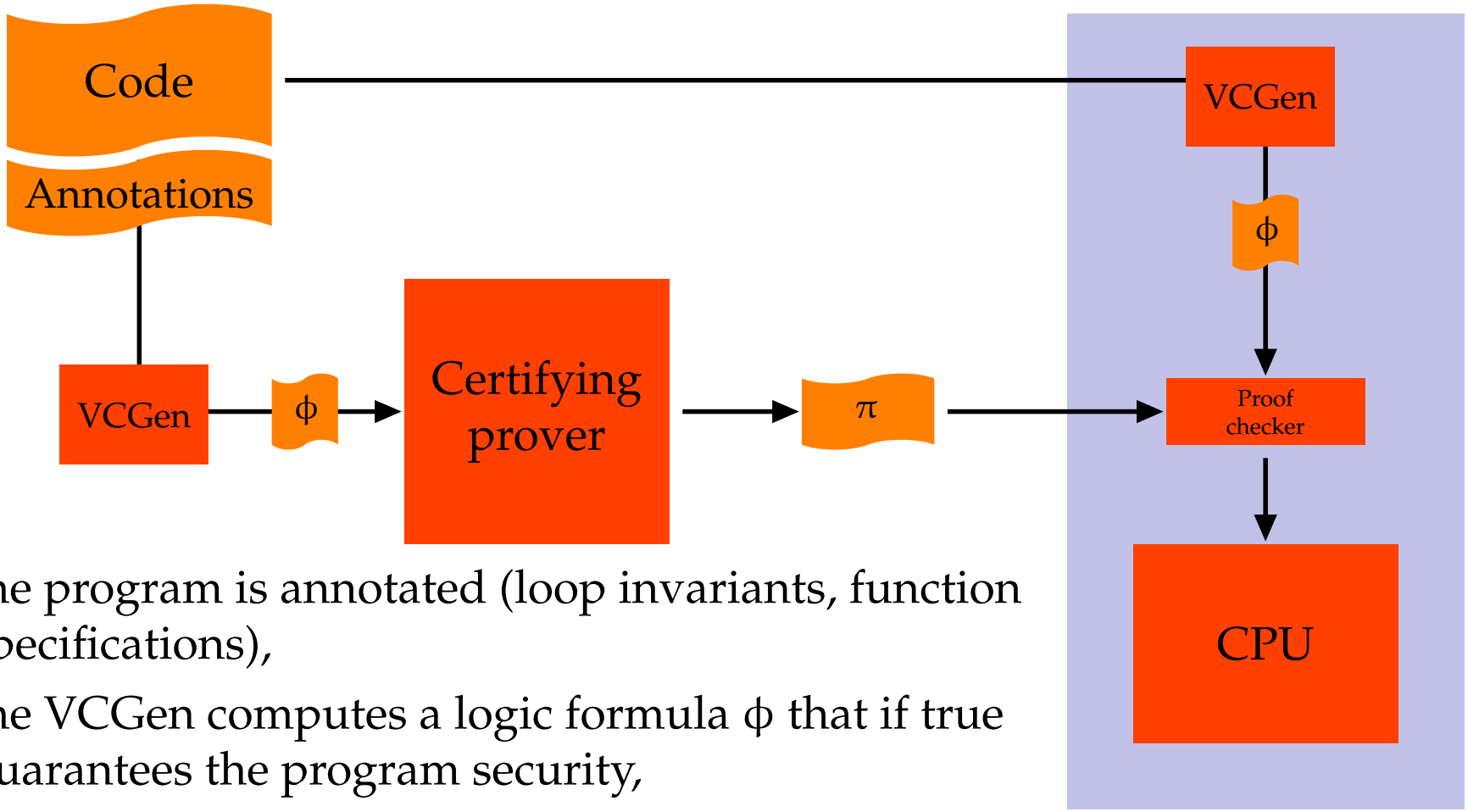
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,

Proof carrying code : standard framework



- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,

Proof carrying code : standard framework



- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π is a valid proof of ϕ.

The representation and checking for proofs

In this seminal work Necula and Lee used LF[1]

- a logical framework which allows to define logic systems with their proof rules and provide a generic proof checker

Advantages :

- the verifier is generic, efficient, and small (and then certainly sound)

Disadvantages :

- certificates are big (sometimes 1000×code !)

Variants :

- LF_i is a variant[2] where the proof checker infers by itself fragments of the proof (2.5×code)
- *Oracle-based* proofs[3] reduces drastically this factor (12% of the code)

[1] R. Harper, F. Honsell and G. Plotkin. *A framework for defining logics.* Journal of the ACM, 1993.
[2] G.C. Necula and P. Lee. *Efficient Representation and Validation of Proofs.* LICS'98
[3] G.C. Necula and S. P. Rahul. *Oracle-based checking of untrusted software.* POPL'01

Certifying prover

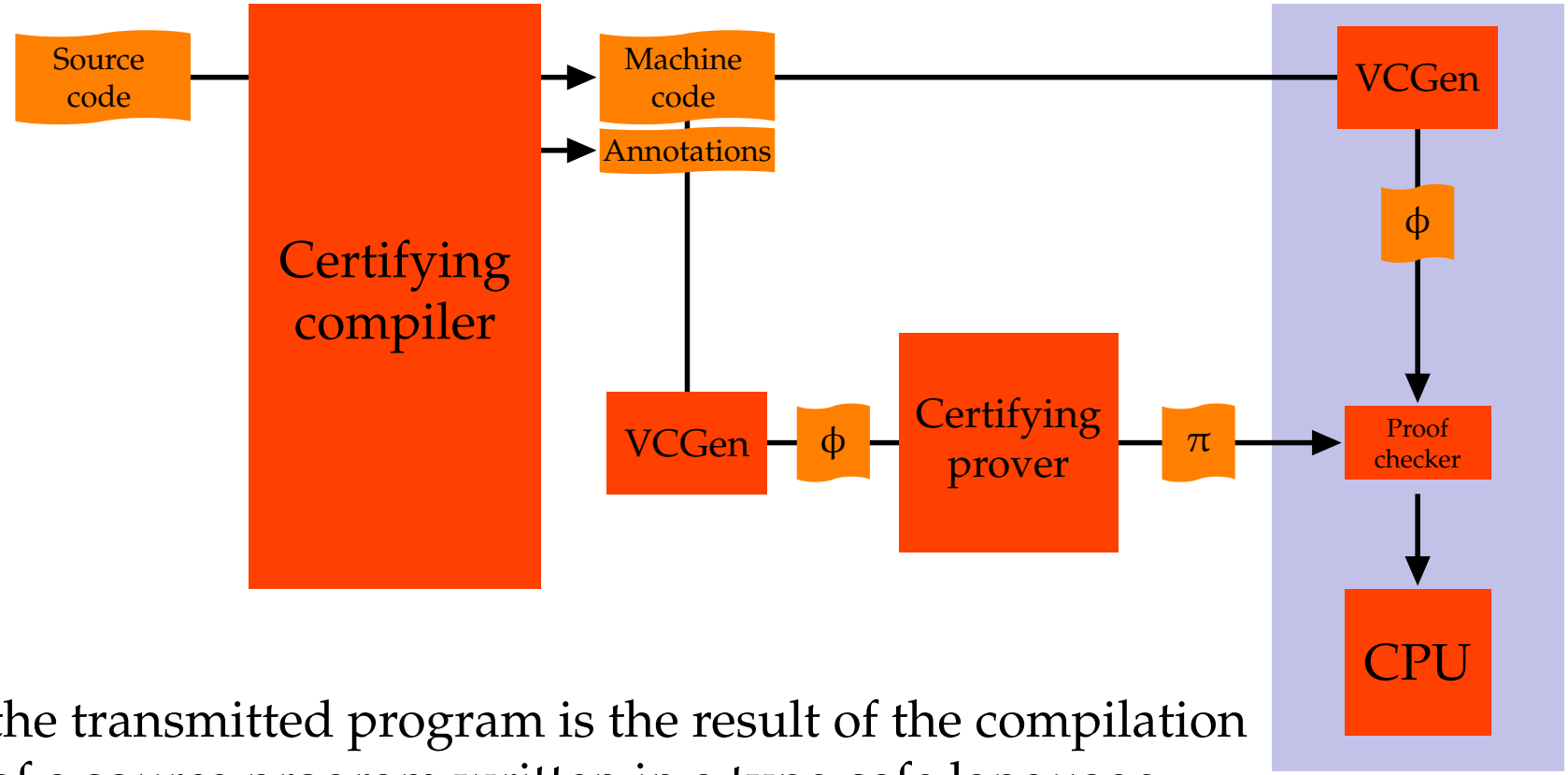
The certifying prover

- automatically proves the verification conditions (VC)
  - VC must fall in some logic fragments whose decision procedures have been implemented in the prover
- in the PCC context, proving is not sufficient, detailed proof must be generated too
  - like decision procedures in skeptical proof assistants (Coq, Isabelle, HOL light,...)
  - proof producing decision procedures are more and more considered as an important software engineering practice to develop proof assistants

Necula's certifying prover includes

- congruence closure and linear arithmetic decision procedures
- with a Nelson-Oppen architecture for cooperating decision procedures

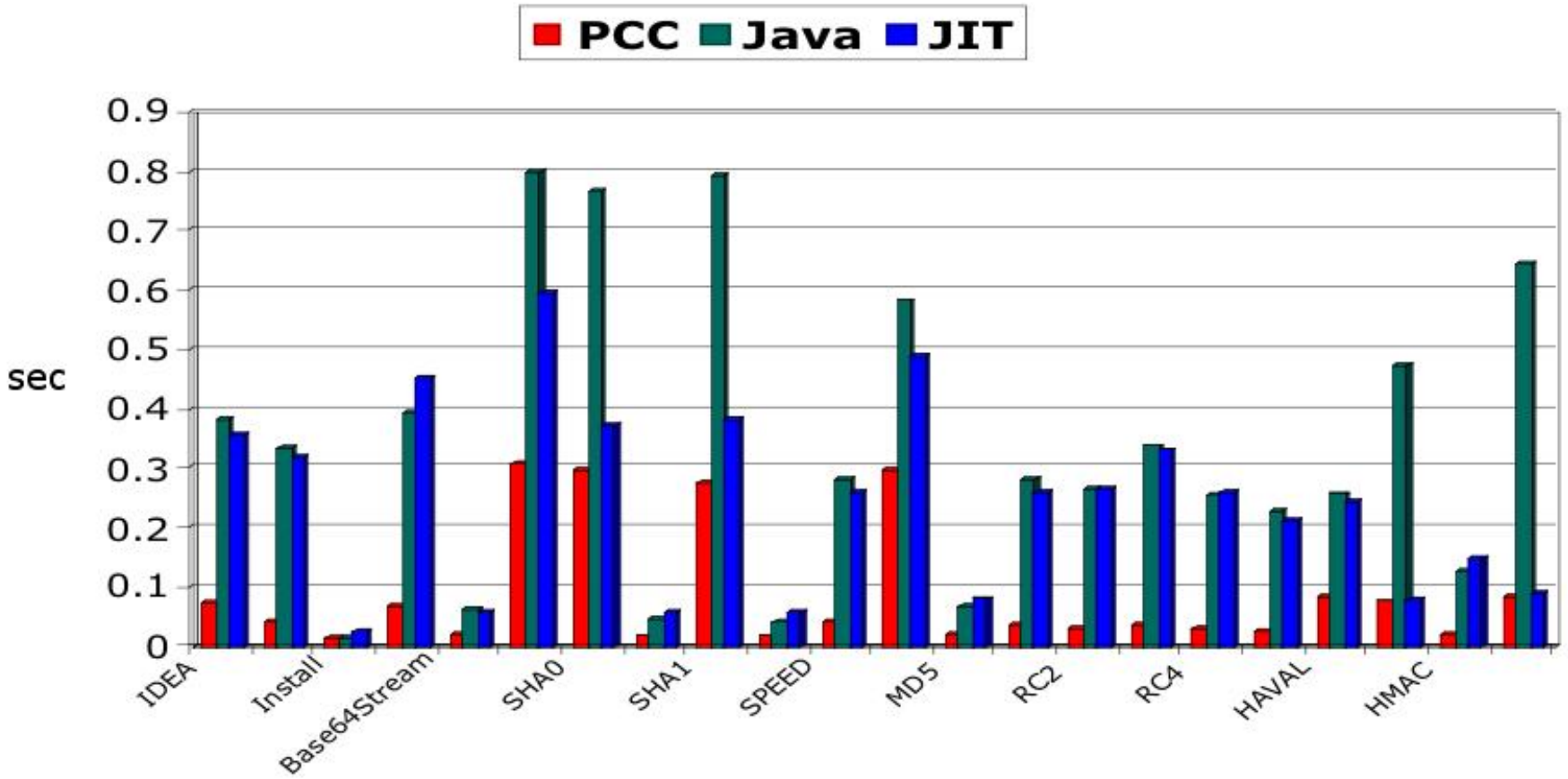
Annotation generation



- the transmitted program is the result of the compilation of a source program written in a type-safe language
- the role of the certifying compiler is
  - to check type-safety of the source program
  - to generate corresponding annotations in the machine code to help the VCGen

One example of PCC's success

The Touchstone system[4] verifies that optimized native machine code produced by a special Java compiler is memory safe.



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

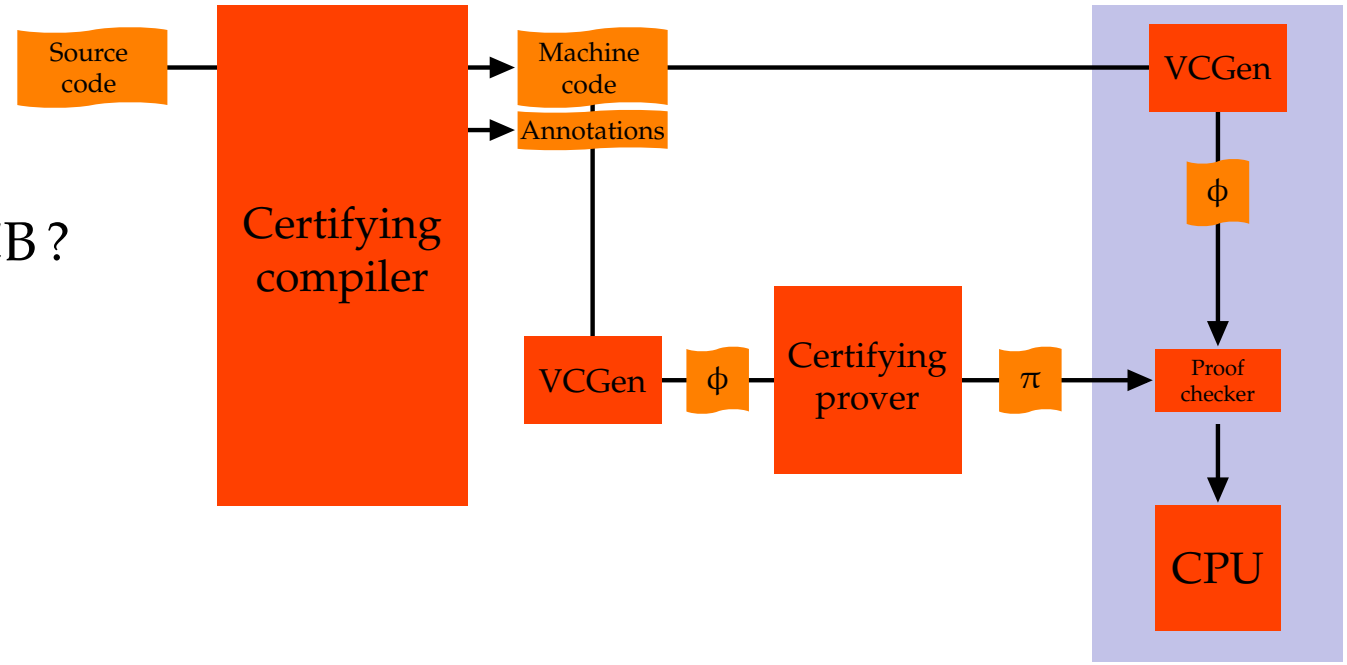
Intermediate conclusions on standard PCC

- an astonish mix between logic, program verification and concrete security issues,
- still a busy research area,
- PCC must demonstrate its ability to enforce more complex security policies while conciliating many features :
  - small certificates,
  - efficient verifier,
  - sound verifier,
  - effective tools to build certificates,
  - effective integration in tomorrow global computers.

Trusted Computing Base (TCB)

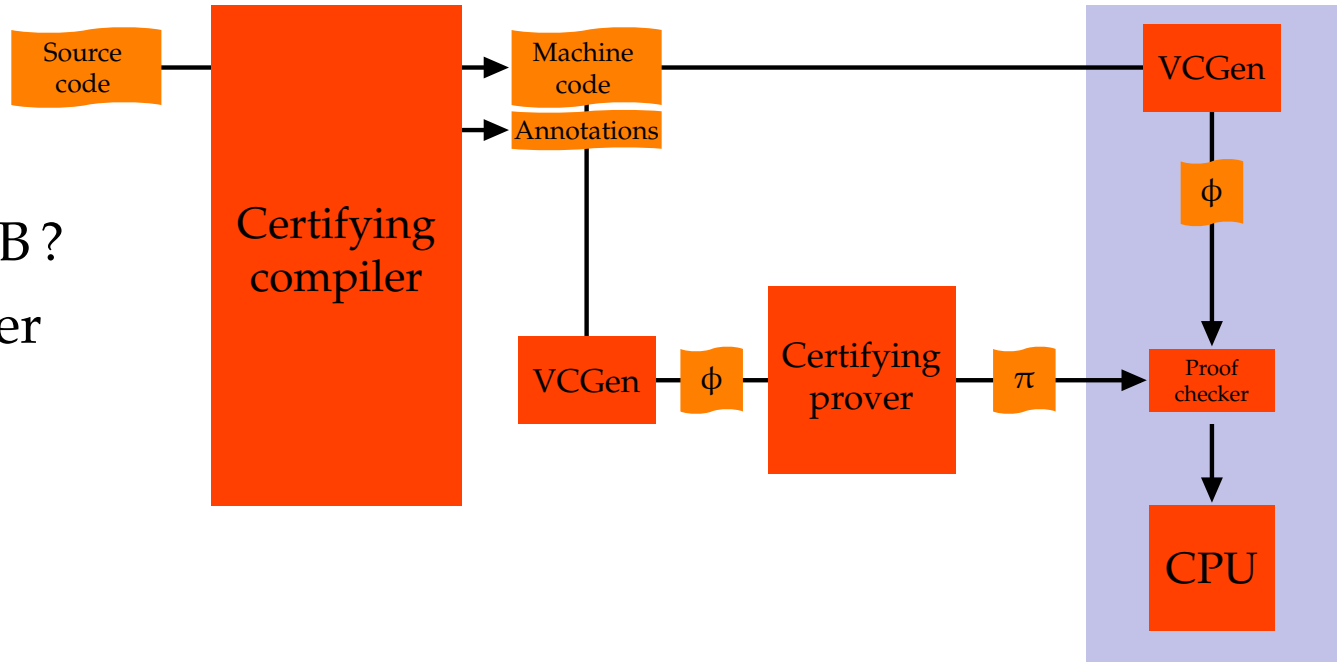
The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB ?



Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

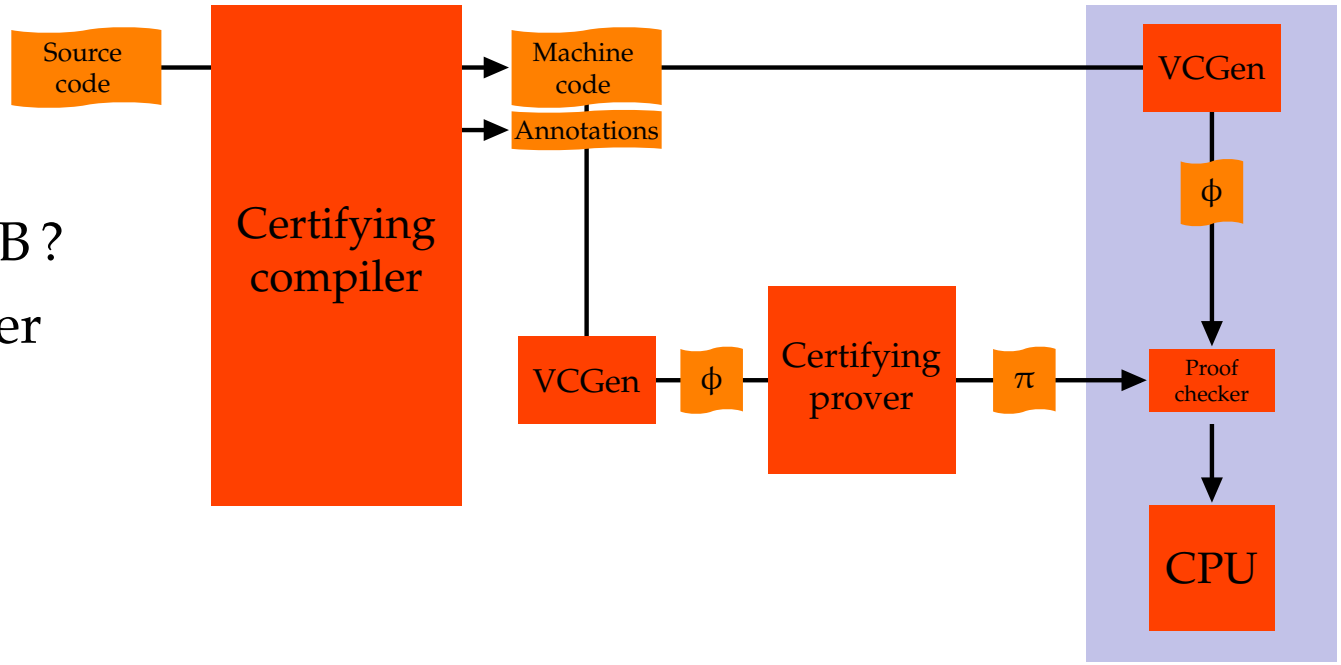


What is the PCC TCB ?

- the proof checker
- the VCGen
- the CPU

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.



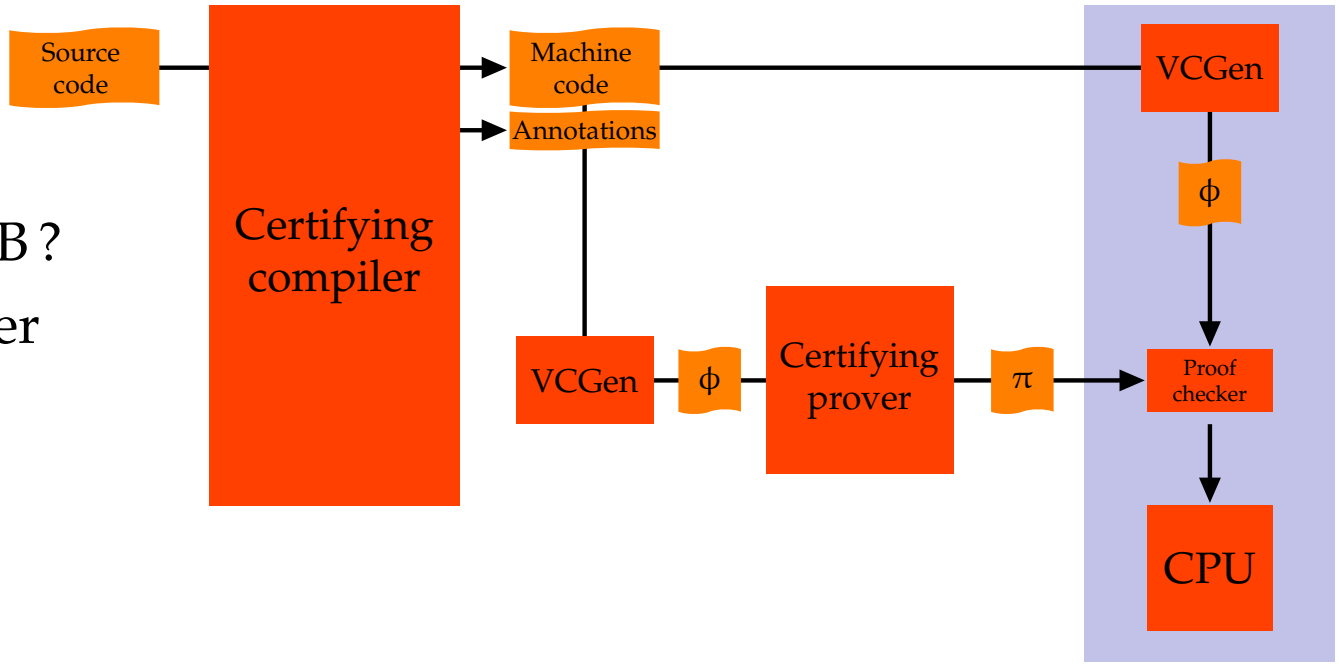
What is the PCC TCB ?

- the proof checker
- the VCGen
- the CPU

You don't need to trust ...

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.



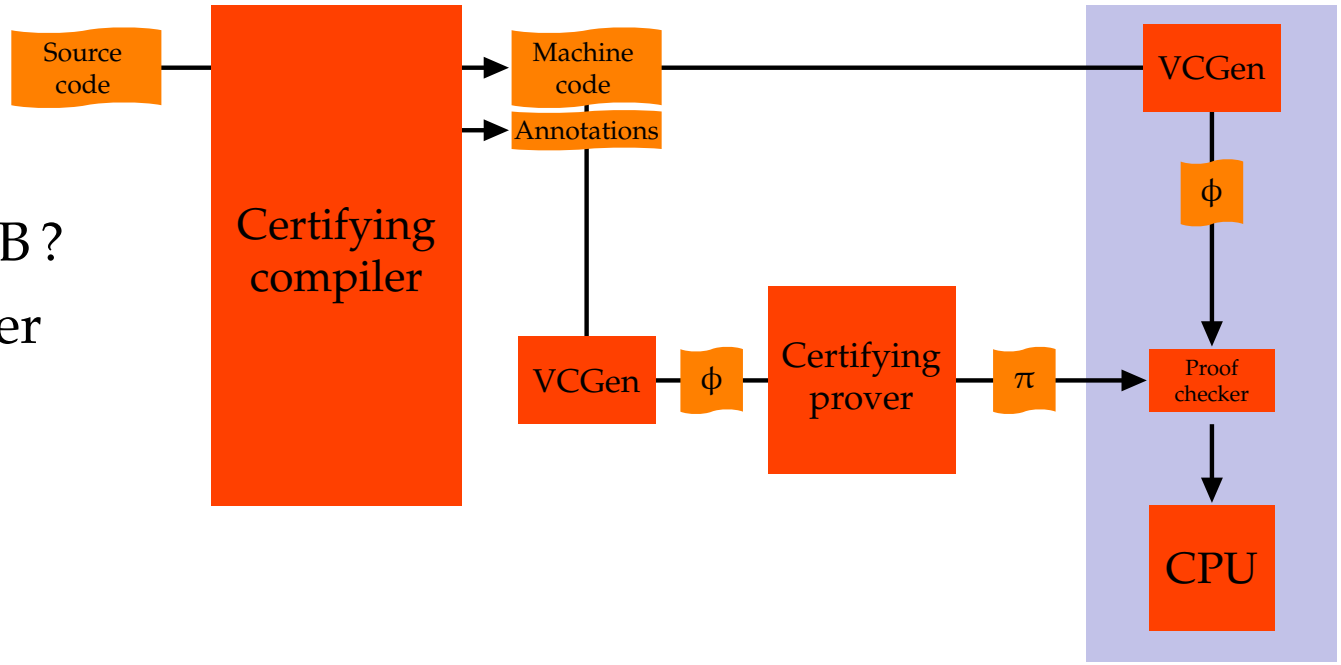
What is the PCC TCB ?

- the proof checker
- the VCGen
- the CPU

You don't need to trust the compiler ...

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.



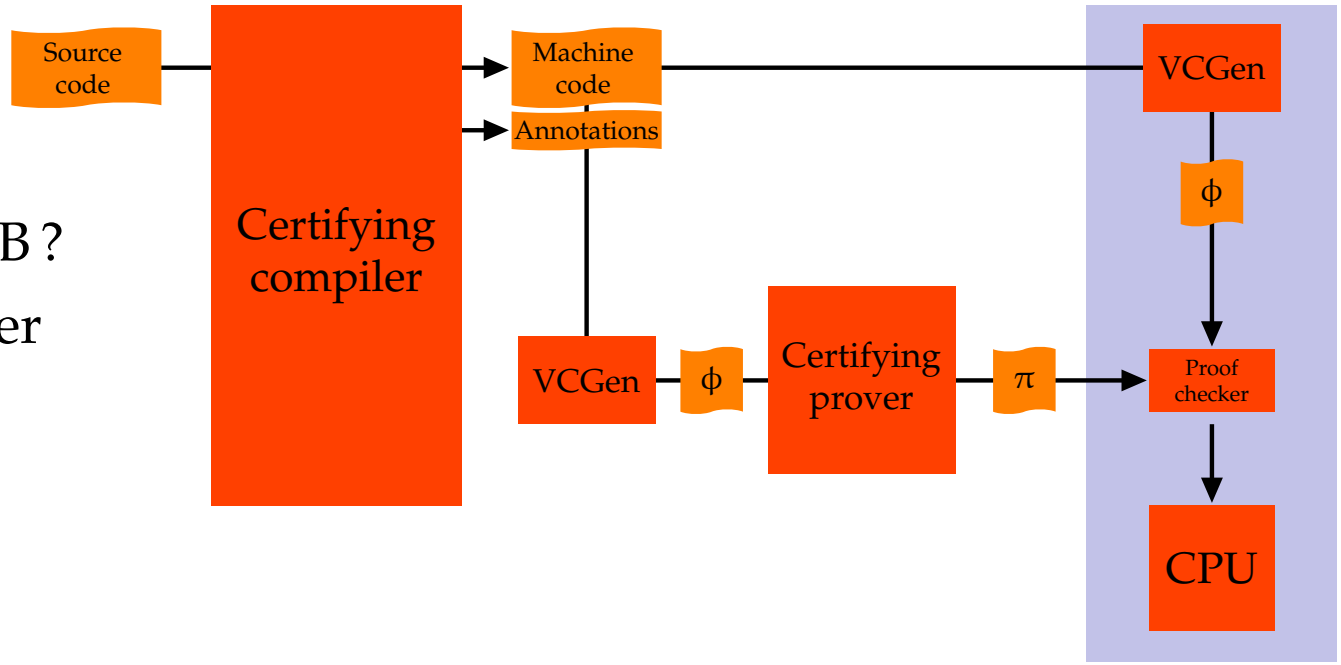
What is the PCC TCB ?

- the proof checker
- the VCGen
- the CPU

You don't need to trust the compiler, the annotations ...

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.



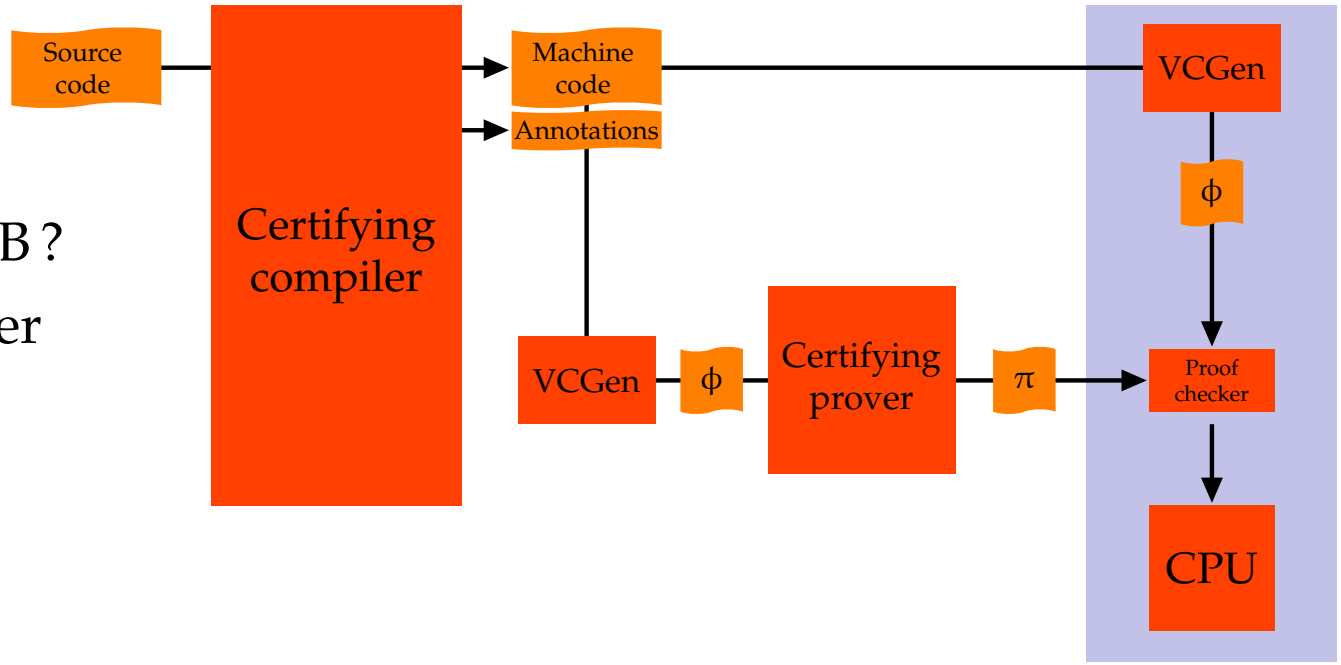
What is the PCC TCB ?

- the proof checker
- the VCGen
- the CPU

You don't need to trust the compiler, the annotations, the prover ...

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.



What is the PCC TCB ?

- the proof checker
- the VCGen
- the CPU

You don't need to trust the compiler, the annotations, the prover, the proof ...

Other instances of PCC (1/2)

An active trend in PCC has focused on soundness

- Touchstone has achieved an impressive level of scalability (programs with about one million instructions)
- but[5] *"[...], there were errors in that code that escaped the thorough testing of the infrastructure"*.
- the weak point was the VCGen (23,000 lines of C...)

The following work have tried to reduce the size of the TCB

- by *simply* removing the VCGen !
  - A.W. Appel. *Foudational Proof-Carrying Code.* LICS'01
- by certifying in a proof assistant the VCGen
  - M. Wildmoser and T. Nipkow. *Asserting Bytecode Safety.* ESOP'05
- by certifying in a proof assistant the checker
  - TAL (next slide), certified abstract interpretation (Lecture 4)

[5] G.C. Necula and R.R. Schneck. *A Sound Framework for Untrusted Verification-Condition Generators.* LICS'03

Other instances of PCC (2/2)

Some work use checkers and proof formats specific to one security property

- Rose's *Lightweight Bytecode Verifier*
  - ensures type-safety of Java bytecode programs,
  - the proof/certificate is a (partial) type annotation,
  - now part of the Sun KVM (JVM for embedded devices).
- TAL[6] Typed Assembly Language for advanced memory safety
- Abstraction-Carrying Code[7] : PCC by abstract interpretation

Such work lose the genericity of the seminal PCC proof checker, but can be machine checked

- Lightweight Bytecode Verifier (Klein & Nipkow, Barthe & Dufay)
- TAL (Krary)
- Abstraction-Carrying Code (Besson & Jensen & Pichardie)

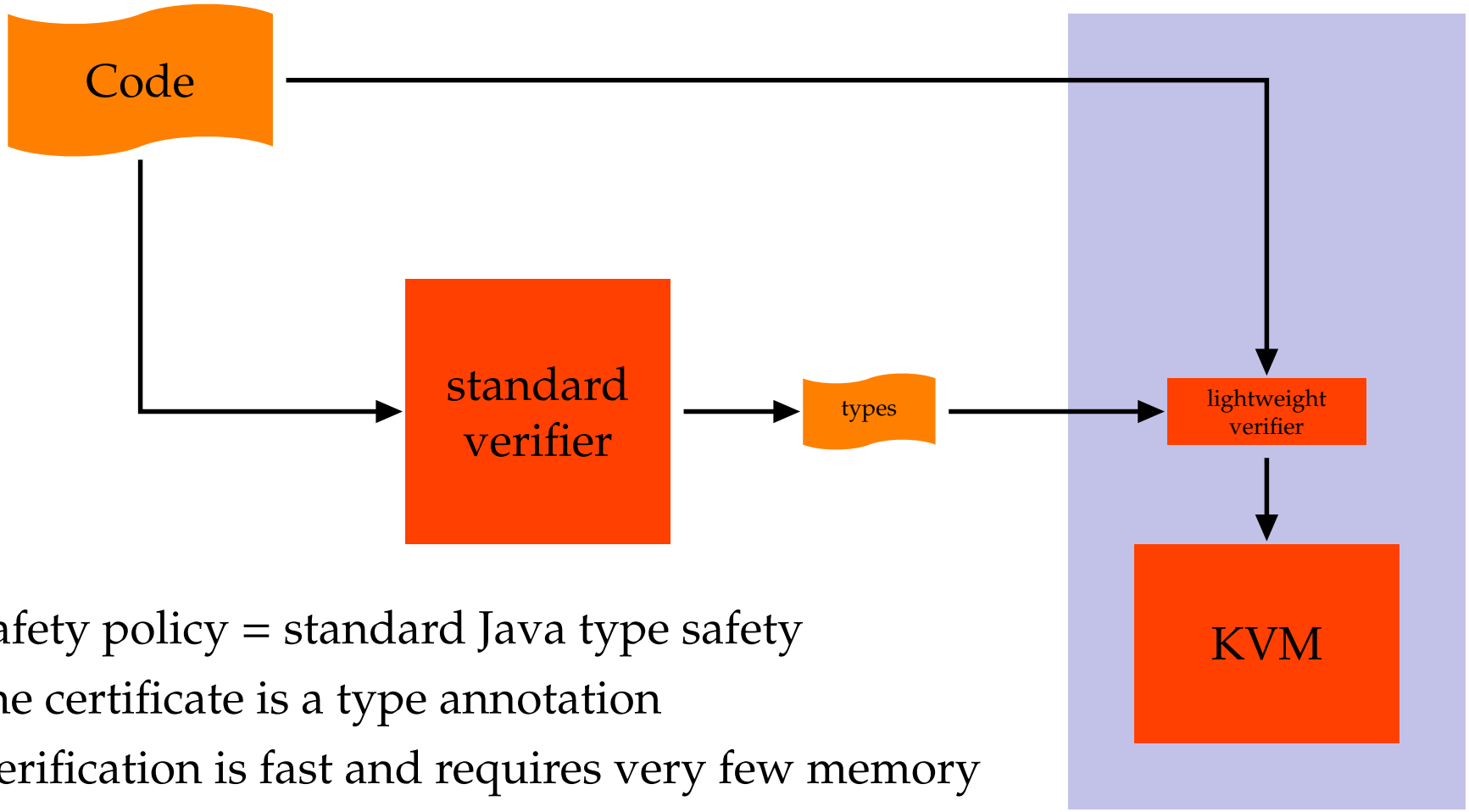
---

[6] G. Morrisett, D. Walker, K. Crary and Neal Glew. *From System F to Typed Assembly Language.* POPL'98

[7] E. Albert, G. Puebla and M. V. Hermenegildo. *Abstraction-Carrying Code.* LPAR'05

PCC *à la Rose*

Rose's *Lightweight Bytecode Verifier* is now part of the Sun KVM (JVM for embedded devices).



- safety policy = standard Java type safety
- the certificate is a type annotation
- verification is fast and requires very few memory

Discussing Rose's approach

Advantages :

- certificates are small,
- certificate verification is fast,
- certificate generation is automatic.

Disadvantages :

- security policy is quite simple
- checker is very ad-hoc (soundness ?)

Discussing Rose's approach

Advantages :

- certificates are small,
- certificate verification is fast,
- certificate generation is automatic.

Disadvantages :

- security policy is quite simple
- checker is very ad-hoc (soundness ?)
  - formally established using a proof assistant (Nipkow & al)

Discussing Rose's approach

Advantages :

- certificates are small,
- certificate verification is fast,
- certificate generation is automatic.

Disadvantages :

- security policy is quite simple
- checker is very ad-hoc (soundness ?)
  - formally established using a proof assistant (Nipkow & al)

How can we generalise the approach while keeping a machine checked soundness proof ?

Our approach

Generalisation

- based on Abstract Interpretation

Machine checked soundness proof

- rely on the methodology/libraries proposed in Pichardie's Phd work

Case study

- array access safety for an imperative fragment of Java

Abstract interpretation

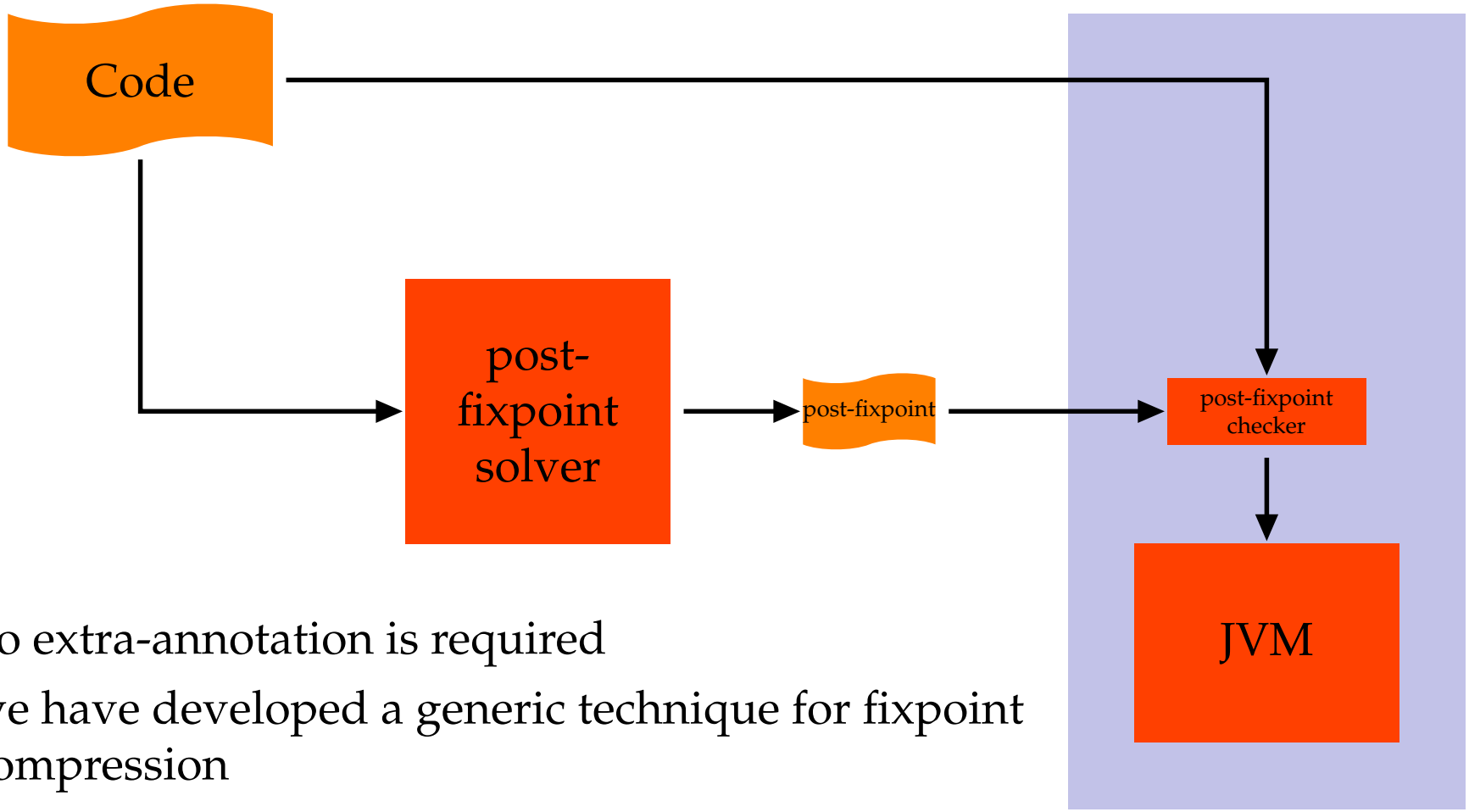
Abstract Interpretation [Cousot and Cousot 77] is a method for designing approximate semantics of programs.

- application to static analysis : static analysers are computable approximate semantics of programs
- a static analysis is presented as a post-fixpoint $[\![p]\!]^\sharp$ of a functional $F^\sharp$ in a lattice

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

- $[\![p]\!]^\sharp$ is computed by complex iterative methods
- but checking a given post-fixpoint is very fast !

Abstract Interpretation based PCC



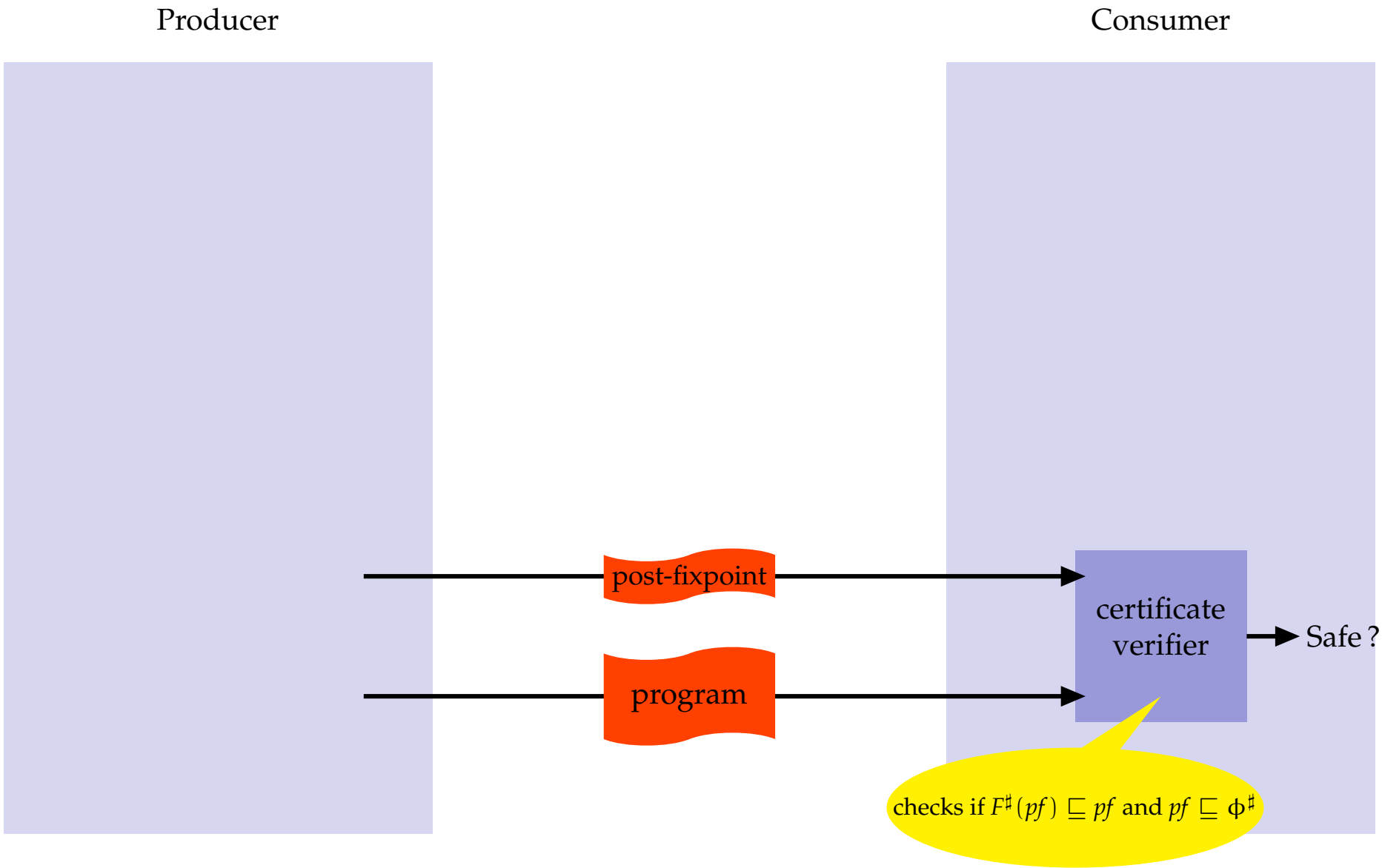
- no extra-annotation is required
- we have developed a generic technique for fixpoint compression
- but each post-fixpoint checker is ad-hoc : must be formally proved sound !

PCC by abstract interpretation

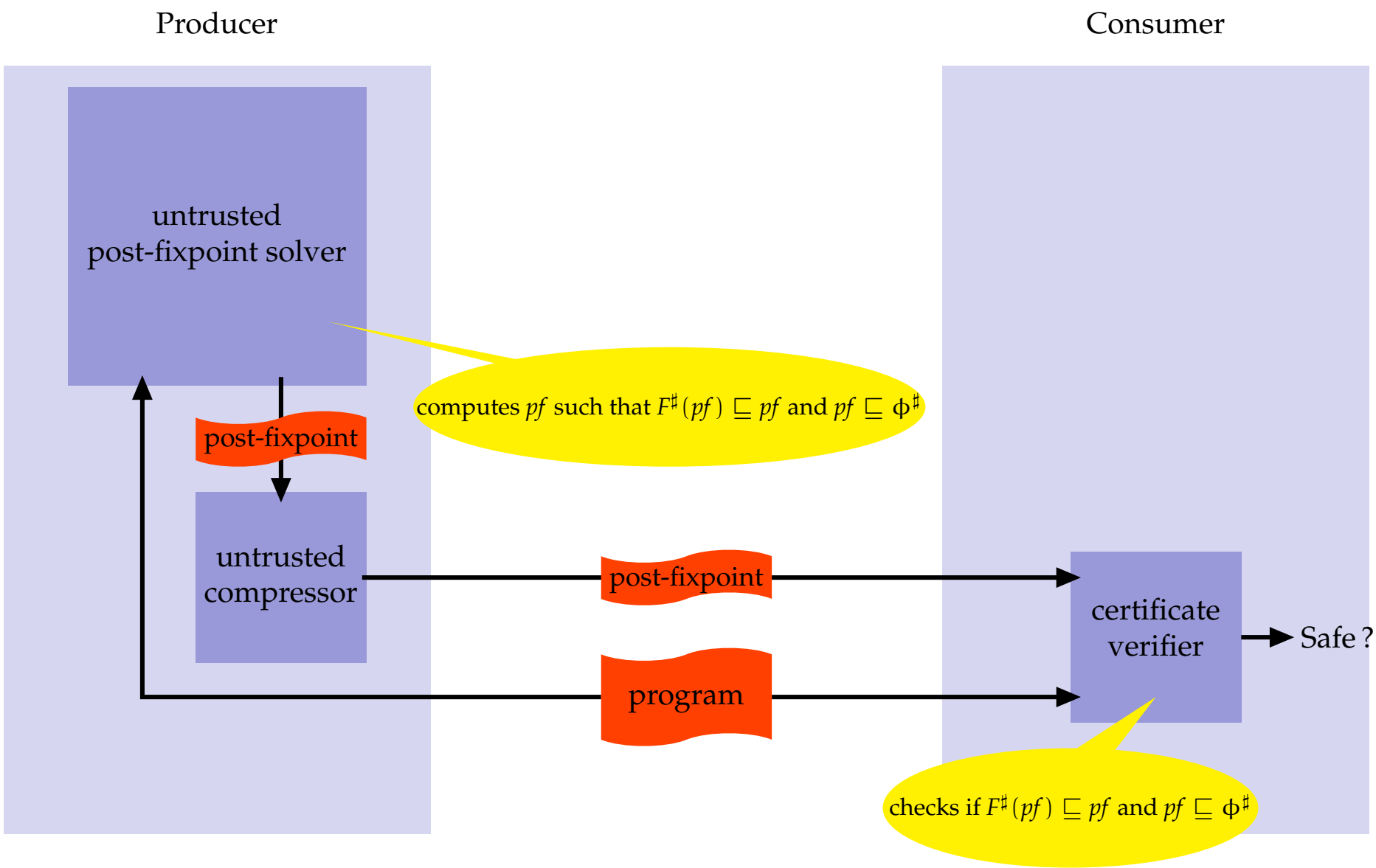
Producer

Consumer

PCC by abstract interpretation



PCC by abstract interpretation



Certified PCC by abstract interpretation

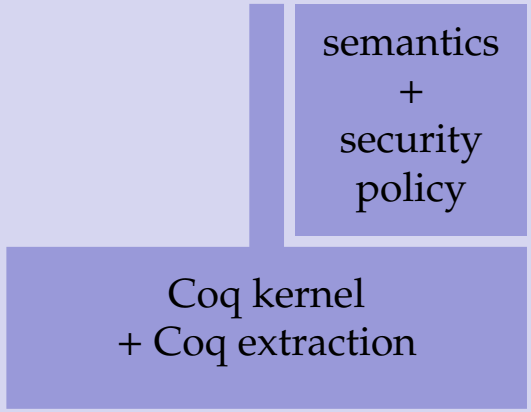
Producer

Consumer

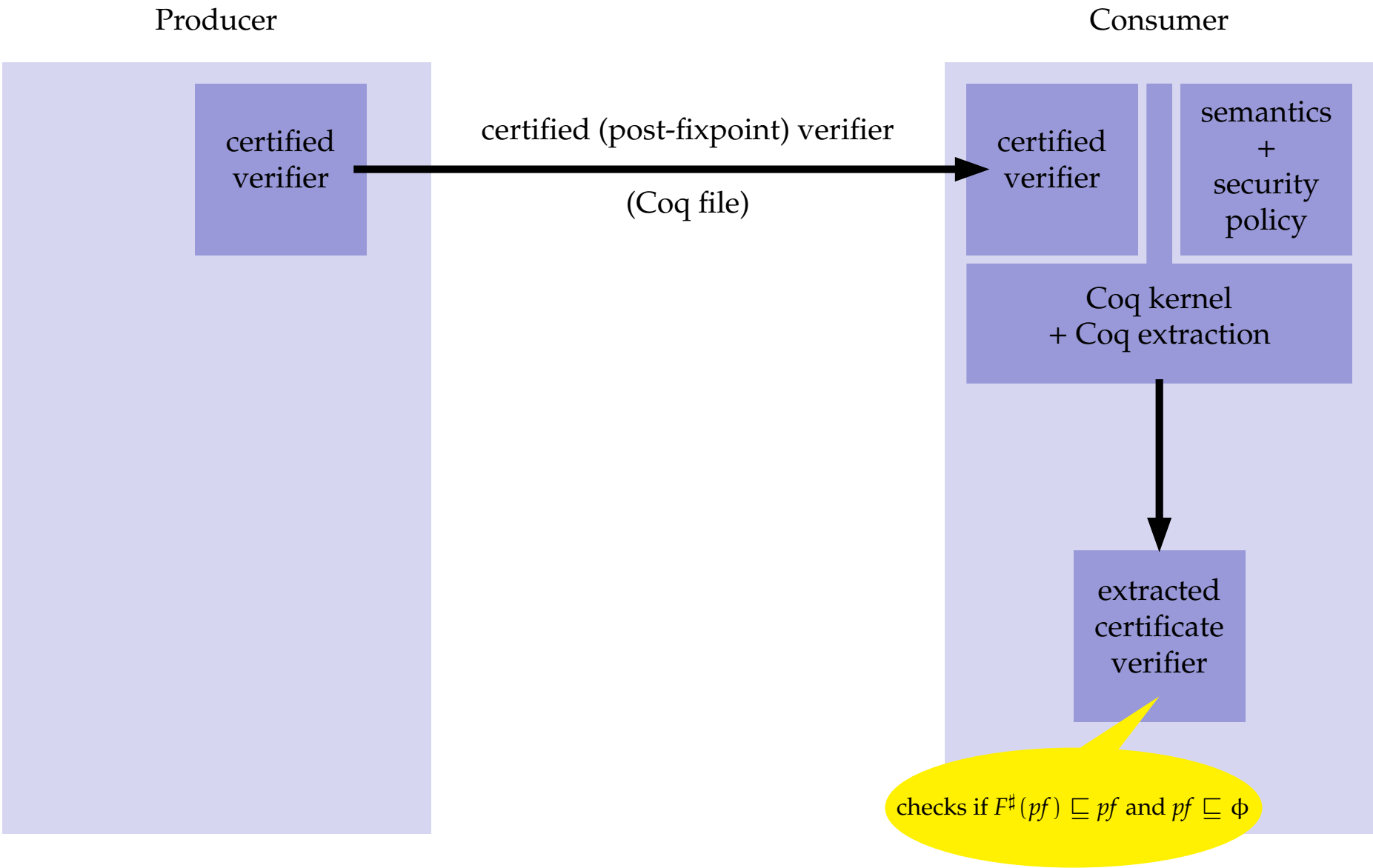
Certified PCC by abstract interpretation

Producer

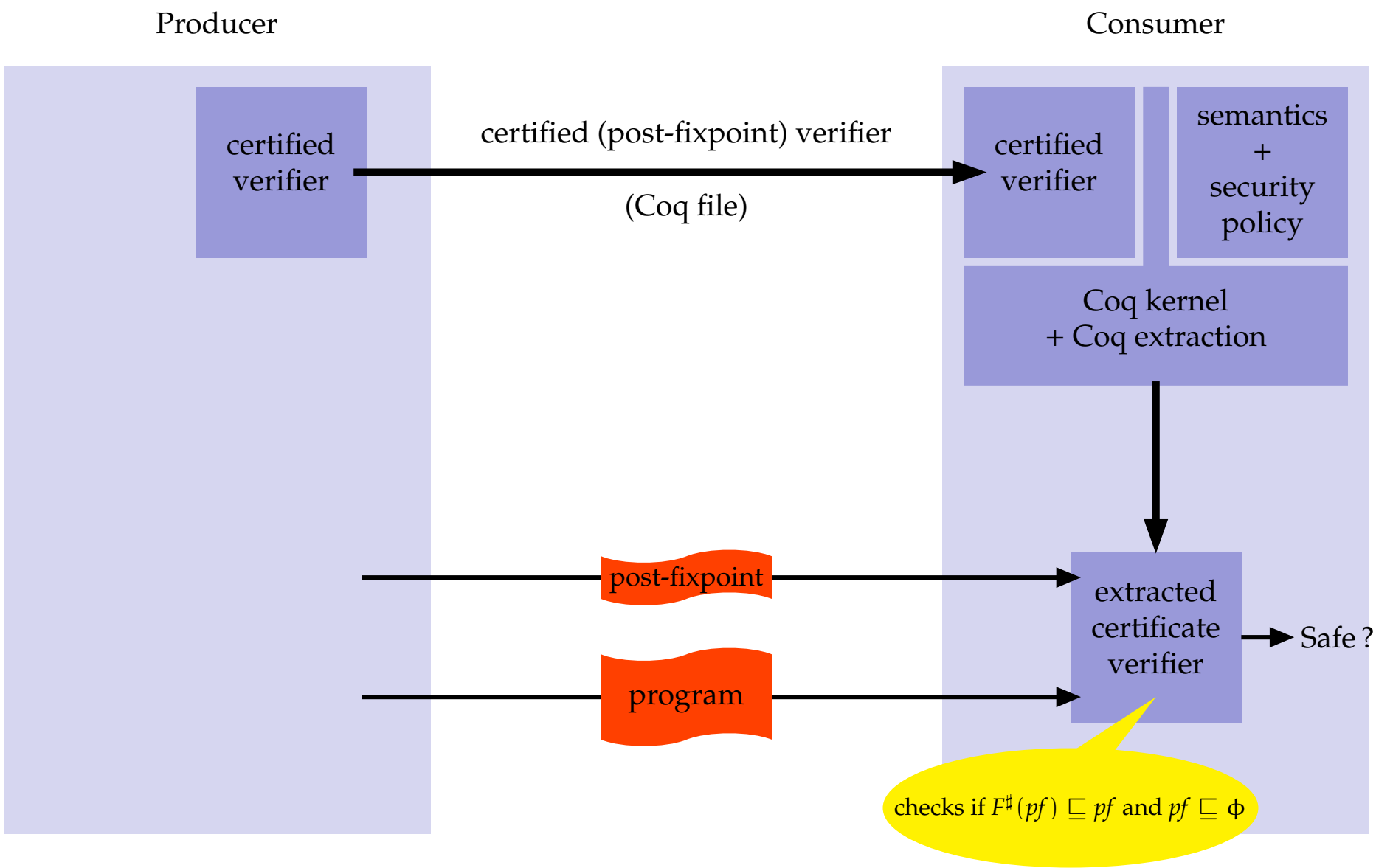
Consumer



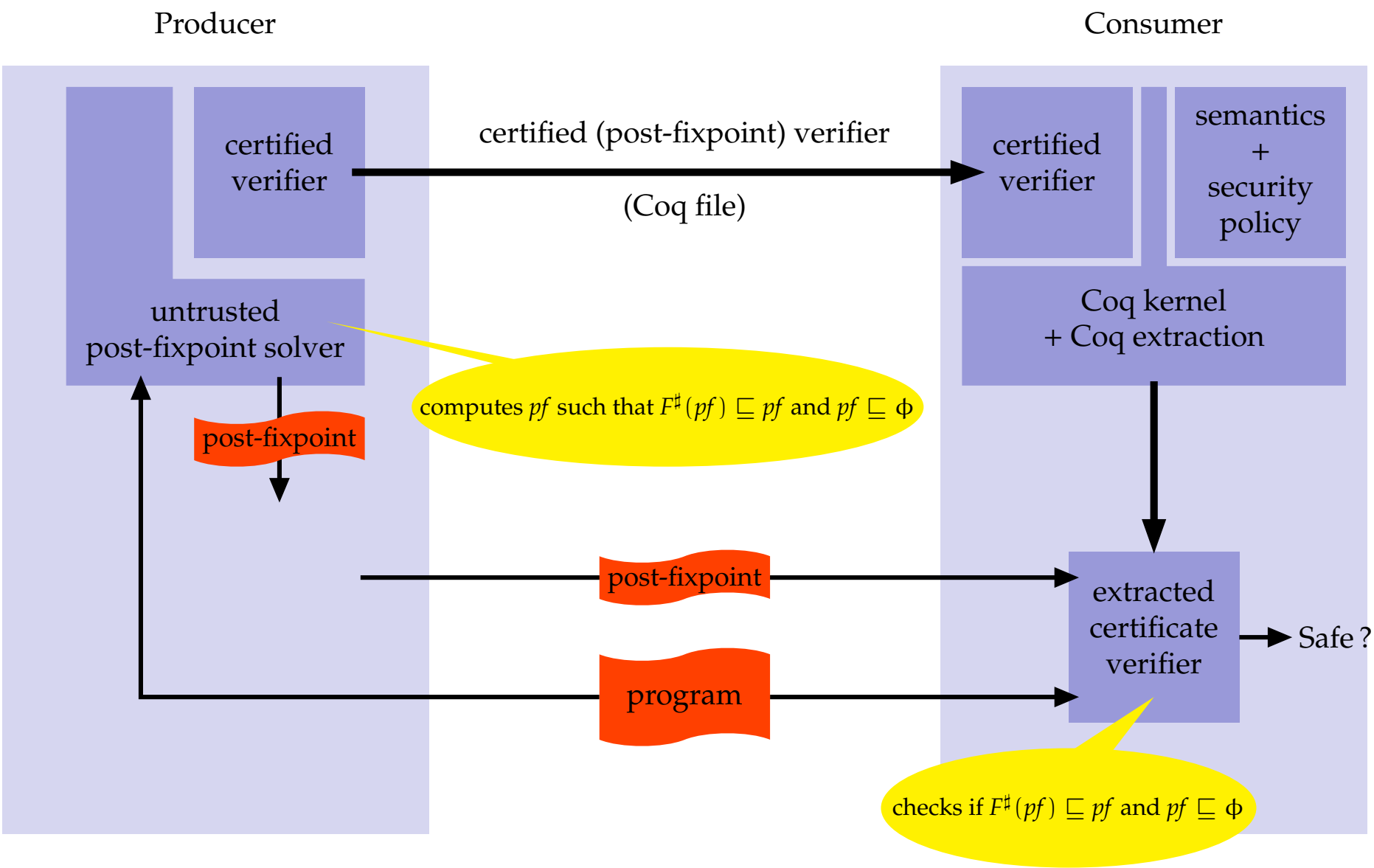
Certified PCC by abstract interpretation



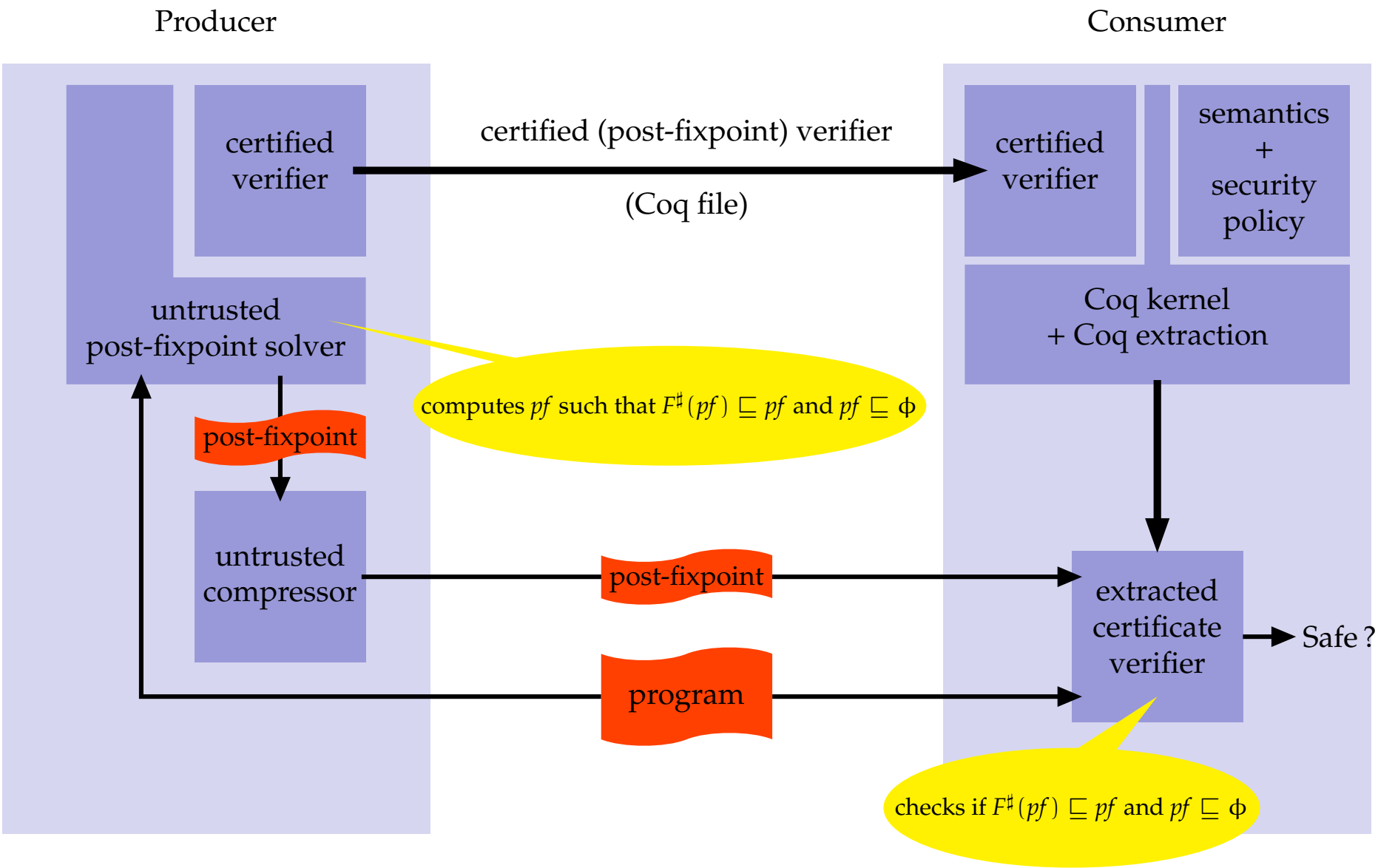
Certified PCC by abstract interpretation



Certified PCC by abstract interpretation



Certified PCC by abstract interpretation



Outline

Polyhedral abstract interpretation

Automatic discovery of linear restraints among variables of a program.
P. Cousot and N. Halbwachs. POPL'78.

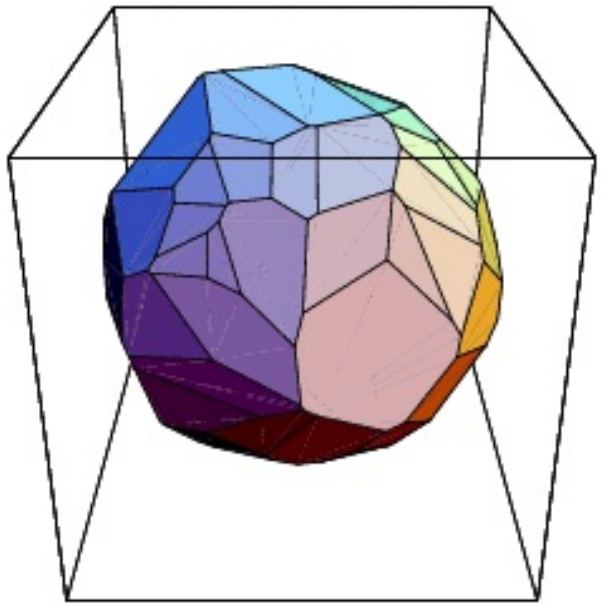
Patrick Cousot

Nicolas Halbwachs

Polyhedral analysis seeks to discover invariant linear equality and inequality relationships among the variables of an imperative program.

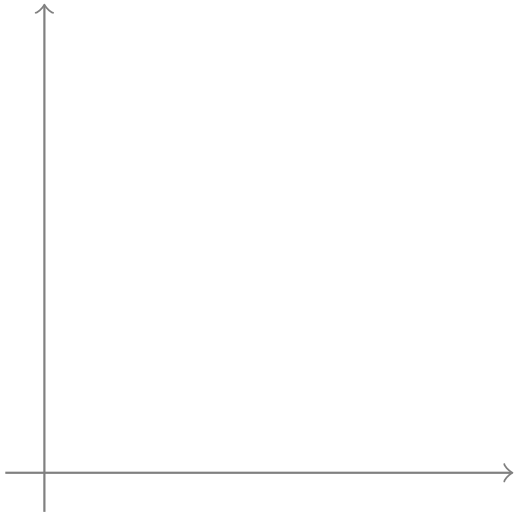
Convex polyhedra

A convex polyhedron can be defined algebraically as the set of solutions to a system of linear inequalities.
Geometrically, it can be defined as a finite intersection of half-spaces.



Polyhedral analysis

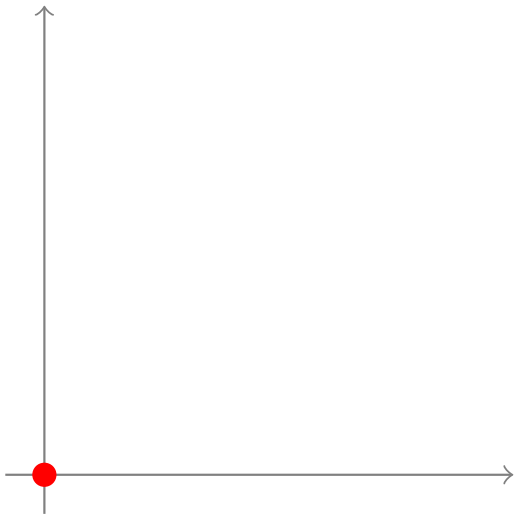
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;


while (x<6) {
  if (?) {

    y = y+2;

  };


  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



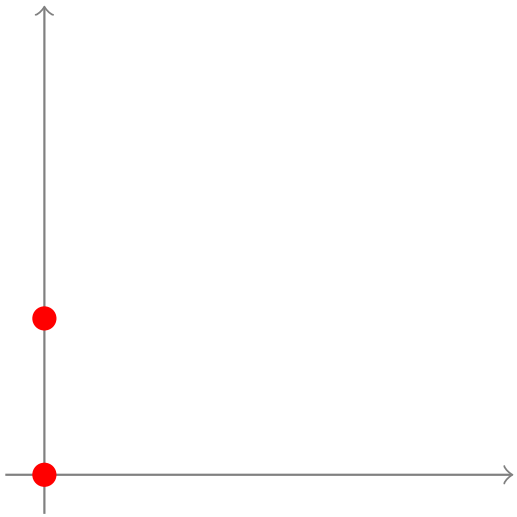
```
x = 0; y = 0;
```
$\{x = 0 \wedge y = 0\}$

```
while (x<6) {
  if (?) {
```
$\{x = 0 \wedge y = 0\}$

```
    y = y+2;

  };

  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



At junction point, we over approximate union by a convex union.

```
x = 0; y = 0;
```
$\{x = 0 \wedge y = 0\}$

```
while (x<6) {
  if (?) {
```
$\{x = 0 \wedge y = 0\}$

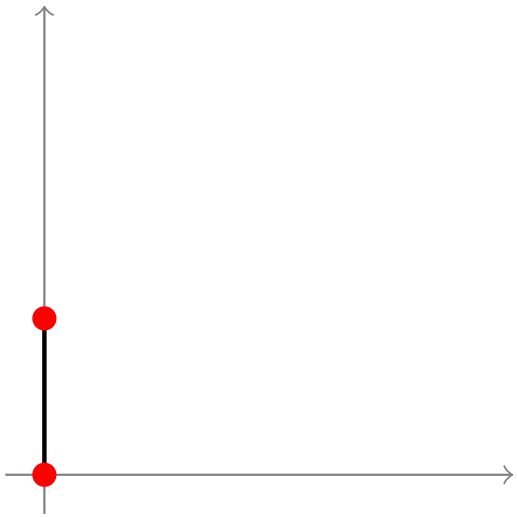
```
    y = y+2;
```
$\{x = 0 \wedge y = 2\}$

```
  };
```
$\{x = 0 \wedge y = 0\} \uplus \{x = 0 \wedge y = 2\}$

```
  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

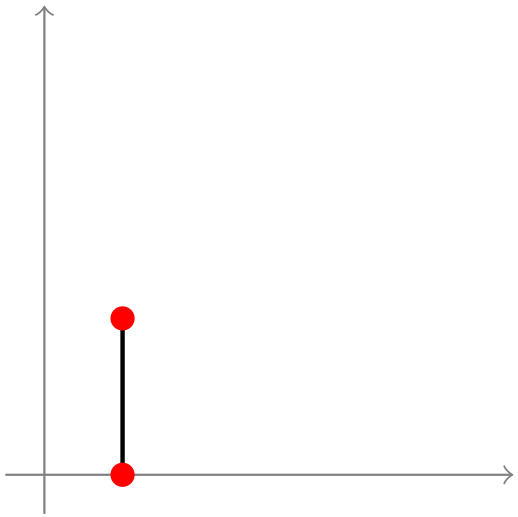


At junction point, we over approximate union by a convex union.

```
x = 0; y = 0;
        {x = 0 ∧ y = 0}

while (x<6) {
  if (?) {
        {x = 0 ∧ y = 0}
    y = y+2;
        {x = 0 ∧ y = 2}
  };
        {x = 0 ∧ 0 ⩽ y ⩽ 2}

  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
```
$\{x = 0 \wedge y = 0\}$

```
while (x<6) {
  if (?) {
```
$\{x = 0 \wedge y = 0\}$

```
    y = y+2;
```
$\{x = 0 \wedge y = 2\}$

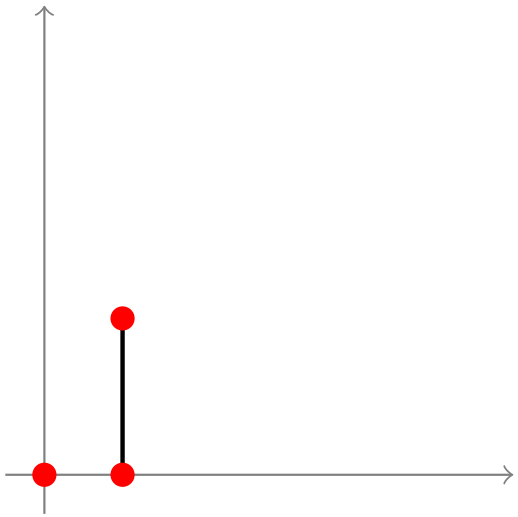
```
  };
```
$\{x = 0 \wedge 0 \leqslant y \leqslant 2\}$

```
  x = x+1;
```
$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
```
$\{x = 0 \wedge y = 0\} \uplus \{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
while (x<6) {
  if (?) {
```
$\{x = 0 \wedge y = 0\}$

```
    y = y+2;
```
$\{x = 0 \wedge y = 2\}$

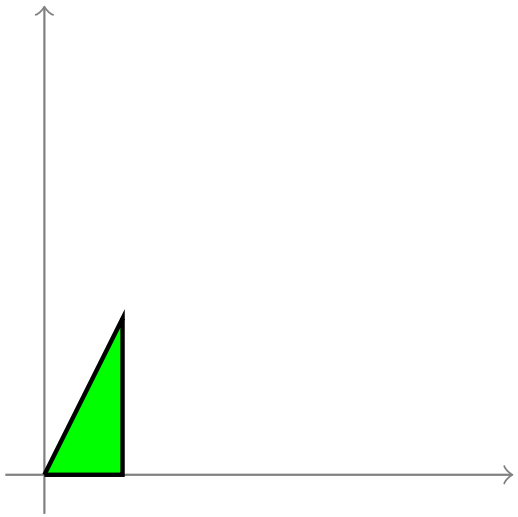
```
  };
```
$\{x = 0 \wedge 0 \leqslant y \leqslant 2\}$

```
  x = x+1;
```
$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```
$\{x = 0 \wedge y = 0\}$

```
    y = y+2;
```
$\{x = 0 \wedge y = 2\}$

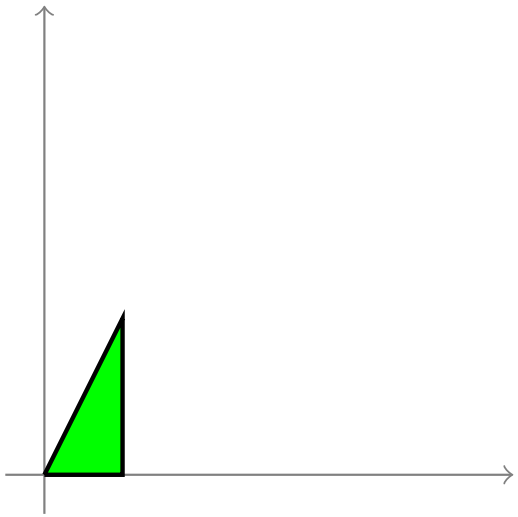
```
  };
```
$\{x = 0 \wedge 0 \leqslant y \leqslant 2\}$

```
  x = x+1;
```
$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

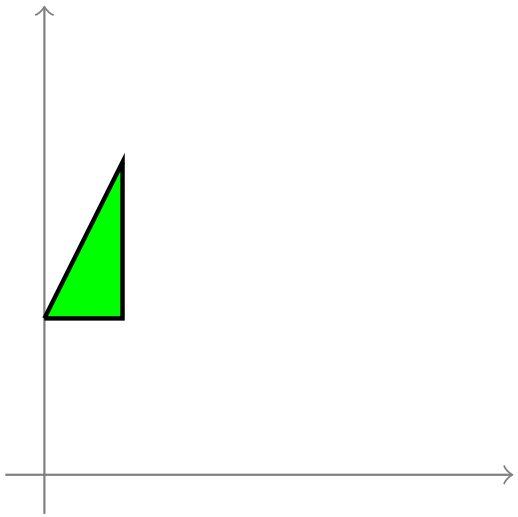
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x = 0 ∧ y = 2}
  };
        {x = 0 ∧ 0 ⩽ y ⩽ 2}

  x = x+1;
        {x = 1 ∧ 0 ⩽ y ⩽ 2}
}
```

Polyhedral analysis

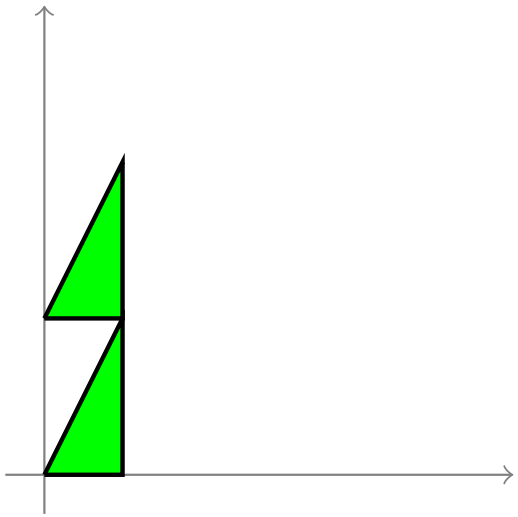
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
        {x = 0 ∧ 0 ⩽ y ⩽ 2}

  x = x+1;
        {x = 1 ∧ 0 ⩽ y ⩽ 2}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
    y = y+2;
```
$\{x \leqslant 1 \wedge 2 \leqslant y \leqslant 2x+2\}$

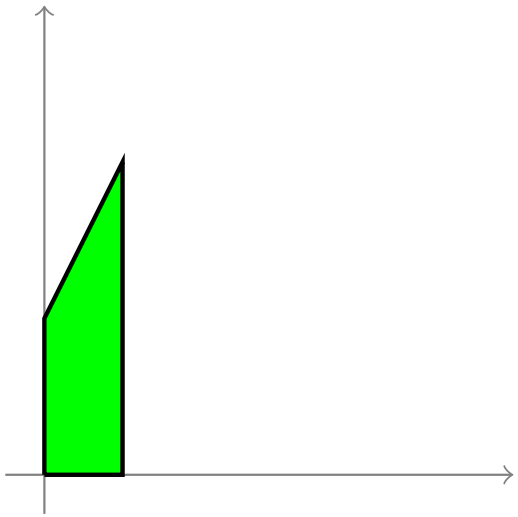
```
  };
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\} \uplus \{x \leqslant 1 \wedge 2 \leqslant y \leqslant 2x+2\}$

```
  x = x+1;
```
$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

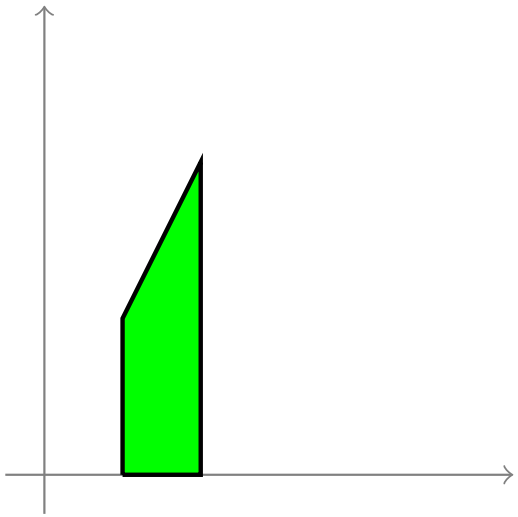
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
        {0 ⩽ x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x + 2}

  x = x+1;
        {x = 1 ∧ 0 ⩽ y ⩽ 2}
}
```

Polyhedral analysis

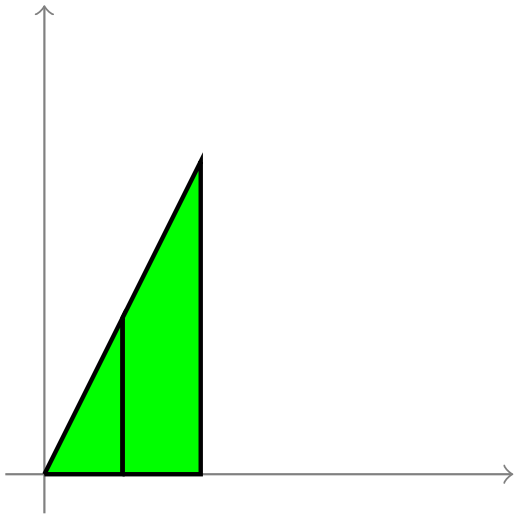
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



```
x = 0; y = 0;
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
        {0 ⩽ x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x + 2}

  x = x+1;
        {1 ⩽ x ⩽ 2 ∧ 0 ⩽ y ⩽ 2x}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

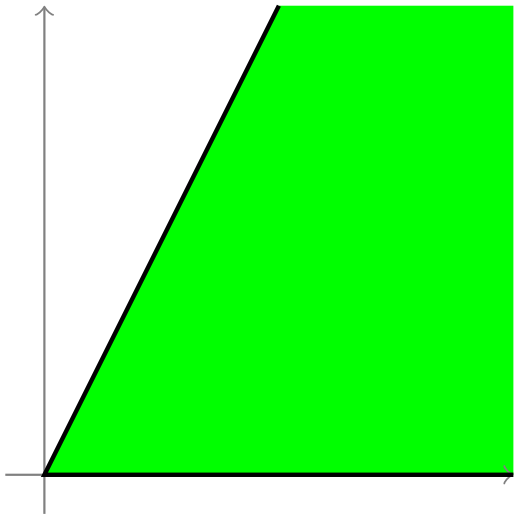


At loop headers, we use heuristics (widening) to ensure finite convergence.

```
x = 0; y = 0;
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
                    ∇ {x ⩽ 2 ∧ 0 ⩽ y ⩽ 2x}
while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
        {0 ⩽ x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x + 2}

  x = x+1;
        {1 ⩽ x ⩽ 2 ∧ 0 ⩽ y ⩽ 2x}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



At loop headers, we use heuristics (widening) to ensure finite convergence.

```
x = 0; y = 0;
        {0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
        {0 ⩽ x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x + 2}

  x = x+1;
        {1 ⩽ x ⩽ 2 ∧ 0 ⩽ y ⩽ 2x}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

By propagation we obtain a post-fixpoint

```
x = 0; y = 0;
```
$\{0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```
$\{0 \leqslant y \leqslant 2x \wedge x \leqslant 5\}$

```
    y = y+2;
```
$\{2 \leqslant y \leqslant 2x+2 \wedge x \leqslant 5\}$

```
  };
```
$\{0 \leqslant y \leqslant 2x+2 \wedge 0 \leqslant x \leqslant 5\}$

```
  x = x+1;
```
$\{0 \leqslant y \leqslant 2x \wedge 1 \leqslant x \leqslant 6\}$

```
}
```
$\{0 \leqslant y \leqslant 2x \wedge 6 \leqslant x\}$

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

By propagation we obtain a post-fixpoint which is enhanced by downward iteration.

```
x = 0; y = 0;
```
$\{0 \leqslant y \leqslant 2x \ \wedge x \leqslant 6\}$

```
while (x<6) {
  if (?) {
```
$\{0 \leqslant y \leqslant 2x \ \wedge \ x \leqslant 5\}$

```
    y = y+2;
```
$\{2 \leqslant y \leqslant 2x + 2 \ \wedge \ x \leqslant 5\}$

```
  };
```
$\{0 \leqslant y \leqslant 2x + 2 \ \wedge \ 0 \leqslant x \leqslant 5\}$

```
  x = x+1;
```
$\{0 \leqslant y \leqslant 2x \ \wedge \ 1 \leqslant x \leqslant 6\}$

```
}
```
$\{0 \leqslant y \leqslant 2x \ \wedge \ 6 = x\}$

Polyhedral analysis

A more complex example.

```
x = 0; y = A;
```
$\{A \leqslant y \leqslant 2x + A \wedge x \leqslant N\}$

```
while (x<N) {
  if (?) {
```
$\{A \leqslant y \leqslant 2x + A \ \wedge \ x \leqslant N - 1\}$

```
    y = y+2;
```
$\{A + 2 \leqslant y \leqslant 2x + A + 2 \ \wedge \ x \leqslant N - 1\}$

```
  };
```
$\{A \leqslant y \leqslant 2x + A + 2 \ \wedge \ 0 \leqslant x \leqslant N - 1\}$

```
  x = x+1;
```
$\{A \leqslant y \leqslant 2x + A \ \wedge \ 1 \leqslant x \leqslant N\}$

```
}
```
$\{A \leqslant y \leqslant 2x + A \ \wedge \ N = x\}$

The analysis accepts to replace some constants by parameters.

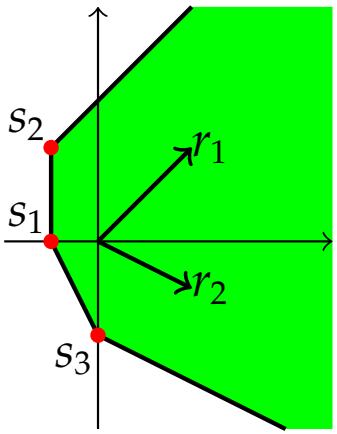
The four polyhedra operations

- $\uplus \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: convex union
  - over-approximates the concrete union in junction points
- $\cap \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: intersection
  - over-approximates the concrete intersection after a conditional intruction
- $[\![\mathtt{x} := e]\!] \in \mathbb{P}_n \to \mathbb{P}_n$: affine transformation
  - over-approximates the affectation of a variable by a linear expression
- $\nabla \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: widening
  - ensures (and accelerate) convergence of (post-)fixpoint iteration
  - includes heuristics to infer loop invariants

```
x = 0; y = 0;
```
$P_0 = [\![\mathtt{y} := 0]\!][\![\mathtt{x} := 0]\!](\mathbb{Q}^2) \nabla P_4$
```
while (x<6) {
  if (?) {
```
$P_1 = P_0 \cap \{x < 6\}$
```
    y = y+2;
```
$P_2 = [\![\mathtt{y} := \mathtt{y} + 2]\!](P_1)$
```
  };
```
$P_3 = P_1 \uplus P_2$
```
  x = x+1;
```
$P_4 = [\![\mathtt{x} := \mathtt{x} + 1]\!](P_3)$
```
}
```
$P_5 = P_0 \cap \{x \geqslant 6\}$

Library for manipulating polyhedra

- Parma Polyhedra Library[8] (PPL), NewPolka : complex C/C++ libraries
- They rely on the Double Description Method
  - polyhedra are managed using two representations in parallel



  - by set of inequalities

$$P = \left\{ (x,y) \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} x \geqslant -1 \\ x - y \geqslant -3 \\ 2x + y \geqslant -2 \\ x + 2y \geqslant -4 \end{array} \right\}$$

  - by set of generators

$$P = \left\{ \lambda_1 s_1 + \lambda_2 s_2 + \lambda_3 s_3 + \mu_1 r_1 + \mu_2 r_2 \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} \lambda_1, \lambda_2, \lambda_3, \mu_1, \mu_2 \in \mathbb{R}^2 \\ \lambda_1 + \lambda_2 + \lambda_3 = 1 \end{array} \right\}$$

  - operations efficiency strongly depends on the chosen representations, so they keep both
- We really don't want this in a Trusted Computes Base !
- But we really don't want to certify this C/C++ libraries neither !

[8] Previous tutorial on polyhedra partially comes from `http://www.cs.unipr.it/ppl/`

Polyhedra in a PCC framework

Join work with F. Besson, T. Jensen and T. Turpin

Develop a checker of analysis results

- minimize the number of operations to certify
- avoid (some of the most) costly operations

The checker will receive a post-fixpoint + a *certificate* of certain polyhedra inclusions to be verified by the checker

We develop one checker for a rich abstract domain based on Farkas lemma

Can accommodate invariants that are obtained

- automatically (intervals, polyhedra,. . .)
- by user-annotation (polynomials, . . .)

A minimal polyhedral tool-kit

For efficiency and simplicity,

- Polyhedra are represented in constraint form prefixed by existentially quantified variables
- Constraints are never normalised

Abstract operators are much simpler :

- Assignments do not trigger quantifier elimination ;

$$[\![x := e]\!](P) = \exists x',\ P[x'/x] \ \wedge\ x = e[x'/x]$$

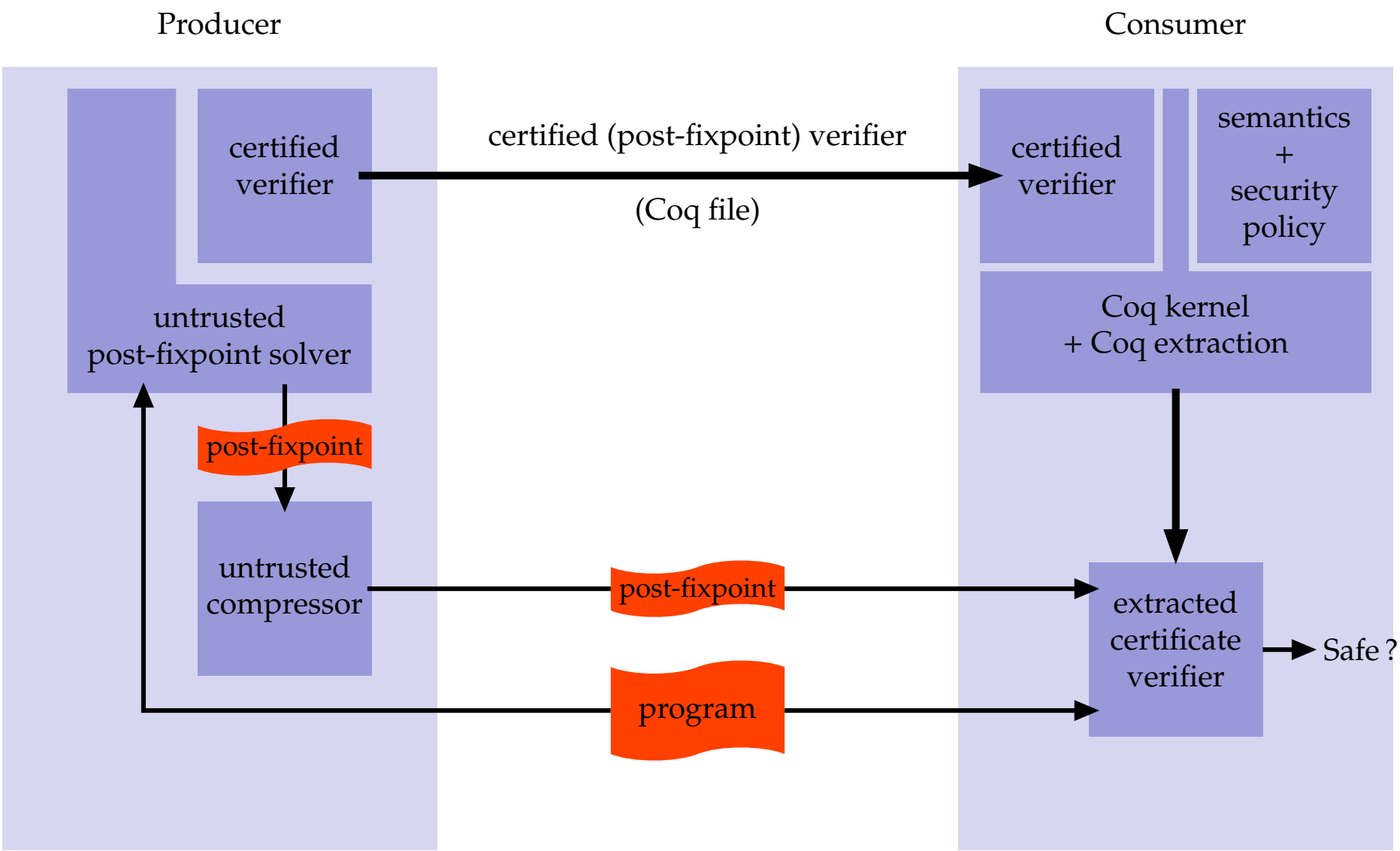
- Intersection is just syntactic union of constraints ;
- (Over-approximations) of Convex Hulls are given as untrusted invariants ;

$$isUpperBound(P, Q, UB) \equiv P \sqsubseteq UB \wedge Q \sqsubseteq UB$$

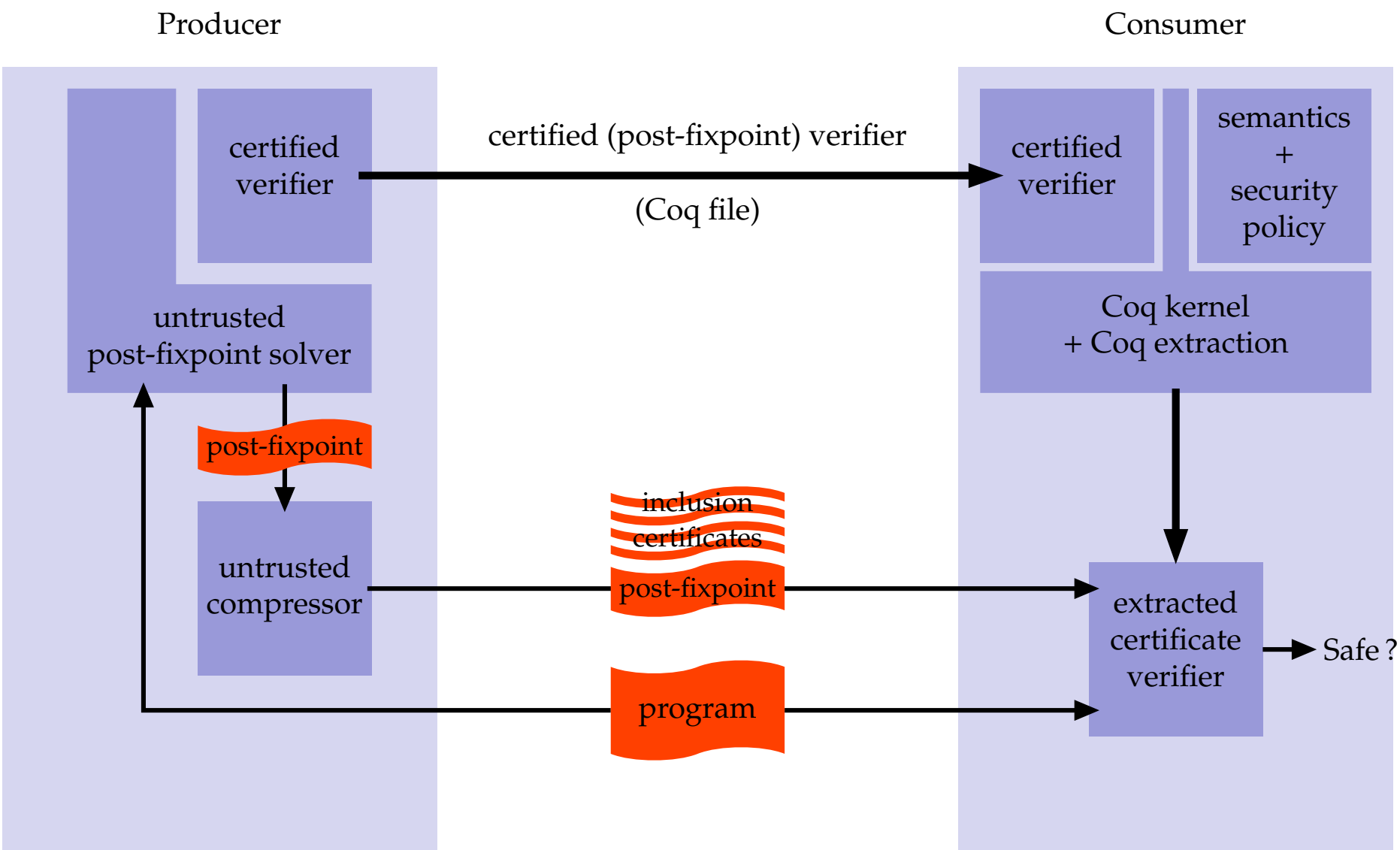
- Polyhedra inclusion is guided by a certificate ;

$$isIncluded(P, Q, Cert) \Rightarrow P \sqsubseteq Q$$

Certified PCC by abstract interpretation



Certified PCC by abstract interpretation



Checking polyhedra inclusion using certificates

- Inclusion reduces to a conjunction of emptiness problems

$$P \sqsubseteq \{q_1 \geqslant c_1, \ldots q_m \geqslant c_m\}$$

if and only if

$$P \cup \{-q_1 \geqslant -c_1 + 1\} = \emptyset \wedge \ldots \wedge P \cup \{-q_m \geqslant -c_m + 1\} = \emptyset$$

- Each emptiness reduces to unsatisfiability of linear constraints

$$\forall x_1, \ldots, x_n, \neg \begin{pmatrix} a_{1,1}, \ldots, a_{1,n} \\ \vdots \\ a_{m,1}, \ldots, a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \geqslant \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Unsatisfiability certificates

Lemma (Farkas's Lemma (Variant))

Let $A \in \mathbb{Q}^{m \times n}$ *and* $b \in \mathbb{Q}^n$.

$$\forall x \in \mathbb{Q}^n, \neg(A \cdot x \geqslant b)$$

if and only if

$$\exists(cert \in \mathbb{Q}^m), cert \geqslant \bar{0}, \text{ such that } \begin{cases} A^t \cdot cert = \bar{0} \\ b^t \cdot cert > 0 \end{cases}$$

Soundness of certificates is easy ($\Leftarrow$)

Démonstration.

Suppose	$A \cdot x \geqslant b$.
Since $cert \geqslant \bar{0}$ we have	$(A \cdot x)^t \cdot cert \geqslant b^t \cdot cert$.
Now	$x^t \cdot (A^t \cdot cert) = (x^t \cdot A^t) \cdot cert = (A \cdot x)^t \cdot cert$.
Hence	$x^t \cdot (A^t \cdot cert) \geqslant b^t \cdot cert$.
Therefore	$x^t.\bar{0} = 0 \geqslant b^t \cdot cert > 0 \rightarrow$ contradiction.

□

Certificate checking

Example

Using the certificate $cert = (1;1;5)$, check that $\begin{pmatrix} 1 & 1 \\ -1 & 4 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \geqslant \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$ has no solutions.

Checking algorithm.

- Check $\begin{pmatrix} 1 & 1 \\ -1 & 4 \\ 0 & 1 \end{pmatrix}^t \cdot \begin{pmatrix} 1 \\ 1 \\ 5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$
- Check $\begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}^t \cdot \begin{pmatrix} 1 \\ 1 \\ 5 \end{pmatrix} > 0.$

□

Checking time complexity is quadratic (matrix-vector product).

Certificate generation by linear programming

Let $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^n$, the set of unsatisfiability certificates is defined as

$$Cert = \left\{ c \;\middle|\; \begin{array}{l} c \geqslant 0 \\ b^t \cdot c > 0 \\ A^t \cdot c = 0 \end{array} \right\}$$

Finding an *extremal* certificate is a linear programming problem

$$min\{c^t \cdot \bar{1} \mid c \in Cert\}$$

that can be solved

- Over $\mathbb{N}$, by linear integer programming algorithms
  (Bad complexity, smallest certificate)
- Over $\mathbb{Q}$, by the Simplex (or interior point methods)
  (Good complexity and small certificate – in practise)

Application : a polyhedral bytecode analyser

We have applied this technique for a Java-like bytecode language with

- (unbounded) integers,
- dynamically created (unidimensional) array of integers,
- static methods (procedures),
- static fields (global variables).

Linear invariant are used to statically checks that all array accesses are within bounds.

It allows to remove the dynamic check used by standard JVM without risk of buffer overflow attack.

In practice we could only try to detect statically some valid array accesses and keep dynamic checks for the other accesses.

Example : binary search

```
static int bsearch(int key, int[] vec) {

   int low = 0, high = vec.length - 1;

   while (0 < high-low) {

      int mid = (low + high) / 2;

      if (key == vec[mid]) return mid;
      else if (key < vec[mid]) high = mid - 1;
      else low = mid + 1;
   }

   return -1;
}
```

Example : binary search

```
  //     PRE:  0 ≤ |vec_0|
static int bsearch(int key, int[] vec) {
  // (I_1) key_0 = key ∧ |vec_0| = |vec| ∧ 0 ≤ |vec_0|
   int low = 0, high = vec.length - 1;
  // (I_2) key_0 = key ∧ |vec_0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec_0|
   while (0 < high-low) {
  // (I_3) key_0 = key ∧ |vec_0| = |vec| ∧ 0 ≤ low < high < |vec_0|
       int mid = (low + high) / 2;
  // (I_4) key_0 = key∧|vec_0| = |vec|∧0 ≤ low < high < |vec_0|∧low+high−1 ≤ 2·mid ≤ low
       if (key == vec[mid]) return mid;
       else if (key < vec[mid]) high = mid - 1;
       else low = mid + 1;
    // (I_5) key_0 = key ∧ |vec_0| = |vec| ∧ −2 + 3 · low ≤ 2 · high + mid ∧ −1 + 2 · low ≤ hi
mid ∧ −1 + low ≤ mid ≤ 1 + high ∧ high ≤ low + mid ∧ 1 + high ≤ 2 · low + mid ∧ 1 + low +
|vec_0| + high ∧ 2 ≤ |vec_0| ∧ 2 + high + mid ≤ |vec_0| + low
   }
  // (I_6) key_0 = key ∧ |vec_0| = |vec| ∧ low − 1 ≤ high ≤ low ∧ 0 ≤ low ∧ high < |vec_0|
   return -1;
} //    POST:  −1 ≤ res < |vec_0|
```

This is a correct post-fixpoint but there is too many informations (too precise) !

Example : binary search

```
  //     PRE: True
static int bsearch(int key, int[] vec) {
  // (I'_1) |vec_0| = |vec| ∧ 0 ≤ |vec_0|
   int low = 0, high = vec.length - 1;
  // (I'_2) |vec_0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec_0|
   while (0 < high-low) {
  // (I'_3) |vec_0| = |vec| ∧ 0 ≤ low < high < |vec_0|
       int mid = (low + high) / 2;
  // (I'_4) |vec| − |vec_0| = 0 ∧ low ≥ 0 ∧ mid − low ≥ 0∧
  //        2 · high − 2 · mid − 1 ≥ 0 ∧ |vec_0| − high − 1 ≥ 0
       if (key == vec[mid]) return mid;
       else if (key < vec[mid]) high = mid - 1;
       else low = mid + 1;
  // (I'_5) |vec_0| = |vec| ∧ −1 + low ≤ high ∧ 0 ≤ low ∧ 5 + 2 · high ≤ 2 · |vec|
   }
  // (I'_6) 0 ≤ |vec_0|
   return -1;
} //    POST: −1 ≤ res < |vec_0|
```

This one is less precise but sufficient to ensure the security policy.

Some preliminary benchmarks

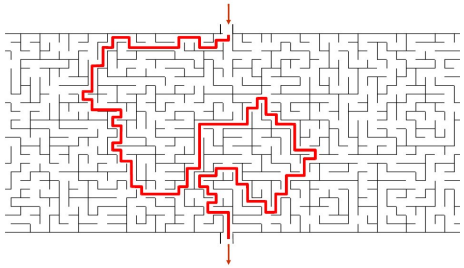
	.class	certificates		checking time	
Program		before	after	before	after
BSearch	515	22	12	0.005	0.007
BubbleSort	528	15	14	0.0005	0.0003
HeapSort	858	72	32	0.053	0.025
QuickSort	833	87	44	0.54	0.25

Class files are given in bytes, certificates in number of constraints, time in seconds.
The two checking times in the last column give the checking time with and without fixpoint pruning.

Conclusions

The two main slogans of PCC

1. program verification should follow the maze metaphor
   - less power consuming for the consumer
   - more easy to trust (or prove correct)
2. TCB must be as small, as formal and generic as possible



The next challenges for PCC

1. PCC tools must be able to enforce more expressive security properties
2. certified PCC must reach the scalability level of standard PCC