Proof-Carrying-Code

Hans-Wolfgang Loidl

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

June 19, 2007

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?

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: define safety policies to control the level of safety; managed through cryptographic signatures on the code.

Windows: Microsoft's Authenticode attaches cryptographic signatures to the code; more or less compulsory in Windows XP for drivers.

Authentication for Mobile Code

The main mechanisms used nowadays are based on **authentication**.

Java: define safety policies to control the level of safety; managed through cryptographic signatures on the code.

Windows: Microsoft's Authenticode attaches cryptographic signatures to the code; 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.

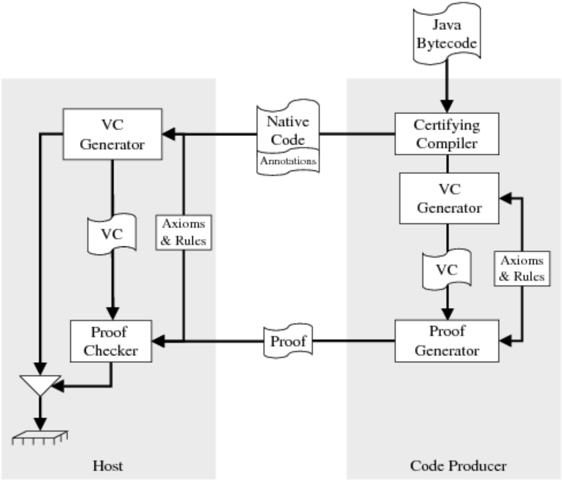
> *PCC is a software mechanism that allows a host system to determine with certainty that it is safe to execute a program supplied by an untrusted source.*

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

An Example: CCured

CCured is a system for checking **pointer-safety** of C programs, developed by the group of George Necula at Berkeley.

Uses a hybrid mechanism of static type checking and run-time checks.

Goal: Prove pointer safety statically, where possible, and minimise required run-time checks.

A Roadmap to a PCC infrastructure

Task of the infrastructure: **Certify** that the **execution** of the program is **well-behaved**.

Several steps to build the infrastructure:

- Formalise **execution** as an **operational semantics** of the language.
- Formalise **well-behaved** as a **safety policy** (type-system)
- **Certify** safety by producing a proof-term (or similar).

The core language

Mini-C language:

$$
\begin{array}{lcl}
e & ::= & x \mid n \mid e_1 \ \mathtt{op} \ e_2 \mid (\tau)e \mid e_1 \oplus e_2 \mid \, !e \\
c & ::= & \mathtt{skip} \mid c_1 ; c_2 \mid e_1 := e_2
\end{array}
$$

Types: standard C types with extension for **pointers into arrays and dynamic types**.

Efficient type inference is possible and demonstrated for this type system.

The CCured type system: pointers

C contains 2 evil pointer operations: arithmetic and casts.

The type system distinguishes between 3 kinds of pointers:

- **Safe pointers**: no arithmetic or casts; represented as an address
- **Sequence pointers**: arithmetic but **no casts**; represented as a region
- **Dynamic pointers**: **casts**, all bets are off! represented as a region

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                        // ptr arithm
    e = *p;                           // read elem
    while ((int)e % 2 == 0) {         // check tag
        e = *(int **)e;               // unbox
    }
    acc += ((int)e >> 1);             // strip tag
}
```

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                      // ptr arithm
    e = *p;                         // read elem
    while ((int)e % 2 == 0) {       // check tag
        e = *(int **)e;             // unbox
    }
    acc += ((int)e >> 1);           // strip tag
}
```

a and p point into an array with elems of type int *

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                          // ptr arithm
    e = *p;                             // read elem
    while ((int)e % 2 == 0) {           // check tag
        e = *(int **)e;                 // unbox
    }
    acc += ((int)e >> 1);               // strip tag
}
```

a is subject to pointer arithm ("sequence pointer")
$\Longrightarrow$ check for out of bounds

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                        // ptr arithm
    e = *p;                           // read elem
    while ((int)e % 2 == 0) {         // check tag
        e = *(int **)e;               // unbox
    }
    acc += ((int)e >> 1);             // strip tag
}
```

p has no arithmetic (“safe pointer”)
$\Longrightarrow$ no bounds check needed

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                      // ptr arithm
    e = *p;                         // read elem
    while ((int)e % 2 == 0) {       // check tag
        e = *(int **)e;             // unbox
    }
    acc += ((int)e >> 1);           // strip tag
}
```

e is subject to a type cast ("dynamic pointer")
$\Longrightarrow$ nothing known about underlying type

Operational Semantics

The value of an integer, or a safe pointer is an integer n; the value of a sequence or dynamic pointer is a **home**, modelled as a pair $\mathbb{N} \times \mathbb{N}$ of start address and offset.

$$v ::= n \mid \langle h, n \rangle$$

Operational Semantics

The value of an integer, or a safe pointer is an integer n; the value of a sequence or dynamic pointer is a **home**, modelled as a pair $\mathbb{N} \times \mathbb{N}$ of start address and offset.

$$v ::= n \mid \langle h, n \rangle$$

Each home is tagged as being an integer or a pointer, and has an associated **kind** and **size** functions. The semantic domain for pointers:

$$\begin{array}{lcl}
\| \texttt{int} \|_H & = & \mathbb{N} \\
\| \texttt{DYNAMIC} \|_H & = & \{\langle h, n \rangle \mid \ h \in H \wedge (h = 0 \vee \texttt{kind}(h) = \mathit{untyped}\} \\
\| \tau \ \texttt{ref SEQ} \|_H & = & \{\langle h, n \rangle \mid \ h \in H \wedge (h = 0 \vee \texttt{kind}(h) = \mathit{typed}(\tau)\} \\
\| \tau \ \texttt{ref SAFE} \|_H & = & \{h + i \mid \ h \in H \wedge 0 \leq i \leq \mathit{size}(h) \ \wedge \\
 & & (h = 0 \vee \texttt{kind}(h) = \mathit{typed}(\tau)\}
\end{array}$$

Operational Semantics (pointers)

$$\frac{\Sigma, M \vdash e_1 \Downarrow \langle h, n_1 \rangle \quad \Sigma, M \vdash e_2 \Downarrow n_2}{\Sigma, M \vdash e_1 \oplus e_2 \Downarrow \langle h_1, n_1 + n_2 \rangle}$$

(Pointer Artihm)

$$\frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle}{\Sigma, M \vdash (\mathtt{int})e \Downarrow h + n} \quad \text{(CastToInt)}$$

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\tau\ \mathtt{ref}\ \mathtt{SEQ})e \Downarrow \langle 0, n \rangle} \quad \text{(CastToSeq)}$$

$$\frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle \quad \mathbf{0} \leq \mathbf{n} \leq \mathtt{size}(\mathbf{h})}{\Sigma, M \vdash (\tau\ \mathtt{ref}\ \mathtt{SAFE})e \Downarrow h + n}$$

(CastToSafe)

Operational Semantics (read operations)

Two kinds of reads, with different obligations for run-time checks:

$$\frac{\Sigma, M \vdash e \Downarrow n \quad \mathbf{n \neq 0}}{\Sigma, M \vdash !e \Downarrow M(n)} \quad (\textsc{SafeRd})$$

$$\frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle \quad \mathbf{h \neq 0} \quad \mathbf{0 \leq n \leq \mathtt{size}(h)}}{\Sigma, M \vdash !e \Downarrow M(h + n)} \quad (\textsc{DynRd})$$

$$\frac{\Sigma, M \vdash e_1 \Downarrow n \quad \mathbf{n \neq 0} \quad \Sigma, M \vdash e_2 \Downarrow v}{\Sigma, M \vdash e_1 := e_2 \Downarrow M(n \mapsto v)} \quad (\textsc{SafeWr})$$

$$\frac{\Sigma, M \vdash e_1 \Downarrow \langle h, n \rangle \quad \mathbf{h \neq 0} \quad \mathbf{0 \leq n \leq \mathtt{size}(h)} \quad \Sigma, M \vdash e_2 \Downarrow v}{\Sigma, M \vdash e_1 := e_2 \Downarrow M(h + n \mapsto v)} \quad (\textsc{DynWr})$$

The CCured type system: rules

The type system keeps track of the kind of pointers.
Rules for converting pointers:

$$\frac{}{\tau \leq \tau}$$

$$\frac{}{\tau \leq \texttt{int}}$$

$$\frac{}{\texttt{int} \leq \tau\ \texttt{ref SEQ}}$$

$$\frac{}{\texttt{int} \leq \texttt{DYNAMIC}}$$

$$\frac{}{\tau\ \texttt{ref SEQ} \leq \tau\ \texttt{ref SAFE}}$$

Typing rules for commands

$\Gamma \vdash c$ means, command c is well-typed.
$\Gamma \vdash e : \tau$ means, expression e has type τ.

$$\frac{}{\Gamma \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2} \qquad \frac{\Gamma \vdash e : \tau\ \texttt{ref SAFE} \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e'}$$

$$\frac{\Gamma \vdash e : \texttt{DYNAMIC} \quad \Gamma \vdash e' : \texttt{DYNAMIC}}{\Gamma \vdash e := e'}$$

Typing rules for expressions

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \ \texttt{op} \ e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash (\tau) e : \tau}$$

$$\frac{}{\Gamma \vdash (\tau \ \texttt{ref SAFE})0 : \tau \ \texttt{ref SAFE}} \qquad \frac{\Gamma \vdash e_1 : \tau \ \texttt{ref SEQ} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \tau \ \texttt{ref SEQ}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{DYNAMIC} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{DYNAMIC}} \qquad \frac{\Gamma \vdash e : \tau \ \texttt{ref SAFE}}{\Gamma \vdash !e : \tau} \qquad \frac{\Gamma \vdash e : \texttt{DYNAMIC}}{\Gamma \vdash !e : \texttt{DYNAMIC}}$$

Safety Policy

The **safety policy** states, that at all times in the execution, the contents of each memory address must correspond to the typing constraints of the home to which it belongs.

Formally, the following predicate must be fulfilled at all times

$$
\begin{array}{rcl}
WF(M_H) & \equiv & \forall h \in H^*.\ \forall i \in \mathbb{N}. 0 \leq i < \texttt{size}(h) \Rightarrow \\
& & (\texttt{kind}(h) = \mathit{untyped} \Rightarrow M(h+i) \in \| \texttt{DYNAMIC} \|_H \wedge \\
& & \texttt{kind}(h) = \mathit{typed}(\tau) \Rightarrow M(h+i) \in \| \tau \|_H
\end{array}
$$

We can prove that this property is preserved by all rules in the type system.

Theorems

We separate run-time failure from rightful termination like this: $\Sigma, M_H \vdash e \Downarrow \mathit{CheckFailed}$ means a run-time check failed during the execution of expression e.

Theorems

We separate run-time failure from rightful termination like this: $\Sigma, M_H \vdash e \Downarrow \mathit{CheckFailed}$ means a run-time check failed during the execution of expression e.

Theorem

(Progress and type preservation) If $\Gamma \vdash e : \tau$ *and* $\Sigma \in \| \Gamma \|_H$ *and* $WF(M_H)$*, then either* $\Sigma, M_H \vdash e \Downarrow \mathit{CheckFailed}$ *or* $\Sigma, M_H \vdash e \Downarrow v$ *and* $v \in \| \tau \|_H$.

Theorems

$\Sigma, M_H \vdash c \Longrightarrow \mathit{CheckFailed}$ means a run-time check failed during the execution of command c.

Theorems

$\Sigma, M_H \vdash c \Longrightarrow \mathit{CheckFailed}$ means a run-time check failed during the execution of command c.

Theorem

(Progress for commands) If $\Gamma \vdash c$ and $\Sigma \in \parallel \Gamma \parallel_h$ and $WF(M_H)$ then either $\Sigma, M_H \vdash c \Longrightarrow \mathit{CheckFailed}$ or $\Sigma, M_H \vdash c \Longrightarrow M'_H$ and M'_H is well-formed.

Main results

- An efficient inference algorithm attaches `ref SEQ`, `ref SAFE`, `DYNAMIC` annotations to plain C code.
- Most of the checks can be done statically.
- The performance overhead of the remaining run-time checks is moderate: 0–150%
- Purely dynamic checks would incur a performance overhead of factors 6–20
- Several array bounds bugs discovered in SPECINT95

Further Reading

CCured: Type-Safe Retrofitting of Legacy Code, in POPL'02 — ACM Symposium on Principles of Programming Languages, 2002. Online Demo at `http://manju.cs.berkeley.edu/ccured/web/index.html`.

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`
- *CCured Demo*,
  `http://manju.cs.berkeley.edu/ccured/web/index.html`

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

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

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

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)

Foundational PCC

Philosophy of Foundational PCC: Minimise the "trusted code base", i.e. the software that needs to be trusted.

Approach of Foundational PCC: Define safety policy directly on the **operational semantics** of the code.

Certificates are proofs over the operational semantics.

Foundational PCC

Philosophy of Foundational PCC: Minimise the "trusted code base", i.e. the software that needs to be trusted.

Approach of Foundational PCC: Define safety policy directly on the **operational semantics** of the code.

Certificates are proofs over the operational semantics.

Pros and cons:

- ☺ **more flexible**: not restricted to a particular type system as the language in which the proofs are phrased;
- ☺ **more secure**: no reliance on VCG.
- ☹ **larger proofs**

Conventional vs Foundational PCC

Re-examine the logic for memory safety, eg.

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

The rule has **built-in knowledge about the type-system**, in this case representing the data layout of the compiler ("*Type specialised PCC*") $\Longrightarrow$ dangerous if soundness of the logic is not checked mechanically!

Logic rules in Foundational PCC

In foundational PCC the rules work on the operational semantics:

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

This looks similar to the previous rule but has a very different meaning: $\models$ is a predicate over the formal model of the computation, and the above rule can be proven as a lemma, $\vdash$ is an encoding of a type-system on top of the operational semantics and thus needs a **soundness proof**.

Example: specifying safe memory access

To specify safety, the operational semantics is written in such a way, that it gets stuck whenever the safety condition is violated.

Example: specifying safe memory access

To specify safety, the operational semantics is written in such a way, that it gets stuck whenever the safety condition is violated.

Example: operational semantics on assembler code.
Safety policy: “only readable addresses are loaded”.
Define a predicate: $\mathit{readable}(x) \equiv 0 \leq x \leq 1000$

Example: specifying safe memory access

To specify safety, the operational semantics is written in such a way, that it gets stuck whenever the safety condition is violated.

Example: operational semantics on assembler code.
Safety policy: "only readable addresses are loaded".
Define a predicate: $readable(x) \equiv 0 \leq x \leq 1000$
The semantics of a load operation `LD ri, c(rj)` is now written as follows:

$$
\begin{array}{lcl}
load(i,j,c) & \equiv & \lambda\ r\ m\ r'\ m'. \\
 & & r'(i) = m(r(j)+c)\ \wedge\ readable(r(j)+c)\ \wedge \\
 & & (\forall x \neq i.\ r'(x) = r(x))\ \wedge\ m' = m
\end{array}
$$

EXAMPLE: SPECIFYING SAFE MEMORY ACCESS

To specify safety, the operational semantics is written in such a way, that it gets stuck whenever the safety condition is violated.

Example: operational semantics on assembler code.
Safety policy: "only readable addresses are loaded".
Define a predicate: $\mathit{readable}(x) \equiv 0 \leq x \leq 1000$
The semantics of a load operation $\mathtt{LD}\ \mathtt{r_i}, \mathtt{c}(\mathtt{r_j})$ is now written as follows:

$$\begin{array}{lcl} \mathit{load}(i, j, c) & \equiv & \lambda\ r\ \ m\ \ r'\ \ m'. \\ & & r'(i) = m(r(j) + c)\ \wedge\ \mathit{readable}(r(j) + c)\ \wedge \\ & & (\forall x \neq i.\ r'(x) = r(x))\ \wedge\ m' = m \end{array}$$

Note: the clause for nothing else changes, quickly becomes awkward when doing these proofs
$\Longrightarrow$ Separation Logic (Reynolds'02) tackles this problem.

Further Reading

Andrew Appel, *Foundational Proof-Carrying Code* in LICS'01 — Symposium on Logic in Computer Science, 2001.
`http://www.cs.princeton.edu/~appel/papers/fpcc.pdf`

Summary

PCC is a powerful, general mechanism for providing safety guarantees for mobile code.

It provides these guarantees without resorting to a trust relationship.

It uses techniques from the areas of type-systems, program verification and logics.

It is a very active research area at the moment.

PCC reading list:
`http://www.tcs.ifi.lmu.de/~hwloidl/PCC/reading.html`