Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression

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 safety 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. Certificates take the form of strategies for reconstructing a fixpoint and are kept small due to a technique for fixpoint compression. The PCC architecture has been implemented and 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 [20] 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" [23].

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 [27] 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 [1]. Untrusted provers are external static analysers that suggest potential (inductive) invariants. These invariants are then reproved inside Isabelle to obtain a transmittable program certificate.

Abstract interpretation is another technique for proving invariants of programs and 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 requires the code consumer to take for granted the semantic correctness of the abstract interpretation. It is thus prone to the same objections as those made against the initial PCC framework where the code consumer had to trust the correctness of the verification condition generator. In this paper we show how to improve on this situation by developing a foundational PCC architecture based on *certified abstract interpretation* [7] 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. This proof object can then be communicated to the code consumer for verification. We describe how this leads to an infrastructure that allows to download specialised 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 [2 ,26]. In Sections and 6 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 syntactic expressions (Section 4). 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.

2 A PCC architecture with certified proof checkers

In the following, we propose an extensible PCC architecture based on abstract interpretation which 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 [9]. 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 general-purpose proof checker and flexible *specialised* proof checkers. Once the proof checker has been installed, the consumer is ready to download the program of the code producer. 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. This certificate can be obtained using optimised, un-trusted fixpoint solvers and compressors since it will be checked upon reception.

We use the program extraction mechanism of Coq to extract efficient Caml checkers from their Coq specification. Extraction is using the proofs-as-programs paradigm to erase those parts of a proof term that only concerns the proof of properties and which do not contribute to the specified computation. A formal account of Coq current extraction can be found in Letouzey's thesis [17]. It would, in principle, have been possible to execute the Coq specification directly since the Coq proof-checker implements strong normalisation of lambda terms. However, this mechanism is at the moment not efficient enough to make such an approach viable (recent progress in the implementation of strong reduction [13] may change this in the future).

The producer and consumer have to formalise 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

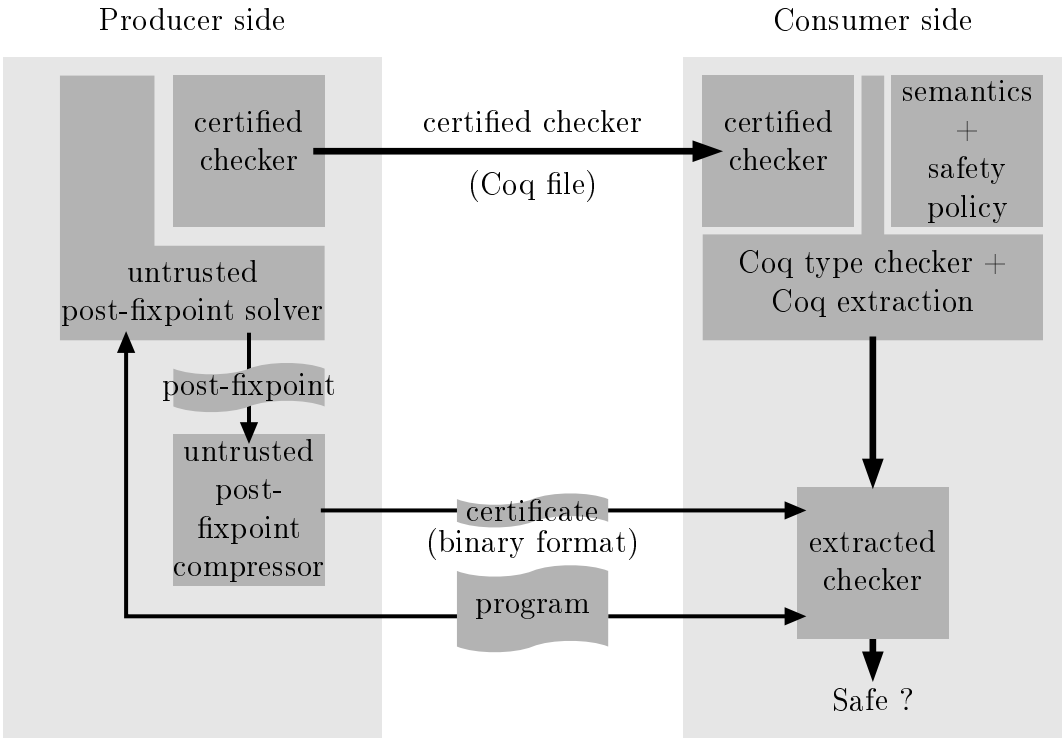

Fig. 1. PCC architecture

definition of the security policy. 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 : $\mathcal{S}_0 \subseteq State$
- for each program, an operational semantics $\rightarrow_p \subseteq State \times State$,
- for each program, a set $Safe_p$ of states that respect the security policy.

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 $\mathcal{S}_0$.

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

Definition 2.1 *A program p is* safe *if $[\![p]\!] \subseteq Safe_p$,* i.e.*, if all its reachable states are safe.*

Together with the Coq proof-checker and extraction mechanism, the specification of the semantics and safety property form the Trusted Computing Base (TCB) of the PCC architecture. In Figure 1, these trusted components are

located in the upper-right corner of the consumer side. Other components of the consumer are not part of the TCB:

- Downloaded checkers (upper left corner) are only trusted if they type-check;
- Extracted checkers inherit trust from type-checked checkers.

Program extraction excluded, the TCB of Figure 1 is exactly the TCB of

analysis. Sections –6 describe how certificate checkers can be built from certified static analyses. We first present an unoptimised checker and then develop techniques for *fixpoint compression* that allow to obtain compact program certificates. In Section 7 we then describe the implementation of a Coq functor `IAChecker` which constructs a module of type `Checker` from any certified analysis.

3 A Coq signature of certified static analyses

The notion of certified analysis is based on previous work on programming a static analyser in Coq [7,24]. We recall the main components of such a formalisation and explain how they are used for proof-carrying code.

3.1 Certified abstract interpretation for PCC

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

$$\forall p \in Pgm,\ analyse(p) = true \Rightarrow [\![p]\!] \subseteq Safe_p$$

The analyser and its Coq correctness proof are built in 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.

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

$$\gamma : \left(State\ , \sqsubseteq\ , \sqcup\ , \sqcap\ , \right) \to (\mathcal{P}(State), \subseteq, \cup, \cap) \tag{1}$$

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

(2) An abstract semantics is then specified as any post-fixpoint of a well-chosen abstract function $F_p \in State \to State$. The correctness of this

[1] Because we only focus on soundness of the abstract interpreters, the classic notion of Galois connection [11] is not mandatory here. Instead we require γ to be a meet morphism, *i.e.* $\gamma(s_1 \sqcap\ s_2) = \gamma(s_1) \cap \gamma(s_2)$. This is equivalent to the existence of the corresponding Galois connection when $\left(State\ , \sqsubseteq\ , \sqcup\ , \sqcap\ , \right)$ is complete and $\sqcap$ denotes the general greatest lower bound (on sets instead of two values).

specification must be proved by establishing that all post-fixpoint are correct approximations of the concrete semantics.

$$\forall p \in Pgm,\ \forall s\ \in State\ ,\ F_p(s\ ) \sqsubseteq\ s\ \ \Rightarrow\ [\![p]\!] \subseteq \gamma(s\ ) \tag{2}$$

(3) A post-fixpoint solver $solve \in Pgm \rightarrow State$ is then defined, based on fixpoint iteration techniques.

$$\forall p \in Pgm,\ F_p(solve(p)) \sqsubseteq\ solve(p) \tag{3}$$

(4) An abstract safety test $Safe_p \in State\ \rightarrow bool$ is defined in the abstract world.

$$\forall p \in Pgm, \forall s\ \in State\ , Safe_p(s\ ) = true \Rightarrow \gamma(s\ ) \subseteq Safe_p \tag{4}$$

Together, these proofs assert that $Safe\ \circ solve$ is a correct analyser.

Step (3) 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 (2) and (4) we have:

Observation 3.1

$$\forall p \in Pgm, (\exists s\ \in State\ ,\ F_p(s\ ) \sqsubseteq\ s\ \wedge Safe_p(s\ ) = true) \Rightarrow [\![p]\!] \subseteq Safe_p$$

In particular, this means that for a proposed certificate $s\ \in State$, our PCC checker only has to test $F_p(s\ ) \sqsubseteq\ s\ \ \wedge\ Safe_p(s\ ) = true$.

3.2 Certified analysis for memory invariants

We now present the Coq definition of certified analyses for languages where the semantic domain is expressed as a set of reachable states, composed of a control point and a memory $State = Ctrl \times Mem$. The abstract domain $State\ = Ctrl \rightarrow Mem$ attaches memory invariants to each control point of a program. We will later show how to compress such abstract states into more compact program certificates that only provide invariants at certain, well-chosen control points. The certified analysis interface is presented in Figure 3.

The first element of this signature is the lattice structure `AbMem` which is the Coq counterpart of *Mem* . 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

```
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:Constraint) (st : AbState) :=
   AbMem.order (expr C (map st (sources C))) (st (target C)).

 Parameter gen_cstr: program → list Cstr.

 Definition Approx (P:program) (St : AbState) :=
   ∀ c, c ∈ (gen_cstr P) → Verif_cstr c St.

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

 Definition Secure (P:program) (St : AbState) :=
   ∀ p check, (p,check) ∈ (gen_AbSafe P) →
check (St p) = true.

 Parameter analysis_correct : ∀ P st,
   Approx P st → Secure P st → ⟦P⟧ ⊆(Safe P).
End CertifiedAnalysis.
```

Fig. 3. The Coq signature of a certified static analysis

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 [24]. Most of the proofs follow standard lattice theory.

The abstract function F_p previously presented now operates on the domain $\Big(Ctrl \to Mem\Big) \to \Big(Ctrl \to Mem\Big)$. Because the number of control points of a program is finite, say n, post-fixpoints of F_p can be represented as n-tuples $(s_1, \ldots, s_n) \in \Big(Mem\Big)^n$ solutions to systems of constraints where a given constraint has the general form $\Big(f(s_{cp_1}, \ldots, s_{cp_k}) \sqsubseteq s_{cp}\Big)$. In the Coq signature, the abstract function F_p is modelled by a list of such constraints generated by the function `gen_cstr`. A constraint $\Big(f(s_{cp_1}, \ldots, s_{cp_k}) \sqsubseteq s_{cp}\Big)$ is represented by a record `Constraint` with three fields: `target` contains the

control point *cp* 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. The predicate `Verif_cstr` defines what it means for an abstract state to satisfy a constraint. Finally, the predicate `Approx` holds when an abstract state verifies all the constraints generated from the program.

Because abstract states in *State* are of the form $Ctrl \rightarrow Mem$, we can split the abstract safety test $Safe_p$ into several local tests of the form

$$(cp, check) \in Ctrl \times (Mem \rightarrow bool).$$

Each test is attached to a specific control point *cp* and ensures that no error state can be reached by a one-step transition out of the state at control point *cp*. For example, a safety test of array bounds checks would check the value of the index before each array access instruction of a 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 Observation 3.1 to our specific abstract domain of states. If an abstract state s verifies all the constraints generated by `gen_constr` (*i.e.* is a post-fixpoint of F_p) and fulfills all safety checks generated by `gen_AbSafe` (*i.e.* $Safe_p(s) = true$), then the program is safe.

4 Enhanced interval analysis for byte codes

To demonstrate the working 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 [10] but has been extended with an abstract domain of *syntactic expressions* to obtain a similar precision at byte code level.

4.1 Syntax and semantics

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, access 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.* [7].

$$
\begin{array}{rcl}
pgm & ::= & (pc\ instr\ pc)^* \\
instr & ::= & \text{Nop} \mid \text{Ipush } i \mid \text{Pop} \mid \text{Dup} \mid \text{Ineg} \mid \text{Iadd} \mid \text{Isub} \mid \text{Imult} \\
& \mid & \text{Load } x \mid \text{Store } x \mid \text{Iinc } x\ n \\
& \mid & \text{Newarray} \mid \text{Arraylength} \mid \text{Iaload} \mid \text{Iastore} \\
& \mid & \text{Goto } pc \mid \text{If\_icmp}cond\ pc \quad cond \in \{\text{eq,ne,lt,le,gt,ge}\} \\
& \mid & \text{Iinput} \mid \text{Ireturn} \mid \text{Return}
\end{array}
$$

The byte code language is given an operational semantics which program states have the form $<cp, h, s, l>$ where cp is a control point to be executed next, h is a heap for storing allocated arrays, s is an operand stack, and l is an environment mapping local variables to values. An array is modelled by a pair consisting of the size of the array and a function that for a given index returns the value stored at that index. A special error state *Error* is used to model execution errors which here arise from indexing an array outside its bounds.

$$
\begin{aligned}
Val &= \mathbb{Z} + Location \\
Stack &= Val^* \\
LocVar &= Var \rightarrow Val \\
Array &= (length : \mathbb{Z}) \times ([0, length{-}1] \rightarrow Val) \\
Heap &= Location \rightarrow Array_\perp \\
State &= (Ctrl \times Heap \times Stack \times LocVar) + Error
\end{aligned}
$$

The operational semantics is defined via a transition relation $\rightarrow$ between states in a standard fashion and will not be explained in detail. Representative rules of the definition of $\rightarrow$ are shown in Figure 4. They illustrate different aspects of the byte code language; in particular how array bound checks are performed when accessing an array.

With the introduction of a specific error state, the set of safe states can simply be defined as all states except the *Error* state.

$$Safe_p = \{s \mid s \neq Error\}$$

.

$$\frac{instrAt_P(p_1, Ipush\ n, p_2)}{<p_1, h, s, l> \rightarrow_P <p_2, h, n :: s, l>} \qquad \frac{instrAt_P(p_1, Load\ x, p_2) \quad l(x) = n}{<p_1, h, s, l> \rightarrow_P <p_2, h, n :: s, l>}$$

$$\frac{instrAt_P(p_1, If\_icmplt\ p, p_2) \quad n_1 < n_2}{<p_1, h, n_2 :: n_1 :: s, l> \rightarrow_P <p, h, s, l>}$$

$$\frac{instrAt_P(p_1, Iaload, p_2) \quad h(ref) = a \quad 0 \leq i < a.length}{<p_1, h, i :: ref) :: s, l> \rightarrow_P <p_2, h, a[i] :: s, l>}$$

$$\frac{instrAt_P(p_1, Iaload, p_2) \quad h(ref) = a \quad \neg\ 0 \leq i < a.length}{<p_1, h, i) :: ref :: s, l> \rightarrow_P Error}$$

Fig. 4. Operational semantics (selected rules)

4.2 Interval analysis

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:

$$\begin{aligned}
Intvl &= \left\{ [a, b] \mid a \in \overline{\mathbb{Z}},\ b \in \overline{\mathbb{Z}}, a = -\infty \vee a \leq b \vee b = +\infty \right\} \\
Num &= Array = Intvl_\perp \\
Val &= \left(Num + Array \right)^\top_\perp \\
Stack &= (Exp^*)^\top_\perp \\
LocVar &= Var \rightarrow Val \\
State &= Ctrl \rightarrow \left(Stack \times LocVar \right)
\end{aligned}$$

The domain of syntactic expression $Exp[\,Val\,]$ is inductively defined by the following rules:

$$\frac{n \in \mathbb{Z}}{const\ n \in Exp[\,Val\,]} \qquad \frac{x \in Var}{var\ x \in Exp[\,Val\,]} \qquad \frac{v \in Val}{absval\ v \in Exp[\,Val\,]}$$

$$\frac{e \in Exp[\,Val\,]}{-e \in Exp[\,Val\,]} \qquad \frac{e_1 \in Exp[\,Val\,], e_2 \in Exp[\,Val\,], op \in \{+, -, *\}}{binop\ op\ e_1\ e_2 \in Exp[\,Val\,]}$$

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 [24].

The novelty of this analysis is the use of an abstract domain $Exp[\,Val\,]$ of syntactic expressions over the base abstract domain Val of abstract values. An 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 $Exp[\,Val\,]$ 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 all corresponding places in the term are related. The exact definition can be found in [6].

$$\frac{instrAt_P(p_1, Ipush\ n, p_2) \quad m_{p_1} = (s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \Big((const\ n) :: s_{p_1}, l_{p_1}\Big)}$$

$$\frac{instrAt_P(p_1, If\_icmplt\ p, p_2) \quad m_{p_1} = (e_2 :: e_1 :: s_{p_1}, l_{p_1})}{m_p \sqsupseteq \Big(s_{p_1}, [\![e_1 < e_2]\!]_{test}(l_{p_1})\Big)}$$

$$\frac{instrAt_P(p_1, If\_icmplt\ p, p_2) \quad m_{p_1} = (e_2 :: e_1 :: s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \Big(s_{p_1}, [\![e_1 \geq e_2]\!]_{test}(l_{p_1})\Big)}$$

Fig. 5. Constraint generation rules (examples)

Several constraint generation rules are presented in Figure (see Appendix A for the comprehensive set of constraints). 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* [10]. 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 is refined by $[\![e_1\ \mathfrak{c}\ e_2]\!]_{test}(l\ )$. The operator $[\![\cdot]\!]_{test} \in LocVar\ \rightarrow LocVar$ over-approximates the set of environments (l, h) which fulfill the guard $e_1\ \mathfrak{c}\ e_2$.

Using the abstract domain $Exp[\,Val\,]$ of syntactic expressions over lattice Val 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 syntactic expressions to abstract stack content enables the analysis to keep information such as that a value is the sum of two variables. Figure 6 provides an ex-

ample of the precision so obtained. The figure contains a Java code snippet and its compiled version in byte code, annotated with an interval certificate. Before executing the instruction `11 : if_icmple 16`, the stack contains the abstract elements $(binop\ +\ (var\ j)\ (var\ i))$ and $(const\ 3)$. Since i is the singleton interval `[100,100]` the analysis can deduce that at the following instruction, `j` is necessarily bigger than or equal to `-96`.

Source example

```
int i = 100;
int j = Input.read_int();
if (j+i>3) { .. }
```

Analysed byte code version

```
...
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <>
 7 : iload j
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <(varj)>
 8 : iload i
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <(var j)::(var i)>
 9 : iadd
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <(binop + (var j) (var i))>
10 : ipush 3
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <(binop + (var j) (var i))::(const 3)>
11 : if_icmple 16
    // [j↦[-96,+oo] ; i↦[100,100]]
    // <>
...
```

Fig. 6. Analysis example

Figure 7 presents an example of an abstract safety test for the interval analysis. Such a test is done at each control point where an instruction *Iaload* is found. We verify that the abstract operand stack has at least 2 elements. The first in an interval abstracting the index where an array element must be found and the second the interval approximating the length of the array. We check that index is positive and smaller than the length of the array. Here, $[\![e]\!]_{expr}(l\ )$ denotes the interval resulting from the evaluation of expression e in the abstract environments l .

$$
\begin{array}{l}
Check_{Iaload}\ (s\ ,l\ )\ = \\
\quad \textbf{match}\ s\ \ \textbf{with} \\
\quad |\ (e_i :: e_{length} :: q)\ \rightarrow \\
\quad\quad \textbf{let}\ [i_1, i_2] = [\![e_i]\!]_{expr}(l\ )\ \textbf{and}\ [l_1, l_2] = [\![e_{length}]\!]_{expr}(l\ )\ \textbf{in} \\
\quad\quad\quad 0 \leq i_1\ \&\ i_2 < l_1 \\
\quad |\ \_ \rightarrow \textbf{false}
\end{array}
$$

Fig. 7. Safety test for *Iaload* instructions

5 Certificate checkers

The checker component of the PCC architecture is the critical part that has to be provably sound as well as space- and time-efficient. In the following, we describe how to generate checkers and certificates that fulfil these requirements. The certificates attach a piece of information to a subset of the control points. Each such piece of information can be checked by evaluating the constraint associated with the corresponding control point. Hence, certificate size and certificate checking are linear in the size of the program.

The method for constructing certificate checkers is generic and applies to any certified analysis. It is expressed as a functor

```
Module type AIChecker (CertifiedAnalysis) : Checker
```

which takes as argument a `CertifiedAnalysis` (the interface of which was defined in Figure 3) and returns a `Checker` (*cf.* Figure 2). Central to the construction of such a functor is Observation 3.1 which establishes the link between the checker and the certified analysis and which provides a property that has to be proved for any new proof checker. This property defines the notion of a fixpoint checker:

Definition 5.1 (Fixpoint checker) *Given a program P and a certificate cert, a fixpoint checker is a function*

$$checker : Pgm \rightarrow Certificate \rightarrow bool$$

satisfying that if $checker(P, cert) = true$, then there exists an abstract state s which at the same time

- *approximates the program semantics ($F_{\mathsf{P}}(s\ ) \sqsubseteq\ s\ )$ and*
- *respects the safety policy ($Safe_{\mathsf{P}}(s\ ) = true$)*

Observation 3.1, when combined with the property `analysis_correct` from

the signature `CertifiedAnalysis`, yields that success of a fixpoint checker ($checker(P, cert) = true$) implies program safety. This proves the `checker_ok` theorem of the `Checker` interface—see Theorem .6.

The simplest certificate to check is just an abstract state s . The algorithm of the checker for such certificates is simple: check that all the generated verification conditions are satisfied by the proposed s . Such a naive checker can be written as

```
Let checker p s =
 List.for_all (verif_cstr s ) (gen_cstr p) &&
 List.for_all (verif_AbSafe s ) (gen_AbSafe p)
```

This checker is directly executable because we have provided constructive definitions of the abstract domain operations throughout the specification. The algorithm trivially satisfies the Definition .1: if the `checker` returns `true` then the certificate s verifies all the verification conditions (those imposed by the analysis itself and those imposed by the safety policy). In terms of complexity, this naive algorithm fulfils the requirement of having a runtime complexity that is linear in the program size. For each instruction, it checks the constraints imposed by the analysis and the safety requirements. Verifying such a constraint amounts to computing an abstract transfer function and an ordering test $\sqsubseteq$. The size of the certificate is also linear in the program size—each program point stores an element of the abstract domain. This can, however, be improved. In the following we explain how to design checkers that require significantly smaller certificates.

5.1 Strategies for reconstructing certificates

The naive algorithm requires certificates that provide a *complete* solution of the analysis: an abstract memory state is attached to each control point. We now describe a proof checker which (implicitly) recomputes the complete solution from a sparse certificate. The core of this checker is a reconstruction algorithm which takes as input a program and a strategy that is interpreted step by step. Upon success, it returns a *tagged* abstract state from which one can extract (after tag erasure) a correct and safe abstract state.

$$reconstruct : Pgm \rightarrow Strategy \rightarrow option(Ctrl \rightarrow TagMem)$$

The datatypes for strategy commands and tagged memories are given below:

```
Inductive TagMem : Set :=
| Undef
| Hint (mem:AbMem.t)
```

```
| Checked (mem:AbMem.t).

Inductive command : Set :=
| Assign  (cp:Ctrl) (m:AbMem.t)
| Eval (cp:Ctrl).

Definition Strategy := list command.
```

Tags are used to keep track of the reconstruction status of a control point and carry the following intuitive meaning. For a control point `cp`:

- `Undef` means that the abstract memory attached to `cp` has not been reconstructed yet;
- `Hint mem` means that `mem` is proposed as an (untrusted) invariant for `cp`;
- `Checked mem` means that `mem` satisfies the constraint and the safety condition associated with `cp`.

The reconstruction algorithm is essentially an interpreter of strategies which updates an abstract state s and at the same time keeps track of the number `chck` of states that are tagged `Checked`. Reconstruction starts from an undefined tagged abstract state (with `chck` $= 0$) and consumes a strategy command at a time. Each command updates the current tagged abstract state and triggers local verification conditions.

- The command `Assign cp mem` explicitly provides a (presumably) sound abstract memory `mem` for the control point `cp`. If this control point is already set (its tag is different from `Undef`) then the reconstruction fails. Otherwise, if `mem` verifies the local safety policy at control point `cp`, the abstract state is updated ($s :=s$ [`cp` $\rightarrow$ `Hint mem`]).
- The command `Eval cp` computes the least abstract memory `mem` which verifies the constraints imposed by the analysis on control point `cp`. The behaviour changes slightly depending on the tag already attached to `cp`.
  - If it is `Undef` and `mem` verifies the safety condition of control point `cp`, then $s :=s$ [`cp` $\rightarrow$ `Checked mem`] and `chck` is incremented;
  - If it is `Hint mem'` and `mem` $\sqsubseteq$ `mem'`, then $s :=s$ [`cp` $\rightarrow$ `Checked mem'`] and `chck` is incremented.
  - If it is `Checked mem'` then the reconstruction fails.
- If there are no more commands and the number `chck` of checked states equals the number of control points in s , then reconstruction succeeds and returns *Some* s . Otherwise it fails.

Correctness of the reconstruction algorithm amounts to proving that if the reconstruction succeeds, it outputs a tagged abstract state that is a correct approximation of the program and for which all control points satisfy the safety policy. To argue the correctness of the reconstruction, we introduce

the notion of partial correctness of the tagged abstract states at intermediate stages of the computation.

Definition 5.2 *A tagged abstract state s is* partially correct *if every control point cp is tagged as follows:*

- *If a control point is tagged* `Checked mem` *then*
  - `mem` *verifies the local checks on cp imposed by the safety policy;*
  - `mem` *verifies the constraints on cp imposed by the analysis*
- *If a program point is tagged* `Hint mem` *then* `mem` *only verifies the local checks on* `cp` *imposed by the safety policy;*

The soundness proof of the reconstruction algorithm is divided into two parts. Lemma .3 states that the reconstruction algorithm only returns partially correct tagged abstract states.

Lemma 5.3 (Correct Reconstruction) *Given program P and strategy strat, if reconstruct$(P, strat) = Some\ s$ then s is* partially correct

It can be proved (and this has been done in Coq) that each command updates the tagged abstract state such that the invariant is preserved. The result then follows by induction on the strategy length.

Lemma .3 does not ensure that the reconstruction is complete: for example, the totally undefined abstract state is partially correct. However, the counter of checked state and the final check on this counter makes it straightforward to prove the following Lemma .4 which ensures that, at the end of the reconstruction, all control points have a `Checked` tag attached.

Lemma 5.4 (Complete Reconstruction) *Given program P and strategy strat. If reconstruct$(P, strat) = Some\ s$ then*

$$\forall(cp \in P), \exists mem, s\ (cp) = \texttt{Checked}(mem)$$

5.2 Optimisation of the reconstruction algorithm

The strategies presented so far explicitly yield a witness s that satisfies the verification conditions of the analysis. However, according to Observation 3.1 it suffices for the checker to ensure the existence of such a witness—there is no need to reconstruct it. This observation leads to an optimised reconstruction algorithm which exploits this weaker requirement to *drop* on the fly abstract memories that are no longer needed by the verification process. This reduces the memory usage of the reconstruction algorithm by keeping the size of the tagged abstract state as small as possible.

The strategy language is enriched with a `Drop cp` command and the set of memory tags is extended with a `Done` value. A program point can be marked `Done` when it has been checked (*i.e,* its tag is `Checked mem`) and the computed value is no longer needed to evaluate other constraints. In this case, the effect of the `Drop cp` is to set a `Done` tag. As a side effect, the abstract memory `mem` may be garbage-collected. Lemma . formalises the correctness of strategies with `Drop` commands. It says that if a strategy with `Drop` commands succeeds then the same strategy with all `Drop` commands removed will also succeed. Since success of `Drop`-free strategies implies safety, this suffices to ensure the existence of the desired witness.

Lemma 5.5 (Implicit Reconstruction) *For any strategy strat, if the reconstruction succeeds, a reconstruction using the same strategy without* `Drop` *also succeeds.*

The proof relies on the fact that `Done` tags can only be obtained from `Checked` tags. As a result, if the implicit reconstruction drops a control point, there exists a `Checked` tag that would be computed by a strategy that replaces a `Drop` by a no-op. By combining this intermediate result with the partial correctness (Lemma .3) and completeness (Lemma .4) of the explicit reconstruction algorithm, we can conclude the existence of an abstract state that correctly approximates the program and which respects the safety policy. Thus, the optimised proof checker is a fixpoint checker (*cf.* Definition .1).

Theorem 5.6 (Checker) *Define*

$$checker(P, strat) \equiv (reconstruct(P, strat) \neq None).$$

For all programs P *and strategies strat,*

$$if\ \ checker(P, strat) = true\ \ then\ \exists s\ , F_{\mathsf{P}}(s\ ) \sqsubseteq\ s\ \ \ and\ \ Safe_{\mathsf{P}}(s\ ) = true$$

6 Generating Certificates

The generation of strategies (a code producer task) is not safety-critical for the PCC infrastructure. However, for our PCC scheme to be feasible, efficient strategies are necessary. In this section we first show that for any given fixpoint a strategy can be generated, and then show how these strategies can be optimised. We introduce the notion of *winning* strategies which are strategies that verify certain well-formedness conditions with respect to the dependencies between control points. These dependencies informally express that the abstract memory at one control point is needed for the computation of the abstract memory at another control point, and are defined formally as follows.

Definition 6.1 *Let P be a program. The dependencies of a control point cp are all the control points that appear in the rhs of a constraint with target p.*

$$Depends(cp) = \{cp' \mid \exists c \in gen\_cstr(P), c.target = cp \wedge cp' \in c.sources\}$$

Definition 6.2 *Let P be a program and s be an abstract state that is a correct abstraction of P ($F_P(s) \sqsubseteq s$) and which respects the safety policy ($Safe_P(s) = true$). A* winning *strategy is such that for each control point cp*

(1) there exists one and only one $Eval(cp)$;
(2) there exists at most one $Assign(cp, s(cp))$;
(3) an $Assign(cp, s(cp))$ never occurs after an $Eval(cp)$;
(4) there exists at most one $Drop(cp)$;
(5) for all $cp' \in Depends(cp)$, we have that
  (a) $Eval(cp)$ occurs after $Eval(cp')$ or $Assign(cp', s(cp'))$;
  (b) $Drop(cp')$ never occurs before $Eval(cp')$;
  (c) $Drop(cp')$ never occurs before $Eval(cp)$;

The essential property of *winning* strategies is their existence:

Lemma 6.3 *Forall program P and abstract state s such that $F_P(s) \sqsubseteq s$ and $Safe_P(s) = true$, there exists a* winning *strategy strat such that*

$$checker(P, strat) = true.$$

PROOF: Consider the strategy made of *Assign* commands followed by *Eval* commands.

$$Assign(p_1, s(p_1)); \ldots; Assign(p_n, s(p_n)); Eval(p_1); \ldots; Eval(p_n)$$

Conditions 1,2,3 and a of Definition 6.2 are trivially fulfilled because the control points are assigned and evaluated once and all the assignments are made before the evaluations begin. Finally, conditions 4, b and c are vacuously true because there are no *Drop* commands.

This naive strategy gives rise to the naive checker described in Section . It requires a whole s and evaluates all the control points. As such, it is not a very interesting strategy. Nonetheless, its existence allows us to state the relative *completeness* of our checkers.

Theorem 6.4 *Given program P and abstract state s such that $F_P(s) \sqsubseteq s$ and $Safe_p(s) = true$, there exists a certificate cert which ascertains $Safe(P)$.*

PROOF: Take as certificate a *winning* strategy which by Lemma 6.3 always exists. On such a winning strategy, the checker succeeds. The correctness of the checker now implies that the theorem holds.

The generation of *winning* strategies can be done using the following scheme. Choose an order on the control points and generate `Eval` commands for each variable in that order, issuing `Assign` commands when evaluation requires the value of a variable that has not yet been visited. `Drop` commands can be inserted as soon as a variable is no longer needed to evaluate those variables that have not yet been visited. By choosing different orderings, different kinds of strategies will be generated. For example, fixing to evaluate control points in increasing order leads to the lightweight byte code verifier of Rose [2]. In this kind of strategies, the `Eval` commands are implicit and only need to be prefixed by `Assign` commands for each back-edge in the dependency graph. This leads to very compact strategies at the expense of being sub-optimal in memory usage. We return to this point in Section 9 on related work.

More memory-efficient strategies can be obtained by taking into account the specific topological properties of the control-flow graph (see Definition 6.1). We here list optimisations for some standard intermediate code structures:

Sequential graphs. A sequential graph is a graph for which a control point has only a single predecessor and a single successor. Such graphs are obtained from the analysis of basic blocks. They allow a straightforward strategy which works in constant memory and alternates a `Eval` command and a `Drop` command of the predecessor control point.

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

Such a strategy can be coded efficiently by intervals of program counters.

Loop free graphs. For a directed acyclic graph (DAG), a topological traversal of the graph is a *winning* strategy that does not require a single `Assign` command. It is possible to further optimise this strategy by picking a traversal that allows to insert `Drop` commands as early as possible. This improves the memory usage of the checker.

Reducible graphs. Reducible graphs are obtained from structured programming languages. As such, our byte code may not structured but any code generated from structured ones will be. For those graphs, an efficient strategy consists in placing `Assign` commands at loop-headers. Given these loop-headers, the rest of the graph can be decomposed into DAGs for which the DAG strategy applies.

7 Implementation

We use the program extraction mechanism of Coq to speed up the computations both on the producer and the consumer side. The extraction mechanism in Coq produces Caml programs from Coq terms by eliding those parts of the terms that do not have computational content. Such parts are only necessary to ensure the well typing of the Coq term (and thereby the correctness of the corresponding programs) but are not required for executing the programs.

Strictly speaking, nothing needs to be certified in Coq on the producer side, but parts of the extracted checker can nonetheless be reused. In order to obtain a working analyser, the code extracted from a `CertifiedAnalysis` structure must be combined with a fixpoint iterator for solving the constraint systems. Such an iterator is a reusable component independent of the specific analysis. If the extracted code does not scale well, subparts of the abstract domains can be substituted for hand-coded operators in a modular way. This might be relevant for numeric-intensive computations for which purely functional implementations cannot compete with the arithmetics of the processor. These optimisations are local to the producer and serve to speed up the computation of a certificate. They may be unsafe but can at worse lead to certificates that will not be accepted by a certified checker.

On the consumer size, the specification of the certificate checker is a module of type `Checker` (presented in Section 2). Because the fixpoint 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 extracted Caml `checker` function must be applied to a program `p` and a certificate `cert`. Some care must be exercised when deciding on the format of `p` and `cert`. The Coq extraction of function is correct only if the extracted function is evaluated on arguments that are well-typed in Coq (see Letouzey's PhD thesis [17] for a formal statement) but the extracted Caml function will have a more permissive type and will thus return a result on arguments which the Coq version of the function would not accept. This means that, potentially,

a malicious producer could propose a certificate that would be rejected by Coq but accepted by the Caml type checker. The output of the extracted checker on such a certificate is unspecified by its Coq correctness statement and would provide a security hole in the architecture.

To avoid this pitfall, we define the Coq certificate type so that it is in bijection with the corresponding extracted type. In our implementation we choose a certificate format of form:

```
Definition certificate : Set := list bool.
```

Hence certificates are directly manipulated as lists of bits. It is the responsibility of the consumer to open and close the stream file and convert it into a correct list of bits. The producer must then not only propose a certified checker written in Coq but also a Coq parser to parse the bit-stream certificates. Programming such a parser is not difficult since no proof (except termination) is needed. The main Caml file of the consumer checker then has the structure

```
let _ =
  let file = Sys.argv.(1) in
  let p = Parser.parse_main (file^".class") in
  let s = ReadBit.get_stream (file^".pcc") in
    if Coq.BytecodeChecker.checker p s
    then Printf.printf "program␣safe.\n"
    else Printf.printf "bad␣certificate.\n"
```

This clearly exhibits the three components of the consumer checker:

- the byte code parser `Parser.parse_main`,
- the function `ReadBit.get_stream` to open, close and transform a channel into a list of bit
- the extracted checker `Coq.BytecodeChecker.checker`

The functions `Parser.parse_main` and `ReadBit.get_stream` are part of the trusted base whereas the function `Coq.BytecodeChecker.checker`, which was defined above, is certified by Theorem .6.

8 Experiments

We have tested our PCC framework by applying the improved interval analysis described in Section 4 on a number of array-manipulating algorithms for generating certificates for a safety policy stating that the programs do not make array accesses that are out of bounds. 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 this safety 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 enhanced interval analysis described in Section 4 is sufficiently precise to be able to verify that all array accesses are safe.

Figure 8 and 9 presents some measurements pertinent to the certification. Measures about the efficiency of the certificate verification are given Figure 8. The last column shows 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. Figure 9 shows size of various elements: source and byte code programs, full and compress fixpoints, and at last binary certificates. Compress certificates contain only one abstract memories for each back-edge in the dependency graph. The binary certificates are obtained from the compress fixpoints by an ad-hoc binary encoding. Such binary files are then decoded into strategy which follow the increasing order on program points. Note that this encoding/decoding phase is not part of the trusted base.

Program	checking time (sec.)	analyser/checker
BubbleSort	0.01	440/110
HeapSort	0.0 0	8001/381
QuickSort	0.060	8910/40
Convolution	0.010	460/92
FloydWarshall	0.020	23114/163
PolynomProduct	0.010	1 0669/133

Fig. 8. Efficiency experiments on various algorithms

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, 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 several orders of magnitude. 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.

Program	.java	.class	complete fixpoint	compressed fixpoint	binary certificate
BubbleSort	440	28	3640	182	44
HeapSort	1044	8 8	173 2	332	63
QuickSort	1078	96	2 288	629	1 8
Convolution	378	42	2942	19	2
FloydWarshall	417	96	7180	346	134
PolynomProduct	09	604	366	308	87

Fig. 9. Experiments on various algorithms: size in bytes

9 Related Work

The *VeryPCC* project conducted by Nipkow *et al.* aims at providing a foundational PCC framework verified within the Isabelle/HOL theorem prover. Their PCC infrastructure [28] is based on a dedicated safety logic that is used to express local program properties and the overall safety policy. The core of the framework is a generic VCGen that generates verification conditions in the safety logic from the program's control flow graph. The VCG is parameterised on a weakest precondition transformer *wpF* that for a given instruction in the program and a given post-condition in the safety logic finds a weakest precondition in the safety logic. This *wpF* transformer must be proved correct with respect to the operational semantics of the particular programming language. One difference with the work presented here is that the VCG works on programs annotated with loop invariants. These loop invariants can be provided by an un-certified data flow analyser but they will then have to be re-proved in Isabelle by the code producer in order to obtain a proof that can be communicated to the code consumer. This user interaction limits the scalability of the approach as soon as the invariants cannot be proved by the Isabelle decision procedures. Moreover, proof terms are Isabelle proof scripts that have to be rerun. Because tactics can boil down to proof search, the efficiency of the proof checking is not clear. By using an abstract interpretation certified within Coq, the analyser directly produces a proof (namely, a post-fixpoint) that can be communicated and understood by the proof checker.

The *Mobile Resource Guarantee* (MRG) project [,4] has produced a fundamental PCC infrastructure for proving properties related to the resource consumption of a code with explicit memory management. For example, they want to establish that a given code can avoid dynamic memory allocation by re-cycling memory that is no longer being used. Initially, the functional source code is submitted to an advanced static analysis that will provide information

about memory consumption. This information is then used to compile into an imperative intermediate code. To reason about intermediate code annotated with memory consumption information, they build an intermediate layer of customised inference rules from a generic program logic. The soundness of this logic is checked in Isabelle. Certificate checking is now reduced to checking a proof in this dedicated logic. The MRG work shares with us 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 does not propose a particular methodology for producing such proofs but can use a variety of type inference mechanisms. In one instantiation of the framework [19], the actual certificate checking in is done within Isabelle using dedicated proof techniques. In contrast, we propose a particular methodology based on certified abstract interpretation. Our use of post-fixpoints and their formalisation in constructive logic allowed to obtain a proof checker that is both certifiable and efficient.

The *Open Verifier Framework* [8] is a proposal for strengthening the trust in the infrastructure without sacrificing efficiency. It is more flexible and more secure than standard PCC. The soundness depends on a core (trusted) condition generator. For flexibility, condition generators can be installed dynamically to enrich the platform, without having to be trusted by the core. The interaction is governed by the following protocol. The core is generating strongest postconditions; *custom components* generate a weakening together with a machine-checkable proof that it is correct. To ease the design of such custom components, a scripting language provides a flexible way to describe on-the-fly abstractions. On the other hand, a *foundational* custom component would not have to argue its correctness at each inference step.

Albert, Hermenegildo and Puebla have proposed to use abstract interpretation for automatically producing analysis-carrying code [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 fixpoint is sent to the code consumer who uses the abstract interpreter to check in one iteration that the certificate is a fixpoint. Our work improves over this approach in three ways. First, our FPCC approach provides transmittable proofs of correctness of our analysers which means that they do not have to be part of the trusted computing base—this is not dealt with in [1]. Second, the certificates in [1] are complete fixpoints (the *analysis answer tables*) which could be further compacted with our fixpoint compression algorithm. Finally, their approach works for a high-level source code language (CLP) whereas we have directly addressed the problem of analysing byte code.

For PCC, the size of proof terms has been a recurring problem. Several approaches have been proposed to tackle this problem. Necula and Lee [21] enhance the LF type-checker with an efficient reconstruction algorithm that

allows a more compact representation of proofs. Works closer to ours are the oracle-based checkers of Necula and Rahul [22] who, instead of transmitting a proof term, sends as certificate an oracle (a stream of bits) that guides an higher-order logic interpreter in his proof search. A variation of this idea has been implemented in a *foundational* PCC framework by Wu, Appel and Stump [29]. Unlike our approach based on certified checkers, the logic interpreters are part of the TCB. The type of certificates are otherwise rather different but it is interesting to observe that in both cases, it is possible to generate quite small certificates.

Lightweight Bytecode Verification for the KVM developed by Rose and Rose [2 ,26] includes a compression scheme for stack maps (that correspond to our certificates) based on converting a data flow problem into a *lightweight data flow problem*. Compared to our algorithm, their stack map compression allows to evaluate certificates on the fly as constraint generation proceeds. It has the consequence that the strategy is pre-determined and fixed: the constraints must be solved in the order they are generated. Unlike our garbage-collecting checker, their strategy for `Drop`ping values is hard-coded and may not be optimal. Furthermore, backward control points cannot be dropped at all. For the same reason, the number of `Assign` may not be optimal. Our algorithm is more flexible and accommodates more efficient strategies. It has also the advantage that new strategies do not require a new correctness proof.

10 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 is taken advantage of in 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 *syntactic 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 syntactic 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 should be of interest to other analyses of low-level code.

The whole Coq development, including a working checker, is available for download at `http://www.irisa.fr/lande/pichardie/PCC/`.

Several issues remain open for further investigation.

- The theory of strategies for reconstruction fixpoints from Section could be developed further, notably with the aim of determining general conditions for the existence of optimal strategies. Furthermore, the trade-off between the length of a strategy (and hence its execution time) and its memory consumption should be elucidated.
- The class of security policies considered should be enlarged to include temporal policies and policies related to the way the code consumes the resources of the host machine. Here, we have chosen to deal with the array-out-of-bounds policy, to make the presentation focused but the framework can accommodate other policies as long as there are certified analysers to find the relevant information.
- We have illustrated our PCC framework with an interval-based analysis but the framework is prepared to accommodate more precise relational analyses such as *e.g.*, octagon-based analyses [18] as implemented in the industrial strength C program analyser Astree [12]. An interesting, concrete illustration of how optimised and certified analysers co-exist in our framework would be to use the highly optimised (but non-certified) abstract domains of Astree for building certificates that would then be checked by a checker built from a certified but non-optimised octagon byte code analyser.

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] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In *Proc. of the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices workshop (CASSIS'04)*, pages 1–26. Springer LNCS vol. 3362, 2005.

[5] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *Proc. of 11th Int. Conf. of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, pages 347–362. Springer LNCS vol. 3452, 2005.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astrée analyser. In M. Sagiv, editor, *Proc. of 14th European Symp. on Programming (ESOP'05)*, number 3444 in LNCS, pages 21–30. Springer, 2005.

[13] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 235–246, New York, NY, USA, 2002. ACM Press.

[14] John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on*

Formal Methods in Computer-Aided Design (FMCAD'96), volume 1166, pages 265–269, 1996.

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

[16] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

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

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

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

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

[21] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Proc. of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, 1998.

[22] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proc. of the 28th ACM Symp. on Principles of Programming Languages (POPL'01)*, pages 142–154. ACM Press, 2001.

[23] 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.

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

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

[26] Eva Rose and Kristoffer Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

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

[28] Martin Wildmoser, Tobias Nipkow, Gerwin Klein, and Sebastian Nanz. Prototyping proof carrying code. In J-J Levy, E. W. Mayr, and J. C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics, TC1 3rd Int. Conf. on Theoretical Computer Science (TCS2004)*, pages 333–347. Kluwer, 2004.

[29] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Proc. of the 5th ACM international conference on Principles and Practice of Declarative Programming (PPDP '03)*, pages 264–274. ACM Press, 2003.

A Constraint rules for the byte code interval analysis

$$\frac{instrAt_P(p_1, \text{Nop}, p_2)}{m_{p_2} \sqsupseteq m_{p_1}}$$

$$\frac{instrAt_P(p_1, \text{Ipush } n, p_2) \quad m_{p_1} = (s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big((const\ n) :: s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Pop, p_2) \quad m_{p_1} = (v\ :: s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big(s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Dup, p_2) \quad m_{p_1} = (v\ :: s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big(v\ :: v\ :: s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Ineg, p_2) \quad m_{p_1} = (v\ :: s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big((-\ v\ ) :: s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Iadd, p_2) \quad m_{p_1} = (v_1 :: v_2 :: s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big((binop\ +\ v_2\ v_1) :: s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Isub, p_2) \quad m_{p_1} = (v_1 :: v_2 :: s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big((binop\ -\ v_2\ v_1) :: s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Imult, p_2) \quad m_{p_1} = (v_1 :: v_2 :: s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big((binop\ \times\ v_2\ v_1) :: s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Load\ x, p_2) \quad m_{p_1} = (s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big((var\ x) :: s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Store\ x, p_2) \quad m_{p_1} = (e :: s_{p_1}, l_{p_1}) \quad v\ = [\![e]\!]_{expr}(l_{p_1})}{m_{p_2} \sqsupseteq \big(s_{p_1}[x \mapsto absval\ v\ ], l_{p_1}[x \mapsto v\ ]\big)}$$

$$\frac{instrAt_P(p_1, Iinc\ x\ n, p_2) \quad m_{p_1} = (s_{p_1}, l_{p_1}) \quad v\ = [\![e]\!]_{expr}(l_{p_1})}{m_{p_2} \sqsupseteq \big(s_{p_1}[x \mapsto absval\ v\ ], l_{p_1}[x \mapsto [\![binop\ +\ e\ (const\ 1)]\!]_{expr}(l_{p_1})]\big)}$$

$$\frac{instrAt_P(p_1, If\_icmplt\ p, p_2) \quad m_{p_1} = (e_2 :: e_1 :: s_{p_1}, l_{p_1})}{m_p \sqsupseteq \big(s_{p_1}, [\![e_1 < e_2]\!]_{test}(l_{p_1})\big)}$$

$$\frac{instrAt_P(p_1, If\_icmplt\ p, p_2) \quad m_{p_1} = (e_2 :: e_1 :: s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq \big(s_{p_1}, [\![e_1 \geq e_2]\!]_{test}(l_{p_1})\big)}$$

$$\frac{instrAt_P(p_1, Newarray, p_2) \quad m_{p_1} = (e :: s_{p_1}, l_{p_1})}{m_p \sqsupseteq \big((absval\ [\![e]\!]_{expr}(l_{p_1})) :: s_{p_1}, l_{p_1}\big)}$$

$$\frac{instrAt_P(p_1, Arraylength, p_2) \quad m_{p_1} = (v \ :: s_{p_1}, l_{p_1})}{m_p \sqsupseteq \left(v \ :: s_{p_1}, l_{p_1}\right)}$$

$$\frac{instrAt_P(p_1, Iaload, p_2) \quad m_{p_1} = (v_1 :: v_2 :: s_{p_1}, l_{p_1})}{m_p \sqsupseteq \left(\top :: s_{p_1}, l_{p_1}\right)}$$

$$\frac{instrAt_P(p_1, Iastore, p_2) \quad m_{p_1} = (v_1 :: v_2 :: v_3 :: s_{p_1}, l_{p_1})}{m_p \sqsupseteq \left(s_{p_1}, l_{p_1}\right)}$$

$$\frac{instrAt_P(p_1, Goto \ p, p_2)}{m_p \sqsupseteq m_{p_1}}$$

$$\frac{instrAt_P(p_1, Iinput \ p, p_2) \quad m_{p_1} = (s_{p_1}, l_{p_1})}{m_{p_2} \sqsupseteq (\top :: s_{p_1}, l_{p_1})}$$

$l\ [x \mapsto v]$ denotes the abstract local variables where value of variable x as been updated to v. $s\ [x \mapsto v]$ denotes the stack of expressions where all occurrences of $(varx)$ has been replaced by v.