A PCC Architecture based on Certified Abstract Interpretation

Frédéric Besson and Thomas Jensen and David Pichardie

Irisa, Campus de Beaulieu, F-35042 Rennes, France

Abstract

Proof-Carrying Code (PCC) is a technique for downloading mobile code on a host machine while ensuring that the code adheres to the host's security policy. We show how certified abstract interpretation can be used to build a PCC architecture where the code producer can produce program certificates automatically. Code consumers use proof checkers derived from certified analysers to check certificates. Proof checkers carry their own correctness proofs and accepting a new proof checker amounts to type checking the checker in Coq. Fixpoint compression techniques are used to obtain compact certificates. The PCC architecture has been evaluated experimentally on a byte code language for which we have designed an interval analysis that allows to generate certificates ascertaining that no array-out-of-bounds accesses will occur.

1 Introduction

Proof-Carrying Code (PCC) is a technique for downloading mobile code on a host machine while ensuring that the code adheres to the host's safety policy. The basic idea is that the code producer sends the code with a proof (in a suitably chosen logic) that the code is secure. Upon reception of the code, the code consumer submits the proof to a proof checker for the logic. Thus, in the basic PCC architecture, the only components that have to be trusted are the program logic, the proof checker of the logic and the formalisation of the safety property in this logic. Neither the mobile code nor the proposed safety proof have to be trusted.

In his seminal work, Necula [15] axiomatises the program using a Hoare-like logic. For a given safety policy, this logic comes together with a *verification condition generator* (VCGen) that generates lemmas, the proofs of which are sufficient to ensure the property. For each lemma, a machine-checkable proof term has to be generated by the code producer. One weakness of the initial approach is that the soundness of the verification condition generator is not proved but taken for granted, having as consequence that "there were errors in that code that escaped the thorough testing of the infrastructure" [16].

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The *foundational proof carrying code* (FPCC) of Appel [2,3] gives stronger semantic foundations to PCC by generating verification conditions directly from the operational semantics rather than from some program logic, but the proofs are accordingly more complicated to produce. An alternative approach is presented by Nipkow and Wildmoser [20] who prove the soundness of a *weakest precondition* calculus with respect to the byte code semantics for a reasonable subset of Java byte code. Verification conditions are proved using a hybrid approach that use both trusted and untrusted provers. An example of a trusted prover is the byte code verifier that Klein and Nipkow have formalised and proved correct in Isabelle [10]. Untrusted provers are external static analysers that suggest potential (inductive) invariants. These invariants are then reproved inside Isabelle to obtain a transmittable program certificate.

Albert, Hermenegildo and Puebla have proposed to use the fixpoint generated by an abstract interpretation as the certificate. Their analysis-carrying code approach [1] is a PCC framework for constraint logic programs in which the checker verifies that a proposed certificate is a fixpoint of an abstract interpretation of the communicated program. This solves the problem of producing the certificates automatically but is prone to the same objections as those made against the initial PCC framework*i.e.*, how can the code consumer be sure that the checker is semantically correct.

In this paper we show how to remedy this latter problem by developing a foundational PCC architecture based on *certified abstract interpretation* [5] which is a technique for extracting a static analyser from the constructive proof of its semantic correctness. The technique produces at the same time an analyser and a proof object certifying its semantic correctness. We describe how this leads to an infrastructure that allows to download customised proof-checkers carrying their own correctness proof (Section 2). These proof checkers are derived automatically in a functorial way from a certified analysis.

An important issue in PCC is that of optimising (*i.e.*, minimising) the size of certificates. In the context of abstract interpretation-based PCC, this amounts to the compression of fixpoints, as *e.g.* it is done in lightweight byte code verification. In Section 4 we propose a fully automatic fixpoint compression algorithm that generates compressed certificates from the results of untrusted static analysers.

We have evaluated the feasibility of the approach and the efficiency of the fixpoint compression on the problem of communicating proof that a byte code program will not perform any illegal array accesses. As part of this experiment we have defined (and certified) an interval analysis for byte code that combines the standard interval-based abstract interpretation with a modicum of symbolic evaluation, resulting in a novel abstract domain of lattice expressions (Section 3.3). This extension is required in order to have a sufficiently precise analysis; at the same time it shows that complex analyses are within reach of certification and hence can be used for foundational, abstract interpretation-based PCC.

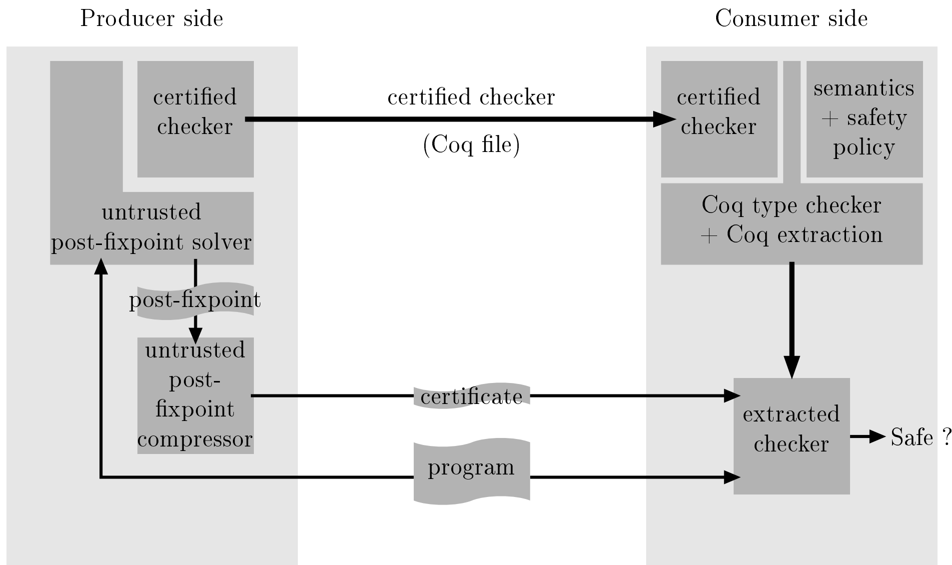

Fig. 1. PCC architecture

2 Proof Carrying Proof Checkers

In the following, we propose an extensible PCC architecture that allows to download dedicated, certified proof-checkers safely. The architecture, summarised in Figure 1, is bootstrapped by the code consumer with a general purpose proof checker, here Coq [7]. The certification of a program is done using a two-step protocol between the code producer and the code consumer. In the first step, the producer queries the consumer in order to know whether it possesses the relevant proof-checker. If not, the producer sends the checker together with its soundness proof. This soundness proof is then verified automatically by a general purpose proof checker (here, the Coq type checker) and if verification succeeds, the now certified checker is installed. In this way the architecture combines the advantages of both a trustworthy core proof checker and flexible specialised proof checkers. Once the proof checker has been installed, the consumer is ready to download the program of the provider. As it is customary in PCC, the code producer sends the program packaged with a certificate to be checked by the previously downloaded proof checker.

The producer and consumer have to agree on a formal meaning of what it means for a program to be safe. This is done by providing a Coq specification of the semantics (here, a small-step operational semantics) of the program together with a semantic definition of the security property. We restrict our attention to safety properties that must hold for all reachable execution states. More precisely, the Coq specification provides:

- the type of programs Pgm,
- a semantic domain State,
- a set of initial semantic states : $\mathcal{S}_0 \subseteq \text{State}$

- for each program, an operational semantics $\rightarrow_p \subseteq \text{State} \times \text{State}$,
- for each program, a set of states that respect the security policy: $\text{Safe}_p \subseteq \text{State}$

As usual, we write $\rightarrow_p^*$ for the reflexive transitive closure of the transition relation of the program p. The collecting semantics of a program p is defined as the set of all reachable states by $\rightarrow_p$, starting from an element of $\mathcal{S}_0$

$$[\![p]\!] = \{\ s \in \text{State} \mid \exists s_0 \in \mathcal{S}_0,\ s_0 \rightarrow_p^* s\ \}$$

Definition 2.1 A program is *safe* if all its reachable states are safe.

$$[\![p]\!] \subseteq \text{Safe}_p$$

Given a program p, the code producer has to provide a machine-checkable proof that p is safe. These proofs can be tedious and time-consuming to produce by hand. In this paper, we show how to use abstract interpretation to construct program certificates in a fully automatic way. In this approach, programs are automatically annotated with program properties (elements of abstract domains) together with a *reconstruction strategy* (to be described in detail in Section 4). A reconstruction strategy consists of a series of steps that allow to verify that the program properties form a program invariant that implies the security policy.

The certified checkers implement the signature expressed by the Coq module `Checker` in Figure 2. This module first contains a definition of the format of certificates. The function `checker` takes two arguments: a program `P` and a candidate certificate `cert` generated by an untrusted external prover. If the `checker` function returns `true`, the companion theorem `checker_ok` ensures that the program is safe, as defined in Definition 2.1. Thus, the successful type checking of a module against the signature `Checker` proves that the checker is correct.

```
Module Type Checker.
  Parameter certificate:Set.
  Parameter checker : program → certificate → bool.
  Parameter checker_ok : ∀ P cert,
  checker P cert = true → [[P]] ⊆ (Safe P).
End Checker.
```

Fig. 2. Interface for certified proof checkers

In this paper we propose a generic method to construct such a certified checker from a certified static analysis. Section 3 presents the notion of certified static analysis. In Section 4 we define a Coq functor which constructs a module of type `Checker` from any certified analysis.

3 Certified abstract interpretation for PCC

The notion of certified analysis is based on previous work on programming a static analyser in Coq [5,17]. We recall the main components of such a formalisation and explain how they are used for proof carrying code.

3.1 Certified static analysis

A certified analysis is a Coq function analyse $\in$ Pgm $\rightarrow$ bool which for a given program p either proves the safety of p (and returns true) or fails:

$$\forall p \in \mathrm{Pgm},\ \mathrm{analyse}(p) = \mathrm{true} \Rightarrow [\![p]\!] \subseteq \mathrm{Safe}_p$$

The analyser and its Coq correctness proof are built following four main steps. We stress that the following domains, functions and relations are all Coq objects that for presentational purposes are written using ordinary mathematical notation.

(i) An abstract domain $\left(\mathrm{State}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp\right)$ with a lattice structure is introduced, $\sqsubseteq^\sharp$ modelling the relative precision of elements in $\mathrm{State}^\sharp$. In the concrete world, property precision is modelled with the partial order $\subseteq$. The concrete and abstract worlds are linked by a concretisation function

$$\gamma : \left(\mathrm{State}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp\right) \rightarrow (\mathcal{P}(\mathrm{State}), \subseteq, \cup, \cap)$$

An abstract object $s^\sharp \in \mathrm{State}^\sharp$ is said to be a *correct approximation* of a concrete state $s \in$ State if and only if $s \in \gamma(s^\sharp)$ [1].

(ii) An abstract semantics is then specified as any post-fixpoint of a well-chosen abstract function $F_p^\sharp \in \mathrm{State}^\sharp \rightarrow \mathrm{State}^\sharp$. The correctness of this specification must be proved by establishing that all post-fixpoint are correct approximations of the concrete semantics.

(1) $\forall p \in \mathrm{Pgm},\ \forall s^\sharp \in \mathrm{State}^\sharp,\ F_p^\sharp(s^\sharp) \sqsubseteq^\sharp s^\sharp \ \Rightarrow\ [\![p]\!] \subseteq \gamma(s^\sharp)$

(iii) A post-fixpoint solver solve $\in$ Pgm $\rightarrow \mathrm{State}^\sharp$ is then defined, based on fixpoint iteration techniques.

(2) $\forall p \in \mathrm{Pgm},\ F_p^\sharp(\mathrm{solve}(p)) \sqsubseteq^\sharp \mathrm{solve}(p)$

(iv) An abstract safety test $\mathrm{Safe}_p^\sharp \in \mathrm{State}^\sharp \rightarrow$ bool is defined in the abstract world. For all programs p, if an abstract safety test succeeds on a correct approximation of $[\![p]\!]$, then p is safe.

(3) $\forall p \in \mathrm{Pgm}, \forall s^\sharp \in \mathrm{State}^\sharp, [\![p]\!] \subseteq \gamma(s^\sharp) \wedge \mathrm{Safe}_p^\sharp(s^\sharp) = \mathrm{true} \Rightarrow [\![p]\!] \subseteq \mathrm{Safe}_p$

[1] Because we only focus on soundness of the abstract interpreters, the classic notion of Galois connection [9] is not mandatory here. Instead we require γ to be a meet morphism, *i.e.* $\gamma(s_1^\sharp \sqcap^\sharp s_2^\sharp) = \gamma(s_1^\sharp) \sqcap \gamma(s_2^\sharp)$. This assumption is equivalent to the existence of the corresponding Galois connection when $\left(\mathrm{State}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp\right)$ is complete and $\sqcap^\sharp$ denotes the general greatest lower bound (on sets instead of two values).

Together, these proofs assert that $\mathrm{Safe}^\sharp \circ \mathrm{solve}$ is a correct analyser.

Step (2) constructs a post-fixpoint that serves as certificate for showing that the program is safe. However, for our PCC context it is important to observe that it is only *the existence* of such a post-fixpoint that matters for proving safety. Formally, by combining (1) and (3) we have that :

(4) $\forall p \in \mathrm{Pgm}, (\exists s^\sharp \in \mathrm{State}^\sharp,\ F_p^\sharp(s^\sharp) \sqsubseteq^\sharp s^\sharp \wedge \mathrm{Safe}_p^\sharp(s^\sharp) = \mathrm{true}) \Rightarrow [\![p]\!] \subseteq \mathrm{Safe}_p$

In particular, this means that for a proposed certificate $s^\sharp \in \mathrm{State}^\sharp$, our PCC checker only has to test $F_p^\sharp(s^\sharp) \sqsubseteq^\sharp s^\sharp \ \wedge\ \mathrm{Safe}_p^\sharp(s^\sharp) = \mathrm{true}$.

3.2 Certified analysis for memory invariants

We now present the Coq definition of certified analysis for languages where the semantic domain is expressed as a set of reachable states, composed of a control point and a memory $\mathrm{State} = \mathrm{Ctrl} \times \mathrm{Mem}$. The abstract domain $\mathrm{State}^\sharp = \mathrm{Ctrl} \to \mathrm{Mem}^\sharp$ attaches memory invariants to each control point of a program. The certified analysis interface is presented in Figure 3.

```
Module Type CertifiedAnalysis.
Declare Module AbMem : Lattice.

Definition AbState := Ctrl → AbMem.t.

Record Constraint : Set :=
 { target : Ctrl; expr : list AbMem.t → AbMem.t;
   sources: list Ctrl }

Definition Verif_cstr (C:Cstr) (s♯ : AbState) : Prop :=
   AbMem.order (expr C (map s♯ (sources C))) (s♯ (target C)).

Parameter gen_cstr : program → list Cstr.

Parameter genAbSafe: program → list (Ctrl*(AbMem.t → bool)).

Parameter analysis_correct : ∀ P s♯,
(∀ c, In c (gen_cstr P) → Verif_cstr c s♯) →
(∀ pc check,In (pc,check) (genAbSafe P) → check(s♯ pc)=true)→
⟦P⟧ ⊆(Safe P).
End CertifiedAnalysis.
```

Fig. 3. The Coq signature of certified static analysis

The first element of this signature is the lattice structure containing $\mathrm{Mem}^\sharp$. The Coq lattice signature provides the standard definition of lattices (partial order, least upper bound, greatest lower bound with their properties). The carrier of the lattice is represented by `AbMem.t` and the partial order by

AbMem.order. A number of lattice operations exist for designing new abstract domains. As part of our certified static analysis project, we have developed a lattice library in Coq, containing base lattices (finite sets, intervals, ...) and domain constructors (sum, product, function) that permit to construct new abstract domains by composing these basic blocks [17]. Most of the proofs follow standard lattice theory.

The abstract function $F_p^\sharp$ previously presented now operates on the domain $\left(\mathrm{Ctrl} \to \mathrm{Mem}^\sharp\right) \to \left(\mathrm{Ctrl} \to \mathrm{Mem}^\sharp\right)$. Because the number of control points for a given program is finite, post-fixpoints of $F_p^\sharp$ can be represented as solutions of a constraint system:

(5) $\{m_1^\sharp \sqsupseteq f_1(m_1^\sharp, \ldots, m_n^\sharp), \ldots, m_n^\sharp \sqsupseteq f_n(m_1^\sharp, \ldots, m_n^\sharp)\}$

A constraint $\left(m_{pc}^\sharp \sqsupseteq f(m_1^\sharp, \ldots, m_n^\sharp)\right)$ is coded by a record of type Constraint with three fields: target contains the control point pc targeted by the constraint; expr computes the right-hand side of the constraint and sources contains the list of control points which appear in the definition of f. Verif_cstr is a predicate which defines when an abstract state satisfies a constraint and gen_cstr is the constraint generation function which collects all constraints of a program.

Because abstract states $\mathrm{State}^\sharp$ are of the form $\mathrm{Ctrl} \to \mathrm{Mem}^\sharp$, we can split the abstract safety test $\mathrm{Safe}_p^\sharp$ into several local tests of the form

$$(\mathrm{pc}, \mathrm{check}) \in \mathrm{Ctrl} \times (Mem^\sharp \to \mathrm{bool}).$$

Each test is attached to a specific control point pc and ensures that no error state can be reached by a one-step transition out of the state at control point pc. For example, a safety test of array bounds checks would check the value of the index on top of the operand stack before each array access instruction of a byte code program. The check generation is realised by a function genAbSafe which returns, for a given program, a list of local tests.

The last element of the signature is a proof analysis_correct that states the global correctness of the constraint generator gen_cstr and the abstract test generator gen_AbSafe. It is a direct specialisation of the property (4) for our context. If an abstract state $s^\sharp$ verifies all the constraints generated by gen_constr (*i.e.* is a post-fixpoint of $F_p^\sharp$) and fulfills all safety checks generated by gen_AbSafe (*i.e.* $\mathrm{Safe}_p^\sharp(s^\sharp) = \mathrm{true}$), then the program is safe.

3.3 Case study: interval analysis for byte codes

To demonstrate the workings of our PCC framework and to test its feasibility we have developed an interval analysis for a simple byte code language. The analysis is based on existing interval analyses for high-level structured languages [8] but has been extended with an abstract domain of *lattice expressions* to obtain a similar precision at byte-code level. We give a succinct description of the analysis and refer the reader to a companion technical report [4] for a detailed description.

The byte code instruction set contains operators for stack and local variable manipulations and for integer arithmetic. Instructions on arrays permit to create, obtain the size of, index and update arrays. The flow of control can be modified unconditionally (with Goto) and conditionally with the family of instructions If_icmp*cond* which compare the top elements of the run-time stack and branch according to the outcome. Finally, there are instructions for inputting and returning values. This language is sufficiently general to illustrate the novelties of our approach and perform experiments on code obtained from compilation of Java source code. An extension to the object-oriented layer would follow the lines of the certified analysis for object-oriented (Java Card) byte code already developed by Cachera *et al.* [5]. The formal semantics of the language is standard and can be found in [4].

$$
\begin{array}{rcl}
\mathit{pgm} & ::= & (\mathit{pc}\ \mathit{instr}\ \mathit{pc})^* \\
\mathit{instr} & ::= & \mathrm{Nop} \mid \mathrm{Ipush}\ i \mid \mathrm{Pop} \mid \mathrm{Dup} \mid \mathrm{Ineg} \mid \mathrm{Iadd} \mid \mathrm{Isub} \mid \mathrm{Imult} \\
& \mid & \mathrm{Iload}\ x \mid \mathrm{Istore}\ x \mid \mathrm{Iinc}\ x\ n \mid \mathrm{Newarray} \mid \mathrm{Arraylength} \\
& \mid & \mathrm{Goto}\ \mathit{pc} \mid \mathrm{If_icmp}\mathit{cond}\ \mathit{pc} \quad \mathit{cond} \in \{\mathrm{eq,ne,lt,le,gt,ge}\} \\
& \mid & \mathrm{Iinput} \mid \mathrm{Ireturn} \mid \mathrm{Return}
\end{array}
$$

Interval analysis uses the set Intvl of intervals over $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, \infty\}$ to approximate integer values. The other kind of values are the references to arrays. We abstract arrays by their size which is also represented by an interval. The abstract domains for the analysis are defined as follows:

$$
\begin{array}{rcl}
\mathrm{Intvl} & = & \{\ [a,b] \mid a \in \overline{\mathbb{Z}},\ b \in \overline{\mathbb{Z}},\ a \leq b\ \} \\
\mathrm{Num}^\sharp & = & \mathrm{Ref}^\sharp\ =\ \mathrm{Intvl}_\bot \\
\mathrm{Val}^\sharp & = & \left(\mathrm{Num}^\sharp + \mathrm{Ref}^\sharp\right)_\bot^\top \\
\mathrm{Exp}[\mathrm{Val}^\sharp] & = & \mathrm{const}\ n \mid \mathrm{var}\ x \mid \mathrm{absval}\ v^\sharp \mid \mathrm{unop\ op}\ e \mid \mathrm{binop\ op}\ e_1\ e_2 \\
\mathrm{Stack}^\sharp & = & (\mathrm{Exp}^*)_\bot^\top \\
\mathrm{LocVar}^\sharp & = & \mathrm{Var} \to \mathrm{Val}^\sharp \\
\mathrm{State}^\sharp & = & \mathrm{Ctrl} \to \left(\mathrm{Stack}^\sharp \times \mathrm{LocVar}^\sharp\right)
\end{array}
$$

For each abstract domain defined above we build the corresponding Coq lattice structure by simply combining lattice functors. We use here the lattice library proposed in [17].

The novelty [2] of this analysis is the use of an abstract domain $\mathrm{Exp}[\mathrm{Val}^\sharp]$ of syntactic expressions over the base abstract domain $\mathrm{Val}^\sharp$ of abstract values. An

[2] A similar technique was recently proposed by Antoine Miné [12] for improving the precision of interval analysis using symbolic evaluation in a context of structured languages. Such techniques become essential for low level languages which do not directly manipulate expressions.

example of such an abstract element is binop + (var j) (const 42) which when evaluated will result in the interval obtained by applying interval arithmetic to the interval associated with local variable j and the constant 42. The order imposed on $\text{Exp}[\text{Val}^\sharp]$ is the order of the underlying lattice extended to expressions by stipulating that two expressions are in the order relation if they have the same term structure and if abstract values at a given place in the term are related. The exact definition can be found in [4].

$$\frac{\text{instrAt}_{\text{P}}(p_1, \text{Ipush } n, p_2) \quad m^\sharp_{p_1} = (s^\sharp_{p_1}, l^\sharp_{p_1})}{m^\sharp_{p_2} \sqsupseteq \left((\text{const } n) :: s^\sharp_{p_1}, l^\sharp_{p_1}\right)}$$

$$\frac{\text{instrAt}_{\text{P}}(p_1, \text{If_icmplt } p, p_2) \quad m^\sharp_{p_1} = (e_2 :: e_1 :: s^\sharp_{p_1}, l^\sharp_{p_1})}{m^\sharp_{p} \sqsupseteq \left(s^\sharp_{p_1}, [\![e_1 < e_2]\!]^\sharp_{\text{test}}(l^\sharp_{p_1})\right)}$$

$$\frac{\text{instrAt}_{\text{P}}(p_1, \text{If_icmplt } p, p_2) \quad m^\sharp_{p_1} = (e_2 :: e_1 :: s^\sharp_{p_1}, l^\sharp_{p_1})}{m^\sharp_{p_2} \sqsupseteq \left(s^\sharp_{p_1}, [\![e_1 \geq e_2]\!]^\sharp_{\text{test}}(l^\sharp_{p_1})\right)}$$

Fig. 4. Constraint generation rules (examples)

Several constraint generation rules are presented in Figure 4. Among these, constraints which model test-and-jump instructions are of particular interest because they make use of the notion of *backward abstract interpretation of expressions* [8]. It allows to restrict the destination state of the jump according to the information obtained by the test. When a guard of the form $e_1 \mathfrak{c} e_2$ is verified (with $\mathfrak{c}$ a comparison operator and e_1 and e_2 some expression), the current abstract environment $l^\sharp$ is refined by $[\![e_1 \mathfrak{c} e_2]\!]_{\text{test}}(l^\sharp)$. The operator $[\![\cdot]\!]^\sharp_{\text{test}} \in \text{LocVar}^\sharp \to \text{LocVar}^\sharp$ over-approximates the set of environments (l, h) which fulfill the guard $e_1 \mathfrak{c} e_2$.

Using the abstract domain $\text{Exp}[\text{Val}^\sharp]$ of syntactic expressions over lattice $\text{Val}^\sharp$ has a significant impact on the precision of the analysis (and hence on the certificates that can be generated) because it allows to preserve information obtained through the evaluation of conditional expressions. At source level, a test such as `j+i>3` provides information about the possible values of `i` and `j` that can be exploited in the branches of a conditional statement. At byte code level, this link between variables `i` and `j` is lost (even when these corresponds to local variables in the byte code) because these values have to be pushed onto the stack before they can be compared. Using lattice expressions to abstract stack content enables the analysis to keep information such as that a value is the sum of two variables. For details of the backwards analysis and the lattice of lattice expressions, see [4].

4 Certificates for Static Analyses

A valid certificate is any object from which a checker can construct a proof that *there exists an abstract state* $s^\sharp \in \text{State}^\sharp$ *which satisfies all the constraints and passes all the abstract security tests.* Given such an abstract state $s^\sharp$, the simplest certificate is the abstract state itself. The task of the checker is then reduced to verifying that $s^\sharp$ satisfies all the constraints and passes all abstract security tests. For a fixed number of program variables, such certificates are linear in the number of program locations and can thus be checked in linear time.

Both the running time and space consumption of this naive checking can be improved considerably, notably by a better handling of basic blocks. For k bytecodes, the certificate would contain k abstract memories, the checking of which requires the computation of k transfer functions and k tests with the $\sqsubseteq^\sharp$ order relation. For basic blocks, our optimized algorithm only requires a single abstract memory (updated in-place) and saves the $\sqsubseteq$ tests. To achieve this, the core of our optimised checker is not a post-fixpoint checker but a *post-fixpoint reconstruction* algorithm which interprets certificates that encode *post-fixpoint reconstruction strategies* consisting of lists of commands. Upon success, the reconstruction returns a *tagged* abstract state which witnesses the program safety. The Coq datatypes for strategy commands and tags are:

```
Inductive TagMem : Set :=
  | Undef
  | Hint (m:AbMem.t)
  | Checked (m:AbMem.t)
  | Done.
```

```
Inductive command : Set :=
| Assign (c:Ctrl) (m:AbMem.t)
| Eval (c:Ctrl)
| Drop (c:Ctrl).
```

The reconstruction starts from the undefined abstract state. It updates tags and triggers local verification conditions according to the current command:

- The command `Assign(pc,mem)` proposes an (untrusted) abstract memory `mem` for control point `pc`. If `pc` is tagged `Undef` and `mem` verifies the abstract security checks, `pc` is tagged by `Hint(mem)`.
- `Eval pc` commands to set the tag of `pc`. To do so, one computes the (least) abstract memory `mem` which verifies the constraints on control point `pc`.
  - · If `pc` is tagged `Undef` and `mem` verifies the security tests, `pc` is updated with the tag `Checked(mem)`.
  - · If `pc` is tagged `Hint mem'`, and `mem` $\sqsubseteq$ `mem'` then `pc` is updated with the tag `Checked(mem')`.
- The command `Drop(pc)` marks `pc` with a `Done` tag as long as `pc` was tagged `Checked` before. This allows to discard (garbage collect) abstract memories not needed for the rest of the computation.

In any other case, the reconstruction fails. This algorithm is coded as a Coq function `reconstruct` which takes as arguments a set of constraints, a

set of safety checks and a certificate, and returns a tagged abstract state if the reconstruction succeeds. Otherwise, it fails. Because the reconstruction algorithm is analysis independent, certificate checkers can be constructed in a generic fashion from any certified static analysis. This is expressed as a functor

```
Module AIChecker (CA:CertifiedAnalysis) : Checker.
 ...
 Definition certificate := list command.
 Definition checker (p:program) (cert:certificate):bool:=
  reconstruct (CA.gen_cstr p) (CA.gen_AbSafe p) cert ≠ Fail.
  ...
End AIChecker.
```

which takes as argument a `CertifiedAnalysis` (*cf.* Figure 3) and returns a `Checker`, the interface of which was defined in Figure 2.

The success of our scheme depends on how easy it is to automate the generation of certificates. In our setting, we use a standard *chaotic (post)-fixpoint iterator* back-end of abstract interpretation based analysers to produce a post-fixpoint. To obtain a certificate from a post-fixpoint, a *reconstruction strategy* has to be devised. Our present strategy takes advantage of the graph structure of the control flow graph:

- Sequential graphs corresponding to basic blocks allow a straightforward strategy which works in constant memory and alternates an `Eval` command and a `Drop` command of the predecessor control point.

$$\mathrm{Assign}(p_0, m_0); \mathrm{Eval}(p_1); \mathrm{Drop}(p_0) \ldots \mathrm{Eval}(p_n); \mathrm{Drop}(p_{n-1})$$

- For a directed acyclic graph (DAG), a strategy based on topological traversal of the graph does not require any `Assign` command. It is possible to further optimise this strategy by picking a traversal that allows to insert `Drop` commands as early as possible, improving the memory usage of the checker.
- Reducible graphs allow a strategy that minimises `Assign` commands by putting them at loop-headers. Given these loop-headers, the rest of the graph can be decomposed into DAGs for which the DAG strategy applies.

The language of strategy commands can encode the lightweight byte code verifier of Rose [18]. In her algorithm, the evaluation order of the control points is hard-coded so that control points are evaluated in increasing order and values of the fixpoint are explicitly given for all loop-headers. The byte code comes equipped with the value of the fixpoint at all program points that are the target of a back-edge in the control graph. For an applet with n control points of which $q_1, \ldots, q_k$ are targets of back-edges, the lightweight byte code verification strategy can be expressed in our strategy language as

$$\mathrm{Assign}_{q_1}; \ldots; \mathrm{Assign}_{q_k}; \mathrm{Eval}_1; \ldots; \mathrm{Eval}_n.$$

5 Implementation and experiments

In this section, we give some details of how the PCC architecture has been programmed in Coq and Caml together with some benchmarks for certificate generation done with this implementation. On the code producer side, nothing needs to be certified since the only obligation is to produce a certificate for the given program. It is nevertheless natural to share the constraint generation function with the certified checker. To get a working byte code analyser, it suffices then to solve the constraints with any (un-certified) highly optimised iterative fixpoint solver.

On the consumer size, the constraint generation function and the fixpoint reconstruction algorithm belong to a module of type `Checker` as defined in Section 2. It is in principle possible to execute the Coq definitions of the checker within Coq but it would be rather slow. Instead, we use the Coq program extraction mechanism to extract Caml functions from its Coq definition. This yields a Caml `checker` function to be applied to a program `p` and a certificate `cert`. The use of the Coq extraction mechanism requires a little care since the Coq extraction of functions is correct only if the extracted function is evaluated on arguments that are well-typed in Coq (see [11] for a formal statement). To avoid this pitfall, a certificate is coded up as a simple bit-stream which is then parsed into a Coq structure by a Coq-type-checked parser (details omitted).

We have tested our PCC framework on a number of array-manipulating algorithms to check how the certificate generation behaves on byte codes generated by compilation of Java source programs. The whole Coq development, including a working checker, is available for download[3]. The test programs have been chosen because they are all array manipulation-intensive and hence require precise certificates in order to show that they respect the security policy. We have generated and checked certificates for three classical sorting algorithms (bubble sort, heap sort and quick sort), the Floyd-Warshall algorithm for shortest past computation, and algorithms for polynomial product and vector convolution. For each algorithm, the improved interval analysis described in Section 3.3 is sufficiently precise to be able to verify that all array accesses are safe. Figure 5 presents some measurements pertinent to the certification: the size of the source and byte code, the size of the certificates, the time for checking the certificates and the ratio between the number of constraints that an analyser had to evaluate to construct the certificate and the number of constraints that the checker had to evaluate. It should be stressed that the analyser used to construct the certificates uses efficient iteration algorithms based on widening and narrowing operators to accelerate convergence. Appendix A give details of the analysis of bubble sort.

Two things are worth noting here. First, the size of the certificates is much (sometimes an order of magnitude) smaller than the code it certifies. Second,

[3] `http://www.irisa.fr/lande/pichardie/PCC/`

Program	.java (bytes)	.class (bytes)	certificate (bytes)	checker (sec.)	analyser/ checker
BubbleSort	440	528	32	0.015	440/110
HeapSort	1044	858	63	0.050	8001/381
QuickSort	1078	965	124	0.060	8910/405
Convolution	378	542	52	0.010	460/92
FloydWarshall	417	596	134	0.020	23114/163
PolynomProduct	509	604	87	0.010	150669/133

Fig. 5. Experiments on various algorithms

the ratio between the number of evaluations of constraints used by the analyser by far exceeds the number of evaluations used by the checker to verify the certificate—sometimes by order of magnitudes. The six programs are moderate in size but are sufficiently complex to show that the PCC infrastructure can be used to generate compact, non-trivial program certificates which can be checked more efficiently than they can be produced.

6 Related Work

The *VeryPCC* project conducted by Nipkow *et al.* aims at providing a foundational PCC framework verified within the Isabelle/HOL theorem prover [20]. The core of the framework is a generic VCGen based on a weakest precondition calculus proved correct with respect to the operational semantics of a Java-like bytecode language. An important difference from our work is that the VCGen works on programs annotated with loop invariants that have to be re-proved in Isabelle. In this scheme, proof terms are Isabelle proof scripts that have to be rerun. Tactics can boil down to proof search so the complexity of the proof checking can be high. In contrast, by using an abstract interpretation certified within Coq, the analyser directly produces a proof (namely, a post-fixpoint) that can be communicated and verified efficiently by the proof checker.

The *Mobile Resource Guarantee* (MRG) project has also designed an Isabelle based PCC infrastructure. It aims at proving properties which guarantee that the resource consumption (*e.g,* space and time) of a code does not exceed a given bound. To do that, they have proved inside Isabelle the soundness of a powerful resource-aware type system. Once again, certificates are Isabelle scripts. This work shares with ours the idea of installing a dedicated proof checker that comes with its own correctness proof which can be verified with respect to an operational semantics. The approach described in [14] does not propose a methodology for producing such proofs however, whereas we

are able to propose a methodology based on certified abstract interpretation. The actual certificate checking in [14] is done within Isabelle using dedicated proof techniques and is not efficient. Our use of post-fixpoints and their formalization in constructive logic allowed to obtain a proof checker that is both certifiable and efficient.

The *Open Verifier Framework* [6] is a proposal to strengthen the trust in the infrastructure without sacrificing efficiency. The soundness depends on a core (trusted) condition generator. For flexibility, specialised (untrusted) condition generators can be downloaded and imported to enrich the platform. The core is generating *strongest postconditions*; specialised components generate a *weakening* together with a machine-checkable proof that it is correct. Our proof checkers can be understood as compiled specialised components. Because their soundness is established once for all, it has the advantage that these trusted components execute without any runtime penalty.

Albert, Hermenegildo and Puebla have proposed to use abstract interpretation for automatically producing analysis-carrying code. In [1] they develop a PCC framework for constraint logic programs in which a CLP abstract interpreter calculates a program invariant (a fixpoint) that is sufficient to imply a given security policy. The certificate is a fixpoint that is checked by a single step of the abstract interpreter. The most notable difference with respect to our approach is that their analysis is not certified and hence must be trusted by the code consumer. Our analysers are not part of the trusted computing base because our FPCC architecture requires that the code producer transmits proofs of their correctness.

Lightweight Bytecode Verification developed by Rose [18] includes a compression scheme for stack maps (that corresponds to our certificates). Their stack map compression allows to evaluate certificates on the fly as constraint generation proceeds. This leads to a hard-coded evaluation order that might not be optimal. Our strategy-based algorithm is more flexible and accommodates strategies that in certain cases are more efficient.

7 Conclusions and Further Work

We have developed a foundational PCC architecture based on certified static analysis. Compared to other PCC proposals, this approach allows to employ static analyses as certificate generators in a seamless and automatic manner, without having to re-prove proposed invariants inside a given theorem prover. The strong semantic foundations of the theory of abstract interpretation and its recent formalisation inside the Coq proof assistant enables the construction of a certified proof checker from the certified static analyses. Such certified proof checkers can then be installed dynamically by a code consumer who can check the validity of the checker by type checking it in Coq.

Instead of sending explicit representations of certificates with a mobile code, we encode certificates as *strategies* that the code consumer executes in

order to reconstruct a suitable post-fixpoint that will imply the given security policy. Such strategies are generated from certificates and can be further tuned to minimise memory consumption of the checker. Indeed, proof checkers only need to verify the *existence* of a suitable post-fixpoint, without having to re-create it in its entirety. This takes advantage of the garbage-collecting strategies that we have defined.

The architecture has been implemented and tested with a certified interval analysis of array-manipulating byte code in order to generate certificates attesting that a given code will not attempt to access an array outside its bounds. The interval analysis uses a novel kind of abstract domains in which *lattice expressions* are mixed with abstract values. This symbolic representation allows to keep track of the expression used to compute a particular abstract value—an information which is otherwise lost when compiling from high-level languages to byte code. The lattice expressions add just enough relational information to the otherwise non-relational interval analysis to deal properly with the propagation of the information obtained from conditional instructions. This analysis technique (which was already briefly sketched in [19]) should be of general interest to the analysis of low-level code. Further along, it would be interesting to develop a certified, truly relational analysis based *e.g.*, on octagons [13].

References

[1] Elvira Albert, German Puebla, and Manuel Hermenegildo. Abstraction-carrying code. In *Proc. of 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, Springer LNAI vol. 3452, pages 380–397, 2004.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Proc. of 16th IEEE Symp. on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[3] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. of the 27th ACM Symp. on Principles of Programming languages (POPL'00)*, pages 243–253. ACM, 2000.

[4] Frédéric Besson, Thomas Jensen, and David Pichardie. A PCC architecture based on abstract interpretation. Technical Report RR–5751, INRIA, Nov. 2005.

[5] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *Proc. of 13th European Symp. on Programming (ESOP'04)*, pages 385–400. Springer LNCS vol. 2986, 2004.

[6] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The open verifier framework for foundational verifiers. In Greg Morrisett and Manuel Fähndrich, editors, *Proc. of 2nd International Workshop on Types in Languages Design and Implementation (TLDI'05)*. ACM, 2005.

[7] The Coq Proof Assistant. http://coq.inria.fr/.

[8] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[10] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.

[11] Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[12] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains.
In *VMCAI'06*, volume 3855 of *LNCS*, pages 348–363. Springer, 2002. http://www.di.ens.fr/~mine/publi/article-mine-VMCAI06.pdf.

[13] Antoine Miné. The octogon abstract domain. In *Proc. of the 8th Working Conference On Reverse Engineering (WCRE 01)*, pages 310–320. IEEE, 2001.

[14] Alberto Momigliano and Lennart Beringer. Certification of resource consumption: from types to logic programming. *Assoc. for Logic Programming Newsletter*, 18(2), May 2005.

[15] George C. Necula. Proof-carrying code. In *Proc. of 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM, 1997.

[16] George C. Necula and Robert R. Schneck. A sound framework for untrusted verification-condition generators. In *Proc. of 18th IEEE Symp. on Logic In Computer Science (LICS 2003)*, pages 248–260, 2003.

[17] David Pichardie. *Interprétation abstraite en logique intuitioniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université de Rennes 1, Sept. 2005.

[18] Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

[19] Martin Wildmoser, Amine Chaieb, and Tobias Nipkow. Bytecode analysis for proof carrying code. In *Proc. of 1st Workshop on Bytecode Semantics, Verification and Transformation, Electronic Notes in Computer Science*, 2005.

[20] Martin Wildmoser and Tobias Nipkow. Asserting bytecode safety. In *Proc. of the 15th European Symp. on Programming (ESOP'05)*, 2005.

A A full example: bubble sort

Below is listed the source code of the program `BubbleSort`. The result of the analysis run on the corresponding byte code program is given in comments.

The symbol U represents an initialised value and topV the top of abstract values. The compiled version of BubbleSort.java is 69 bytecodes long. From 70 abstract memories computed, the post-fixpoint compressor keeps only 3 stackmaps. The stackmaps are automatically chosen at the heads of the for loops (lines 3, 7 and 8) which correspond to back-edge in the byte code program. 256 bits are then necessary to encode these stackmaps into a bit stream format. Two iteration strategies are able to compute a correct post-fixpoint which is sufficiently precise to prove the safety of BubbleSort. The first strategy is the classic iteration from bottom, without acceleration. If this computation terminates it computes the least post-fixed point of the constraint system. This iteration strategy is not ensured to terminate and requires about n^2 iterations (of each constraints) to terminate in the example of BubbleSort, with n the (static) size of the array t. The second strategy uses chaotic iteration with widening and narrowing. It only needs 3 iterations of each constraints to reach a safe post-fixpoint.

```
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: [0,20] ; t: int[20  ,20] ; tmp: U]
 3:     for (i=0;i<n;i++) {
  // [j: U ; n: [20,20] ; i: U ; 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: topV ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: topV ]
 8:   for (j=0; j<n-1-i; j++)
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: topV ]
 9:   if (t[j+1] < t[j]) {
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: topV ]
10:   tmp = t[j];
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: topV ]
11:   t[j] = t[j+1];
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: topV ]
12:   t[j+1] = tmp;
  // [j: [0,18] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: topV ]
13:   }
```

```
  // [j: [1,19] ; n: [20,20] ; i: [0,18] ; t: int[20,20] ; tmp: topV ]
14: };
  // [j: topV  ; n: [20,20] ; i: [19,19] ; t: int[20,20] ; tmp: topV ]
15: Tab.print_tab(t);
  // [j: topV  ; n: [20,20] ; i: [19,19] ; t: int[20,20] ; tmp: topV ]
}}
```