Semantic program analysis for software safety and security

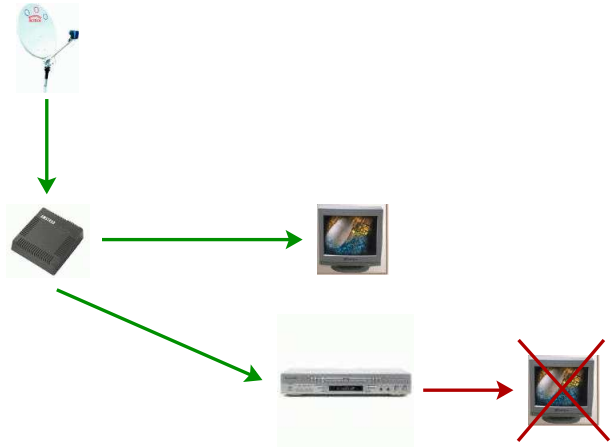
Thomas Jensen

IRISA

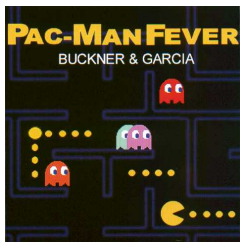
June 26, 2007

Outline of the talk

- Cryptographic protocols



- Mobile code security



Plan

Software safety and security

Cryptographic protocol verification

- proper use of cryptographic primitives
- examine all accessible to prove confidentiality

Validating access control to data and resources in Java software

- using Java stack inspection for access control,
- using the Java Card firewall,
- using the Java MIDP security architecture for embedded devices

Checking for buffer overflows and data race conditions

Semantic program analysis

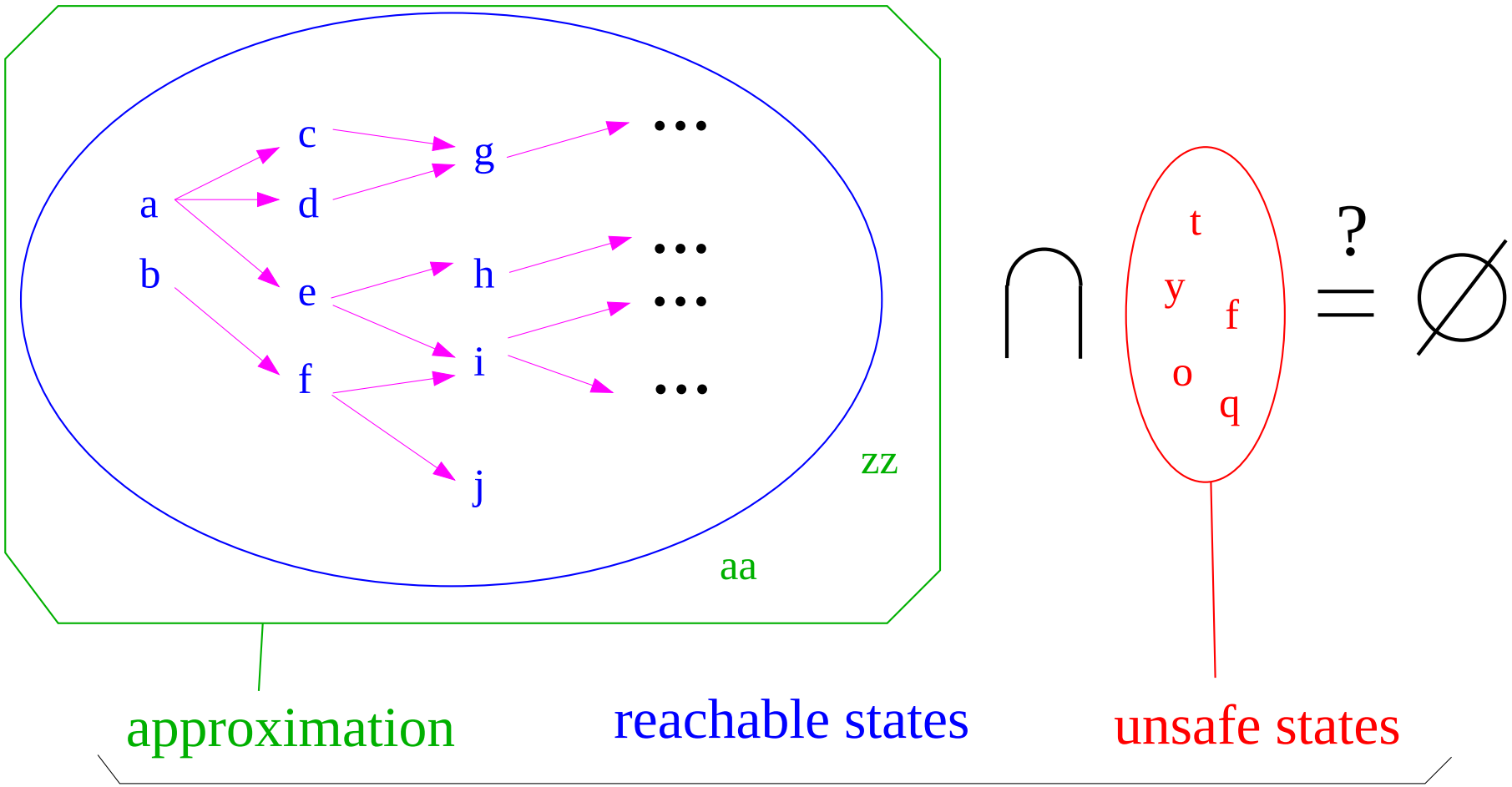
The goals of static program analysis

- To prove properties about the run-time behaviour of a program
- In a fully automatic way
- Without actually executing this program
- Accepting to give approximate answers

Solid foundations for designing an analyser

- Abstract Interpretation gives a guideline
  - to formalise analyses
  - to prove their soundness with respect to the semantics of the programming language
- Resolution of constraints on lattices by iteration and symbolic computation

Computing reachable states



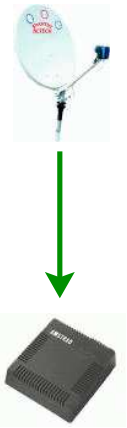
IRISA-related application domains

Smart card (Java Card) software

- data flow analysis of Java Card applets
  - array bounds check, firewall analysis, resource usage
- test case generation for a Java Card VM

Digital Rights Management

- Verification of the Thomson R&D "Smart Right" protocol for domestic networks
  - "view-once" properties for "Video-on-Demand"



Mobile telephones (Java MIDP)

- semantic modeling of interactive navigation graphs ;
- verifying the access control to resources (SMS, confidential data) in applets.

Plan

Cryptographic protocols

"Alice and Bob"-style protocol specification

The Diffie-Hellman example

1 - $A \hookrightarrow B : G^{N_a}$
2 - $B \hookrightarrow A : G^{N_b}$
3 - $A \hookrightarrow B : \{Nsecret\}_K$

$$K = (G^{N_a})^{N_b} = (G^{N_b})^{N_a}$$

"Man in the middle" attack

1 - $A \hookrightarrow I : G^{N_a}$
2 - $I \hookrightarrow B : G^{N_i}$
3 - $B \hookrightarrow I : G^{N_b}$
4 - $I \hookrightarrow A : G^{N_i}$
5 - $A \hookrightarrow I : \{secret\}_{K_1}$
6 - $I \hookrightarrow B : \{secret\}_{K_2}$

$$K_2 = (G^{N_b})^{N_i}$$
$$K_1 = (G^{N_i})^{N_a}$$

Cryptographic protocols

"Alice and Bob"-style protocol specification

The Diffie-Hellman example

1 - $A \hookrightarrow B : G^{N_a}$
2 - $B \hookrightarrow A : G^{N_b}$
3 - $A \hookrightarrow B : \{Nsecret\}_K$

$$K = (G^{N_a})^{N_b} = (G^{N_b})^{N_a}$$

"Man in the middle" attack

1 - $A \hookrightarrow I : G^{N_a}$
2 - $I \hookrightarrow B : G^{N_i}$
3 - $B \hookrightarrow I : G^{N_b}$
4 - $I \hookrightarrow A : G^{N_i}$
5 - $A \hookrightarrow I : \{secret\}_{K_1}$
6 - $I \hookrightarrow B : \{secret\}_{K_2}$

$$K_2 = (G^{N_b})^{N_i}$$
$$K_1 = (G^{N_i})^{N_a}$$

Protocol verification

Manual search for attacks

- Requires an expert
- Long and costly process

Security proofs in a computational model

- Complexity-based proofs
- Offer the best guarantees
- Very little automation

Verification with the symbolic Dolev-Yao model

- Idealised model of cryptography
- Fully automated attack detection
- Semi-automatic security proofs
- Some proofs carry over to the computational model

Symbolic specification and verification

Protocol specification

- Formal
- Done after design!

  $\Rightarrow$ Protocol completely finished and known
- In order to prove classical properties

  Essentially confidentiality and authentication

Automatic protocol verification

- Detecting attacks that violate desired properties
- Prove absence of attacks for an un-bounded number of
  - agents
  - interleaved sessions
  - simple intruder operations

Symbolic specification and verification

Protocol specification

- Formal
- Done after design!

  $\Rightarrow$ Protocol completely finished and known
- In order to prove classical properties

  Essentially confidentiality and authentication

Automatic protocol verification

- Detecting attacks that violate desired properties
- Prove absence of attacks for an un-bounded number of
  - agents
  - interleaved sessions
  - simple intruder operations

Protocol specification languages

Various formats exists

- ProVerif [Blanchet]
- Prouvé [Kremer, Laknech, Treinen]
- AVISPA (HLPSL) [Armando, et col.]

Alice	Bob
role alice (A,B: agent, ...) local State: nat, Na,Nb: text init State:= 0 transition 0. State=0 /\ RCV(start) =|> State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb) /\ secret(Na',na,{A,B}) 2. State=2 /\ RCV({Na.Nb'}_Ka) =|> State':= 4 /\ SND({Nb'}_Kb) end role	role bob(A, B: agent, ...) local State : nat, Na,Nb: text init State:= 1 transition 1. State= 1 /\ RCV({Na'.A}_Kb) =|> State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka) /\ secret(Nb',nb,{A,B}) 3. State= 3 /\ RCV({Nb'}_Kb) =|> State':= 5 /\ end role

Industrial protocol development

Protocol specification

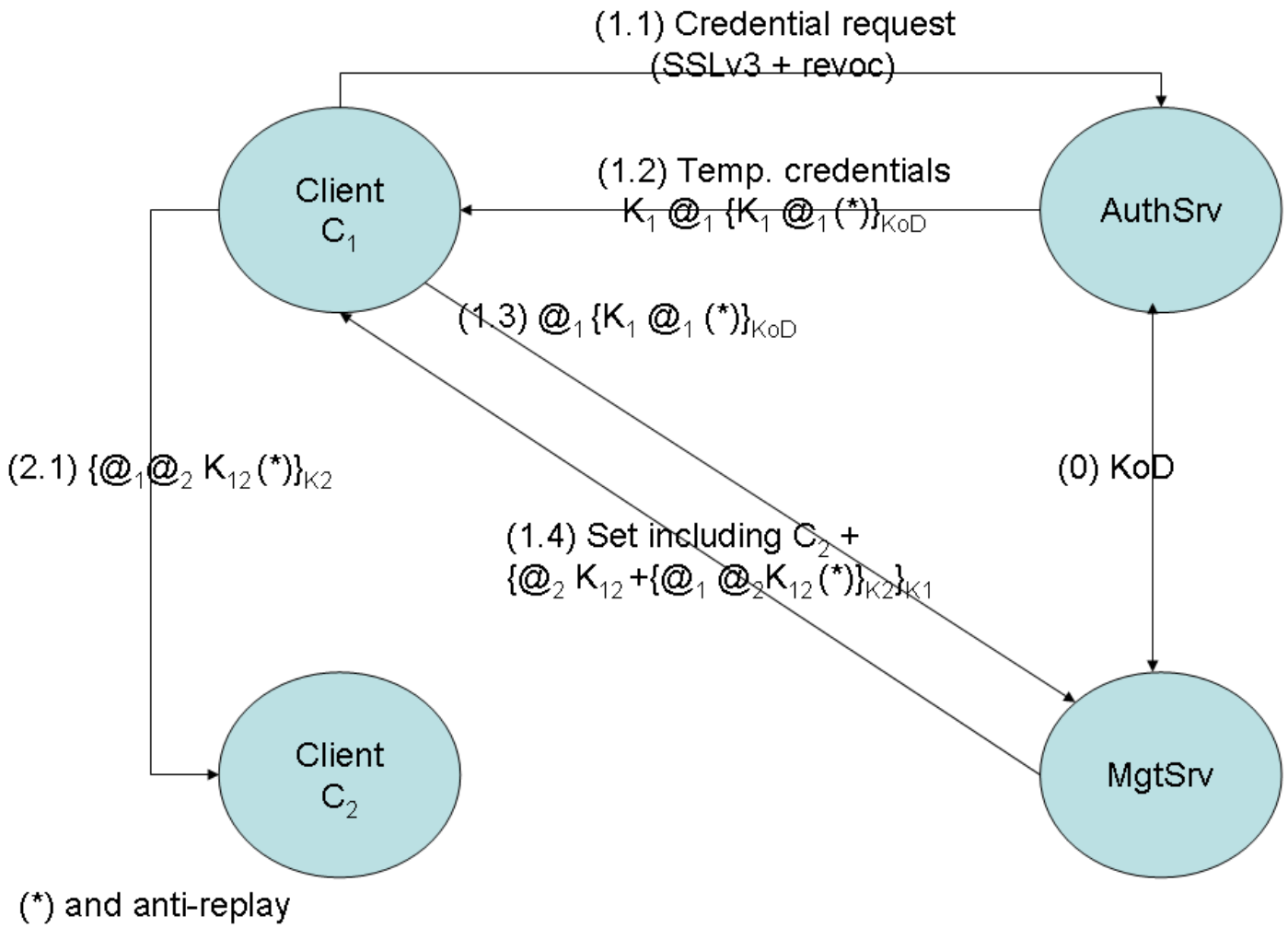
- partial: mixed up with development
- Informal: documents for design/discussion/patents
- Ad-hoc: specific deployment environment

Protocol verification

- Non-standard properties (Ex. "anonymous authentication")
- Rare because formal specification is already a lot of work!
- Formal verification *a posteriori* is not enough
- Verification during development would be ideal

Major tools for industrial protocol specification

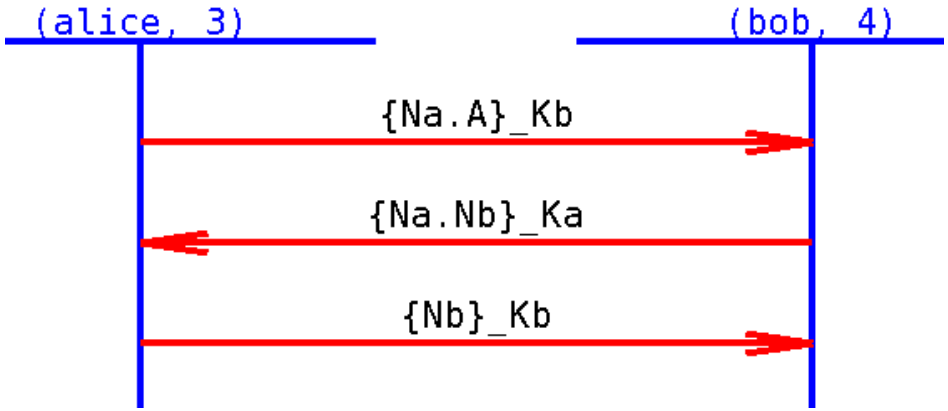
- Word for technical documents and patents
- White board and Powerpoint for design



Reconcile formalism and intuition

```
role alice (A,B: agent, ...)
local State: nat, Na,Nb: text
init State:= 0
transition
0. State=0 /\ RCV(start) =|>
   State':= 2 /\ Na' := new()
   /\ SND({Na'.A}_Kb)
   /\ secret(Na',na,{A,B})

2. State=2 /\ RCV({Na.Nb'}_Ka) =|>
   State':= 4 /\ SND({Nb'}_Kb)
end role
```



The solution: AVISPA + SPAN

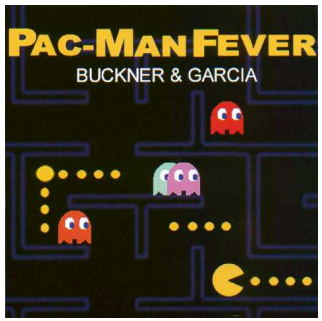
Plan

Mobile code dilemma

Mobile code dilemma

Untrusted code

Host system



Safe ?

- The untrusted code may cause damage to the system
  - internal structure corruption
  - illegal memory access
- The untrusted code may use too many resources
  - CPU, memory, SMS...

Bubble sort

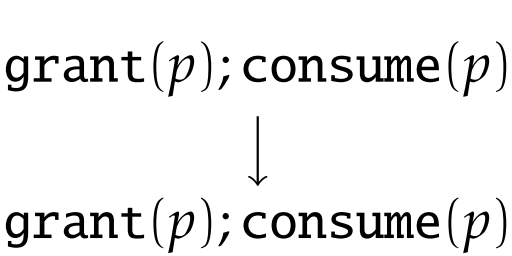
```
class BubbleSort {
public static void main(String[] argv ) {
 int i,j,tmp,n;
 0: n = 20;
 1: int[] t = new int[n];
 2: Input.init();
 3: for (i=0;i<n;i++) {
 4:   t[i] = Input.read_int();
 5: };
 6: Tab.print_tab(t);
 7: for (i=0; i<n-1; i++) {
 8:   for (j=0; j<n-1-i; j++)
 9:     if (t[j+1] < t[j]) {
10:       tmp = t[j];
11:       t[j] = t[j+1];
12:       t[j+1] = tmp;
13:     }
14: };
15: Tab.print_tab(t);
}}
```

```
class BubbleSort {
public static void main(String[] argv ) {
 int i,j,tmp,n;
    // [j: U ; n: U ; i: U ; t: U ; tmp: U    ]
 0: n = 20;
  // [j: U ; n: [20,20] ; i: U ; t:  U   ; tmp: U]
 1: int[] t = new int[n];
  // [j: U ; n: [20,20] ; i: U ; t: int[20,20] ; tmp: U]
 2: Input.init();
  // [j: U ; n: [20,20] ; i: U ; t: int[20 ,20] ; tmp: U]
 3:     for (i=0;i<n;i++) {
  // [j: U ; n: [20,20] ; i: [0,19] ; t: int[20,20] ; tmp: U]
 4: t[i] = Input.read_int();
  // [j: U ; n: [20,20] ; i: [0,19] ; t: int[20,20] ; tmp: U]
 5: };
  // [j: U ; n: [20,20] ; i: [20,20] ; t: int[20,20] ; tmp: U]
 6: Tab.print_tab(t);
  // [j: U ; n: [20,20] ; i: [0,20] ; t: int[20,20] ; tmp: U]
 7: for (i=0; i<n-1; i++) {
  // [j: U ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
 8:    for (j=0; j<n-1-i; j++)
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
 9:    if (t[j+1] < t[j]) {
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
10:    tmp = t[j];
...
}}
```

```
class BubbleSort {
public static void main(String[] argv ) {
 int i,j,tmp,n;
    // [j: U ; n: U ; i: U ; t: U ; tmp: U    ]
 0: n = 20;
  // [j: U ; n: [20,20] ; i: U ; t:  U   ; tmp: U]
 1: int[] t = new int[n];
  // [j: U ; n: [20,20] ; i: U ; t: int[20,20] ; tmp: U]
 2: Input.init();
  // [j: U ; n: [20,20] ; i: U ; t: int[20 ,20] ; tmp: U]
 3:     for (i=0;i<n;i++) {
  // [j: U ; n: [20,20] ; i: [0,19] ; t: int[20,20] ; tmp: U]
 4: t[i] = Input.read_int();
  // [j: U ; n: [20,20] ; i: [0,19] ; t: int[20,20] ; tmp: U]
 5: };
  // [j: U ; n: [20,20] ; i: [20,20] ; t: int[20,20] ; tmp: U]
 6: Tab.print_tab(t);
  // [j: U ; n: [20,20] ; i: [0,20] ; t: int[20,20] ; tmp: U]
 7: for (i=0; i<n-1; i++) {
  // [j: U ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
 8:    for (j=0; j<n-1-i; j++)
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
 9:    if (t[j+1] < t[j]) {
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
10:    tmp = t[j];
...
}}
```

```
class BubbleSort {
public static void main(String[] argv ) {
 int i,j,tmp,n;
    // [j: U ; n: U ; i: U ; t: U ; tmp: U    ]
 0: n = 20;
  // [j: U ; n: [20,20] ; i: U ; t:  U   ; tmp: U]
 1: int[] t = new int[n];
  // [j: U ; n: [20,20] ; i: U ; t: int[20,20] ; tmp: U]
 2: Input.init();
  // [j: U ; n: [20,20] ; i: U ; t: int[20 ,20] ; tmp: U]
 3:    for (i=0;i<n;i++) {
  // [j: U ; n: [20,20] ; i: [0,19] ; t: int[20,20] ; tmp: U]
 4: t[i] = Input.read_int();
  // [j: U ; n: [20,20] ; i: [0,19] ; t: int[20,20] ; tmp: U]
 5: };
  // [j: U ; n: [20,20] ; i: [20,20] ; t: int[20,20] ; tmp: U]
 6: Tab.print_tab(t);
  // [j: U ; n: [20,20] ; i: [0,20] ; t: int[20,20] ; tmp: U]
 7: for (i=0; i<n-1; i++) {
  // [j: U ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
 8:   for (j=0; j<n-1-i; j++)
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
 9:   if (t[j+1] < t[j]) {
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: U ]
10:   tmp = t[j];
...
}}
```

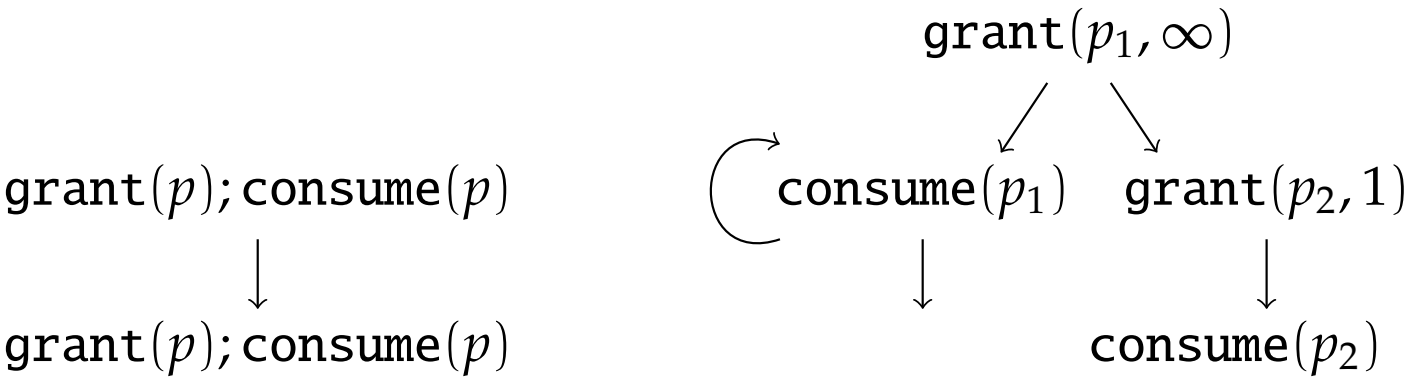
Interactive resource control on Java telephones



In the current MIDP model

- Permission request (one-shot) appear just before the actual use of the permission

Interactive resource control on Java telephones



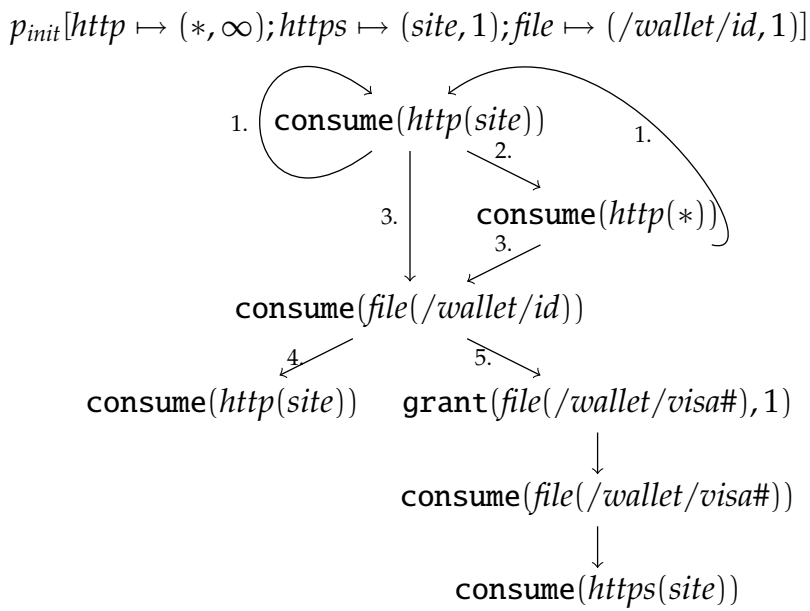
In the current MIDP model

- Permission request (one-shot) appear just before the actual use of the permission

Access control for interactive devices

Access control model for Java MIDP (mobile phones):

- permissions based on origin (operator, public,. . .)
- interactive access control to resources (SMS, http, . . .)
- security screens for granting "one-shot" permissions



Proposal: a more flexible access control model

- separating permission granting from consumption
- permissions with multiplicities ("the applet can send 5 SMSs")
- basic security policy to check:

  an applet will not attempt to use permissions it does not have

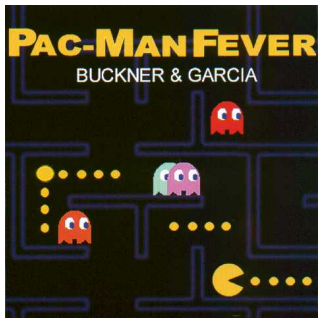
- a static permission flow analysis to check this

Plan

Mobile code dilemmas...

Untrusted code

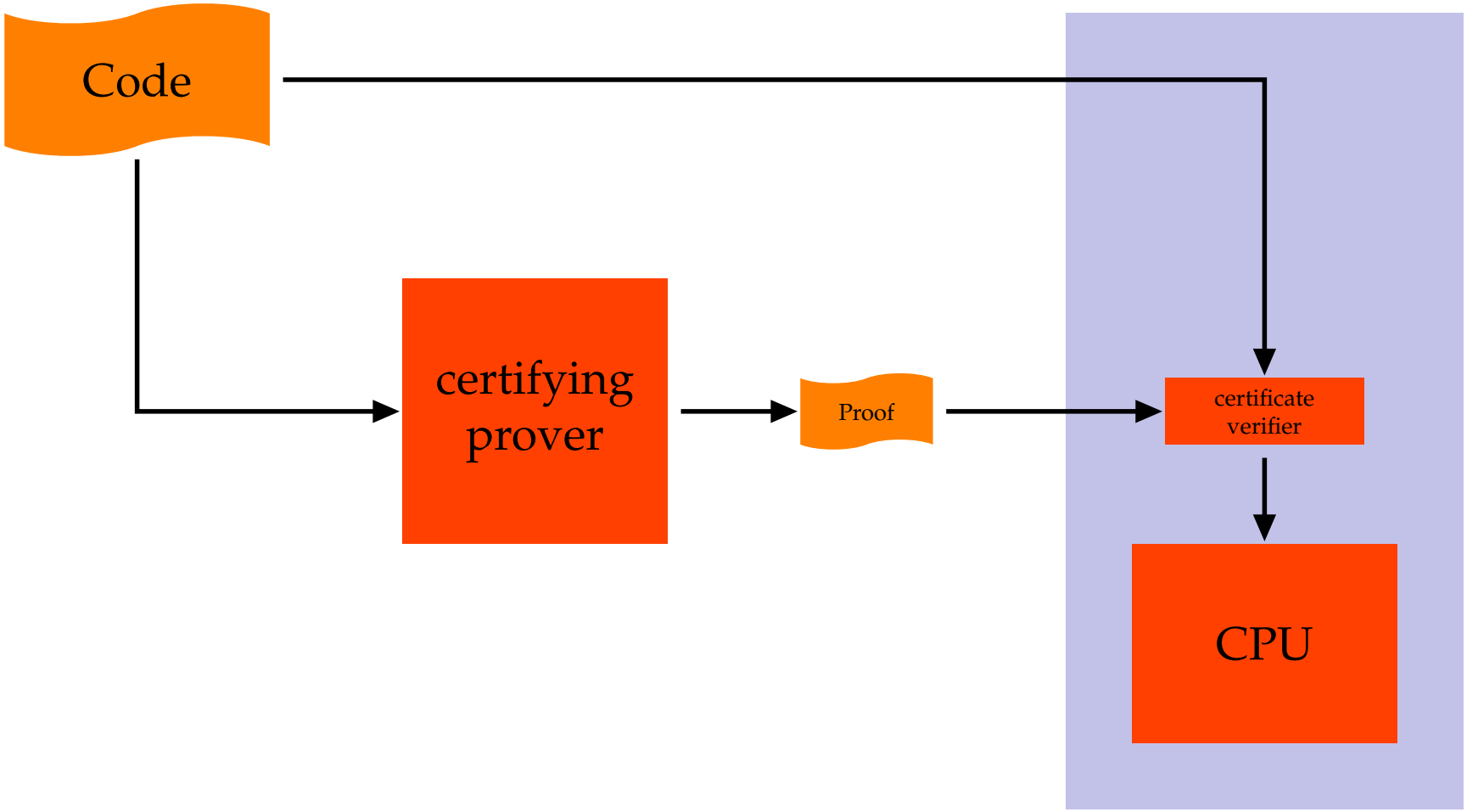
Host system



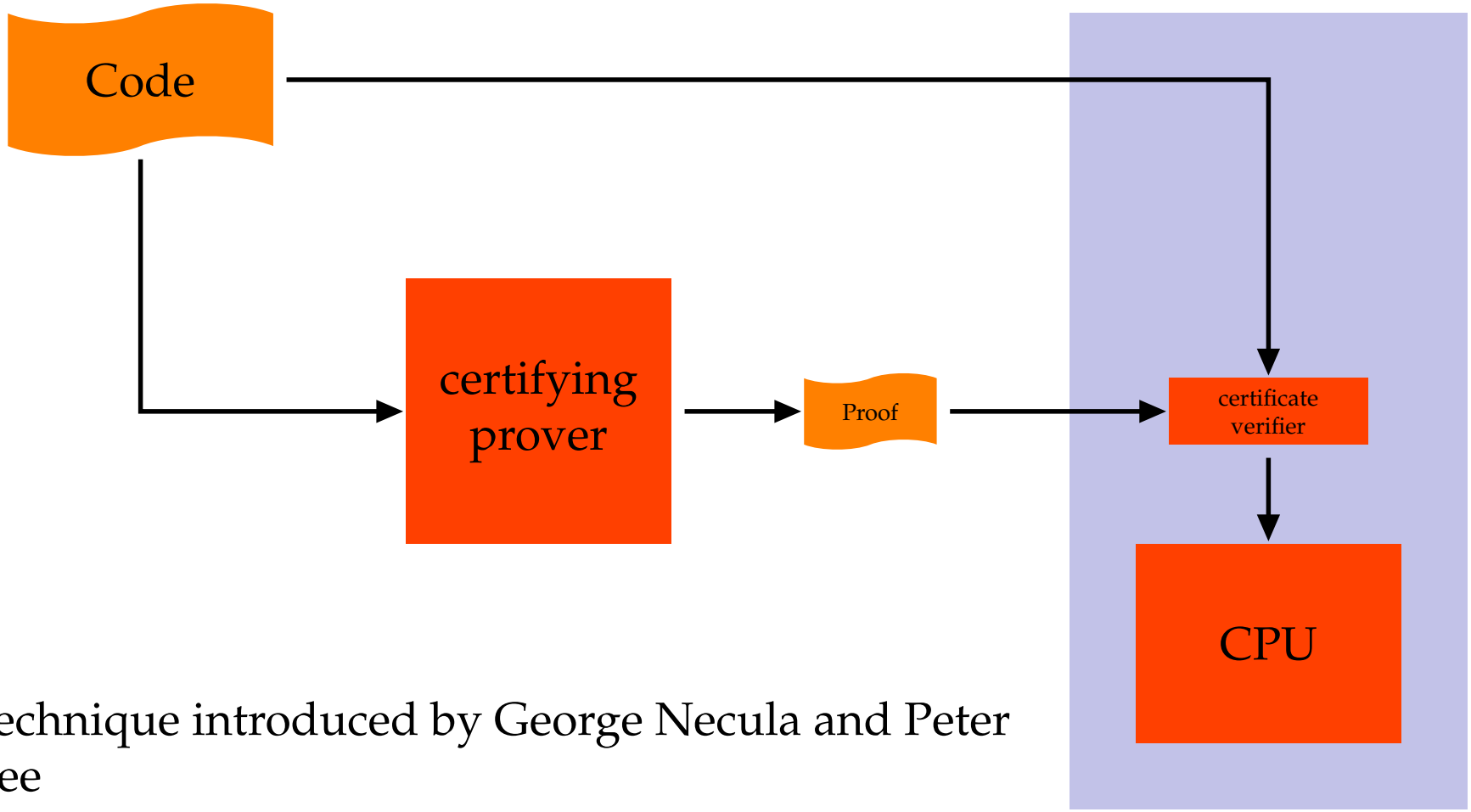
Safe ?

- The untrusted code may cause damage to the system
  - internal structure corruption
  - illegal memory access
- The untrusted code may use too many resources
  - CPU, memory, SMS...

Proof carrying code

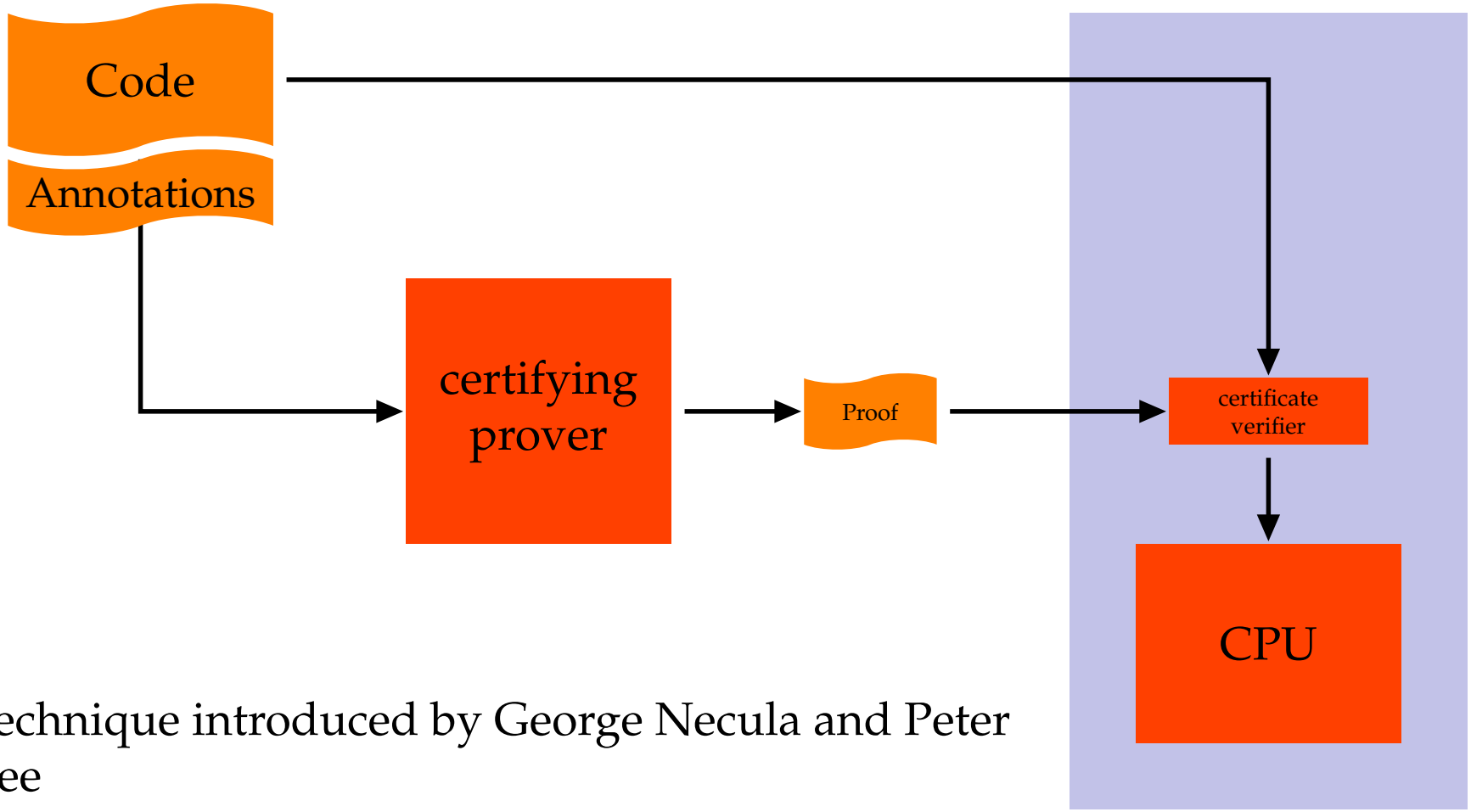


Proof carrying code: standard framework



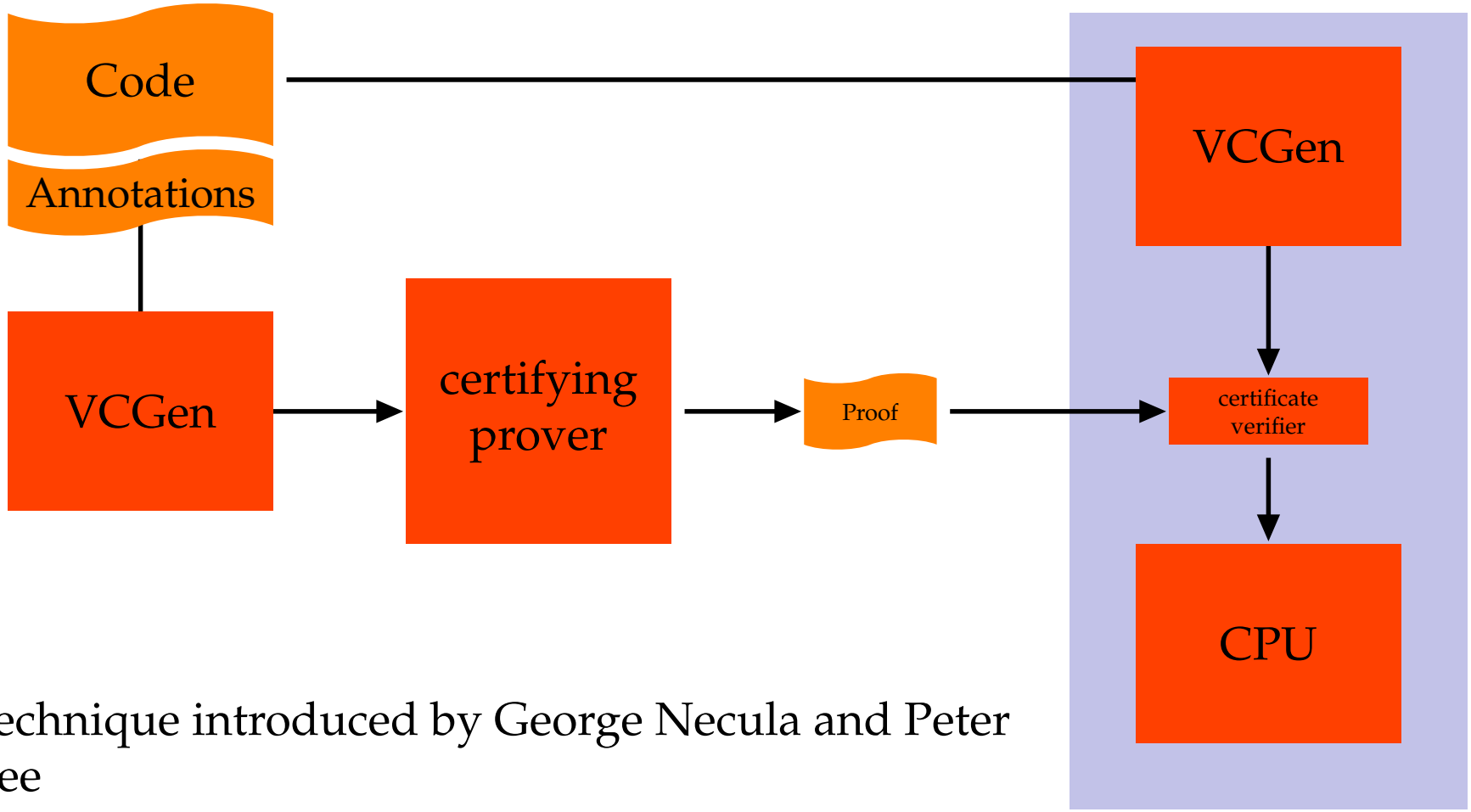
- Technique introduced by George Necula and Peter Lee
- Standard PCC architectures are based on axiomatic semantics

Proof carrying code: standard framework



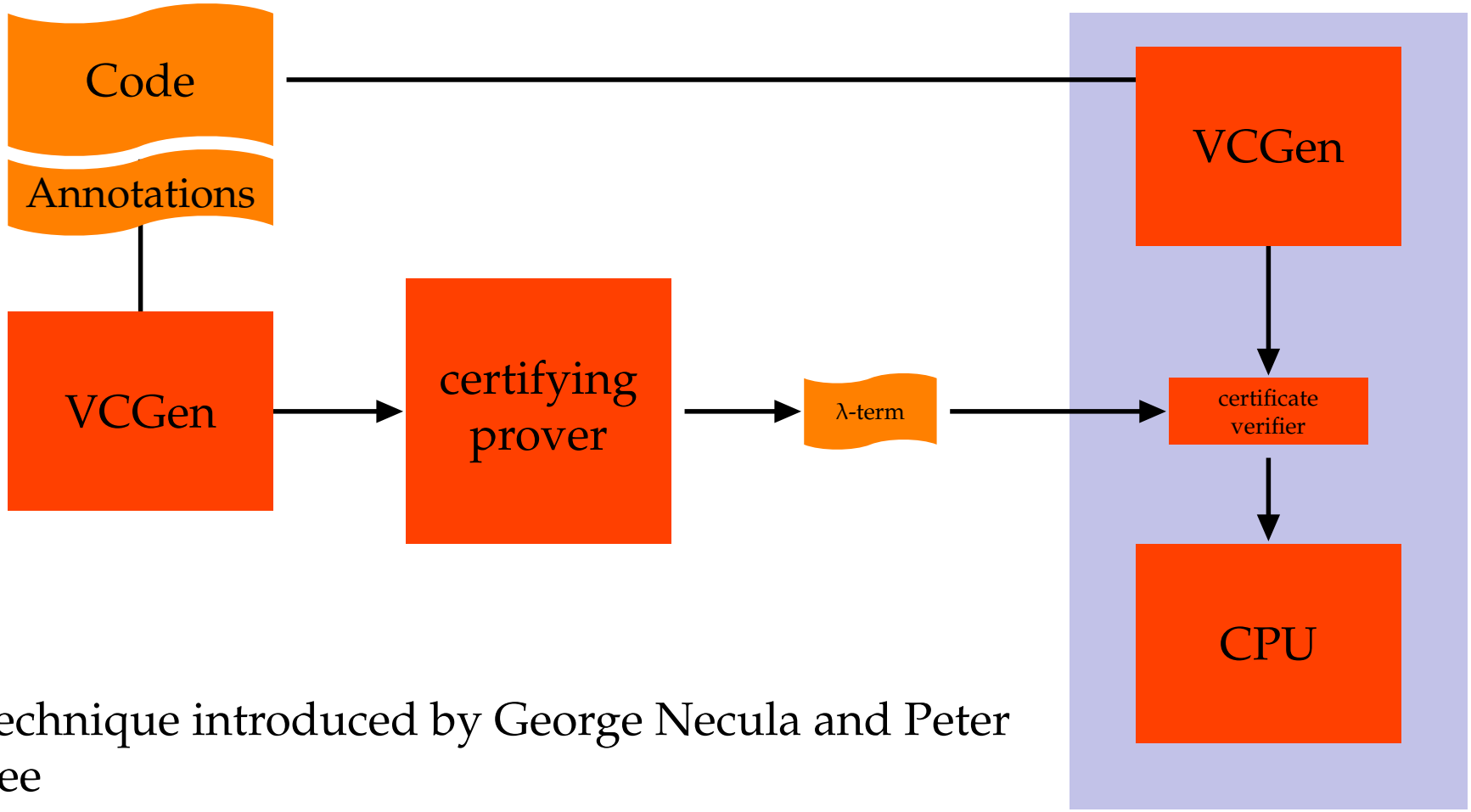
- Technique introduced by George Necula and Peter Lee
- Standard PCC architectures are based on axiomatic semantics

Proof carrying code: standard framework



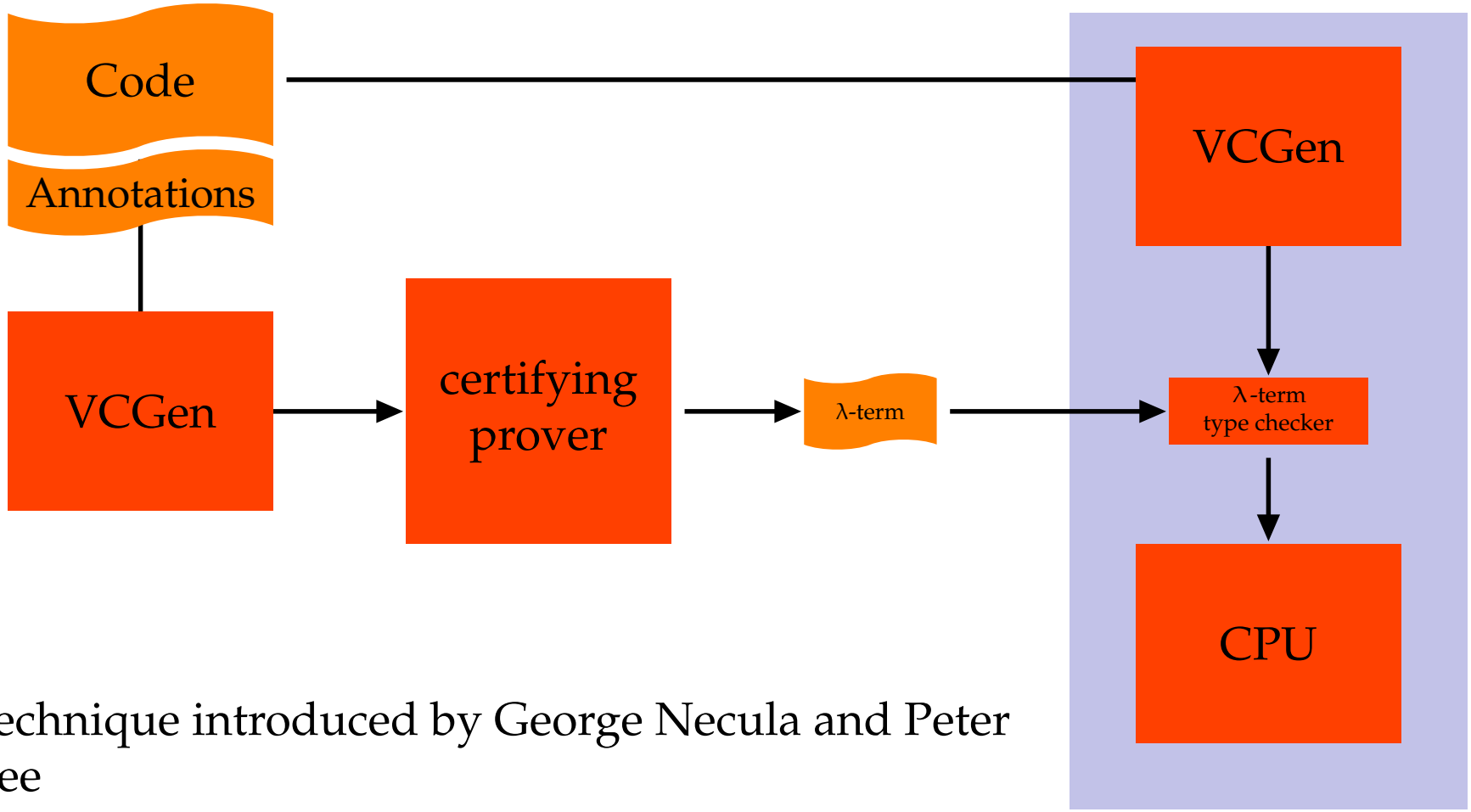
- Technique introduced by George Necula and Peter Lee
- Standard PCC architectures are based on axiomatic semantics

Proof carrying code: standard framework



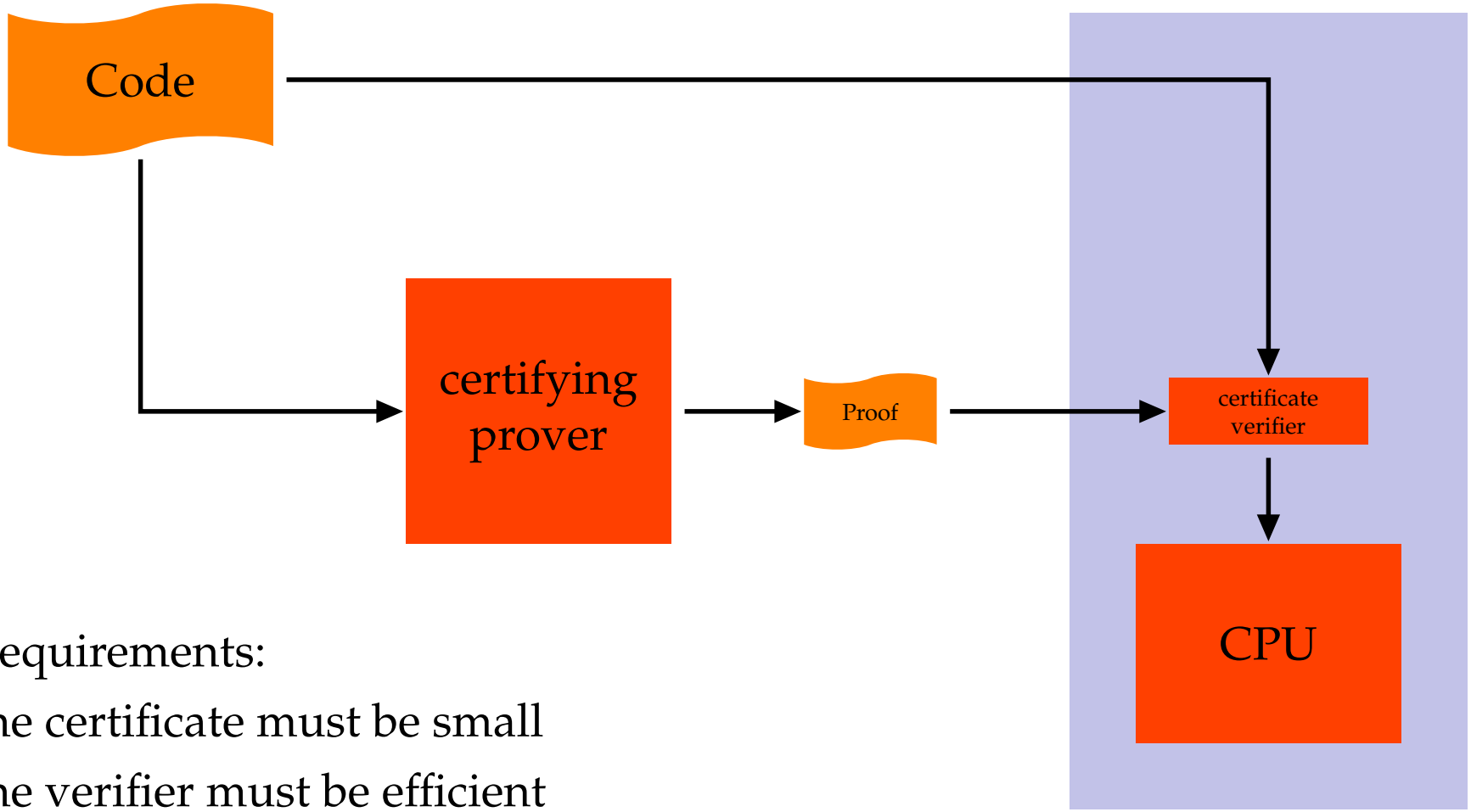
- Technique introduced by George Necula and Peter Lee
- Standard PCC architectures are based on axiomatic semantics

Proof carrying code: standard framework



- Technique introduced by George Necula and Peter Lee
- Standard PCC architectures are based on axiomatic semantics

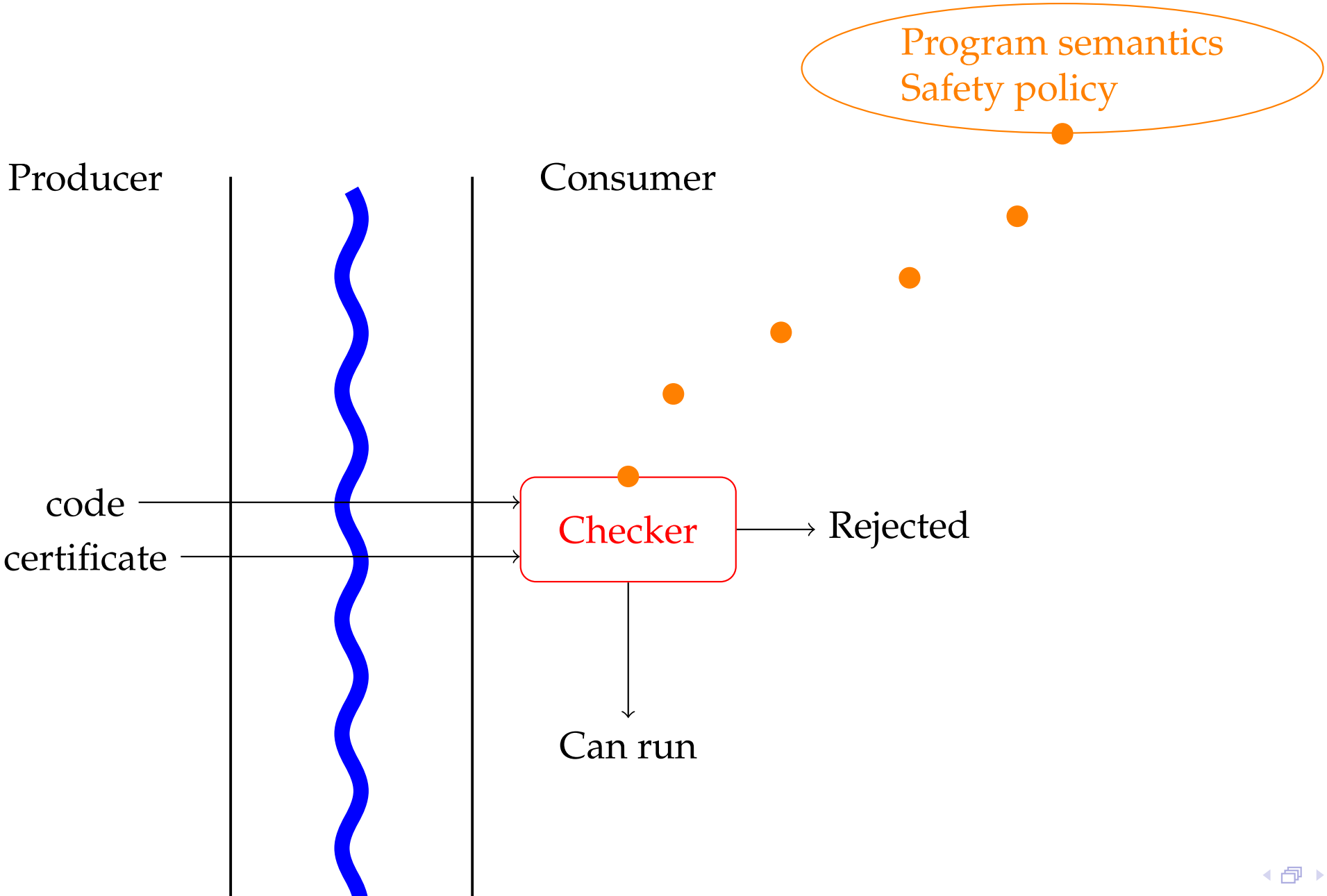
Proof carrying code



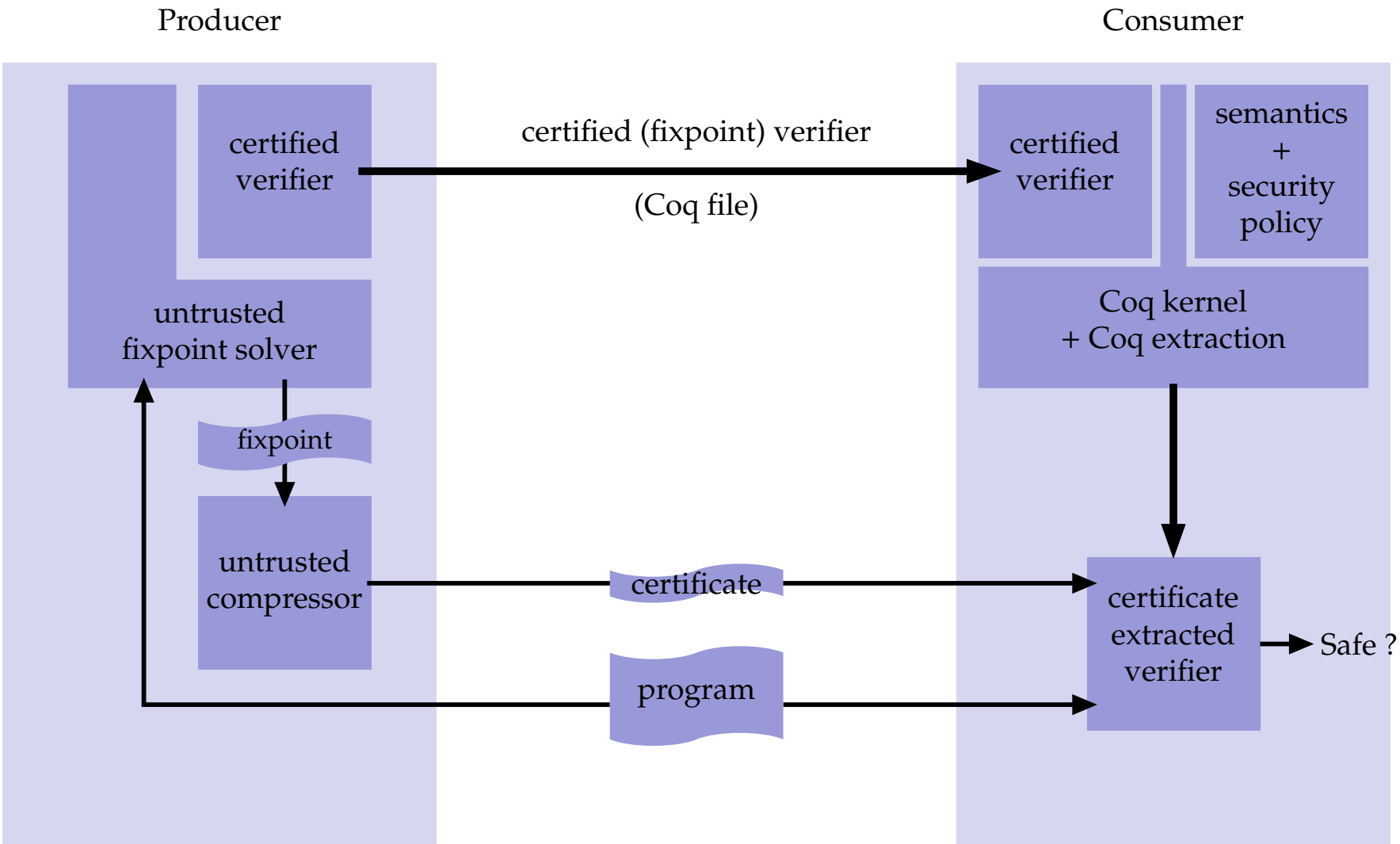
PCC requirements:

- the certificate must be small
- the verifier must be efficient
- soundness of the certifying prover is not critical
- soundness of the verifier is critical
  - What about extracting a certified verifier ?

The weakest link is . . . the checker



Proof-carrying code from abstract interpretation



Conclusions

Software security is a open-ended challenge for developers.

- cryptographic protocols are complex distributed algorithms,
- mobile code multiplies and make code inspection a major endeavour

Program analysis can help

- in the development phase, to understand consequences of design choices
- in the deployment phase, to communicate trust in mobile code
  - by generating program certificates automatically
  - by providing machine-checkable certificate checkers

Logic-based software certificates

- exist for certain verifications (on-device byte code verification)
- have potential to cover many more security-related properties

Semantic program analysis for software safety and security

Thomas Jensen

IRISA

June 26, 2007