Some Issues on Incremental 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. The abstraction thus plays the role of safety certificate and its generation (and validation) is carried out automatically by a fixed-point analyzer. 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 this abstract, we overview the main issues on *incremental* ACC. In particular, in the context of logic programming, we discuss both the generation of incremental certificates and the design of an incremental checking algorithm for untrusted *update*s of a (trusted) program, i.e., when a producer provides a modified version of a previously validated program. By update, we refer to any arbitrary change on a program, i.e., the extension of the program with new procedures, the deletion of existing procedures and the replacement of existing procedures by new versions for them. We also discuss how each kind of update affects the incremental extension in terms of accuracy and correctness.

1 Introduction

Proof-Carrying Code (PCC) [**?**] 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 the level of proving to the level of checking, a task which should be much simpler, efficient, and automatic than generating the original certificate.

Abstraction-carrying code (ACC) [**?**] 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. Lightweight bytecode verification [?] is another PCC method which relies on analysis techniques (namely on type analysis in the style of those used for Java bytecode verification [?]) to generate and check certificates in the context of the Java Card language. 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.

Existing 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, in a non incremental way. However, there are situations which are not well suited to this simple model and which instead require only rechecking certain parts of the analysis graph which has already been validated. 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. We characterize the different kind of updates, or modifications over a program. In particular, we include:

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 discuss the influence of the different kinds of updates on an *incremental* extension to PCC in terms of correctness and efficiency. We also outline the main issues on the generation of incremental certificates and the design of incremental checkers.

The paper is organized as follows. Section 2 introduces briefly some notation and preliminary notions on abstract interpretation and ACC. In Section 3, we present a general view of incremental ACC. In Section 4 we describe the different kinds of updates over a program and the way they affect the certification and checking phases. Section 5 reviews the notion of full certificate and proposes the use of incremental certificate. In Section 6, we discuss the extensions needed on a non-incremental checking algorithm in order to support incrementality and we sketch the new tasks of an incremental checking algorithm. Finally, Section 7 concludes.

2 Abstraction-Carrying Code

Our work relies on the abstract interpretation-based analysis algorithm of [**?**] for (Constraint) Logic Programming, (C)LP. We assume some familiarity with abstract interpretation (see [**?**]), (C)LP (see, e.g., [**?**, **?**]) and PCC [**?**].

Very briefly, *terms* are constructed from variables (e.g., x), *functors* (e.g., f) and *predicates* (e.g., p). We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ, where $x_i \neq x_j$, if $i \neq j$, and t_i are terms. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} \equiv \rho^{-1}\rho \equiv id$. A *constraint* is a conjunction of expressions built from predefined predicates (such as 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 *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.

An abstract interpretation-based certifier is a function CERTIFIER: $Prog \times ADom \times AInt \mapsto ACert$ which for a given program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and an abstract safety policy $I_\alpha \in AInt$ generates an abstract certificate $Cert_\alpha \in ACert$, 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. In the following, using the same subscript α, we denote that I_α and $Cert_\alpha$ are specifications given as abstract semantic values of D_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* $\langle D_\alpha, \sqsubseteq\rangle \in ADom$ 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 $\alpha : 2^D \rightarrow D_\alpha$, and a *concretization* function $\gamma : D_\alpha \rightarrow 2^D$. 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 [**?**]

$$\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$$

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 $[\![P]\!]_\alpha = \mathrm{lfp}(S_P^\alpha)$. The operator S_P^α is the abstract counterpart of S_P.

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

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

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

Note that the certificate can be stricter than the safety specification and it is only required that I_α is implied by $Cert_\alpha$.
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* × *ADom* × *ACert* ↦ *bool* which for a program $P \in \mathit{Prog}$, an abstract domain $D_\alpha \in \mathit{ADom}$ and certificate $Cert_\alpha \in \mathit{ACert}$ checks whether $Cert_\alpha$ is a fixpoint of S_P^α or not:

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

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 A General View of Incremental ACC

Figures 1 and 2 present a general view of the incremental certification and incremental checking processes respectively. In Figure 1, the producer starts from an Updated Program, U_P, w.r.t. a previously certified Program, P. It first retrieves from disk P and its certificate, Cert, computed in the previous certification phase.

Next, the process "$\ominus$" compares both programs and returns the differences between them, $Upd(P)$, i.e, the program Updates which applied to P results in U_P, written as $Upd(P) = U_P \ominus P$. Note that, from an implementation perspective, a program update should contain both the new updates to be applied to the program and instructions on where to place and remove such new code. This can be easily done by using the traditional Unix *diff* format for coding program updates. An Incremental Certifier generates from Cert, P and $Upd(P)$ an incremental certificate, Inc_Cert, which can be used by the consumer to validate the new updates. The package "$Upd(P)$+Inc_Cert" is submitted to the code consumer. Finally, in order to have a compositional incremental approach, the producer has to update the original certificate and program with the new updates. Thus, the resulting Ext_Cert and U_P are stored in disk replacing Cert and P, respectively.

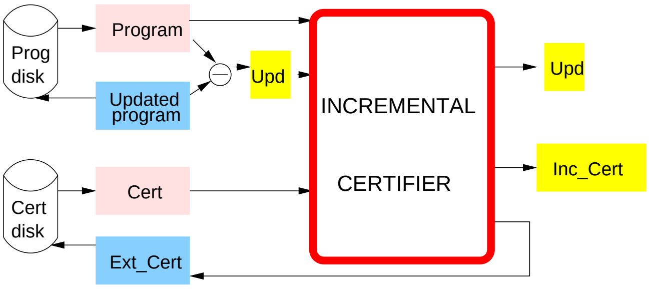

Fig. 1. Incremental Certification in Abstraction-Carrying Code

In Figure 2, the consumer receives the untrusted package. In order to validate the incoming update w.r.t. the provided (incremental) certificate, it first retrieves P and Cert from disk. Next, it reconstructs the updated program by using an operator "$\oplus$" which applies the update to P and generates $U_P = P \oplus Upd(P)$. This can implemented by using a program in the spirit of the traditional Unix *patch* command as $\oplus$ operator. An Incremental Checker now efficiently validates the new modification by using the stored data and the incoming incremental certificate. If the validation succeeds (returns ok), the checker will have reconstructed the full certificate. As before, the updated program and extended certificate are stored in disk (superseding the previous versions) for future (incremental) updates. In order to simplify our scheme, we assume that the safety policy and the generation of the verification condition [?] are embedded within the certifier and checker. However, in an incremental approach, producer and consumer could perfectly agree on a new safety policy to be implied by the modification.

It should be noted that this does not affect our incremental approach and the verification condition would be generated exactly as in non incremental PCC.

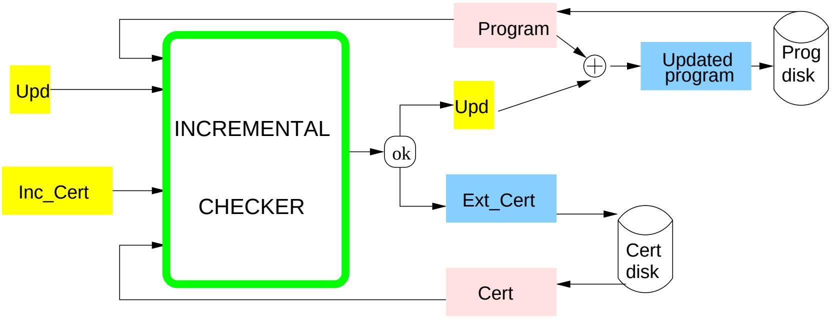

Fig. 2. Incremental Checking in Abstraction-Carrying Code

4 Characterization of Updates

Let us now characterize the types of updates we consider and how they can be dealt within the ACC scheme in the context of logic programming. Given a program P, we define an *update* of P, written as $Upd(P)$, 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 and:

- $Add(A)$ is the set of rules which are to be added to P for predicate p. This includes both the case of addition of new predicates, when p did not exist in P, as well as the extension of additional rules (or functionality) for p, if it existed.
- $Del(A)$ is the set of rules which are to be removed from P for predicate p.

Note that, for the sake of simplicity, we do not include the instructions on where to place and remove such code in the formalization of our method. We distinguish three classes of updates:

- the *addition* of predicates occurs when $\forall A,\ Del(A) = \emptyset\ \wedge\ \exists A,\ Add(A) \neq \emptyset$,
- the *deletion* of predicates occurs if $\forall A,\ Add(A) = \emptyset\ \wedge\ \exists A,\ Del(A) \neq \emptyset$ and
- the remaining cases are considered *arbitrary changes*.

Addition of Procedures. When a program P is extended with new predicates or new clauses for existing predicates, the original certificate $Cert_\alpha$ is not guaranteed to be a fixpoint any longer, because the contribution of the new rules can lead to a more general answer. Consider P^{add} the program after applying some additions and $Cert_\alpha^{add}$ the certificate computed from scratch for P^{add}. Then, $Cert_\alpha \sqsubseteq Cert_\alpha^{add}$. This means that $Cert_\alpha$ is no longer valid. Therefore, we need to perform the least upper bound (*lub*) of the contribution of the new rules and submit, together with the extension, the new certificate $Cert_\alpha^{add}$ (or the difference of both certificates). The consumer will thus test the safety policy w.r.t. $Cert_\alpha^{add}$. Consider the abstract operation $\mathsf{Alub}(CP_1, CP_2)$ which performs the abstract disjunction of two descriptions. Then, we define $Cert_\alpha^{add} = \mathsf{Alub}(Cert_\alpha, [\![P^{add}]\!]_\alpha)$ and submit the incremental certificate Cert which is defined as the (abstract) difference $Cert_\alpha^{add} - Cert_\alpha$. The notion of incremental certificate is the issue of Section 5.

Deletion of Procedures. The first thing to note is that in order to entail the safety policy, unlike extensions over the program, we need not change the certificate at all when some predicates are deleted. Consider P^{del} the program after applying some deletions and $Cert_\alpha^{del}$ the certificate computed from scratch for P^{del}. The original certificate $Cert_\alpha$ is trivially guaranteed to be a fixpoint (hence a correct certificate), because the contribution of the rules was conjoined (by computing the lub) to give $Cert_\alpha$ and so it still correctly describes the contribution of each remaining rule. By applying Equation 2, $Cert_\alpha$ is still valid for P^{del} w.r.t. I_α since $Cert_\alpha \sqsubseteq I_\alpha$. Therefore, more accuracy is not needed to ensure compliance with the safety policy. This suggests that the incremental certificate can be empty and the checking process does not have to check any predicate. However, it can happen that a new, more precise, safety policy is agreed by the consumer and producer. Also, this accuracy could be required in a later modification. Although $Cert_\alpha$ is a correct certificate, it is possibly less *accurate* than $Cert_\alpha^{del}$, i.e., $Cert_\alpha^{del} \sqsubseteq Cert_\alpha$. It is therefore interesting to define the corresponding incremental algorithm for reconstructing $Cert_\alpha^{del}$ and checking the deletions and the propagation of their effects.

Arbitrary Changes. The case of arbitrary changes considers that rules can both be deleted from and added to an already validated program. In this case, the new certificate for the modified program can be either equal, more or less precise than the original one, or even not comparable. Imagine that an arbitrary change replaces a rule R_a, which contributes to a fixpoint $Cert_\alpha^a$, with a new one R_b which contributes to a fixpoint $Cert_\alpha^b$ such that $Cert_\alpha^{ab} = \mathsf{Alub}(Cert_\alpha^a, Cert_\alpha^b)$ and $Cert_\alpha^a \sqsubset Cert_\alpha^{ab}$ and $Cert_\alpha^b \sqsubset Cert_\alpha^{ab}$. The point is that we cannot just compute an approximation of the new rule and compute the lub with to the previous fixpoint, i.e., we cannot use $Cert_\alpha^{ab}$ as certificate and have to provide the more accurate $Cert_\alpha^b$. The reason is that it might be possible to attest the safety policy by independently using $Cert_\alpha^a$ and $Cert_\alpha^b$ while it cannot be implied by using their lub $Cert_\alpha^{ab}$. This happens for certain safety policies which contain disjunctions, i.e., $Cert_\alpha^a \vee Cert_\alpha^b$ does not correspond to their lub $Cert_\alpha^{ab}$. Therefore, arbitrary changes require a precise recomputation of the new fixpoint and its proper checking.

As a practical remark, an arbitrary update can be decomposed into an addition and a deletion and then handled as the first cases. We have pointed out the difference because correctness and accuracy requirements are different in each particular case, as we have seen above.

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 the standard implementation of app for appending lists:

$$P_0 \equiv \begin{cases} (\mathtt{rev}_1)\ \mathtt{rev(X,Y)} :- \mathtt{X} = [\ ],\ \mathtt{Y} = [\ ]. \\ (\mathtt{rev}_2)\ \mathtt{rev(X,Y)} :- \mathtt{X} = [\mathtt{U}|\mathtt{V}],\ \mathtt{rev(V,W)},\ \mathtt{T} = [\mathtt{U}],\ \mathtt{app(W,T,Y)}. \\ (\mathtt{app}_1)\ \mathtt{app(X,Y,Z)} :- \mathtt{X} = [\ ],\ \mathtt{Y} = \mathtt{Z}. \\ (\mathtt{app}_4)\ \mathtt{app(X,Y,Z)} :- \mathtt{X} = [\mathtt{U}|\mathtt{V}],\ \mathtt{Z} = [\mathtt{U}|\mathtt{W}],\ \mathtt{app(V,Y,W)}. \end{cases}$$

Suppose now that the consumer modifies P_0 introducing two more base cases for app (e.g., added automatically by a partial evaluator [?]):

$(\mathtt{app}_2)\ \mathtt{app(X,Y,Z)} :- \mathtt{X} = [\mathtt{U}],\ \mathtt{Z} = [\mathtt{U}|\mathtt{Y}].$
$(\mathtt{app}_3)\ \mathtt{app(X,Y,Z)} :- \mathtt{X} = [\mathtt{U,V}],\ \mathtt{Z} = [\mathtt{U,V}|\mathtt{Y}].$

The producer must send to the consumer the set $Upd(P_0)$, composed of the unique tuple:

$$\langle \mathsf{app}(\mathtt{X,Y,Z}), Add(\mathsf{app}(\mathtt{X,Y,Z})), Del(\mathsf{app}(\mathtt{X,Y,Z}))\rangle$$

where $Add(\mathsf{app}(\mathtt{X,Y,Z})) = \{\mathsf{app}_2, \mathsf{app}_3\}$ and $Del(\mathsf{app}(\mathtt{X,Y,Z})) = \emptyset$, i.e., we are in the case of an *addition* of predicates only. Let us name P_1 to the program resulting from adding rules app_2 and app_3 to P_0. Note that these rules do not add any further information to the program (i.e., the certificate for P_0 and P_1 would remain the same and, as we will see, the *incremental certificate is empty*).

Consider now the following new definition for predicate app which is a specialization of the previous app to concatenate lists of a's of the same length:

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

The update consists in deleting all rules for predicate app in P_1 and replacing them by $\mathtt{Napp}_1$ and $\mathtt{Napp}_2$. Let P_2 be the resulting program. $Upd(P_1)$ is composed again of a unique tuple with the following information:

$Add(\mathsf{app}(\mathtt{X,Y,Z})) = \{\mathtt{Napp}_1, \mathtt{Napp}_2\}$
$Del(\mathsf{app}(\mathtt{X,Y,Z})) = \{\mathsf{app}_1, \mathsf{app}_2, \mathsf{app}_3, \mathsf{app}_4\}$

In this case, we are in the case of an *arbitrary change*, and as we will show in Example 5, the *incremental certificate* will not be empty in this case (since by using the abstract domain *Def* in Example 2, the fixpoint for P_2 will change w.r.t. the one for P_1). □

5 Incremental Certificates

Although ACC and incremental ACC, as outlined above, are general proposals not tied to any particular programming paradigm, our developments for incremental ACC (as well as for the original ACC framework [?]) are formalized in the context of (C)LP. A main idea in ACC [?] is that a *certificate*, Cert, is automatically generated by using the *complete* set of *entries* returned by an abstract fixpoint analysis algorithm. For concreteness, we rely on an abstract interpretation-based analysis algorithm in the style of the generic analyzer of [?].

The analysis algorithm of [?], which we refer to as Analyze, given a program P and an abstract domain D_α, receives a set of *call patterns* $S_\alpha \in AAtom$ of Abstract Atoms (or call patterns) which are a description of the calling modes into the program, and constructs an *analysis graph* [?] for S_α which is an *abstraction* of the (possibly infinite) set of (possibly infinite) trees explored by the concrete execution of initial calls described by S_α in P. A *call pattern* $A : CP \in AAtom$ is composed of an atom in base form, $A \equiv p(X_1, \ldots, X_n)$, and a description in the abstract domain, CP, for A.

The program analysis graph computed by Analyze(S_α) for P in D_α can be implicitly represented by means of two data structures, the *answer table* (AT) and the *dependency arc table* (DAT), which are the output of the algorithm of Analyze. Each element (or *entry*) in the answer table takes the form $A : CP \mapsto AP$ such that, for the atom A, CP is its *call* description and AP its *success* (or answer) description in the abstract domain. Informally, such entry should be interpreted as "the answer pattern for calls to A satisfying precondition CP accomplies postcondition AP". The dependency arc table is not relevant now, although it is fundamental in the design of the incremental checking, as we will see later. All the details and the formalization of the algorithm can be found in [?].

Our proposal for the incremental checking is that, if the consumer keeps the original (fixed-point) abstraction Cert, it is possible to provide only the program updates and the incremental certificate Inc_Cert. Concretely, given:

- an update $Upd(P)$ of P,
- the certificate Cert for P and S_α,
- the certificate Ext_Cert for $P \oplus Upd(P)$ and S_α

we define Inc_Cert, the *incremental certificate* for $Upd(P)$ w.r.t. Cert, as the difference of certificates Ext_Cert and Cert, i.e., the set of entries in Ext_Cert not occurring in Cert. The first obvious advantage is that the size of the transmitted certificate can be considerably reduced. Let us see an example.

Example 2. Consider program P_0 in Example 1. The description domain that we are going to use in our examples is the *definite Boolean functions* [?], denoted Def. 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 $\mathtt{X} \wedge (\mathtt{Y} \leftrightarrow (\mathtt{U} \wedge \mathtt{V}))$. 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 set of calling patterns $S_\alpha = \{\mathtt{rev(X,\ Y)} : \top\}$, i.e., no entry information is provided on X nor Y. Analyze($\{\mathtt{rev(X,Y)} : \top\}$) returns in the answer table, AT, the following entries:

$$(A_1) \quad \mathtt{rev(X,Y)} : \top \mapsto \mathtt{X} \leftrightarrow \mathtt{Y}$$
$$(A_2) \quad \mathtt{app(X,Y,Z)} : \top \mapsto (\mathtt{X} \wedge \mathtt{Y}) \leftrightarrow \mathtt{Z}$$

The certificate Cert for this example is then composed of the entries A_1 and A_2. Consider now the addition of rules app_2 and app_3 in P_0, i.e., program P_1 of Example 1. The analysis algorithm of [?] returns as Ext_Cert the same answer table AT as for P_0, since the added 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. □

6 Incremental Checking

Intuitively, an abstract interpretation-based checking algorithm (like the one in [?]) receives as input a program P, a set of abstract atoms S_α and a certificate Cert 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 an answer table AT_{mem} (initially empty) 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 accepted and $AT_{mem} \equiv$ Cert.

6.1 Checking with Dependencies

In order to define an incremental checking, the checking algorithm in [?] needs to be modified to compute (and store) also the dependencies between the atoms in the answer table. In [?], we have instrumented a checking algorithm for full certificates with a *Dependency Arc Table.* This structure, DAT, is not required by non incremental checkers but it is fundamental to support an incremental design.

The DAT returned by Analyze is composed of arcs (or *dependencies*) of the form $A_k : CP \Rightarrow B_{k,i} : CP'$ associated to a program rule A_k:-$B_{k,1}, \ldots, B_{k,n}$ with $i \in \{1, ..n\}$. The intended meaning of such a dependency is that the answer for $A_k : CP$ depends on the answer for $B_{k,i} : CP'$, say AP. Thus, if AP changes with the update of some rule for $B_{k,i}$ then, the arc $A_k : CP \Rightarrow B_{k,i} : CP'$ 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 $B_{k,i}$, i.e., we do not need to process the part A_k:-$B_{k,1}, \dots, B_{k,i-1}$ because it is not affected by the changes.

In the following, we assume that CHECKER is a non incremental checker such that, if the call CHECKER$(P, S_\alpha, \mathsf{Cert})$ does not fail, then it returns the reconstructed answer table AT_{mem} and the set of dependencies DAT_{mem} which have been generated. In such a case, se say that Cert has been *checked* or *accepted.* By the correctness of the checker [?], the reconstructed structures contain exactly the same data as the answer table and the dependency arc table computed by the analysis algorithm ANALYZE(S_α) for the program P.

Example 3. Consider the program P_0 in Example 1. ANALYZE returns, together with AT, the following dependency arc table:

$$\begin{array}{l}(D_1)\ \mathtt{rev(X,Y)} : \top \Rightarrow \mathtt{rev(V,W)} : \top \\ (D_2)\ \mathtt{rev(X,Y)} : \top \Rightarrow \mathtt{app(W,T,Y)} : \top \\ (D_3)\ \mathtt{app(X,Y,Z)} : \top \Rightarrow \mathtt{app(V,Y,W)} : \top\end{array}$$

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 $\mathtt{rev}_2$ for $\mathtt{rev}$ 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 (rules $\mathtt{rev}_2$ and $\mathtt{app}_4$ respectively). The detailed steps performed by the algorithm can be found in [?]. Note that, the CHECKER executed for the call pattern at hand, computes (and stores) in AT_{mem} the entries A_1, A_2 in Example 2, and, after traversing rules $\mathtt{rev}_2$ and $\mathtt{app}_4$, it stores in DAT_{mem} the dependencies D_1, D_2 and D_3. □

6.2 Additional Tasks of an Incremental Checker

In order to support incrementality, 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. Thus, we denote by $AT_{persist}$, $DAT_{persist}$ and $P_{persist}$ the copy in persistent memory (i.e., in disk) of such structures. Now, we outline in a very general way the additional tasks that an incremental checking algorithm (INC_CHECK in the following) has to perform. The complete code of the algorithm can be found in [?]. It receives as input parameters an update $Upd(P)$ of the original program P, a set of abstract atoms $S_\alpha \in AAtom$ and the incremental certificate Inc_Cert for $Upd(P)$ w.r.t. Cert. The following tasks are carried out by an incremental checker:

Step 1) It retrieves from memory $AT_{mem} := AT_{persist}$, $DAT_{mem} := DAT_{persist}$ and $P := P_{persist}$ (stored in persistent memory in a previous checking phase) and generates $P_{mem} := P \oplus Upd(P)$.

Step 2) It rechecks all entries in AT_{mem} which have been directly affected by an update. Concretely, for each $A : CP \in AT_{mem}$, such that A has an entry in $Upd(P)$, a call to CHECKER$(P \oplus Upd(P), \{A : CP\}, \mathsf{Inc_Cert})$ is generated, marking the entry as *checked* (or *accepted*) $A : CP^{check}$. This guarantees that the incremental checking process is done in one pass (i.e., rules used to check $A : CP$ are traversed at most once).

Step 3) It propagates and rechecks the indirect effect of these changes by inspecting the dependencies in DAT_{mem}. Thus, for all $A : CP^{check} \in \mathsf{Inc\_Cert}$, if there exists a dependency of the form $B : CP_B \Rightarrow A : CP$ (modulo renaming) in DAT_{mem} such that $B : CP_B$ is not marked as checked, then a call to CHECKER($P \oplus \mathit{Upd}(P), \{B : CP_B\}, \mathsf{Inc\_Cert}$) is generated and $B : CP_B$ is marked as checked. This process is repeated until there are no dependencies satisfying the above condition. Note that the condition $A : CP^{check} \in \mathsf{Inc\_Cert}$ ensures that the answer for $A : CP$ has changed w.r.t. Cert. Otherwise nothing has to be done (this will allow us to reduce the checking time w.r.t a full checking process for P and Ext_Cert).

Step 4) If it does not issue an Error then it removes from AT_{mem} those entries corresponding to deleted rules. We can identify them by exploring DAT_{mem}. Concretely, for all $A : CP \in AT_{mem}$, $A : CP \notin S_\alpha$, if there not exists a dependency $B : CP' \Rightarrow A : CP$ in DAT_{mem} then remove $A : CP$ from AT_{mem}.

Step 5) It stores $AT_{persist} := AT_{mem}$, $DAT_{persist} := DAT_{mem}$ and $P_{persist} := P_{mem}$.

Our first example is intended to illustrate a situation in which the task performed by the incremental checker can be optimized such that it only checks a part of the analysis graph.

Example 4. Consider the addition of rules app_2 and app_3 to program P_0, which results in program P_1 (Example 1). As shown in Example 2, the incremental certificate Inc_Cert associated to such an update is empty. The incremental checking algorithm INC_CHECK proceeds as follows:

Step 1) AT_{mem} and DAT_{mem} are initialized with A_1,A_2 (Example 2) and D_1,D_2 and D_3 (Example 3) respectively. $P_{mem} \equiv P_1$.

Step 2) Since $\mathsf{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) : \top \in AT_{mem}$ and $\mathit{Add}(\mathsf{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}))$ is not empty, then a call to CHECKER(P_1,$\{\mathsf{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) : \top\}$,Inc_Cert) is generated in order to ensure that the fixpoint is preserved. Now, $\mathsf{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) : \top$ is marked as checked.

Step 3) No checking has to be done since Inc_Cert is empty.

Step 4) Nothing is done since $\mathsf{app}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) : \top$ occurs at the right hand side of dependency D_3.

Step 5) Finally, once Inc_Cert has been validated, the consumer stores the answer table AT_{mem}, the dependency arc table DAT_{mem} and the program P_{mem} in disk with the same values as in Step 1. □

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

Example 5. The update consists in deleting all rules for predicate `app` in program P_1 of Example 1 (which results in program P_2), and replacing them by $\mathtt{Napp}_1$ and $\mathtt{Napp}_2$. After running the ANALYZE for P_2, the following answer table and dependencies are computed:

$$\begin{array}{ll}
(NA_1) & \mathtt{rev(X,Y)} : \top \mapsto \mathtt{X} \wedge \mathtt{Y} \\
(NA_2) & \mathtt{app(X,Y,Z)} : \top \mapsto \mathtt{X} \wedge \mathtt{Y} \wedge \mathtt{Z} \\
(NA_3) & \mathtt{app(X,Y,Z)} : \mathtt{X} \mapsto \mathtt{X} \wedge \mathtt{Y} \wedge \mathtt{Z} \\
(ND_1) & \mathtt{rev(X,Y)} : \top \Rightarrow \mathtt{rev(V,W)} : \top \\
(ND_2) & \mathtt{rev(X,Y)} : \top \Rightarrow \mathtt{app(W,T,Y)} : \mathtt{W} \\
(ND_3) & \mathtt{app(X,Y,Z)} : \mathtt{X} \Rightarrow \mathtt{app(V,U,W)} : \mathtt{V}
\end{array}$$

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).[1] 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). The incremental certificate Inc_Cert contains NA_3 as it corresponds to a new calling pattern and contains also NA_1 and NA_2 since their answers have changed w.r.t. the ones stored in Cert (Example 2). Let us illustrate the incremental checking process carried out to validate this update.

Step 1) We retrieve from disk the answer table, dependency arc table and the program stored in Step 4 of Example 4. Now $P_{mem} \equiv P_2$.

Step 2) Similar to Step 2 of Example 4, but considering the new rules for `app`.

Step 3) Since we have the dependency $D_2 \in DAT_{mem}$ and $\mathtt{app(X,Y,Z)} : \top \in$ Inc_Cert, a call to CHECKER($\mathtt{rev(X,Y)} : \top$,P_2,Inc_Cert) is generated to ensure that the new fixpoint for $\mathtt{rev(X,Y)} : \top$ is valid. In the checking process, when traversing the rule $\mathtt{rev}_2$, the new call pattern $\mathtt{app(X,Y,Z)} : \mathtt{X}$ occurs and it is also validated by calling to CHECKER. When traversing rule $\mathtt{Napp}_2$, the dependency D_3 is replaced by the new one ND_3 in DAT_{mem}, and the call pattern is marked as checked. Similarly, the dependency D_2 is replaced by the new one ND_2 and $\mathtt{rev(X,Y)} : \top$ is marked as checked. Now, all call patterns have been checked and the process finishes.

Step 4) The entry NA_2 is removed from AT_{mem} since it does not occur at the right hand side of any dependency.

Step 5) The consumer mems the answer table $AT_{mem} := \{NA_1, NA_3\}$, the dependency arc table $DAT_{mem} := \{ND_1, ND_2, ND_3\}$ and the program $P_{mem} := P_2$ in disk. □

[1] Note hat NA_3 is subsumed by NA_2 and we could indeed only submit NA_2. The incremental checking algorithm should be modified to search entries which are equal or more general than the required one.

The definition of the algorithm INC_CHECK can be found in [?], together with the proof of the correctness of the algorithm. Informally, correctness amounts to saying that if INC_CHECK does not issue an error, then it returns as computed answer table the extended certificate Ext_Cert for the updated program. Moreover, we ensure that it does not iterate during the reconstruction of any answer.

7 Conclusions

Our proposal 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 by 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. Nevertheless, there can sometimes be situations where storage resources can be very limited, while runtime resources for performing upcoming checkings could still be sufficient. 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., [?]).

Acknowledgments

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 Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. The authors would like to thank the anonymous referees of WLPE for their useful comments.

References