An Incremental Approach to Abstraction-Carrying Code*

Elvira Albert[1], Puri Arenas[1], and Germán Puebla[2]

[1] Complutense University of Madrid, {elvira,puri}@sip.ucm.es
[2] Technical University of Madrid, german@fi.upm.es

Abstract. *Abstraction-Carrying Code* (ACC) has recently been proposed as a framework for Proof-Carrying Code (PCC) in which the code supplier provides a program together with an *abstraction* (or abstract model of the program) whose validity entails compliance with a predefined safety policy. Existing approaches for PCC are developed under the assumption that the consumer reads and validates the entire program w.r.t. the *full* certificate at once, in a non incremental way. In the context of ACC, we propose an *incremental* approach to PCC for the generation of certificates and the checking of untrusted *update*s of a (trusted) program, i.e., when a producer provides a modified version of a previously validated program. Our proposal is that, if the consumer keeps the original (fixed-point) abstraction, it is possible to provide only the program updates and the incremental certificate (i.e., the *difference* of abstractions). Furthermore, it is now possible to define an *incremental checking* algorithm which, given the new updates and its incremental certificate, only re-checks the fixpoint for each procedure affected by the updates and the propagation of the effect of these fixpoint changes. As a consequence, both certificate transmission time and checking time can be reduced significantly.

1 Introduction

Proof-Carrying Code (PCC) [13] is a general technique for mobile code safety which proposes to associate safety information in the form of a *certificate* to programs. The certificate (or proof) is created at compile time by the *certifier* on the code supplier side, and it is packaged along with the code. The consumer who receives or downloads the (untrusted) code+certificate package can then run a *checker* which by an efficient inspection of the code and the certificate can verify the validity of the certificate and thus compliance with the safety policy. The key benefit of this "certificate-based" approach to mobile code safety is that the consumer's task is reduced from proving to checking, a task which should be much simpler, efficient, and automatic than generating the original certificate.

* This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish MEC under the TIN-2005-09207 *MERIT* project, and the Regional CAM under the S-0505/TIC/0407 *PROMESAS* project.

Abstraction-Carrying Code (ACC) [4] has been recently proposed as an enabling technology for PCC in which an *abstraction* (i.e., an abstract model of the program) plays the role of certificate. An important feature of ACC is that not only the checking, but also the generation of the abstraction (or fixpoint) is *automatically* carried out by a fixed-point analyzer. In this paper, we will consider analyzers which construct a program *analysis graph* which is interpreted as an abstraction of the (possibly infinite) set of states explored by the concrete execution. Essentially, the certification/analysis carried out by the supplier is an iterative process which repeatedly traverses the analysis graph until a fixpoint is reached. A key idea in ACC is that, since the certificate is a fixpoint, a single pass over the analysis graph is sufficient to validate the certificate in the consumer side. The ACC framework and our work here are applied at the source-level while in existing PCC frameworks the code supplier typically packages the certificate with the *object* code rather than with the *source* code (both are untrusted). This is without loss of generality because both the ideas in ACC and in our current incremental proposal could also be applied to bytecode.

Non incremental models for PCC (ACC among them) are based on checkers which receive a "certificate+program" package and read and validate the entire program w.r.t. its certificate at once . However, there are situations which are not well suited to this simple model. In particular, we consider possible untrusted *updates* of a validated (trusted) code, i.e., a code producer can (periodically) send to its consumers new updates of a previously submitted package. By updates, we mean any modification over a program including: 1) the *addition* of new data/procedures and the extension of already existing procedures with new functionalities, 2) the *deletion* of procedures or parts of them and 3) the *replacement* of certain (parts of) procedures by new versions for them. In such a context of frequent software updates, it appears inefficient to submit a full certificate (superseding the original one) and to perform the checking of the entire updated program from scratch, as needs to be done with current systems. In the context of ACC, we investigate an *incremental* approach to PCC by considering any arbitrary program update over the original program.

When a program is updated, a new fixpoint has to be computed for the updated program. Such fixpoint differs from the original fixpoint stored in the certificate in a) the new fixpoint for each procedure affected by the changes and b) the update of certain (existing) fixpoints affected by the propagation of the effect of a). However, certain parts of the original certificate may not be affected by the changes. Thus, if the consumer still keeps the original abstraction, it is possible to provide, along with the program updates, only the *difference* of both abstractions, i.e., the *incremental certificate*. The first obvious advantage of an incremental approach is that the size of the certificate may be substantially reduced by submitting only the increment.

Moreover, the task performed by the checker can also be further reduced in incremental PCC. In principle, a non-incremental checker (like the one in [4]) requires a whole traversal of the analysis graph where the entire program + updates is checked against the (full) certificate. However, it is now possible to define

an *incremental checking* algorithm which, given the updates and its incremental certificate, only rechecks the part of the analysis graph for the procedures which have been affected by the updates and, also, propagates and rechecks the effect of these changes. In order to perform such propagation of changes, the *dependencies* between the nodes of the original analysis graph have to be computed and stored by the consumers, together with the original certificate. With this, the checking process is carried out in a single pass over the part of the abstraction affected by the updates. Thus, the second advantage of our incremental approach is that checking time is further reduced.

2 Abstraction-Carrying Code

We assume some familiarity with abstract interpretation (see [6]), (Constraint) Logic Programming (C)LP (see, e.g., [11, 10]) and PCC [13].

An abstract interpretation-based certifier is a function CERTIFIER: $\mathit{Prog} \times \mathit{ADom} \times \mathit{Approx} \mapsto \mathit{Approx}$ which for a given program $P \in \mathit{Prog}$, an abstract domain $D_\alpha \in \mathit{ADom}$ and an abstract safety policy $I_\alpha \in \mathit{Approx}$ generates an abstract certificate $\mathit{Cert}_\alpha \in \mathit{Approx}$, by using an abstract interpreter for D_α, such that the certificate entails that P satisfies I_α. An abstract safety policy I_α is a specification of the safety requirements given in terms of the abstract domain D_α. We denote that I_α and Cert_α are specifications given as abstract semantic values of D_α by using the same subscript α. The basics for defining such certifiers (and their corresponding checkers) in ACC are summarized in the following five points:

Approximation. We consider a *description (or abstract) domain* $\langle D_\alpha, \sqsubseteq \rangle \in \mathit{ADom}$ and its corresponding *concrete domain* $\langle 2^D, \subseteq \rangle$, both with a complete lattice structure. Description (or abstract) values and sets of concrete values are related by an *abstraction* function $\alpha : 2^D \to D_\alpha$, and a *concretization* function $\gamma : D_\alpha \to 2^D$. The pair $\langle \alpha, \gamma \rangle$ forms a Galois connection. The concrete and abstract domains must be related in such a way that the following condition holds [6] $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. In general $\sqsubseteq$ is induced by $\subseteq$ and α. Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in a precise sense.

Analysis. We consider the class of *fixed-point semantics* in which a (monotonic) semantic operator, S_P, is associated to each program P. The meaning of the program, $[\![P]\!]$, is defined as the least fixed point of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. If S_P is continuous, the least fixed point is the limit of an iterative process involving at most ω applications of S_P starting from the bottom element of the lattice. Using abstract interpretation, we can usually only compute $[\![P]\!]_\alpha$, as $[\![P]\!]_\alpha = \mathrm{lfp}(S^\alpha_P)$. The operator S^α_P is the abstract counterpart of S_P.

$$\mathsf{analyzer}(P, D_\alpha) = \mathrm{lfp}(S^\alpha_P) = [\![P]\!]_\alpha \tag{1}$$

Correctness of analysis ensures that $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$, i.e., $[\![P]\!] \in \gamma([\![P]\!]_\alpha)$. Thus, such *abstraction* can be used as certificate.

Certificate. Let $Cert_\alpha$ be a safe approximation of $[\![P]\!]_\alpha$. If an abstract safety specification I_α can be proved w.r.t. $Cert_\alpha$, then P satisfies the safety policy and $Cert_\alpha$ is a valid certificate:

$$Cert_\alpha \text{ is } \textit{a valid certificate} \text{ for } P \text{ w.r.t. } I_\alpha \text{ iff } Cert_\alpha \sqsubseteq I_\alpha \tag{2}$$

Together, Equations (1) and (2) define a certifier which provides program fixpoints, $[\![P]\!]_\alpha$, as certificates which entail a given safety policy, i.e., by taking $Cert_\alpha = [\![P]\!]_\alpha$.

Checking. A checker is a function CHECKER: $Prog \times ADom \times Approx \mapsto bool$ which for a program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and certificate $Cert_\alpha \in Approx$ checks whether $Cert_\alpha$ is a fixpoint of S_P^α or not:

$$\text{CHECKER}(P, D_\alpha, Cert_\alpha) \textit{ returns true iff } (S_P^\alpha(Cert_\alpha) \equiv Cert_\alpha) \tag{3}$$

Verification Condition Regeneration. To retain the safety guarantees, the consumer must regenerate a trustworthy verification condition –Equation (2)– and use the incoming certificate to test for adherence of the safety policy.

$$P \textit{ is trusted iff } Cert_\alpha \sqsubseteq I_\alpha \tag{4}$$

A fundamental idea in ACC is that, while analysis –Equation (1)– is an iterative process, checking –Equation (3)– is guaranteed to be done in a *single pass* over the abstraction.

3 Notions on Certificates

Although ACC and Incremental ACC are general proposals not tied to any particular programming paradigm, our developments for incremental ACC (as well as for the original ACC framework [4]) are formalized in the context of (C)LP. Very briefly, a *constraint* is essentially a conjunction of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and t_i are terms. A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H :- D where H, the *head*, is an atom and D, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program* $P \in Prog$, or *program*, is a finite set of rules. Program rules are assumed to be normalized: only distinct variables are allowed to occur as arguments to atoms. Furthermore, we require that each rule defining a predicate p has identical sequence of variables $x_{p_1}, \ldots x_{p_n}$ in the head atom, i.e., $p(x_{p_1}, \ldots x_{p_n})$. We call this the *base form* of p. This is not restrictive since programs can always be normalized, and it will facilitate the presentation of the checking algorithms.

3.1 The Notion of Full Certificate

For concreteness, we rely on an abstract interpretation-based analysis algorithm in the style of the generic analyzer of [7]. This goal-dependent analysis algorithm,

which we refer to as analyzer, given a program P and abstract domain D_α, receives a set $Q_\alpha \in AAtom$ of Abstract Atoms (or *call patterns*) and constructs an *analysis graph* [5] for Q_α. The elements of Q_α are pairs of the form $A : CP$ where A is a procedure descriptor and CP is an abstract substitution (i.e., a condition of the run-time bindings) of A expressed as $CP \in D_\alpha$.[1] Then, the analysis graph is an abstraction of the (possibly infinite) set of (possibly infinite) trees explored by the concrete execution of initial calls described by Q_α in P. The program analysis graph computed by analyzer(Q_α) for P in D_α can be implicitly represented by means of two data structures, the *answer table* and the *dependency arc table* (which are in fact the result of the analysis algorithm).

- *Answer Table (AT).* Its entries correspond to the *nodes* in the analysis graph. They are of the form $A : CP \mapsto AP$, where A is always an atom in base form. They should be interpreted as "the answer pattern for calls to A satisfying precondition (or call pattern), CP, accomplishes postcondition (or answer pattern), AP." AP and CP are abstract substitutions in D_α.
- *Dependency Arc Table (DAT).* Dependencies correspond to the *arcs* in the analysis graph. The intended meaning of a dependency $A_k : CP \Rightarrow B_{k,i} : CP_1$ associated to a program rule A_k:-$B_{k,1}, \ldots, B_{k,n}$ with $i \in \{1, ..n\}$, is that the answer for $A_k : CP$ depends on the answer for $B_{k,i} : CP_1$, say AP_1. Thus, if AP_1 changes with the update of some rule for $B_{k,i}$ then, the *arc* $A_k : CP \Rightarrow B_{k,i} : CP_1$ must be reprocessed in order to compute the new answer for $A_k : CP$. This is to say that the rule for A_k has to be processed again starting from atom $B_{k,i}$.

All the details and the formalization of the analysis algorithm analyzer can be found in [7]. Certification in ACC [4] consists in using the *complete* set of entries stored in the answer table as certificate. Dependencies are not needed for certificate generation neither for non-incremental checking though they will be fundamental later for incremental certificate checking.

Definition 1 (certificate [4]). *Let $P \in Prog$, $D_\alpha \in ADom$ and $Q_\alpha \in AAtom$. We define* Cert $\in Approx$, *the* certificate *for P and Q_α, as the set of entries stored in the answer table computed by* analyzer(Q_α) *[7] for P in D_α.*

Example 1. The next example shows a piece of a module which contains the following (normalized) program for the naive reversal of a list and uses an implementation of app with several base cases (e.g., added automatically by a partial evaluator [8] for efficiency purposes).

```
(rev1) rev(X,Y) :- X = [ ], Y = [ ].
(rev2) rev(X,Y) :- X = [U|V], rev(V,W), T = [U], app(W,T,Y).
(app1) app(X,Y,Z) :- X = [ ], Y = Z.
(app2) app(X,Y,Z) :- X = [U], Z = [U|Y].
(app3) app(X,Y,Z) :- X = [U,V], Z = [U,V|Y].
(app4) app(X,Y,Z) :- X = [U|V], Z = [U|W], app(V,Y,W).
```

[1] We sometimes omit the subscript α from Q_α when it is clear from the context.

The description domain that we use in our examples is the domain *Pos* of Positive Boolean functions [12]. The key idea in this description is to use implication to capture groundness dependencies. The reading of the function $x \rightarrow y$ is "if the program variable x is (becomes) ground, so is (does) program variable y." For example, the best description of the constraint $\mathtt{f(X,Y)} = \mathtt{f(a,g(U,V))}$ is $\mathtt{X} \wedge (\mathtt{Y} \leftrightarrow (\mathtt{U} \wedge \mathtt{V}))$. Groundness information is of great importance as a safety property in order to verify that (C)LP programs are "well moded" (i.e., arguments are correctly instantiated). The most general description $\top$ does not provide information about any variable. The least general substitution $\bot$ assigns the empty set of values to each variable.

For the analysis of our running example, we consider the calling pattern $\mathtt{rev(X,Y)} : \top$, i.e., no entry information is provided on `X` nor `Y`. $\mathsf{analyzer}(\mathtt{rev(X,Y)} : \top)$ produces **State 0** composed of the following answers and dependencies:

(A_1) $\mathtt{rev(X,Y)} : \top \mapsto \mathtt{X} \leftrightarrow \mathtt{Y}$
(A_2) $\mathtt{app(X,Y,Z)} : \top \mapsto (\mathtt{X} \wedge \mathtt{Y}) \leftrightarrow \mathtt{Z}$
(D_1) $\mathtt{rev(X,Y)} : \top \Rightarrow \mathtt{rev(V,W)} : \top$
(D_2) $\mathtt{rev(X,Y)} : \top \Rightarrow \mathtt{app(W,T,Y)} : \top$
(D_3) $\mathtt{app(X,Y,Z)} : \top \Rightarrow \mathtt{app(V,Y,W)} : \top$

Intuitively, D_2 denotes that the answer for $\mathtt{rev(X,Y)} : \top$ may change if the answer for $\mathtt{app(W,T,Y)} : \top$ changes. In such a case, the second rule for `rev` must be processed again starting from atom $\mathtt{app(W,T,Y)}$ in order to recompute the fixpoint for $\mathtt{rev(X,Y)} : \top$. D_1 and D_3 reflect the recursivity of $\mathtt{rev(X,Y)} : \top$ and $\mathtt{app(W,T,Y)} : \top$, respectively, since they depend on themselves. The detailed steps performed by the algorithm can be found in [7] for the same program without the rules $\mathtt{app}_2$ and $\mathtt{app}_3$. However these rules do not add any further information to the fixpoint computation and the steps performed there still apply to our example. According to Definition 1, the certificate Cert for this example is composed of all entries in the answer table, i.e., A_1 and A_2. □

3.2 The Notion of Incremental Certificate

Given a program P, we define an *update* of P, written as $Upd(P) \in UProg$, as a set of tuples of the form $\langle A, Add(A), Del(A)\rangle$, where $A = p(x_1, \ldots, x_n)$ is an atom in base form, $Add(A)$ is the set of rules which are to be added to P for predicate p[2] and $Del(A)$ is the set of rules which are to be removed from P for predicate p.

When a program is updated, depending on the kind of update, the new certificate for the modified program can be either equal, more or less precise than the original one, or even not comparable. In any case, it appears inefficient to generate, transmit, and check the full certificate Ext_Cert for the updated program U_P defined as $U_P = P \oplus Upd(P)$.[3] Our proposal is that it is possible to submit only the new program update $Upd(P)$ together with the *incremental certificate* Inc_Cert, i.e., the *difference* of Ext_Cert w.r.t. the original Cert.

[2] This includes both the case of addition of new procedures, when p did not exist in P, as well as the extension of additional rules (or functionality) for p, if it existed.

[3] The operator "$\oplus$" applies the update to P and generates $U_P = P \oplus Upd(P)$. This can be implemented by using a program in the spirit of the traditional Unix *patch* command as $\oplus$ operator.

Definition 2 (incremental certificate). *In the conditions of Def. 1, we consider* $Upd(P) \in UProg$. *Let* Cert *be the certificate for* P *and* Q_α. *Let* Ext_Cert *be the certificate for* $P \oplus Upd(P)$ *and* Q_α. *We define* Inc_Cert, *the* incremental certificate *for* $Upd(P)$ *w.r.t.* Cert, *as* Ext_Cert$-$Cert, *where* Ext_Cert$-$Cert *is defined as the set of entries* $B : CP_B \mapsto AP_B \in$ Ext_Cert *such that:*

1. $B : CP_B \mapsto \_ \notin$ Cert *or,*
2. $A : CP_A \mapsto AP_A \in$ Cert, $A : CP_A = B : CP_B$ *and* $AP_A \neq AP_B$ *(modulo renaming).*

The definition of incremental certificate for the particular case of program extensions can be found in [2]. The following example illustrates that updating a program can require the change in the analysis information previously computed for other procedures whose fixpoint is indirectly affected by the updates, although their definitions have not been directly changed.

Example 2. Consider the following new definition for app, which is a specialization of the previous app to concatenate lists of a's of the same length :

$(\texttt{Napp}_1)$ $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) :- \texttt{X} = [\,], \texttt{Y} = [\,], \texttt{Z} = [\,].$
$(\texttt{Napp}_2)$ $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) :- \texttt{X} = [\texttt{a}|\texttt{V}], \texttt{Y} = [\texttt{a}|\texttt{U}], \texttt{Z} = [\texttt{a},\texttt{a}|\texttt{W}], \texttt{app}(\texttt{V},\texttt{U},\texttt{W}).$

The update consists in deleting all rules for app in Ex. 1, and replacing them by $\texttt{Napp}_1$ and $\texttt{Napp}_2$. After running the (incremental) analysis algorithm in [7], the following answer table and dependencies are computed **(State 1)**:

(NA_1) $\texttt{rev}(\texttt{X},\texttt{Y}) : \top \mapsto \texttt{X} \wedge \texttt{Y}$	(ND_1) $\texttt{rev}(\texttt{X},\texttt{Y}) : \top \Rightarrow \texttt{rev}(\texttt{V},\texttt{W}) : \top$
(NA_2) $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) : \top \mapsto \texttt{X} \wedge \texttt{Y} \wedge \texttt{Z}$	(ND_2) $\texttt{rev}(\texttt{X},\texttt{Y}) : \top \Rightarrow \texttt{app}(\texttt{W},\texttt{T},\texttt{Y}) : \texttt{W}$
(NA_3) $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) : \texttt{X} \mapsto \texttt{X} \wedge \texttt{Y} \wedge \texttt{Z}$	(ND_3) $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) : \texttt{X} \Rightarrow \texttt{app}(\texttt{V},\texttt{U},\texttt{W}) : \texttt{V}$

Note that the analysis information has changed because the new definition of app allows inferring that all its arguments are ground upon success (NA_2 and NA_3). This change propagates to the answer of rev and allows inferring that, regardless of the calling pattern, both arguments of rev will be ground on the exit (NA_1). According to Def. 2, the incremental certificate Inc_Cert contains NA_3, as it corresponds to a new calling pattern (by point 1), and also NA_1 and NA_2 since their answers have changed w.r.t. the ones in **State 0** (by point 2). □

Note that in a non incremental framework, the size of certificates can be reduced by using compression techniques as in [3]. This approach is not compatible with the incremental setting we are going to discuss in this paper, because certain information essential for the incremental checker can have been removed by the fixpoint reduction.

4 A Checking Algorithm with Support for Incrementality

In this section, we present a checking algorithm for full certificates which is instrumented with a *Dependency Arc Table* (*DAT* in the following). The DAT stores the dependencies between the atoms in the analysis graph (see Section 3).

```
1: procedure checking(P, Q, Cert, AT_mem, DAT_mem)
2:     AT_mem := ∅;  DAT_mem := ∅;  CP_checked := ∅;
3:     for all A : CP ∈ Q do
4:         process_node(P, A : CP, Cert, AT_mem, DAT_mem, CP_checked);
5:     return Valid;
6: procedure process_node(P, A : CP, Cert, AT_mem, DAT_mem, CP_checked)
7:     if (∃ a renaming σ s.t. σ(A : CP ↦ AP) in Cert) then
8:         add A : CP ↦ σ^{-1}(AP) to AT_mem ;   CP_checked := CP_checked ∪ {A : CP};
9:     else return Error;
10:    process_set_of_rules(P, P|_A, A : CP ↦ σ^{-1}(AP), Cert,
                            AT_mem, DAT_mem, CP_checked);
11: procedure process_set_of_rules(P, R, A : CP ↦ AP, Cert,
                                   AT_mem, DAT_mem, CP_checked)
12:    for all rule A_k ← B_{k,1}, ..., B_{k,n_k} in R do
13:        W := vars(A_k, B_{k,1}, ..., B_{k,n_k});
14:        CP_b := Aextend(CP, vars(B_{k,1}, ..., B_{k,n_k}));
15:        CPR_b := Arestrict(CP_b, B_{k,1});
16:        CP_a := process_rule(P, A : CP, A_k ← B_{k,1}, ..., B_{k,n_k}, W, CP_b, CPR_b, Cert,
                                AT_mem, DAT_mem, CP_checked);
17:        AP_1 := Arestrict(CP_a, vars(A_k)) ; AP_2 := Alub(AP_1, AP);
18:        if (AP <> AP_2) then return Error;
19: procedure process_rule(P, A : CP, A_k ← B_{k,j}, ..., B_{k,n_k}, W, CP_b, CPR_b, Cert,
                           AT_mem, DAT_mem, CP_checked)
20:    for all B_{k,i} in the rule body i = j, ..., n_k do
21:        CP_a := process_arc(P, A : CP, B_{k,i} : CPR_b, CP_b, W, Cert,
                               AT_mem, DAT_mem, CP_checked);
22:        if (i <> n_k) then   CPR_a := Arestrict(CP_a, var(B_{k,i+1}));
23:        CP_b := CP_a;  CPR_b := CPR_a;
24:    return CP_a;
25: procedure process_arc(P, A : CP, B_{k,i} : CPR_b, CP_b, W, Cert,
                          AT_mem, DAT_mem, CP_checked)
26:    if (B_{k,i} is a constraint) then  CP_a := Aadd(B_{k,i}, CP_b);
27:    else
28:        if (∄ a renaming σ s.t. σ(B_{k,i} : CPR_b ↦ AP') in AT_mem) then
29:            process_node (P, B_{k,i} : CPR_b, Cert, AT_mem, DAT_mem, CP_checked);
30:        AP_1 := Aextend (ρ^{-1}(AP), W); where ρ is a renaming s.t.
                                            ρ(B_{k,i} : CPR_b ↦ AP) in AT_mem
31:        CP_a := Aconj (CP_b, AP_1);
32:        add A : CP ⇒ B_{k,i} to DAT_mem;
33:    return CP_a;
```

Fig. 1. Checking with Support for Incrementality (Algorithm 1)

This structure is not required by non incremental checkers [4] but it is fundamental to support an incremental design.

Algorithm 1 presents our checker, which receives as parameters a program P, a set Q of call patterns, the certificate Cert returned by analyzer, and two input/output variables AT_{mem} and DAT_{mem} (initially empty) and constructs a program analysis graph in a single iteration by assuming the fixpoint information in Cert. While the graph is being constructed, the obtained answers are

stored in AT_{mem} and compared with the corresponding fixpoints stored in Cert. If any of the computed answers is not consistent with the certificate (i.e., it is greater than the fixpoint), the certificate is considered invalid and the program is rejected. Otherwise, Cert gets checked. The checker returns the reconstructed answer table AT_{mem} and the set of dependencies DAT_{mem} which have been traversed. A detailed explanation of this algorithm can be found in [2] (where only program extensions are considered and the parameter $CP_{checked}$ is not needed). Algorithm 1 is parametric w.r.t. the abstract domain of interest D_α and it is hence defined in terms of five abstract operations on D_α:

- $\mathsf{Arestrict}(CP, V)$ performs the abstract restriction of a description CP to the set of variables in the set V, denoted $vars(V)$;
- $\mathsf{Aextend}(CP, V)$ extends the description CP to the variables in the set V;
- $\mathsf{Aadd}(C, CP)$ performs the abstract operation of conjoining the constraint C with the description CP;
- $\mathsf{Aconj}(CP_1, CP_2)$ performs the abstract conjunction of two descriptions;
- $\mathsf{Alub}(CP_1, CP_2)$ performs the abstract disjunction of two descriptions.

Example 3. The abstract operations for the domain Pos (Ex. 1) are:

$\mathsf{Arestrict}(CP, V) = \exists_{-V} CP$ $\quad$ $\mathsf{Aconj}(CP_1, CP_2) = CP_1 \wedge CP_2$
$\mathsf{Alub}(CP_1, CP_2) = CP_1 \sqcup CP_2$ $\quad$ $\mathsf{Aextend}(CP, V) = CP$
$\mathsf{Aadd}(C, CP) = \alpha_{Def}(C) \wedge CP$ $\quad$ $\alpha_{Def}(X = t) = (X \leftrightarrow \bigwedge\{Y \in vars(t)\})$

where $\exists_{-V} F$ represents $\exists v_1, \ldots, v_n F$, $\{v_1, \ldots, v_n\} = vars(F) - V$, and $\sqcup$ is the least upper bound (lub) operation over the Pos lattice. For instance, $\mathsf{Aconj}(X, Y \leftrightarrow (X \wedge Z)) = X \wedge (Y \leftrightarrow Z)$. $\mathsf{Aadd}(X = [U|V], Y) = (X \leftrightarrow (U \wedge V)) \wedge Y$. $\mathsf{Alub}(X, Y) = X \vee Y$. As an example of checking, we illustrate the steps carried out by the checker to validate the rules $\mathtt{app}_1$ and $\mathtt{app}_4$ of Ex. 1 w.r.t. a certificate Cert composed of the entry A_2. We take as call pattern $\mathtt{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) : \top$. Consider the call to procedure process_node for $\mathtt{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) : \top$. The entry A_2 is added (L8) to AT_{mem} (initially empty), and $\mathtt{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) : \top$ is marked as checked by inserting it in $CP_{checked}$. A call to process_set_of_rules is generated for the call at hand w.r.t. $\mathtt{app}_1$ and $\mathtt{app}_4$ (L10). Consider the processing of the two rules.

1. The call to process_rule for $\mathtt{app}_1$ (L16) executes process_arc (L21) for each of the two constraints in the body. The final answer $CP_a \equiv \mathtt{X} \wedge (\mathtt{Y} \leftrightarrow \mathtt{Z})$ (L16) for $\mathtt{app}_1$ is built up from the abstract conjunction (L31) between $\mathtt{X}$ (partial answer from first constraint) and $\mathtt{Y} \leftrightarrow \mathtt{Z}$ (from second constraint). Since the least upper bound (L17) between CP_a and the answer A_2 is A_2, then no Error is issued (L18) and the first rule $\mathtt{app}_1$ gets successfully checked.
2. As before, the call to process_rule for $\mathtt{app}_4$ executes process_arc for the first two constraints and computes as (partial) solution $CP_a \equiv (\mathtt{X} \leftrightarrow (\mathtt{U} \wedge \mathtt{V})) \wedge (\mathtt{Z} \leftrightarrow (\mathtt{U} \wedge \mathtt{W}))$ (L21). Since we are not in the last atom of the rule (L22), CP_a is restricted to the variables in $\mathtt{app}(\mathtt{V}, \mathtt{Y}, \mathtt{W})$, giving as result $CPR_a \equiv \top$. Now, the next call to process_arc for the rightmost body atom $\mathtt{app}(\mathtt{V}, \mathtt{Y}, \mathtt{W}) : \top$ computes as final solution $(\mathtt{X} \leftrightarrow (\mathtt{U} \wedge \mathtt{V})) \wedge (\mathtt{Z} \leftrightarrow (\mathtt{U} \wedge \mathtt{W})) \wedge (\mathtt{V} \wedge (\mathtt{Y} \leftrightarrow \mathtt{W}))$, which is simplified to A_2. The corresponding dependency is stored in DAT_{mem}.

Thus, the call to process_rule for app_4 computes as solution A_2 (L16), the same answer stored in Cert, and no Error is issued (L18).

Therefore, both rules have been successfully checked in one pass over them and the checker returns Valid. □

In order to support an incremental extension, the final values of the data structures AT_{mem}, DAT_{mem} and P must be available after the end of the execution of the checker. We denote by $AT_{persist}$, $DAT_{persist}$ and $P_{persist}$ the copy in persistent memory (i.e., in disk) of such structures.

Definition 3 (checker). *We define function* CHECKER*:*$Prog \times Approx \times AAtom \times ADom \mapsto$ *boolean which takes a program* $P \in Prog$ *and its certificate* Cert $\in Approx$ *for* $Q_\alpha \in AAtom$ *in* $D_\alpha \in ADom$ *and it returns the result of* checking(P, Q_α, Cert, AT_{mem}, DAT_{mem})*. If it does not issue an* Error*, then it stores in memory* $AT_{persist} := AT_{mem}$, $DAT_{persist} := DAT_{mem}$ *and* $P_{persist} := P$.

5 Incremental Checking

In this section, we propose an incremental checking algorithm which deals with all possible updates over a program in a unified form. The basic idea is that the task performed by an incremental checker has to be optimized such that it only: a) rechecks the part of the abstraction for the procedures which have been directly affected by an update and, b) propagates and rechecks the indirect effect of these changes. In order to do this, we will take as starting point the checker in Algorithm 1. Its DAT will allow the incremental algorithm to propagate the changes and carry out the process in a single pass over the subgraph affected by the updates. Algorithm 2 presents our implementation of this intuition. We start by removing all (possibly incorrect or inaccurate) information *directly* affected by the updates from the answer table and DAT (i.e., the information for the updated procedures) and, then, we check it from scratch against the answers provided in the incremental certificate. If the "direct" checking succeeds, we proceed to check the information *indirectly* affected by such changes in a similar way (i.e., delete the information for them from answer and DAT and recheck it from scratch). This iterative process successfully finishes when all directly and indirectly affected information gets checked. Otherwise, an Error is issued.

The incremental checker is defined as follows: replace the procedure checking by the new procedure incremental_checking in Algorithm 2 and use the remaining procedures defined in Algorithm 1. Below we enumerate the points which should be done in a way or another in any incremental checking algorithm beyond the analysis of logic programs.

1. *Retrieve stored data.* After checking the original package, the structures $AT_{persist}$, $DAT_{persist}$ and the program $P_{persist}$ have been stored in persistent memory (see Definition 3). Our checker retrieves such stored data and initializes, respectively, the parameters AT_{mem}, DAT_{mem} and P with them.

```
 1: procedure incremental_checking(P, Upd(P), Inc_Cert, AT_mem, DAT_mem)
 2:     P_mem := P ⊕ Upd(P);  update_answer_table(AT_mem, Inc_Cert);
 3:     call_patterns_to_check(Upd(P), AT_mem, CP_tocheck);
 4:     CP_checked := ∅;        % call patterns already checked
 5:     check_affected_entries(P_mem, Inc_Cert, AT_mem, DAT_mem, CP_tocheck, CP_checked);
 6:     return Valid;
 7: procedure update_answer_table(AT_mem, Inc_Cert)
 8:     for all entry A : CP ↦ AP in AT_mem do
 9:         if (∃ A : CP ↦ AP_A in Inc_Cert and AP ≠ AP_A (modulo renaming))
               then replace entry for A : CP ↦ AP in AT_mem by A : CP ↦ AP_A;
10: procedure call_patterns_to_check(Upd(P), AT_mem, CP_tocheck)
11:     CP_tocheck := ∅;          % call patterns required to be checked
12:     for all entry A : CP ↦ _ ∈ AT_mem do
13:         if A is updated in Upd(P) then  CP_tocheck := CP_tocheck ∪ {A : CP};
14: procedure check_affected_entries(P_mem, Inc_Cert, AT_mem, DAT_mem,
                                     CP_tocheck, CP_checked)
15:     while CP_tocheck ! = ∅ do
16:         select A : CP from CP_tocheck;
17:         remove_previous_info(A : CP, AT_mem, DAT_mem);
18:         if A : CP ∉ Inc_Cert then
19:             let A : CP ↦ AP the entry for A : CP in AT_mem;
20:             Inc_Cert = Inc_Cert ∪ {A : CP ↦ AP}; propagate := false;
21:         else propagate := true;
22:         process_node(P_mem, A : CP, Inc_Cert, AT_mem, DAT_mem, CP_checked);
23:         CP_tocheck := CP_tocheck − CP_checked;
24:         if propagate then  propagate_effects(A : CP, DAT_mem,
                                                 CP_tocheck, CP_checked);
25: procedure remove_previous_info(A : CP, AT_mem, DAT_mem)
26:     remove entry for A : CP from AT_mem;
27:     remove from DAT_mem all dependencies of the form A : CP ⇒ _ ;
28: procedure propagate_effects(A : CP, DAT_mem, CP_tocheck, CP_checked)
29:     for all B : CP_B ⇒ A : CP ∈ DAT_mem do
30:         if B : CP_B ∉ CP_checked ∪ CP_tocheck then
31:             CP_tocheck := CP_tocheck ∪ {B : CP_B};
```

Fig. 2. Incremental Checking (Algorithm 2)

2. *Update program and answer table.* Prior to proceeding with the proper checking, the incoming updates $Upd(P)$ are applied (by means of the operator $\oplus$) to P in order to generate P_{mem} (L2). Also, the procedure update_answer_table updates the answers for those call patterns in AT_{mem} which have a different answer in Inc_Cert (L8-9). The new entries not yet present in AT_{mem} will be asserted upon request, as in the usual checking process (L8 of Algorithm 1).
3. *Initialize call patterns to check.* The procedure call_patterns_to_check initializes the set $CP_{tocheck}$ with those call patterns with an entry in AT_{mem} which correspond to a rule directly affected by an update (L12-13). During the execution of the checker, the set $CP_{tocheck}$ will be dynamically extended to include the additional call patterns whose checking is indirectly affected by the propagation of changes (L31).
4. *Check affected procedures.* Procedure check_affected_entries launches the checking of all procedures affected by the updates, i.e., the call patterns in $CP_{tocheck} - CP_{checked}$. The set $CP_{checked}$ is used to avoid rechecking the same call pat-

tern more than once, if it appears several times in the analysis subgraph to be checked. Three actions are taken in order to check a call pattern:[4] remove its analysis information (L17), proceed to check it by calling process_node of Algorithm 1 (L22) and, propagate the effects of type b) if needed (L24). We only propagate effects if the answer provided in Inc_Cert for the call pattern at hand is different from that originally stored in $AT_{persist}$ (L21). As a technical detail, in L20, we add to Inc_Cert the information which, although has not changed w.r.t. AT_{mem}, needs to be checked and, therefore, it must be available in Inc_Cert (or process_node would issue an error in L9 of Algorithm 1).

5. *Remove previous analysis information.* Before proceeding with the checking, we need to get rid of previous (possibly incorrect or inaccurate) analysis information. Procedure remove_previous_info eliminates the entry to be checked from AT_{mem} (L26) and all its dependencies from DAT_{mem} (L27).
6. *Propagate effects.* After processing the updated rules, the procedure propagate_effects introduces in the set $CP_{tocheck}$ (L31) the calling patterns whose answer depends on the updated one, i.e., those which are indirectly affected by the updates. Their checking will be later required in L15.
7. *Store data.* Upon return, the checker has to store the computed AT_{mem}, DAT_{mem} and P_{mem}, respectively, in $AT_{persist}$, $DAT_{persist}$, and $P_{persist}$ for achieving a compositional design of our incremental approach.

Definition 4 (incremental checker). *We define function* INCR_CHECKER*:* $UProg \times Approx \times \mapsto$ *boolean which takes* $Upd(P) \in UProg$ *and its incremental certificate* Inc_Cert $\in Approx$ *and 1) it retrieves from memory* $AT_{mem} := AT_{persist}$, $DAT_{mem} := DAT_{persist}$ *and* $P := P_{persist}$ *and 2) it returns the result of* incremental_checking(P, $Upd(P)$, Inc_Cert, AT_{mem}, DAT_{mem}) *for* P. *If it does not issue an* Error, *then it stores* $AT_{persist} := AT_{mem}$, $DAT_{persist} := DAT_{mem}$ *and* $P_{persist} := P_{mem}$.

Note that the safety policy has to be tested w.r.t. the answer table for the extended program. Therefore, the checker has reconstructed, from Inc_Cert, the answer table returned by analyzer for the extended program, Ext_Cert, in order to test for adherence to the safety policy –Equation (4), i.e., $AT_{persist} \equiv$ Ext_Cert.

The following example illustrates a situation in which the task performed by the incremental checker is optimized to only check a part of the abstraction.

Example 4. Consider the deletion of rules $\mathtt{app}_2$ and $\mathtt{app}_3$ of Example 1. The analysis algorithm of [7] returns the same state **(State 0)** since the eliminated rules do not affect the fixpoint result, i.e., they do not add any further information. Thus, the incremental certificate Inc_Cert associated to such an update is empty. The checking algorithm proceeds as follows. Initially, AT_{mem} and DAT_{mem} are initialized with the values in **State 0**. P_{mem} is composed of the rules $\mathtt{rev}_1$, $\mathtt{rev}_2$,

[4] Note that an updated rule which does not match any entry in AT_{mem} does not need to be processed by now. Its processing may be required by some other new rule or they can simply not be affected by the checking process.

app_1 and app_4. Procedure update_answer_table (L2) does not modify AT_{mem}. The execution of procedure call_patterns_to_check (L3) adds $E_1 \equiv \mathsf{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top$ to $CP_{tocheck}$. Procedure check_affected_entries selects E_1 from $CP_{tocheck}$. The next call to remove_previous_info (L17) removes A_2 from AT_{mem} and D_3 from DAT_{mem}. It then inserts A_2 in Inc_Cert. The variable "*propagate*" takes the value *false*. We now jump to the non incremental checking with a call to procedure process_node (L22). This process corresponds exactly to the checking illustrated in Example 3. Upon return from process_node (since variable "*propagate*" is *false*), no effects have to be propagated.

The important point to note is that the incremental checker has not had to recheck the rules for `rev` since its answer is not affected by the deletion. Once Inc_Cert has been validated, the consumer memoizes AT_{mem}, DAT_{mem} (which are those of **State 0**) and P_{mem} in disk. □

Our second example is intended to show how to propagate the effect of a change to the part of the analysis graph affected by such update.

Example 5. Let us illustrate the checking process carried out to validate the update proposed in Example 2 with an incremental certificate, Inc_Cert, which contains the entries NA_1, NA_2 and NA_3. The incremental checker retrieves **State 0** from disk. Next, procedure update_answer_table returns as new AT_{mem} the entries NA_1 and NA_2 which replace the old entries A_1 and A_2, respectively. Then, the set $CP_{tocheck}$ is initialized with $E_1 \equiv \mathsf{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \top$. Procedure check_affected_entries first executes remove_previous_info, which eliminates E_1 from AT_{mem} and dependency D_3 from DAT_{mem}. Moreover, the variable "*propagate*" is initialized to *true*. This annotates that effects have to be propagated later. The execution of process_node for E_1 succeeds and adds the dependency D_3 to DAT_{mem} and the set $CP_{checked}$ is returned with E_1 marked as checked. Upon return, since the variable "*propagate*" is *true*, a call to propagate_effects is generated which forces the checking of `rev`. After inspecting D_2 and D_3 (the two dependencies for E_1), only the entry $E_2 \equiv \mathtt{rev}(\mathtt{X},\mathtt{Y}) : \top$ is added to $CP_{tocheck}$. The dependency for D_3 will not be checked because E_1 has been already processed (hence, it belongs to $CP_{checked}$). Now, procedure check_affected_entries takes E_2 from $CP_{tocheck}$, and similarly to the previous case, successfully executes process_node, and replaces D_2 by ND_2. During the checking of rule $\mathtt{rev}_2$, a new call to process_node is generated for $E_3 \equiv \mathsf{app}(\mathtt{X},\mathtt{Y},\mathtt{Z}) : \mathtt{X}$ which introduces E_3 in $CP_{checked}$, and replaces the dependency D_3 in DAT_{mem} by the new one ND_3 of Example 2. Upon return, since the variable "*propagate*" is *true*, a call to propagate_effects is generated from it. But the affected dependency D_1 is not processed because E_1 was processed already and belongs to $CP_{checked}$. The conclusion is that a single pass has been performed on the three provided entries in order to validate the certificate. □

The following theorem establishes the correctness of incremental checking. The proof can be found in [1].

Theorem 1 (correctness). *Let $P \in Prog$, $Upd(P) \in UProg$, $D_\alpha \in ADom$ and $Q_\alpha \in AAtom$. Let* Cert *be the certificate for P and Q_α,* Ext_Cert *the cer-*

tificate for $P \oplus Upd(P)$ *and* Q_α *and* Inc_Cert *the incremental certificate for* $Upd(P)$ *w.r.t.* Cert. *If* INCR_CHECKER*(*$Upd(P)$, Inc_Cert*) does not issue an* Error*, then the validation of* Inc_Cert *is done in a single pass over* Inc_Cert *and* $AT_{persist} \equiv AT_{mem}$*,* $DAT_{persist} \equiv DAT_{mem}$*, where* AT_{mem} *and* DAT_{mem} *are, respectively, the answer table and DAT returned by* checking*(*$P \oplus Upd(p), Q_\alpha$*,* Ext_Cert*,* AT_{mem}, DAT_{mem}*).*

Efforts for coming up with incremental approaches are known in the context of program analysis (see [17, 7, 14, 15]) and program verification (see [18, 9, 16]). Our work is more closely related to incremental program analysis, although the design of our incremental checking algorithm is notably different from the design of an incremental analyzer (like the ones in [7, 14]). In particular, the treatment of deletions and arbitrary changes is completely different. In our case, we can take advantage of the information provided in the certificate in order to avoid the need to compute the strongly connected components (see [7]). This was necessary in the analyzer in order to ensure the correctness of the incremental algorithm. Unlike [7, 14], we have integrated in a single algorithm all incremental updates over a program in a seamless way. In [2], we have identified the particular optimization for the addition of rules to a program.

6 Conclusions

Our approach to incremental ACC aims at reducing the size of certificates and the checking time when a supplier provides an untrusted update of a (previously) validated package. Essentially, when a program is subject to an update, the incremental certificate we propose contains only the *difference* between the original certificate for the initial program and the new certificate for the updated one. Checking time is reduced by traversing only those parts of the abstraction which are affected by the changes rather than the whole abstraction. An important point to note is that our incremental approach requires the original certificate and the dependency arc table to be stored on the consumer side for upcoming updates. The appropriateness of using the incremental approach will therefore depend on the particular features of the consumer system and the frequency of software updates. In general, our approach seems to be more suitable when the consumer prefers to minimize as much as possible the waiting time for receiving and validating the certificate while storage requirements are not scarce. We believe that, in everyday practice, time-consuming safety tests would be avoided by many users, while they would probably accept to store the safety certificate and dependencies associated to the package. We are now in the process of extending the ACC implementation already available in the `CiaoPP` system to support incrementality. Our preliminary results in certificate reduction are very promising. We expect optimizations in the checking time similar to those achieved in the case of incremental analysis (see, e.g., [7]).

References

1. E. Albert, P. Arenas, and G. Puebla. An Incremental Approach to Abstraction-Carrying Code. Technical Report CLIP3/2006, Technical University of Madrid (UPM), School of Computer Science, UPM, March 2006.
2. E. Albert, P. Arenas, and G. Puebla. Incremental Certificates and Checkers for Abstraction-Carrying Code. In *Proc. of WITS 2006*, March 2006.
3. E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Reduced Certificates for Abstraction-Carrying Code. In *Proc. of ICLP 2006*, Springer LNCS. To appear.
4. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, Springer LNAI 3452, pp. 380–397, 2005.
5. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
6. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL 1977*, ACM, pp.238–252, 1977.
7. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
8. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
9. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer LNCS 2031, pp. 98–112, 2001.
10. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
11. Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
12. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(4):181–196, 1993.
13. G. Necula. Proof-Carrying Code. In *Proc. of POPL 1997*, pp. 106–119. ACM Press, 1997.
14. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *Proc. SAS'96*, Springer LNCS 1145,pp. 270–284, 1996.
15. B. Ryder. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, 1988.
16. O.V. Sokolsky and S.A. Smolka. Incremental model checking in the modal μ-calculus. In *Computer Aided Verification, Proc. 6th International Conference*, Springer LNCS 818, pp. 351–363, 1994.
17. Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proc. PLDI'97*, pp. 31–43, 1997.
18. M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode Analysis for Proof Carrying Code. In *Proc. Bytecode'05*, ENTCS 141, pp. 19–34. Elsevier, 2005.