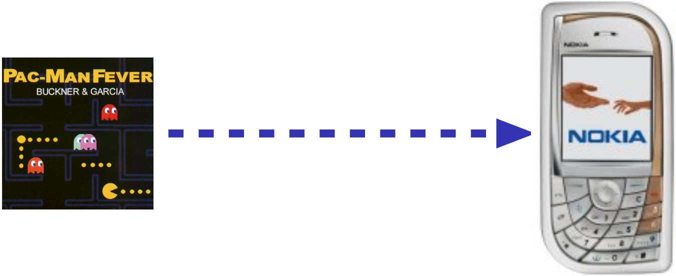
Proof Carrying Code : a quick tour

Types Summer School 2007 - Bertinoro - Italy

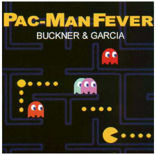
David Pichardie

INRIA Rennes - Bretagne Atlantique

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

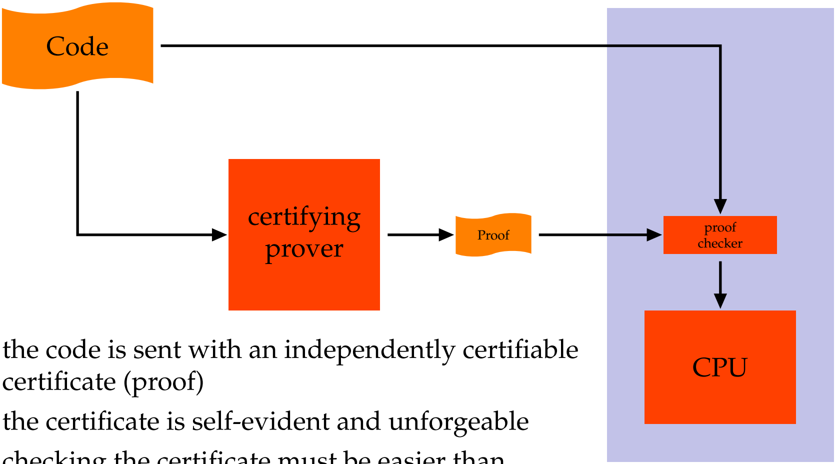
Solutions

- 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

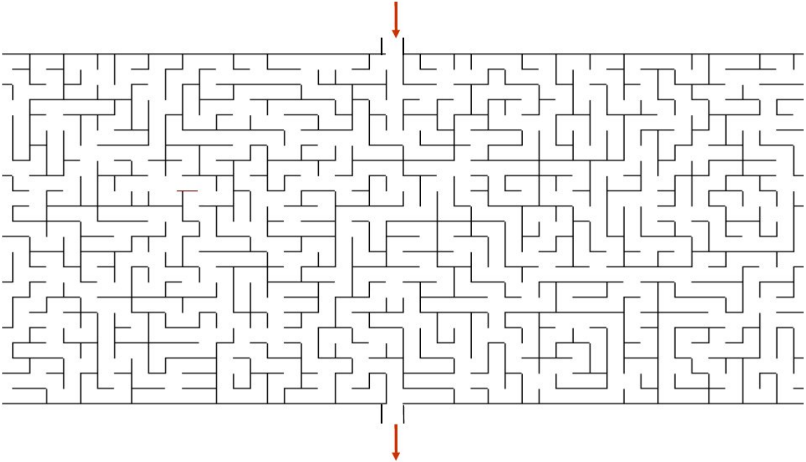
Proof carrying code : principles



- 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

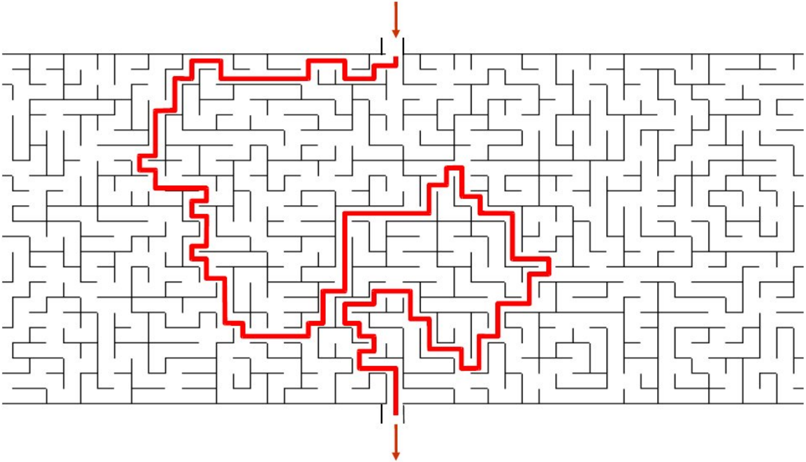
©G. Necula



program = maze

The maze metaphor

©G. Necula

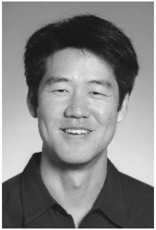


program = maze

proof = red path

Plan

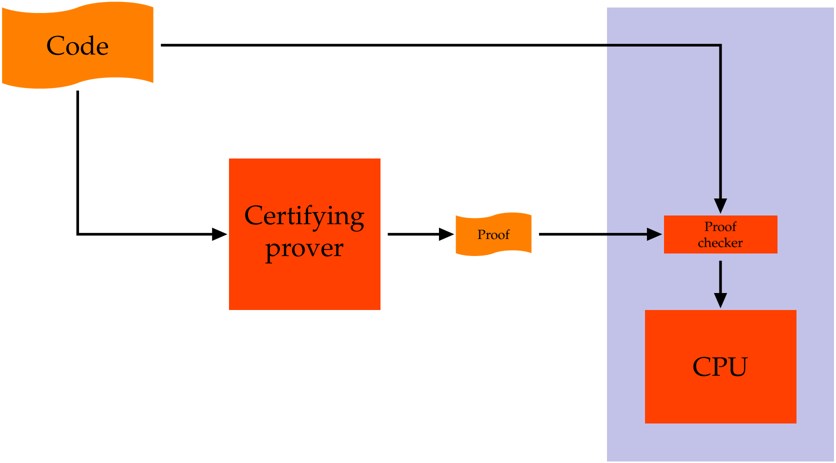
The Proof Carrying Code's pioneers



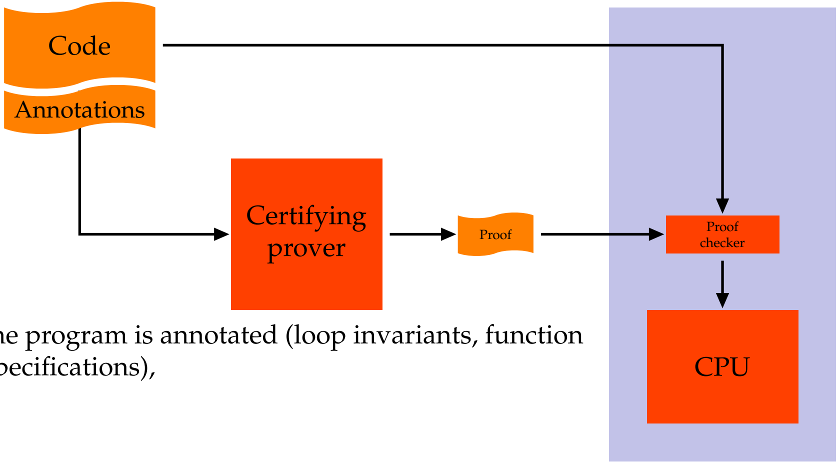
First proposed by Georges Necula (Berkley) and Peter Lee (CMU).

- 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

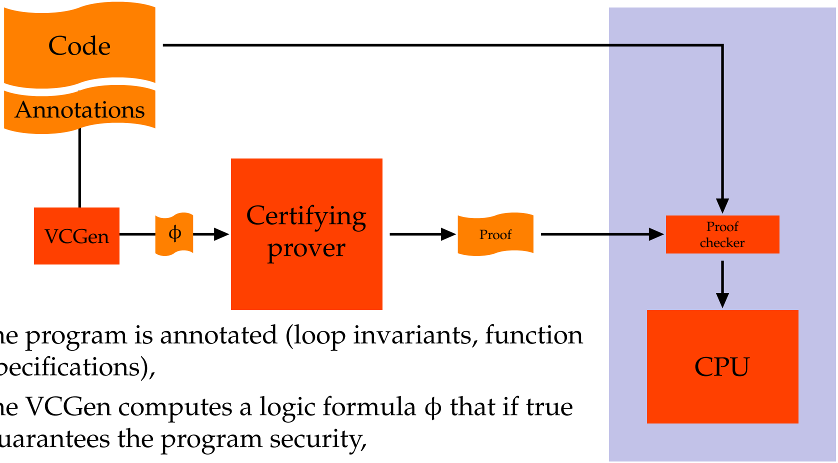


Proof carrying code : standard framework



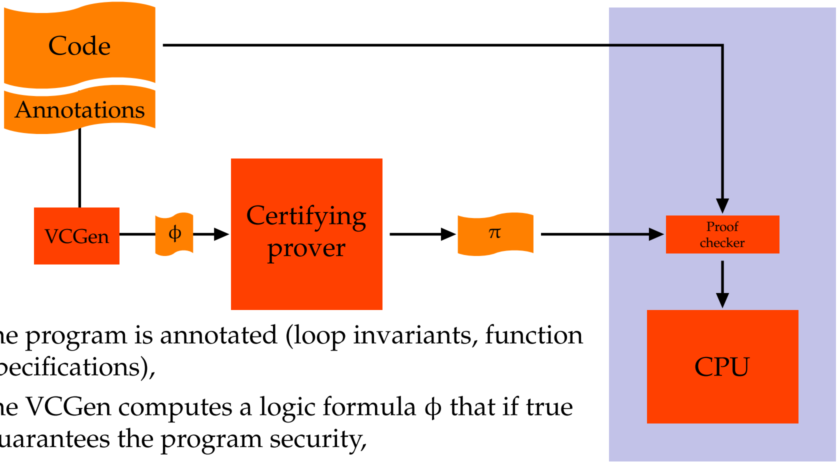
- the program is annotated (loop invariants, function specifications),

Proof carrying code : standard framework



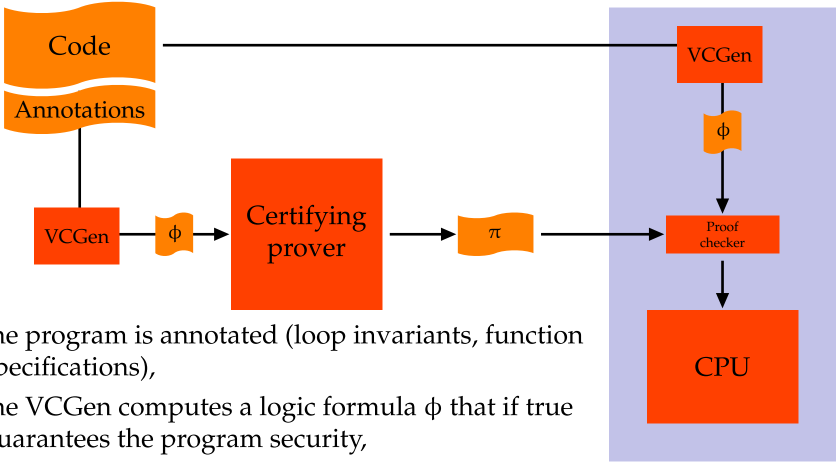
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,

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 $1000\times$code !)

Variants :

- LF_i is a variant[2] where the proof checker infers by itself fragments of the proof ($2.5\times$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

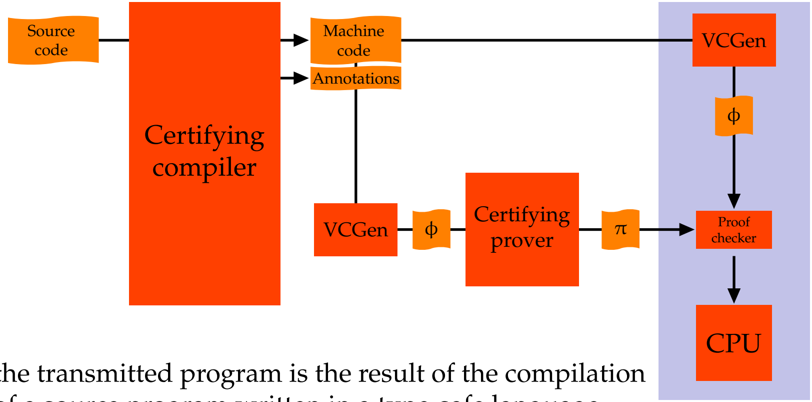
The certifying prover

- 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

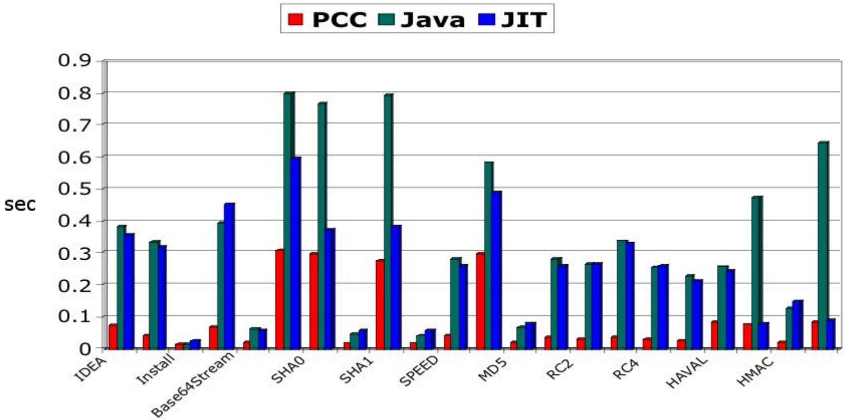
Annotation generation



- 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

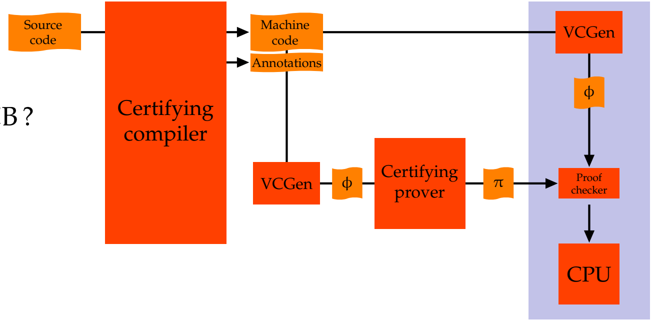
Intermediate conclusions on standard PCC

- 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 ?

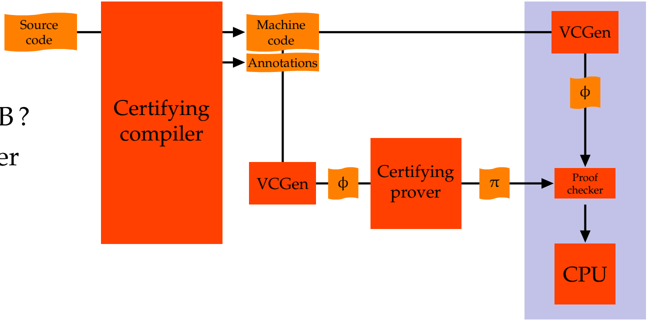


Trusted Computing Base (TCB)

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

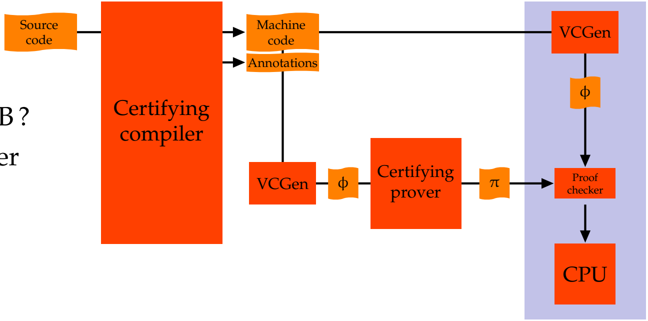


Trusted Computing Base (TCB)

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



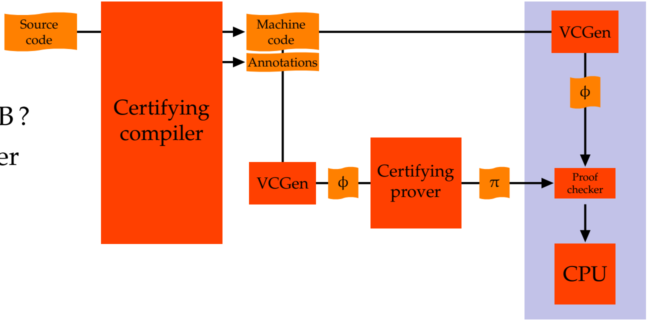
You don't need to trust ...

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



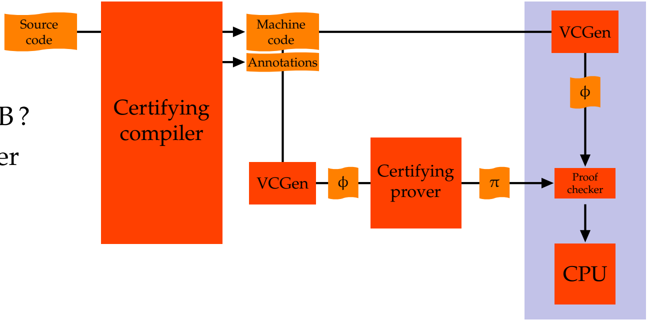
You don't need to trust the compiler ...

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



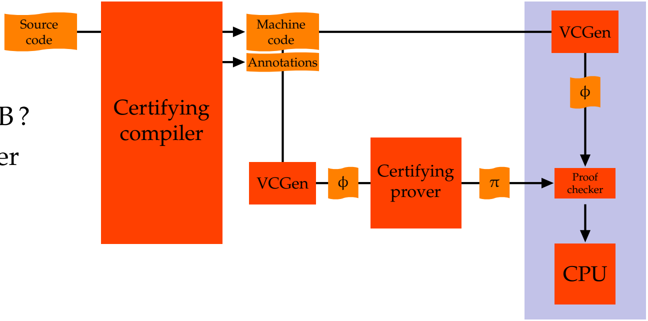
You don't need to trust the compiler, the annotations ...

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



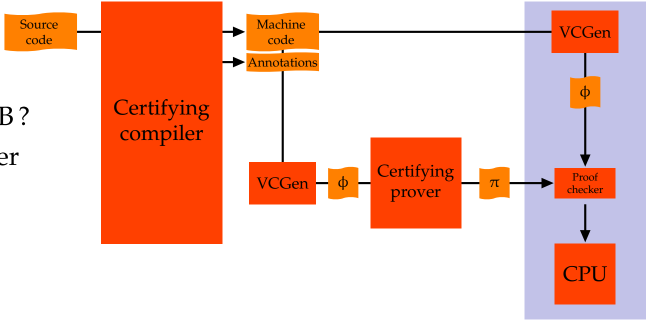
You don't need to trust the compiler, the annotations, the prover ...

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

Plan

Perspectives : Mobius project

- PCC for Java mobile code,
- 16 European partners,
- started in 2005 for 4 years,
- coordinated by INRIA

The goals of the Mobius project

- Certified PCC
  - PCC soundness must be machine-checked
  - Mobius uses the Coq proof assistant
- Security policy beyond memory-safety
  - information flow : public outputs should not depends on confidential data
  - resource usage : memory usage, billable actions,...
  - functional correctness : proof-transforming compilation (Lecture 2)
- Innovative PCC certificate formats : proof by reflection (Lecture 3)
- Program verification
  - Multi-threaded programs
  - Extensive tool support
- ... see `http://mobius.inria.fr`

Certified PCC

First component : Bicolano, an operational model of the Java Virtual Machine

- the basis for all machine checked proofs in Mobius
- JVM have been already modeled in proof assistants (Isabelle, ACL2, Coq)
- but Bicolano have some particularities :
    - targets the CLDC platform (Java for mobile devices)
    - uses intensively the Coq module system
        - some components are described as abstract types to be independent from any particular implementation choice
        - efficient implementations provided (using functional maps)
    - currently restricted to sequential programs but a multi-threaded extension is foreseen

Formal semantics : the weak point of proofs on programming language

Two examples of theorem

Theorem

$x^n + y^n = z^n$ has no non-zero integer solutions for x, y and z when $n > 2$.

Theorem

for all programs p, $\mathrm{analyse}(p)$ computes a correct approximation of $[\![p]\!]$.

Formal semantics : the weak point of proofs on programming language

Two examples of theorem

Theorem

$x^n + y^n = z^n$ has no non-zero integer solutions for x, y and z when $n > 2$.

- depends on the definition of $\mathbb{N}, \mathbb{Z}, +, >$ and the power function.

Theorem

for all programs p, $\mathrm{analyse}(p)$ *computes a correct approximation of* $[\![p]\!]$.

Formal semantics : the weak point of proofs on programming language

Two examples of theorem

Theorem

$x^n + y^n = z^n$ has no non-zero integer solutions for x, y and z when $n > 2$.

- depends on the definition of $\mathbb{N}, \mathbb{Z}, +, >$ and the power function.

Theorem

for all programs p, $\mathrm{analyse}(p)$ *computes a correct approximation of* $[\![p]\!]$.

- depends on the definition of $[\![\cdot]\!]$

Formal semantics : the weak point of proofs on programming language

Two examples of theorem

Theorem

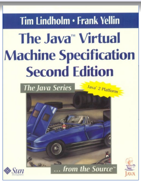
$x^n + y^n = z^n$ has no non-zero integer solutions for x, y and z when $n > 2$.

- depends on the definition of $\mathbb{N}, \mathbb{Z}, +, >$ and the power function.

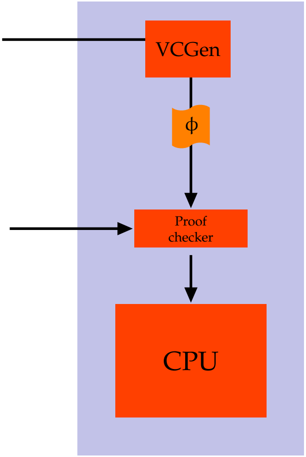
Theorem

for all programs p, $\mathrm{analyse}(p)$ *computes a correct approximation of* $[\![p]\!]$.

- depends on the definition of $[\![\cdot]\!]$
- a 400 pages book !

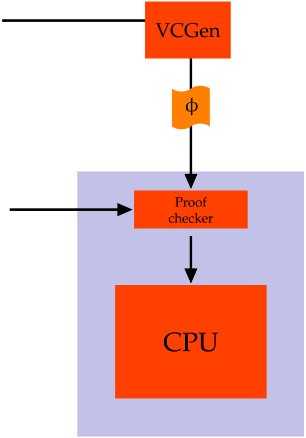


TCB of certified PCC



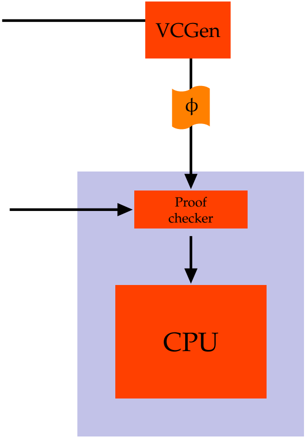
1. In standard PCC

TCB of certified PCC



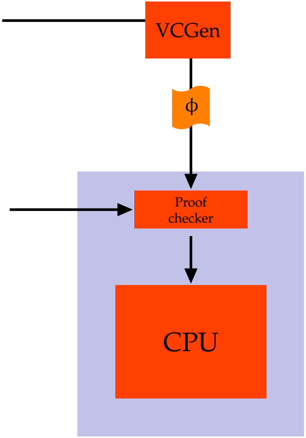
1. In standard PCC
2. If the VCGen is proved correct

TCB of certified PCC



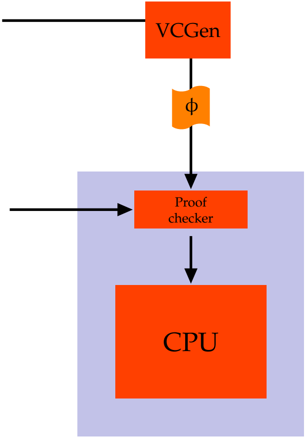
1. In standard PCC
2. If the VCGen is proved correct
   - + the proof checker of the VCGen soundness proof (could be the same as for the code proof)

TCB of certified PCC



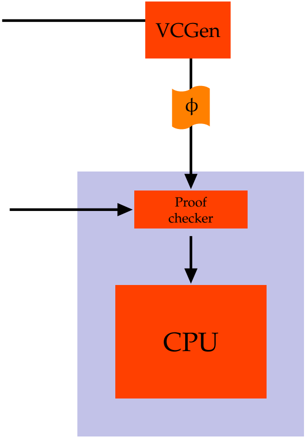
1. In standard PCC
2. If the VCGen is proved correct
   - + the proof checker of the VCGen soundness proof (could be the same as for the code proof)
   - + the formal definition of the language semantics

TCB of certified PCC



1. In standard PCC
2. If the VCGen is proved correct
   - + the proof checker of the VCGen soundness proof (could be the same as for the code proof)
   - + the formal definition of the language semantics
   - + the formal definition of the security property

TCB of certified PCC



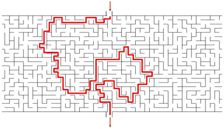
1. In standard PCC
2. If the VCGen is proved correct
   - + the proof checker of the VCGen soundness proof (could be the same as for the code proof)
   - + the formal definition of the language semantics
   - + the formal definition of the security property

This is still a large TCB but I prefer a TCB with large formal definitions than with 20,000 lines of C code. But this is a matter of taste !

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