Proof-Carrying-Code

Applying formal methods in a distributed world

Hans-Wolfgang Loidl

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

June 30, 2005

Motivation

Downloading software over the network is nowadays common-place.

But who says that the software does what it promises to do?

Who protects the consumer from malicious software or other undesirable side-effects?

$\Longrightarrow$ **Mechanisms for ensuring that a program is "well-behaved" are needed.**

Authentication for Mobile Code

The main mechanisms used nowadays are based on authentication.
Java:

- Originally a sandbox model where all code is untrusted and executed in a secure environment (sandbox)
- Since version 1.2 security policies can be defined to have more fine-grained control over the level of security defined. Managed through cryptographic signatures on the code.

Authentication for Mobile Code

Windows:

- Microsoft's Authenticode attaches cryptographic signatures to the code.
- User can distinguish code from different providers.
- Very widely used — more or less compulsory in Windows XP for drivers.

But, all these mechanisms say nothing about the code, only about the supplier of the code!

Whom do you trust completely?

Maybe that's not such a good idea!

Microsoft Security Bulletin MS01-017

Who should read this bulletin: All customers using Microsoft® products.

Technical description: In mid-March 2001, VeriSign, Inc., advised Microsoft that on January 29 and 30, 2001, it issued two VeriSign Class 3 code-signing digital certificates to an individual who fraudulently claimed to be a Microsoft employee. …

Impact of vulnerability: Attacker could digitally sign code using the name "Microsoft Corporation".

Proof-Carrying-Code (PCC): The idea

Goal: Safe execution of untrusted code.

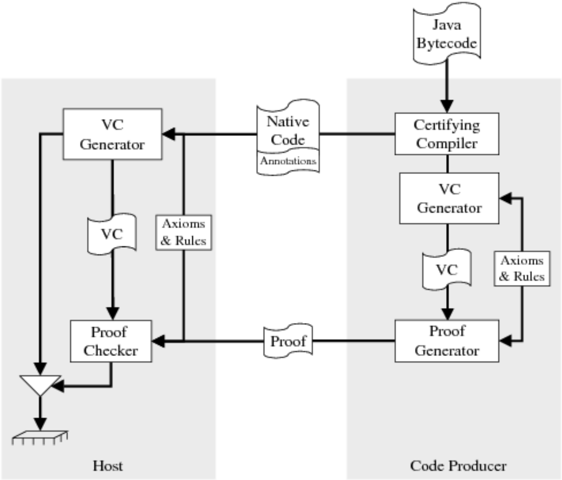
> *PCC is a software mechanism that allows a host system to determine with certainty that it is safe to execute a program supplied by an untrusted source.*

Method: Together with the code, a *certificate* describing its behaviour is sent.

This certificate is a condensed form of a formal proof of this behaviour.

Before execution, the consumer can check the behaviour, by running the proof against the program.

A PCC architecture



Program Verification Techniques

Many techniques for PCC come from the area of **program verification**. Main differences:

General program verification

- is trying to **verify good behaviour** (correctness).
- is usually interactive
- requires at least programmer annotations as invariants to the program

Program Verification Techniques

Many techniques for PCC come from the area of **program verification**. Main differences:

General program verification

- is trying to **verify good behaviour** (correctness).
- is usually interactive
- requires at least programmer annotations as invariants to the program

PCC

- is trying to **falsify bad behaviour**
- must be automatic
- may be based on inferred information from the high-level

Program Verification Techniques

Many techniques for PCC come from the area of **program verification**. Main differences:

General program verification

- is trying to **verify good behaviour** (correctness).
- is usually interactive
- requires at least programmer annotations as invariants to the program

PCC

- is trying to **falsify bad behaviour**
- must be automatic
- may be based on inferred information from the high-level

Observation: Checking a proof is much simpler than creating one

PCC: Selling Points

Advantages of PCC over present-day mechanisms:

- General mechanism for many different safety policies
- Behaviour can be checked before execution
- Certificates are tamper-proof
- Proofs may be hard to generate (producer) but are easy to check (consumer)

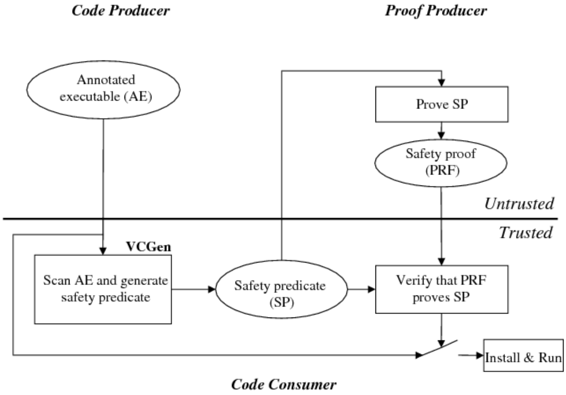
A more general framework

In general the code producer might be different from the proof producer. In this case, validation becomes a 2 stage process

- The code consumer receives an annotated program
- Run the VCG to generate the property to be proven
- Transfer the VCs to a proof producer
- Check the proof delivered by the proof producer w.r.t. VCs

Note that no trust relationship is needed here either, since the final check is still done on the code consumer side.

A PCC architecture



Variants to this framework

In particular scenarios the following specialisations might be appropriate:

- **Classic PCC:** The code producer may run VCG itself and retrieve a proof from proof producer, to then send a code/certificate pair to the consumer
  Advantage: saves a communication step.

Variants to this framework

In particular scenarios the following specialisations might be appropriate:

- **Classic PCC:** The code producer may run VCG itself and retrieve a proof from proof producer, to then send a code/certificate pair to the consumer
  Advantage: saves a communication step.
- **Simple PCC:** For *very* simple properties the code consumer might do proof generation
  Advantage: avoids the transmission of an entire proof.

Variants to this framework

In particular scenarios the following specialisations might be appropriate:

- **Classic PCC:** The code producer may run VCG itself and retrieve a proof from proof producer, to then send a code/certificate pair to the consumer
  Advantage: saves a communication step.
- **Simple PCC:** For *very* simple properties the code consumer might do proof generation
  Advantage: avoids the transmission of an entire proof.
- **Delegated Checking:** Separate code consumer and code executer by introducing a trust relationship between both.
  Advantage: handle complex certificates for programs on tiny systems, e.g. smartcards.

What does "well-behaved" mean?

PCC is a general framework and can be instantiated to many different **safety policies**.

A safety policy defines the meaning of "well-behaved".

What does "well-behaved" mean?

PCC is a general framework and can be instantiated to many different **safety policies**.

A safety policy defines the meaning of "well-behaved".

Examples:

- (functional) correctness
- type correctness ([1])
- array bounds and memory access (CCured)
- resource-consumption (MRG)
- network interaction (e.g. packet filtering [2])

Main Challenges of PCC

PCC is a very powerful mechanism. Coming up with an efficient implementation of such a mechanism is a challenging task.

The main problems are

- Certificate Size
- Size of the trusted code base (TCB)
- Performance of validation

Certificate Size

A certificate is a formal proof, and can be encoded as e.g. LF Term.

BUT: such proof terms include a lot of repetition
$\Longrightarrow$ huge certificates

Approaches to reduce certificate size:

- Compress the general proof term and do reconstruction on the consumer side
- Transmit only hints in the certificate (oracle strings)
- Embed the proving infrastructure into a theorem prover and use its tactic language

Size of the Trusted Code Base (TCB)

The PCC architecture relies on the correctness of components such as VC-generation and validation.

But these components are complex and implementation is error-prone.

Approaches for reducing size of TCB:

- Use proven/established software
- Build everything up from basics **foundational PCC** (Appel)

Performance

Even though validation is fast compared to proof generation, it is on the critical path of using remote code
$\Longrightarrow$ performance of the validation is crucial for the acceptance of PCC.

Approaches:

- Write your own specialised proof-checker (for a specific domain)
- Use hooks of a general proof-checker, but replace components with more efficient routines, e.g. arithmetic

Certifying Compiler

Producing a certificate amounts to verifying a statement determined by the safety policy for a given program.

Can be very time-consuming but is just a one-off process.

Complexity very much depends on the safety policy.

Validator

The consumer has to check that the received code fulfils the properties defined in safety policy (**validation**).

The consumer needs to establish that

- the certificate corresponds to the program
- the certificate is correct
- the certificate corresponds to the safety policy

Validator

What exactly is proven?

The safety policy is typically encoded as a pre-post-condition pair (P/Q) for a program e, and a logic describing how to reason.

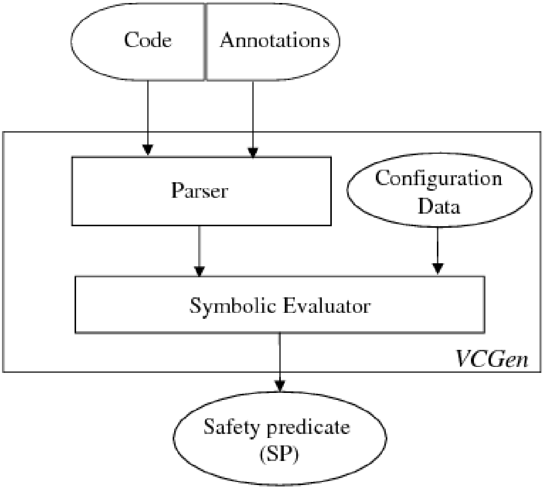
Running the verification condition generator VCG over e and Q, generates a set of conditions, that need to be fulfilled in order for the program to be safe.

The condition that needs to be proven is:

$$P \Longrightarrow VC(e, Q)$$

.

Structure of the VCG



An Example: type-safe assembler code

Scenario: type-safe assembler code as target of the compilation of a high-level language.

Compiler guarantees type-safety, but what about imported **foreign code** for level data access?

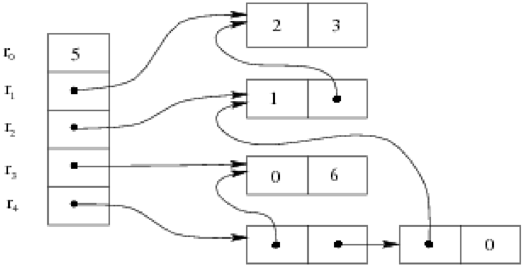
PCC approach: define an appropriate safety policy and attach a certificate to the foreign code imported.

SML code for sum

```
datatype T = Int of int | Pair of int * int
fun sum (l : T list) =
 let
   fun foldr f a nil    = a
     | foldr f a (h::t) = foldr f (f(a,h)) t
 in
   foldr (fn (acc, Int i) => acc + i
           | (acc, Pair(i,j) => acc+i+j))
         0 l
```

Goal: Write an optimised version of sum in assembler code, and make sure the code is type-safe.

Data representation




```
val r0 : int = 5
val r1 : int*int = (2,3)
val r2 : T = Pair r1
val r3 : T = Int 6
val r4 : T list = [r3, r2]
```

Components of the infrastructure

Safety policy: every read operation references a readable address

Language: abstraction over assembler code (akin to DEC assembler)

State: register environment and program counter;

$$
\begin{array}{lcl}
e & ::= & n \mid r_i \mid sel(m, e) \mid e_1 + e_2 \\
m & ::= & r_m \mid upd(m, e_1, e_2)
\end{array}
$$

$sel(m, e)$ is the contents at memory location e in store m;
$upd(m, e_1, e_2)$ is a new store, updating the contents in location e_1 with the value in e_2

Logic: encodes data representation

Language

A subset of DEC assembler code.

$$
\begin{array}{rcl}
e & ::= & \mathtt{ADD}\ \mathtt{r_s}, \mathtt{op}, \mathtt{r_d} \\
  & | & \mathtt{LD}\ \ \mathtt{r_d}, \mathtt{n}(\mathtt{r_s}) \\
  & | & \mathtt{ST}\ \ \mathtt{r_s}, \mathtt{n}(\mathtt{r_d}) \\
  & | & \mathtt{BEQ}\ \mathtt{r_s}, \mathtt{n} \\
  & | & \mathtt{INV}\ I
\end{array}
$$

where r_s, r_d are registers

Operational Semantics of the Abstract Machine

The (small step) operational semantics defines a transition of a state (ρ, pc) when evaluating the code at pc.

$$
\begin{array}{lll}
(\rho, pc) \rightsquigarrow & & \\
\quad (\rho \circ (r_d \mapsto \rho(r_s) + \rho(op)), pc + 1) & \mathit{if} & \Pi_{pc} = \texttt{ADD } \texttt{r}_\texttt{s}, \texttt{op}, \texttt{r}_\texttt{d} \\
\quad (\rho \circ (r_d \mapsto sel(\rho(r_m), \rho(r_s) + n), pc + 1) & \mathit{if} & \Pi_{pc} = \texttt{LD } \texttt{r}_\texttt{d}, \texttt{n}(\texttt{r}_\texttt{s}) \\
 & & \wedge\ r_m \vdash r_s + n : addr \\
\quad (\rho \circ upd(\rho(r_m), \rho(r_d) + n, \rho(r_s)), pc + 1) & \mathit{if} & \Pi_{pc} = \texttt{ST } \texttt{r}_\texttt{s}, \texttt{n}(\texttt{r}_\texttt{d}) \\
 & & \wedge\ r_m \vdash r_d + n : addr \\
\quad (\rho, pc + n + 1) & \mathit{if} & \Pi_{pc} = \texttt{BEQ } \texttt{r}_\texttt{s}, \texttt{n} \ \wedge\ r_s = 0 \\
\quad (\rho, pc + 1) & \mathit{if} & \Pi_{pc} = \texttt{BEQ } \texttt{r}_\texttt{s}, \texttt{n} \ \wedge\ r_s \neq 0 \\
\quad (\rho, pc + 1) & \mathit{if} & \Pi_{pc} = \texttt{INV } I
\end{array}
$$

Logic: Elimination rules

The logic consists of a fragment of first-order predicate logic, and special rules on data-type constructors.

$$\frac{m \vdash e : \tau_1 * \tau_2}{\begin{array}{l} m \vdash e : addr \ \wedge \ m \vdash e + 4 : addr \ \wedge \\ m \vdash sel(m, e) : \tau_1 \ \wedge \ m \vdash sel(m, e+4) : \tau_2 \end{array}} \quad (\textsc{ProdElim})$$

$$\frac{m \vdash e : \tau_1 + \tau_2}{\begin{array}{l} m \vdash e : addr \ \wedge \ m \vdash e + 4 : addr \ \wedge \\ sel(m, e) = 0 \Rightarrow m \vdash sel(m, e+4) : \tau_1 \ \wedge \\ sel(m, e) \neq 0 \Rightarrow m \vdash sel(m, e+4) : \tau_2 \end{array}} \quad (\textsc{SumElim})$$

$$\frac{m \vdash e : \tau \ list \quad e \neq 0}{\begin{array}{l} m \vdash e : addr \ \wedge \ m \vdash e + 4 : addr \ \wedge \\ m \vdash sel(m, e) : \tau \ \wedge \ m \vdash sel(m, e+4) : \tau \ list \end{array}} \quad (\textsc{ListElim})$$

Logic: Introduction rules

$$\frac{m \vdash e_1 : int \quad m \vdash e_2 : int}{m \vdash e_1 + e_2 : int} \qquad (\textsc{SumIntro})$$

$$\frac{}{m \vdash 0 : int} \qquad (\textsc{Const})$$

Assembler code for sum

99sum:MMMMMMMMMM*kill*

```
                                    %r_0 is 1
sum:INV r_m ⊢ r_0 : t list
        MOV r_1, 0                  % initialise acc
L_2     INV r_m ⊢ r_0 : T list ∧ r_m ⊢ r_1 : int
        BEQ r_0, L_14               % is list empty?
        LD  r2,0(r0)                % load head
        LD  r0,4(r0)                % load tail
        LD  r3,0(r2)                % load constructor
        LD  r2,4(r2)                % load data
        BEQ r_3, L_12               % is an integer?
        LD  r3,0(r2)                % load i
        LD  r2,4(r2)                % load j
        ADD r2,r3,r2                % add i and j
L_12    ADD r1,r2,r1                % do the addition
        BR  L_2                     % loop
L_14    MOV r0,r1                   % copy result to r0
        RET
```

Safety property

The interface for the function sum is a pair of pre- and post-conditions, written as a Hoare-style judgement:

$$\{r_m \vdash r_0 : T\ list\}\ \texttt{sum}\ \{r_m \vdash r_0 : int\}$$

A Verification Condition Generator

The VCG computes the side-conditions necessary for the post-condition *Post* to hold after executing the code starting at Π_i.

$$
VC_i = \begin{cases}
(r_s + op / r_d) VC_{i+1} & \textit{if} \quad \Pi_i = \mathtt{ADD}\ \mathtt{r_s, op, r_d} \\
r_m \vdash r_s + n : addr \wedge & \\
(sel(r_m, r_s + n) / r_d) VC_{i+1} & \textit{if} \quad \Pi_i = \mathtt{LD}\ \ \mathtt{r_d, n(r_s)} \\
r_m \vdash r_d + n : addr \wedge & \\
upd(\rho(r_m), \rho(r_d) + n, \rho(r_s)) VC_{i+1} & \textit{if} \quad \Pi_i = \mathtt{ST}\ \ \mathtt{r_s, n(r_d)} \\
(r_s = 0 \Rightarrow VC_{i+n+1}) \wedge & \\
(r_s \neq 0 \Rightarrow VC_{i+1}) & \textit{if} \quad \Pi_i = \mathtt{BEQ}\ \mathtt{r_s, n} \\
Post & \textit{if} \quad \Pi_i = \mathtt{RET} \\
I & \textit{if} \quad \Pi_i = \mathtt{INV}\ I
\end{cases}
$$

VCs generated for this example

The VCs for the example program consist of 2 clauses:

Clause 1: the loop invariant holds when reaching the head of the loop from the function entry

$$r_m \vdash r_0 : T\ list \Rightarrow (r_m \vdash r_0 : T\ list \wedge r_m \vdash r_0 : int)$$

Clause 2: expresses the loop invariant is preserved and it entails the post-condition.

Generating the proof

The code producer generates the VCs and finds a proof by applying rules of a fragment of first-order predicate logic.

This can be done by encoding the program as an LF (Logical Frameworks) signature and the verification condition as an LF type. Finding the proof then means doing type-checking in LF.

Cheaper: make use of high-level (type) information and bring this information down to the low-level representation.

Embedding the logic in a modern theorem prover, powerful tactics (and tactic languages) for proof-search can be used.

A performance bottleneck is often the poor support for arithmetic.

The main theorem

Main theorem:

Theorem

For any program Π, invariants Inv and post-condition Post: if $\triangleright VC(\Pi, Inv, Post)$ and the initial state fulfills the pre-condition Pre, then the program reads only from valid memory locations (as defined by the typing rules) and if the program terminates the final state fulfills the post-condition.

For Further Reading

- George Necula, *Proof-carrying code* in POPL'97 — Symposium on Principles of Programming Languages, Paris, France, 1997.
  `http://raw.cs.berkeley.edu/Papers/pcc_popl97.ps`
- George Necula, *Proof-Carrying Code: Design and Implementation* in Proof and System Reliability, Springer-Verlag, 2002.
  `http://raw.cs.berkeley.edu/Papers/marktoberdorf.pdf`