

A PCC Architecture based on Certified Abstract Interpretation

Frédéric Besson, Thomas Jensen, David Pichardie

▶ **To cite this version:**

Frédéric Besson, Thomas Jensen, David Pichardie. A PCC Architecture based on Certified Abstract Interpretation. [Research Report] PI 1764, 2005, pp.35. inria-00000866

HAL Id: inria-00000866
https://inria.hal.science/inria-00000866

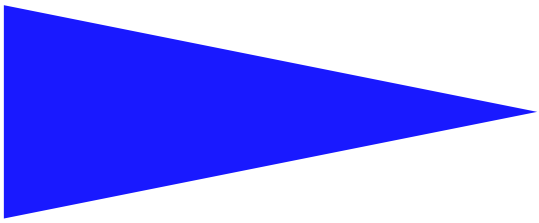
Submitted on 28 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1764



A PCC ARCHITECTURE BASED ON CERTIFIED ABSTRACT INTERPRETATION

FRÉDÉRIC BESSON , THOMAS JENSEN , DAVID PICHARDIE

ISSN 1166-8687

IRISA

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

A PCC Architecture based on Certified Abstract Interpretation

Frédéric Besson[*] , Thomas Jensen[**] , David Pichardie[***]

Systèmes symbolique
Projet Lande

Publication interne n1764 — Novembre 2005 — 35 pages

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. The checking of certificates is accelerated by a technique for (post-)fixpoint compression. 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.

Key-words: Proof Carrying Code, abstract interpretation, proof assistant, Coq

(Résumé : tsvp)

[*] Irisa/Inria
[**] Irisa/CNRS
[***] Irisa/ENS Cachan (Bretagne)

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Une architecture PCC basée sur l'interpétation abstraite certifiée

Résumé : Le *Proof Carrying Code* (PCC) est une technique pour télécharger du code sur une machine hôte tout en assurant que ce code adhère à la polique de sécurité de cette machine. Nous montrons comment l'interprétation abstraite certifiée peut être utilisée pour construire une architecture PCC pour laquelle la production de certificat est automatique pour le producteur de code. Pour vérifier les certificats, les consommateurs de code utilisent les vérificateurs de preuves produits à partir d'analyseurs statiques certifiés. Les vérificateurs de preuves sont eux-mêmes accompagnés de leur preuve de correction et accepter un nouveau vérificateur se ramène à vérifier son bon typage Coq. La vérification de certificats est accélérée par une technique de compression de (post-)points fixes. L'architecture PCC a été évaluée expérimentalement sur un langage à*bytecode* pour lequel nous avons conçu une analyse d'intervalles qui génère des certificats que assurent l'absence d'accès hors des bornes des tableaux.

Mots clés : Proof Carrying Code, interprétation abstraite, assistant de preuve, Coq

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 security 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 security property in this logic. Neither the mobile code nor the proposed security proof have to be trusted.

In their seminal work, Necula and Lee [14, 15] axiomatises the program using a Hoare-like logic. For a given security policy, this logic comes together with a *verification condition generator* VCgen that generates lemmas in a syntax-directed fashion which proofs are sufficient to ensure the property. For each lemma, a machine-checkable proof term has to be generated by the code producer. For each PCC instance, the proof format is customized with axioms (*e.g.*, for arithmetics). Moreover, 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" [18].

To remedy these weaknesses, Appel [2] has proposed the notion of *foundational proof carrying code* (FPCC). Program safety is defined directly in terms of an operational semantics, the verification conditions are generated from the program semantics rather than from some program logic and the safety proof is a proof term of the underlying logic, here LF. This increases the confidence in the correctness of the generated verification conditions. The downside of FPCC is that the construction of the proofs is made significantly more complex because proofs are done directly on the semantics rather than through some customized logic.

One reason for building the FPCC architecture directly upon the operational semantics is that, quoting Appel, "constructing a mechanically-checkable correctness proof of a full VCgen would be a daunting task." This has nevertheless been accomplished by Nipkow and Wildmoser [25, 24] who, for a reasonable subset of Java byte code, prove the soundness of a *weakest precondition* calculus with respect to the byte code semantics. To prove programs, they propose a hybrid approach that allows to use both trusted and untrusted components. An example of a trusted component is the byte code verifier that Klein and Nipkow have formalised and proved correct in Isabelle [10]. Untrusted components are external static analysers that suggest potential (inductive) invariants. These invariants are then reproved inside Isabelle in order to obtain a transmittable program certificate.

In this paper, we present a PCC infrastructure based on *certified abstract interpretation* [4]. Certified abstract interpretation is a technique for extracting a static analyser from the constructive proof of its semantic correctness, producing at the same time an analyser and a proof object certifying its semantic correctness. Using certified abstract interpretation has the following three advantages:

- the PCC infrastructure has semantic foundations as strong as those of FPCC,

- code certificates can be built from results of untrusted static analysers without the need of re-proving these results inside a theorem prover,
- proof carrying proof checkers can be built in such a way that the code consumer can check their correctness automatically using the Coq type checking mechanism.

The third point opens up for the possibility of safely down-loading proof checkers, adding flexibility to the PCC infrastructure.

In the next section we give an overview of the PCC architecture based on certified abstract interpretation. Section 3 contains the syntax and semantics of a byte code language that will serve to exemplify and experiment the architecture. In Sections 4 and 5 we outline the framework of certified abstract interpretation in Coq and instantiates this for a new static analysis for interval analysis on byte code programs. The end result is the Coq signature of a static analyser certified with respect to a security policy. Section 6 describes how proof carrying proof checkers can be built from certified static analyses. We first present an unoptimised proof checker and then develop a proof checker which certificates are *fixpoint reconstruction strategies*. Section 7 explains how to generate efficient strategies that lead to very compact program certificates. Section 8 explains how the Coq program extraction mechanism is used to implement efficient proof checkers and describes an experimental evaluation of the infrastructure in which a collection of array-intensive algorithms are certified to guarantee that the code will not make array accesses that are out of bound.

2 PCC architectures with *ad hoc* proof checkers

The proof checker is the cornerstone of a PCC infrastructure. Previous frameworks have advocated small, well-studied, flexible proof checkers; in particular the proof-checker of the *Logical Framework* (LF). The LF proof checker is small with sound theoretical foundations—this increases confidence in its correctness. The flexibility of LF means that any particular PCC instance can be cast into the framework. However, in this setting, building a proof (an LF term) is at best non-trivial (as in Necula's basic PCC framework) and sometimes infeasible.

At the other end of the PCC spectrum, the Java byte code verifier used in the KVM [23] relies on a sparse type annotation to assess type-safety. This proof checker has been designed for the specific task of byte code checking, which means that it is fast and use small certificates. In addition, the specialised nature of such dedicated proof checkers means that they are often easier to design and consequently prove correct.

In the following, we propose an extensible PCC framework that allows to download *ad hoc* proof-checkers safely. Its architecture is summarised in Figure 1. The architecture induces a two-step protocol between the producer and the consumer. In the first step, the producer queries the consumer to know whether it owns the relevant proof-checker. If not, the producer sends the checker together with its soundness proof. This soundness proof is then verified by a general purpose proof-checker by the consumer who, if the verification

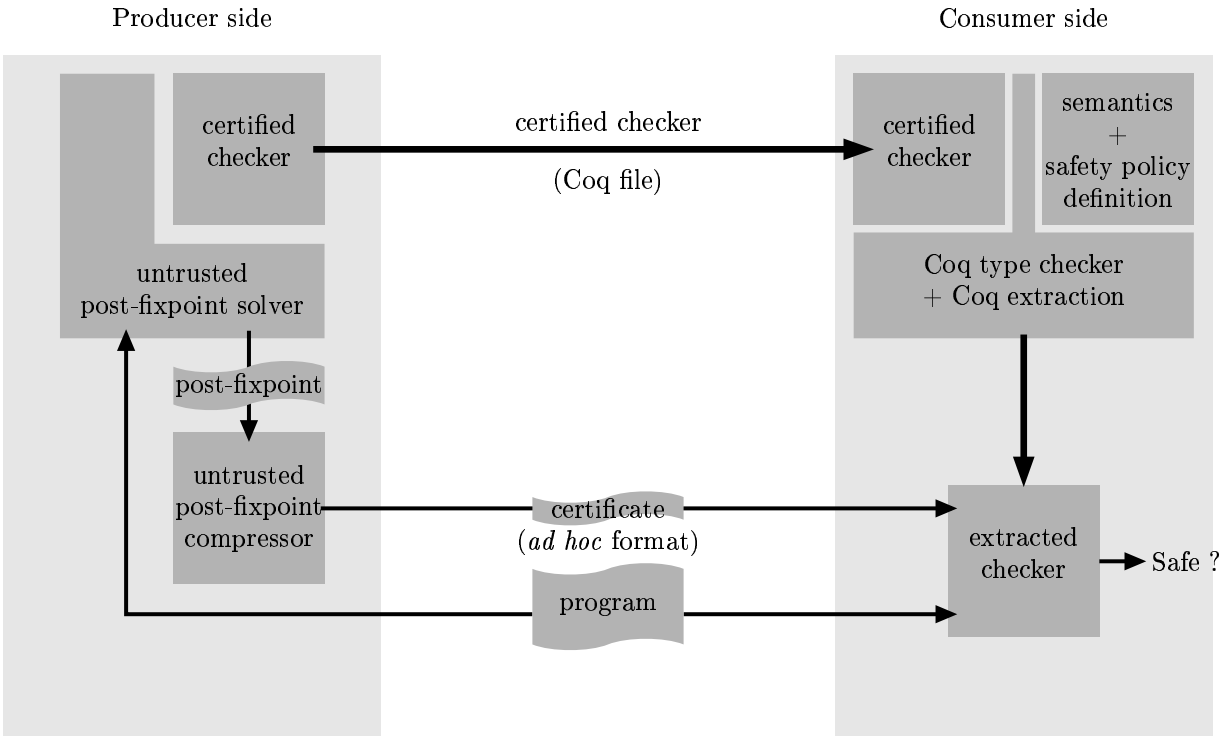


Figure 1: PCC architecture

succeeds, installs the now certified checker. In this way the architecture combines the advantages of both a trustworthy core proof checker and flexible *ad hoc* proof checkers. Once the proof checker has been installed, the consumer is ready to download the program of the producer. As it is customary, the producer sends the program packaged with a certificate to be checked by the previously downloaded proof checker. To bootstrap the infrastructure, the consumer only has to install a general purpose proof checker; our framework is based on the Coq proof assistant [6, 3].

2.1 The trusted computing base

The producer and consumer have to agree on a formal meaning of the statement "A program P satisfies a security property SP". This is done by providing a Coq specification of the semantics (here, a small-step operational semantics) of the program, and provide a semantics interpretation of the security property. We restrict our attention to safety properties that must hold for all the reachable execution states. More precisely, the Coq specification provides:

- the type of programs Pgm,
- a semantic domain for states State,
- a set of initial semantic states : $\mathcal{S}_0 \subset \text{State}$
- a semantic transition relation indexed by programs $\rightarrow_p \subseteq \text{State} \times \text{State}$,
- the semantic interpretation of security policies : a predicate on states $\text{Safe}_p \subseteq \text{State}$, parameterised by program.

As usual, we note $\rightarrow_p^*$ 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 $\mathcal{S}_0$

Definition 2.1

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

Definition 2.2 *A program is* safe *if all its reachable states are safe.*

$$[\![p]\!] \subseteq \text{Safe}_p \tag{1}$$

Given a program p, the code producer has then to provide a machine-checked proof of (1). However, Coq proofs are good example of machine-checked proof, but constructing proofs *interactively* using the Coq proof assistant quickly becomes unfeasible for low level languages. Instead, we propose to use abstract interpretation to automate the construction of program certificates. In this setting, programs are automatically annotated with program properties (elements of abstract domains) together with an reconstruction strategy. Proof checking now amounts to executing the reconstruction strategy, verifying that it leads to an abstract invariant that implies the security policy.

2.2 Signature of certified checkers

The certified checkers must implement the signature expressed by the Coq module `Checker` in Figure 2. 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 2.2. Thus, type checking the proposed checker by the code consumer proves that the checker is correct. In Section 6 we show how to implement such certified checkers using

```
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.
```

Figure 2: Interface for certified proof checkers

certified static analyses.

3 Byte code language: syntax and semantics

The PCC architecture in Figure 1 is programming language independent but to make the presentation more concrete we will use a particular language to illustrate the principles of our approach. Our example programming language is an imperative fragment of Java byte code that operates on a virtual machine with an operand stack, a set of local variables and a heap of arrays. A program is a sequence of triples $(p_1, instr, p_2)$ meaning that control passes from program point p_1 to p_2 while executing $instr$.

The instruction set contains operators for stack manipulations, integer arithmetic and for reading, storing and incrementing local variables. Specific instructions for array manipulations 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. The syntax is summarised in the following table:

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

We have left out the inter-procedural and object-oriented layers of the language. The present fragment is sufficiently general for at the same time illustrating 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 be done much in the same way as arrays are treated here. A certified static analysis for object-oriented (Java Card) byte code was described in [4].

The byte code language is given an operational semantics that operates with program states of the form $< pp, h, s, l >$ where pp is a program point to be executed next, h is a heap for storing allocated arrays, s is an operand stack, and l is an environment of local variables. 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.

$$
\begin{array}{rcl}
\text{Val} & = & \text{num } n \mid \text{ref } \mathit{ref} \quad n \in \mathbb{Z},\ \mathit{ref} \in \text{Location} \\
\text{Stack} & = & \text{Val}^* \\
\text{LocVar} & = & \text{Var} \to \text{Val} \\
\text{Array} & = & (\text{length} : \mathbb{Z}) \times ([0, \text{length}-1] \to \text{Val}) \\
\text{Heap} & = & \text{Location} \to \text{Array}_\perp \\
\text{State} & = & (\text{Ctrl} \times \text{Heap} \times \text{Stack} \times \text{LocVar}) \\
& & + \text{Error}
\end{array}
$$

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

$$\frac{\text{instrAt}_P(p_1, \text{Ipush } n, p_2)}{<p_1, h, s, l> \rightarrow_P <p_2, h, (\text{num } n) :: s, l>}$$

$$\frac{\text{instrAt}_P(p_1, \text{If_icmplt } p, p_2) \quad n_1 < n_2}{<p_1, h, (\text{num } n_2) :: (\text{num } n_1) :: s, l> \rightarrow_P <p, h, s, l>}$$

$$\frac{\text{instrAt}_P(p_1, \text{If_icmplt } p, p_2) \quad \neg\, n_1 < n_2}{<p_1, h, (\text{num } n_2) :: (\text{num } n_1) :: s, l> \rightarrow_P <p_2, h, s, l>}$$

$$\frac{\text{instrAt}_P(p_1, \text{Iload } x, p_2) \quad l(x) = \text{num } n}{<p_1, h, s, l> \rightarrow_P <p_2, h, (\text{num } n) :: s, l>}$$

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

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

This determines the semantics $[\![P]\!]$ of a program P as stipulated in Definition 2.1.

4 Abstract domains

To verify that a program p is safe, we have to check that $[\![p]\!] \subseteq \text{Safe}_p$. Because program semantics are in general uncomputable, computations are done in an abstract world, forgetting properties that are not needed for the verification. Hence an abstract domain $\left(\text{State}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot_{\text{State}^\sharp}\right)$ with a lattice structure is introduced, $\sqsubseteq^\sharp$ modelling the relative precision of elements in $\text{State}^\sharp$. In the concrete world, property precision is modelled with a partial order $\sqsubseteq$ (in our context, the subset partial order). Logical links between concrete and abstract world are done by a concretization function

$$\gamma : \left(\text{State}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \right) \rightarrow (\text{State}, \sqsubseteq, \sqcup, \sqcap)$$

An abstract object $s^\sharp \in \text{State}^\sharp$ is a *correct approximation* of a concrete object $s \in \text{State}$ if and only if $s \sqsubseteq \gamma(s^\sharp)$. Because we only focus on soundness of the abstract interpreters, the classic notion of Galois connection [8] is not mandatory here. Instead we require γ to be a meet morphism, *i.e.* $\gamma(s_1^\sharp \sqcap^\sharp s_2^\sharp) = \gamma(s_1^\sharp) \sqcap \gamma(s_2^\sharp)$[1].

[1] This assumption is equivalent to the existence of the corresponding Galois connection when $(\text{State}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp,)$ is complete and $\sqcap^\sharp$ denotes the general greatest lower bound (on set instead of two values).

4.1 Signatures of lattice structures

The Coq signature below shows (part of) the signature for lattice structures. A definition of a new abstract domain must provide operations `eq`,`order`, `join` and `meet`, together with *proofs* that these operations satisfy the properties `eq_refl`, `eq_sym`, `join_least_upper_bound` *etc.* A number of lattice operations exist for designing new abstract domains. As part of our certified static analysis project, we have developed a generic lattice library, containing base lattices (finite sets, intervals, ...) and domain constructors (sum, product, function) that permit a compositional design of new abstract domains by combining these basic blocks [21]. Most of the proofs follow standard lattice theory. However, we will give examples of a less standard abstract *domain of syntactic expressions* below.

```
Module Type Lattice.

 Parameter t : Set.

 Parameter eq : t → t → Prop.
 Parameter eq_refl : ∀ x : t, eq x x.
 Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  ...
 Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

 Parameter order : t → t → Prop.
 Parameter order_refl : ∀ x y : t, eq x y → order x y.
 Parameter order_antisym : ...
 Parameter order_trans : ...
 Parameter order_dec : ...

 Parameter join  : t → t → t.
 Parameter join_bound1 : ∀ x y : t, order x (join x y).
 Parameter join_bound2 : ∀ x y : t, order y (join x y).
 Parameter join_least_upper_bound : ∀ x y z : t,
    order x z → order y z → order (join x y) z.

 Parameter meet  : t → t → t.
  ...

 Parameter bottom : t.
 Parameter bottom_is_least : ∀ x : t, order bottom x.

End Lattice.
```

4.2 Interval analysis of byte codes

To demonstrate our PCC framework, we develop an interval analysis for the byte code presented in Section 3. The strength of the analysis is comparable to an existing interval

analyses for high-level structured languages [7]. However, to obtain the same precision at the byte code level, it has to be enhanced with an abstract domain of *syntactic expressions*. We first describe this novel kind of abstract domain; Section 5.1 explains in details how this domain is used to precisely model the effects of conditionals.

Interval analysis uses the set Intvl of intervals over $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, \infty\}$ to approximate numeric quantities. In our case, integers values and array sizes are abstracted by intervals. The abstract domains for the analysis are defined in Figure 3.

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

Figure 3: The abstract domains for interval analysis

The novelty of this analysis is the use of an abstract domain $\text{Exp}[\text{Val}^\sharp]$ of syntactic expressions over the base abstract domain $\text{Val}^\sharp$ of abstract values. An example of such an abstract element is binop $+$ (var j) (const 42)) which evaluates to the interval obtained by applying interval arithmetic to the interval that abstracts local variable j and the constant 42. Representing abstract values by such syntactic expressions 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` is a contraint over the possible values of `i` and `j` that can be propagated into the branches of a conditional statement. However, at byte code level, before such a conditional, the evaluation stack only contains a boolean. The explicit constraint between variables `i` and `j` is lost because their values have to be pushed onto the stack before they can be compared. Using syntactic expressions to abstract stack content enables the analysis to reconstruct this information.

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. More precisely,

the order $\sqsubseteq_{\text{Exp}}$ of syntactic expressions is inductively defined by the following rules:

$$\frac{n = m}{(\text{const } n) \sqsubseteq_{\text{Exp}} (\text{const } m)} \qquad \frac{x = y}{(\text{var } x) \sqsubseteq_{\text{Exp}} (\text{var } y)}$$

$$\frac{v_1^\sharp \sqsubseteq_{\text{Var}^\sharp} v_2^\sharp}{(\text{absval } v_1^\sharp) \sqsubseteq_{\text{Exp}} (\text{absval } v_2^\sharp)} \qquad \frac{e_1 \sqsubseteq_{\text{Exp}} e_2}{(\text{neg } e_1) \sqsubseteq_{\text{Exp}} (\text{neg } e_2)}$$

$$\frac{\text{op}_1 = \text{op}_2 \quad e_1 \sqsubseteq_{\text{Exp}} e_2 \quad e_3 \sqsubseteq_{\text{Exp}} e_4}{(\text{binop op}_1\ e_1\ e_3) \sqsubseteq_{\text{Exp}} (\text{binop op}_2\ e_2\ e_4)}$$

The concretization function for Exp[Val$^\sharp$] gives meaning to syntactic expressions by mapping them to sets of values in the concrete domain Val. It is defined in terms of the the concretization function γ_{Val} for the base value domain. Furthermore, since the expressions may contain program variables, the concretization function is parameterised by an environment $l \in \text{LocVar}$ mapping local variables to concrete values.

$$\gamma^l : \text{Exp}[Val^\sharp] \to \wp(\text{Val}) \qquad \text{with } l \in \text{LocVar}$$

$$\begin{aligned}
\gamma^l_{\text{Exp}}(\text{const } m) &= \{\ (\text{num } m)\ \} \\
\gamma^l_{\text{Exp}}(\text{var } x) &= \{\ l(x)\ \} \\
\gamma^l_{\text{Exp}}(\text{absval } v^\sharp) &= \gamma_{\text{Val}}(v^\sharp) \\
\gamma^l_{\text{Exp}}(\text{neg } e) &= \{\ (\text{num } (-n)) \mid (\text{num } n) \in \gamma^l_{\text{Exp}}(e)\ \} \\
\gamma^l_{\text{Exp}}(+\ e_1\ e_2) &= \\
&\left\{ \begin{array}{l|l} (\text{num } (n_1 + n_2)) & \begin{array}{l}(\text{num } n_1) \in \gamma^l_{\text{Exp}}(e_1) \\ (\text{num } n_2) \in \gamma^l_{\text{Exp}}(e_2)\end{array} \\ \ldots & \end{array} \right\}
\end{aligned}$$

Figure 4: Concretization function for Exp[$Val^\sharp$]

Figure 5 provides an example 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`.

Theorem 4.1 *For each abstract domain* D *defined in Figure 3 there exists a Coq structure* `D` *satisfying the signature* `Lattice`.

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]]
     // <(var j)>
 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]]
     // <>
...
```

Figure 5: Analysis example

PROOF: The constructive proofs of standard lattice structures (product, function space, lifting) are given in [20]. The abstract domain of syntactic expressions is a lattice obtained by extending the join and meet operations homomorphically over terms. The machine-checked proof is available on-line [19]

The main result linking the abstract state to the operational semantics states that the concretization function for abstract states is a meet-morphism of lattices. This result is needed for the correctness proof of the abstract interpretation defined in the following section. Let

$$\begin{aligned} \text{Mem} &= \text{Stack} \times \text{Heap} \times \text{LocVar} \\ \text{Mem}^\sharp &= \text{Stack}^\sharp \times \text{LocVar}^\sharp \end{aligned}$$

and suppose given concretization functions for abstract values and environments of local variables,

$$\begin{aligned} &\gamma_{\text{Val}} \in Val^\sharp \to \wp(\text{Val}), \\ &\gamma_{\text{LocVar}} \in \text{LocVar}^\sharp \to \wp(\text{Heap} \times \text{LocVar}) \end{aligned}$$

Definition 4.2 *The concretization function* $\gamma_{\text{Mem}} \in \text{Mem}^\sharp \to \wp(\text{Mem})$ *is defined by*

$$\gamma_{\text{Mem}}(e_1 :: \cdots :: e_n, l^\sharp) = \left\{ (v_1 :: \cdots :: v_n, h, l) \;\middle|\; \begin{array}{l} v_1 \in \gamma^l_{\text{Exp}}(e_1), \ldots, v_n \in \gamma^l_{\text{Exp}}(e_n) \\ (h,l) \in \gamma_{\text{LocVar}}(l^\sharp) \end{array} \right\}$$

Lemma 4.3 *The concretization function*

$$\gamma_{\text{Mem}} \in \text{Mem}^\sharp \to \wp(\text{Mem})$$

is a meet-morphism, provided that $\gamma_{\text{Val}} \in Val^\sharp \to \wp(\text{Val})$ *and* $\gamma_{\text{LocVar}} \in \text{LocVar}^\sharp \to \wp(\text{Heap} \times \text{LocVar})$ *are.*

5 Abstraction Interpretation for PCC

The abstract domains defined in the previous section form the basis on which we now construct correctness proofs of proof-checkers. These proof checkers will be developed in a constructive manner using the abstract interpretation methodology. The end result is a proof (lambda) term that can be sent as a certificate with a proposed proof checker.

The theory of abstract interpretation [8] explains how a correct approximation of the semantics $[\![p]\!]$ of a program p can be computed using an abstract function $F^\sharp \in \text{State}^\sharp \to \text{State}^\sharp$. Theorem 5.1 recalls the classic correctness criterion for $F^\sharp$.

Theorem 5.1 *Given a program* p *and a function* $F^\sharp \in \text{State}^\sharp \to \text{State}^\sharp$*, if* $F^\sharp$ *satisfies*

$$\forall s^\sharp \in \text{State}^\sharp,\ \mathcal{S}_0 \subseteq \gamma(F^\sharp(s^\sharp)) \tag{2}$$

$$\begin{aligned} &\forall s^\sharp \in \text{State}^\sharp,\ \forall s_1, s_2 \in \text{State}, \\ &s_1 \to_p s_2 \wedge s_1 \in \gamma(s^\sharp) \Rightarrow s_2 \in \gamma(F^\sharp(s^\sharp)) \end{aligned} \tag{3}$$

then

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

Hence all post-fixpoints of $F^\sharp$ are correct approximations of $[\![p]\!]$. This is an essential property for PCC: to check that an abstract element $s^\sharp$ is a correct approximation of $[\![p]\!]$ is it sufficient to check it is a post-fixpoint of $F^\sharp$.

In the following, we specialise this theory to languages where the semantics domain is a set of reachable states, made of pairs control point and memory:

$$\mathrm{State} = \mathrm{Ctrl} \times \mathrm{Mem}.$$

An abstract interpreter computes memory invariants at each control point of a program. An abstract domain $\mathrm{Mem}^\sharp$ with a lattice structure is defined and the memory property represented by an abstract memory $m^\sharp$ is given by a concretization function $\gamma : \mathrm{Mem}^\sharp \to \wp(\mathrm{Mem})$. For a program with n control points the result of the analysis is an n-tuple $(m_1^\sharp, \ldots, m_n^\sharp)$ which is specified as a solution of an inequation system (with q constraints).

$$\begin{array}{rcl} m_1^\sharp & \sqsupseteq & f_1(m_1^\sharp, \ldots, m_n^\sharp) \\ & \ldots & \\ m_n^\sharp & \sqsupseteq & f_n(m_1^\sharp, \ldots, m_n^\sharp) \end{array} \tag{4}$$

The previous $F^\sharp$ function is hence seen as a function of $\mathrm{Mem}^{\sharp^n} \to \mathrm{Mem}^{\sharp^n}$ whose post-fixpoint are correct approximations of $[\![p]\!]$.

A very attractive feature of the PCC approach is that the producer is given the harder task of searching for a proof while the consumer is left with the simpler task of checking this proof. Nonetheless, the success of PCC depends on how easy it is to automate the generation of proofs. In general, this process is very difficult. The constraints imposed by standard PCC frameworks make the situation even worse. Indeed, because PCC require a proof term, this largely forbids the direct reuse of off-the-shelf provers (or decision procedures) which yield a yes/no answer. As a result, a PCC security policy often requires either to handcraft a specialised prover or to patch a general prover to generate a convenient proof term.

In our context, there exists a generic off-the-shelf prover that fits our demands: abstract interpretation comes along a generic implementation technique – namely *chaotic fixpoint iteration*. A sound post-fixpoint $(m_1^\sharp, \ldots, m_n^\sharp)$ is iteratively computed by successive approximations until it eventually verifies all the constraints (4) imposed by the analysis. The knowledge of such a post-fixpoint is a crucial step toward the generation of certificates for our *ad hoc* checkers.

5.1 Constraint generation for the interval analysis

Our byte code interval analysis is defined by a set of generation rules that for each instruction generate a constraint on the set of abstract states between the source and destination of the instruction. These constraints take into account the abstract domains of syntactic

expressions described in Section 4.2. Some of these constraints are presented in Figure 6. For example, pushing the constant n on the stack generates the constraint

$$m^\sharp_{p_2} \sqsupseteq \left((\text{const } n) :: s^\sharp_{p_1}, l^\sharp_{p_1}\right)$$

Similarly, the constraints modelling test-and-jump instructions must be able to compare elements that contain syntactic expressions. We use here the notion of *backward abstract interpretation of expressions*[7] 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 fulfils the guard $e_1 \ \mathfrak{c} \ e_2$. In the case of the environment abstraction chosen in this analysis, it is defined by

$$\begin{array}{r} [\![e_1 \ \mathfrak{c} \ e_2]\!]_{\text{test}}(l^\sharp) = [\![e_1]\!]{\downarrow}^\sharp_{\text{expr}} \ (v_1^\sharp)(l^\sharp) \sqcap^\sharp [\![e_2]\!]{\downarrow}^\sharp_{\text{expr}} \ (v_2^\sharp)(l^\sharp) \\ \text{with } (v_1^\sharp, v_2^\sharp) = [\![\mathfrak{c}]\!]{\downarrow}^\sharp_{\text{comp}} \left([\![e_1]\!]{\uparrow}^\sharp_{\text{expr}} \ (l^\sharp), [\![e_2]\!]{\uparrow}^\sharp_{\text{expr}} \ (l^\sharp)\right) \end{array}$$

This implementation is based on several operators:

- $[\![e]\!]{\uparrow}^\sharp_{\text{expr}} \ (l^\sharp) \in \text{Val}^\sharp$ evaluates an expression in a given abstract environment $l^\sharp$.
- $[\![\mathfrak{c}]\!]{\downarrow}^\sharp_{\text{comp}} \in \left(\text{Val}^\sharp \times \text{Val}^\sharp\right) \to \left(\text{Val}^\sharp \times \text{Val}^\sharp\right)$ refines the abstract values which verifies a given condition $\mathfrak{c}$.
- $[\![e]\!] {\downarrow}^\sharp_{\text{expr}} \ (v^\sharp)(l^\sharp) \in \text{LocVar}^\sharp$ refines an abstract environments $l^\sharp$ whose evaluation of expression e gives an abstract value $v^\sharp$.

In the example of Figure 5, the abstract environment $l^\sharp_{11}$ is equal to $\{j \mapsto [-\infty, +\infty];\ i \mapsto [100, 100]\}$. The inequation generated by the analysis for the instruction `if_icmple 16` requires the evaluation of

$$[\![(\text{binop } + \ (\text{var } j) \ (\text{var } i)) \ > \ (\text{const } 3)]\!]^\sharp_{\text{test}}(l^\sharp_{11})$$

By definition of $[\![\cdot]\!]^\sharp_{\text{test}}$, two expressions should then be evaluated in $l^\sharp_{11}$: $[\![(\text{binop} + (\text{var } j) \ (\text{var } i))]\!]{\uparrow}^\sharp_{\text{expr}}$ $(l^\sharp_{11}) = [-\infty, +\infty]$ and $[\![(\text{const } 3)]\!]{\uparrow}^\sharp_{\text{expr}} \ (l^\sharp_{11}) = [3, 3]$. The backward analysis of the condition $>$ is able to refine the first interval: $[\![>]\!]{\downarrow}^\sharp_{\text{comp}} \ ([-\infty, +\infty], [3, 3]) = ([4, +\infty], [3, 3])$. Then this refinement is propagated on $l^\sharp_{11}$ with the backward analysis of expressions: $[\![(\text{binop } +$ $(\text{var } j) \ (\text{var } i))]\!] {\downarrow}^\sharp_{\text{expr}} \ ([4, +\infty])(l^\sharp_{11}) = \{j \mapsto [-96, +\infty];\ i \mapsto [100, 100]\}$. The interested reader should consult [7] for more details about this technique.

Informally, the correctness of the constraint generation means that if an abstract state satisfies all the constraints generated by a program then its concretization must be an over-approximation of the set of reachable states. This is stated formally as follows:

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

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

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

Figure 6: Constraint generation rules (examples)

Lemma 5.2 *Let P be a program, $gen_cstr(P)$ the set of constraints generated from P and $St \in \text{State}^\sharp$ an abstract state. If*

$$\forall \left(m^\sharp_p \sqsupseteq f_p(m^\sharp_1, \ldots, m^\sharp_n)\right) \in gen_cstr(P),\ St_p \sqsupseteq f_p(St)$$

then $[\![P]\!] \subseteq \gamma_P(St)$.

5.2 Abstract Security Policy

The proof of correctness of a proof checker obtained from a certified analysis will ascertain that the checker is capable of verifying that a program adheres to a given security policy. This security policy will thus have to be integrated with the abstract interpretation. In our setting, we split the abstract safety property into several local tests of the form

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

Each check is attached to a specific control point p and ensures that no error state can be reached by a one-step transition out of the state at control point p. For example, in the case of array bounds checking this means that for each control point p with an instruction Iaload, we check that the abstract memory $m^\sharp$ attached to this point verifies

$$\begin{aligned}\forall (h, (\text{num } i) :: (\text{ref } ref) :: s, l) \in \gamma(m^\sharp),& \\ h(ref) = a \ \Rightarrow 0 \leq i < a.&\text{length}\end{aligned}$$

The corresponding $\text{check}_{\text{Iaload}}$ function is hence of the form

$$\begin{aligned}&\text{check}_{\text{Iaload}}\left(e_i :: e_{\text{ref}} :: s^\sharp, l^\sharp\right) = \\ &\quad [\![<]\!]^\sharp_{\text{comp}}\left([\![e_i]\!]^\uparrow_{\text{expr}}(l^\sharp), [0,0]\right) = \bot_{\text{Val}} \\ &\qquad \&\& \\ &\quad [\![\geq]\!]^\sharp_{\text{comp}}\left([\![e_i]\!]^\uparrow_{\text{expr}}(l^\sharp), [\![e_{\text{ref}}]\!]^\uparrow_{\text{expr}}(l^\sharp).\text{length}\right) = \bot_{\text{Val}}\end{aligned}$$

Here the tests of the form $[\![\mathtt{c}]\!]\downarrow^\sharp_{\text{comp}}(v_1^\sharp, v_2^\sharp) = \bot_{\text{Val}}$ check that no concrete value v_1 et v_2 represented by $v_1^\sharp$ and $v_2^\sharp$ can pass the condition $\mathtt{c}$. Such a $\text{check}_{\text{Iaload}}$ function is associated to each control point of a program where an Iaload instruction is found.

The whole check generation is realised by a function *gen_AbSafe* which returns, for a given program, a list of local tests. The correctness of this local test generation means that if a correct approximation St of a program P pass all the local tests in $gen_AbSafe(P)$ then P is safe. This is stated formally in the following lemma:

Lemma 5.3 *Given program P and abstract state St which is a correct approximation of $[\![P]\!]$ (i.e. $[\![P]\!] \subseteq \gamma(St)$). If*

$$\forall (p, \text{check}) \in gen_AbSafe(P),\ \text{check}(St(p)) = \text{true}$$

then $[\![P]\!] \subseteq \text{Safe}(P)$.

5.3 A signature of certified analysers

The final result of the analysis certification is a module of signature `CertifiedAnalysis` given in Figure 7. The module includes a lattice module of abstract states which matches the Coq signature of lattices defined in Section 4.1. The signature also contains the type of constraints `Cstr` and the constraint generation function `gen_cstr` that implements the constraint generation rules from Figure 6. `Verif_cstr` is a predicate to explain what means for an abstract state to verify a constraint. The single proof required by this interface is the proof of parameter `analysis_correct` that states the global correctness of the constraint generator `gen_cstr` and the abstract test generator `gen_AbSafe`. In the case of our bytecode interval analysis, this proof is obtained by combining Lemma 5.2 and Lemma 5.3.

Theorem 5.4 *Given a program P and an abstract state St, if all generated constraints are fulfilled by St*

$$\forall \left(m_p^\sharp \sqsupseteq f_p(m_1^\sharp, \ldots, m_n^\sharp)\right) \in gen_cstr(P),\ St_p \sqsupseteq f_p(St) \tag{5}$$

and if St passes all the abstract checks

$$\forall (p, \text{check}) \in gen_AbSafe(P),\ \text{check}(St(p)) = \text{true} \tag{6}$$

then the program P is safe

$$[\![P]\!] \subseteq \text{Safe}_P$$

In the rest of the paper (and in signature `CertifiedAnalysis`) the condition (5) is written $\text{Approx}(P, St)$ and (6) is written $\text{Secure}(P, St)$.

```
Module Type CertifiedAnalysis.

  Declare Module Lat : Lattice.

  Definition t := Ctrl → Lat.t.

  Parameter Cstr : Set.
  Parameter Target : Cstr → Ctrl.
  Parameter Dep : Cstr → list Ctrl.
  Parameter Eval : Cstr → list Lat.t → Lat.t.

  Definition Verif_cstr (C:Cstr) (St:t) : Prop :=
      Lat.order (Eval C (map St (Dep C)))
                (St (Target C)).

  Parameter gen_cstr : program → list Cstr.

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

  Parameter gen_AbSafe :
         program → list (Ctrl*(Lat.t → bool)).

  Definition Secure (P:program) (St:t) :=
    ∀ p check,
      In (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.
```

Figure 7: The Coq signature of certified static analysis

6 PCC checkers

The checker component of the PCC architecture is the critical part that has to be both sound and efficient in space and speed. Hence, certificates should be small to ensure that their checking is fast. In the following, we describe how to generate proof checkers and certificates that fulfil these requirements. The certificates we generate attach a piece of information to a subset of the program points and each such piece of information can be checked in isolation. Hence, certificate size and certificate checking are linear in the size of the program.

The method for constructing *ad hoc* proof 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` (*cf.* Figure 7) and returns a `Checker` the interface of which was defined in Figure 2.

Central to the design of this module is Lemma 6.1. It is an intermediate result that forms the link between the checker and the certified analysis and which has to be proved for any new proposed proof checker. For the naive checker presented next in Section 6.1) this is straightforward. Later in the section we provide elements of the proof for an optimised checker (Section 6.2) which works with smaller certificates.

Lemma 6.1 *Given a program P and a certificate cert. If the proof checker*

$$\text{checker} : \text{Pgm} \to \text{Certificate} \to \text{bool}$$

satisfies that

$$\text{checker}(P, cert) = \text{true},$$

then there exists an abstract state $S^\#$ which at the same time

- *approximates the program semantics (*$\text{Approx}(P, S^\#)$*) and*
- *respects the security policy (*$\text{Secure}(P, S^\#)$*)*

By combining Lemma 6.1 with the property `analysis_correct` belonging to the signature `CertifiedAnalysis`, it is straightforward to prove that success of the verification ($\text{checker}(P, cert) = \text{true}$) entails program safety (*i.e,* the `checker_ok` theorem of the `Checker` interface).

6.1 A naive checker

The certificate generated by the naive checker is just an abstract state $S^\#$. The algorithm of the checker itself is simple: it checks that all the generated verification conditions are satisfied by the proposed $S^\#$. It can be written simply as

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

where we benefit from the fact that we have provided constructive definitions of the abstract domain operations throughout the specification. As a consequence, the checker is directly executable.

This algorithm trivially satisfies Lemma 6.1. Indeed, 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 security 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 checked the constraints imposed by the analysis and the security requirements. To be more precise, verifying 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 is sub-optimal and the sections that follow explain how to design checkers that require much sparser certificates.

6.2 Strategies for reconstructing certificates

The naive algorithm is given as certificate a *complete* solution of the analysis – an abstract memory state is attached to each program point. In this section, we 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 secure abstract state.

$$\text{reconstruct} : \text{Pgm} \to \text{Strategy} \to \mathit{option}(\text{Ctrl} \to \mathit{TagMem})$$

The datatypes for strategy commands and tagged memories are summarised in Figure 8.

```
Inductive TagMem : Set :=
| Undef
| Hint (m:Mem#)
| Checked (m:Mem#).

Inductive command : Set :=
| Assign  (c:Ctrl) (m:Mem#)
| Eval (c:Ctrl).

Definition Strategy := list command.
```

Figure 8: Tags and commands

Tags are used to keep track of the reconstruction status of a control point. They carry the following intuitive meaning. Consider a tagged abstract state $\mathtt{S}^\#$ and a control point `cp`.

- `Undef` means that the abstract memory to be attached to `cp` has not been reconstructed (yet);
- `Hint mem` means that `mem` has been proposed as an (untrusted) invariant for `cp`;
- `Checked mem` means that all the verification conditions of `pc` are satisfied by `mem`.

The reconstruction starts from an undefined tagged abstract state and consumes the strategy commands one at a time. Each such 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 security policy of control point `cp`, the abstract state is updated ($S^\#{:=}S^\#[cp \rightarrow \mathrm{Hint}(mem)]$).
- 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 security conditions, we have $S^\#{:=}S^\#[cp \rightarrow \mathrm{Checked}(mem)]$
  - If it is `Hint mem'`, and `mem'` subsumes `mem`, we have $S^\#{:=}S^\#[cp \rightarrow \mathrm{Checked}(mem')]$

  In any other case, the reconstruction fails.

If the reconstruction succeeds, it outputs a tagged abstract state for which all program points are tagged `Checked`. A proof sketch of this property is provided in the next section.

6.3 Soundness of reconstruction

The reconstruction algorithm as well as its correctness proof have been formalised in Coq. The central part of the proof of Lemma 6.1 for this checker is that a successful run of the reconstruction algorithm yields a tagged abstract state from which a sound and secure abstract state can be obtained by just erasing tags.

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 6.2 *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 security 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 security policy;*

The soundness proof of the reconstruction algorithm is divided into two parts. Lemma 6.3 states that the reconstruction algorithm only returns partially correct tagged abstract states.

Lemma 6.3 (Correct Reconstruction) *Given program P and strategy strat, if*

$$\text{reconstruct}(P, strat) = Some\ S^{\#}$$

then $S^{\#}$ is partially correct

The proof is by induction over the length the strategy. One proves that each command is updating the tagged abstract state such that the invariant is preserved.

Lemma 6.3 does not ensure that the reconstruction is complete. Indeed, the totally undefined abstract state is partially correct. Lemma 6.4 ensures that, at the end of the reconstruction, all control points have the `Checked` tag attached to them.

Lemma 6.4 (Complete Reconstruction) *Given program P and strategy strat. If*

$$\text{reconstruct}(P, strat) = Some\ S^{\#}$$

then

$$\forall (cp \in P), \exists mem^{\#}, S^{\#}(cp) = \text{Checked}(mem^{\#})$$

There exists a simple algorithm that verifies Lemma 6.4. At the end of the reconstruction, it iterates over the program points to check whether all the program points have been appropriately tagged. To avoid this traversal, the reconstruct algorithm keeps track of the number of control points that have not been tagged with the tag `Checked`. As a result, to prove Lemma 6.4 we simply show (by induction over the length of strategy) that at the end of the reconstruction this number is equal to zero.

6.4 Garbage collecting strategies

The strategies presented so far explicitly yield a witness $S^{\#}$ that satisfies the verification conditions of the analysis. However, in order for a code to be accepted as secure 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 not needed anymore 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.

To allow this behaviour, strategy commands are enriched with a `Drop pp` directive. Symmetrically, tags are enriched with a `Done` value. It is relevant to `Drop` a program point when it has been checked (*i.e,* its tag is `Checked mem`) and the computed value is not needed anymore to evaluate other constraints. In this case, the effect of the `Drop pp` is to set a `Done` tag. As a side-effect, the abstract memory `mem` may be garbage-collected.

In essence, we will not prove that the algorithm computes an abstract state but that it could have returned one – for which we have an algorithm. Lemma 6.5 formalised this intuition. It says that if a strategy with `Drop` commands succeeds then a strategy that replace `Drop` by a nop would have succeeded.

Lemma 6.5 (Implicit Reconstruction) *For all 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 been 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 nop. From this intermediate result and by correctness of the explicit reconstruction algorithm, it is straightforward to prove the following instantiation of Lemma 6.1 for the garbage collecting proof checker.

Theorem 6.6 (Checker) *Let* $\text{checker}(P, strat)$ *be defined by* $\text{reconstruct}(P, strat) \neq \text{None})$. *For all programs* P *and strategies* $strat$*, if*

$$\text{checker}(P, strat) = true$$

then there exists $S^\#$ *such that* $\text{Approx}(P, S^\#)$ *and* $\text{Secure}(P, S^\#)$

7 Generating Certificates

Program	.java size	.class size	certificate size	checker computation time
BubbleSort	440	528	32	0.015
HeapSort	1044	858	63	0.050
QuickSort	1078	965	124	0.060
ConvolutionProduct	378	542	52	0.010
FloydWharshall	417	596	134	0.020
PolynomProduct	509	604	87	0.010

Figure 9: Experiments on various algorithms, size file in bytes, time in secs

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 a strategy can always be generated for any given certificate. Then (Section 7.2) we show when these can be optimised.

7.1 Winning strategies

We introduce the notion of *winning* strategies which are the strategies for which the checkers succeed (proof omitted). The most important condition imposed on them is that dependencies between control points (Definition 7.1) are taken into account.

Definition 7.1 *Let P be a program. The dependencies of a control point p are all the control points that appear in the rhs of a constraint with target p.*

$$\mathrm{Depends}(p) = \left\{ p' \;\middle|\; \begin{array}{l} \exists c \in \mathrm{gen_cstr}(P), \\ \mathrm{Target}(c) = p \wedge p' \in \mathrm{Dep}(c) \end{array} \right\}$$

We can then precisely formalise the notion of winning strategy.

Definition 7.2 *Let P be a program and $S^\#$ be an abstract state such that* $\mathrm{Secure}(P, S^\#)$ *and* $\mathrm{Approx}(P, S^\#)$. *A* winning *strategy is such that for each control point cp*

1. *there exists one and only one* $\mathrm{Eval}(cp)$*;*
2. *there exists at most one* $\mathrm{Assign}(cp, S^\#(cp))$*;*
3. *a* $\mathrm{Assign}(cp, S^\#(cp))$ *always occurs before an* $\mathrm{Eval}(cp)$*;*
4. *there exists at most one* $\mathrm{Drop}(cp)$*;*
5. *for all $cp' \in \mathrm{Depends}(cp)$, we have that*
   (a) $\mathrm{Eval}(cp)$ *occurs after* $\mathrm{Eval}(cp')$ *or* $\mathrm{Assign}(cp', S^\#(cp'))$*;*
   (b) $\mathrm{Drop}(cp')$ *never occurs before* $\mathrm{Eval}(cp')$*;*
   (c) $\mathrm{Drop}(cp')$ *never occurs before* $\mathrm{Eval}(cp)$*;*

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

Lemma 7.3 *Given P be a program, if one can find an abstract state $S^\#$ such that* $\mathrm{Secure}(P, S^\#)$ *and* $\mathrm{Approx}(P, S^\#)$*, then a* winning *strategy always exists.*

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

$$\begin{array}{l} \mathrm{Assign}(p_1, S^\#(p_1)); \ldots; \mathrm{Assign}(p_n, S^\#(p_n)); \\ \mathrm{Eval}(p_1); \ldots; \mathrm{Eval}(p_n) \end{array}$$

Conditions 1,2,3 and 5a of Definition 7.2 are trivially fulfilled because the control points are assigned and evaluated once; all the assignments are made before the evaluations begin. Finally, conditions 4, 5b and 5c are vacuously true because there are simply no Drop commands.

This naive strategy gives rise to the naive checker of Section 6.1. 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 *ad hoc* checkers.

Theorem 7.4 *Given P a program and an abstract state $S^\#$ such that* $\mathrm{Secure}(P, S^\#)$ *and* $\mathrm{Approx}(P, S^\#)$*, there exists a certificate cert which ascertains* $\mathrm{Safe}(P)$*.*

PROOF: Take as certificate a *winning* strategy. By Lemma 7.3, it always exists. Moreover, given a *winning* strategy, the checker succeeds. By correctness of the checker, Theorem 7.4 holds.

We have implemented a *greedy* strategy generator directly based on the definition of *winning* strategies. In essence, it generates `Drop` commands as soon as possible; `Eval` command as much as it can and `Assign` commands as a last resort. Even such a simple algorithm generates strategies of reasonable quality.

7.2 Efficient strategies

In certain cases, the specific topological properties of the control-flow graph (see Definition 7.1) entails that optimal strategies exist. In the following, we consider some classical 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.

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

Note that such a strategy can be coded very efficiently by an interval 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, a strategy that minimises `Assign` consists in putting those commands at loop-headers. Given these loop-headers, the rest of the graph can be decomposed into DAGs for which the DAG strategy applies.

Lightweight graphs Our strategy-based checker can emulate the lightweight byte code verifier of Rose [22]. In this approach, the evaluation order of the control points is hard-coded – control points are evaluated in increasing order. As a result, `Eval` commands are implicit and only few `Assign` commands are provided. This leads to a very compact strategy.

8 Implementation notes

Our implementation uses the program extraction mechanism of Coq to speed up the computations, both on the producer and the consumer side. Intuitively, 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 by the same the correctness of the corresponding programs) but are not necessary to reduce the term to normal form (to evaluate programs).

On the producer side, nothing needs to be certified in Coq, but parts of the extracted checker can nonetheless be reused. In order to obtain a working prototype analyser from the code extracted from a `CertifiedAnalysis` structure, what is required is a fixpoint iterator for solving the constraint systems. Such an iterator is a reusable component independent of the specific analysis Overall, this leads to a lightweight prototyping technique. Moreover, if the extracted code does not scale well enough, subparts of the abstract domains can be substituted by 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 producerx to speed up the computation of a certificate. They may be unsafe but can at the worst lead to certificates that will not be accepted by a certified checker.

On the consumer size, the extracted code is a module of type `Checker` (presented in Section 2). The associated `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 [11] for a formal statement). For example, the Coq certificate format might be the type of sorted integer list, whose corresponding extracted Caml type is that of all integer list, the property of being sorted having only logical and no computational content. Hence, a malicious producer could propose an unsorted list as certificate without being rejected by the Caml type checker. The output of the extracted checker on such a certificate is unspecified by its Coq correctness statement.

To avoid such a hole in our PCC framework, 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:

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

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

The main Caml file of the consumer checker is hence the following

```
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"
```

We see here clearly 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` is certified.

8.1 Experiments

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 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 algorithm (bubble sort, heap sort and quick sort), the Floyd-Wharshall algorithm for shortest past computation, and algorithms for polynomial product and vector convolution. For each algorithm, the interval analysis was sufficiently precise to be able to verify that all array accesses were safe. Figure 9 presents some measurements pertinent to the certification: the size of the source and byte code, the size of the certificates and the time for checking the certificates. The experiments have been done with the lightweight graph method for generating strategies presented in Section 7.

It is worth noting here that the size of the certificates is much (sometimes an order of magnitude) smaller than the code it certifies. Partly as a consequence of this, the certification time is negligible (10–60 ms.) for these byte codes. The six programs do not constitute a proper test suite but are sufficiently complex to allow to conclude that this PCC infrastructure can be used to generate and check non-trivial program certificates.

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 [25] 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 VCG 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 wit 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 dubious. 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 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.

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 [13] 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 [13] is done within Isabelle using dedicated proof techniques and is not efficient. Here, 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* [5] is a proposal to strengthen the trust in the infrastructure without sacrificing efficiency. It tends to bridge the gap between standard PCC and FPCC. It is more flexible and more secure than standard PCC. The soundness depends on a core (trusted) condition generator. For flexibility, custom condition generators can be downloaded and imported to enrich the platform. However, they are not 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 is described. This 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. 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 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 is a recurring problem. Several approaches have been proposed to tackle this problem. Necula and Lee [16] 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 [17] 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 [26]. Like our approach for *ad hoc* 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 [23, 22] 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. Alike our garbage-collecting checker, their strategy for `Dropping` 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. As a result, 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 take advantage of in the garbage-collectin 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 should be of interest to other analyses of low-level code.

Several issues remain open for further investigation.

- The theory of strategies for reconstruction fixpoints from Section 6 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.*, octogon-based analyses [12] as implemented in the industrial strength C program analyser Astree [9]. 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 octogon byte code analyser.

References

[1] Elvira Albert, German Puebla, and Manuel Hermenegildo. Abstraction-carrying code. In *Proc. of 11th International Conference on Logic for Programming Artificial Intel-*

ligence and Reasoning (LPAR'04), number 3452 in LNAI, pages 380–397. Springer, 2004.

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

[3] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer, 2004.

[4] 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)*, number 2986 in LNCS, pages 385 – 400. Springer, 2004.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[15] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. of 2nd Symp. on Operating System Design and Implementation (OSDI '96)*, pages 229–243. USENIX, 1996.

[16] 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.

[17] 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.

[18] 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.

[19] David Pichardie. Coq sources of the PCC infrastructure. `http://www.irisa.fr/lande/pichardie/PCC/`.

[20] David Pichardie. *Certification des analyses statiques : application à la vérification de Java*. PhD thesis, Université de Rennes 1, September 2005.

[21] David Pichardie. Constructive construction of lattices for well-founded fixpoint iteration. Technical Report ????, IRISA, 2005. `http://www.irisa.fr/lande/pichardie/publications/lattice_report.pdf`.

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

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

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

[25] 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 Academic Publishers, 2004.

[26] 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 Bubble sort full example

Figure 10 presents 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 small enough 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, which is the case here. The drawback of this iteration strategy is that it is not ensured to terminate and it 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]
 }

}
```

Figure 10: source code of BubbleSort.java