Incremental Certificates and Checkers for 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 plays thus the role of safety certificate and its generation 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 *original* 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 increments of a (trusted) program, i.e., when a producer provides a new increment of a previously validated program. This increment may not only include new procedures, but also extend the definition of already existing ones. Our proposal is that, if the consumer keeps the original abstraction, it is possible to provide, together with the program increment, only the *difference* of both abstractions, i.e., the incremental certificate. The first obvious advantage is that the size of the transmitted certificate can be considerably reduced. Furthermore, it is now possible to define an *incremental checking* algorithm which, given a program increment and its incremental certificate, only re-checks the fixpoint for each procedure affected by the increment and the propagation of the effect of these fixpoint changes. As a consequence, both certificate transmission time and checking time can be reduced significantly. To the best of our knowledge, this is the first proposal to incremental certificates and incremental checkers for PCC.

1 Introduction

Proof-Carrying Code (PCC) [11] is a general framework 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 that should be much simpler, efficient, and automatic than generating the original certificate.

Abstraction-carrying code (ACC) [2, 6] has been recently proposed as an enabling technology for PCC in which an *abstraction* (or 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 is *automatically* carried out by a fixed-point analyzer. Lightweight bytecode verification [13] is another PCC method which relies on analysis techniques (namely on type analyses in the style of those used for Java bytecode verification [8]) 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. The 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. this *original* 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 of certain parts of the analysis graph which has already been validated. In particular, we consider possible untrusted *increments* (or extensions) of a validated (trusted) code, i.e., a code producer can (periodically) send to its consumers new updates of a previously submitted package. By increments we mean both the addition of new data and also the extension of already existing procedures with new functionalities. In such a context of frequent software extensions, it appears inefficient to 1) submit a full certificate (superseding the original one) and 2) to perform the checking of the extended program from scratch, as needs to be done with current systems. In this work, we investigate an *incremental* approach to PCC, both for the first issue of certificate generation as well as for the checking process.

Regarding the first issue, when a program is extended, a new fixpoint has to be computed for the extended program. Such fixpoint differs from the original fixpoint stored in the certificate in a) the new fixpoint for each procedure affected by the extension 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 have affected by the changes. Our proposal is that, if the consumer still keeps the original abstraction, it is possible to provide, together with the program increment, only the *difference* of both abstractions, i.e., the *incremental certificate*. Essentially, the incremental certificate will contain the subset of the extended fixpoint which has been modified w.r.t. the original fixpoint. The first obvious advantage of our incremental approach is that the size of the certificate may be substantially reduced by submitting only the increment.

The second issue in incremental PCC is that the task performed by the checker can also be further reduced. In principle, a non-incremental checker (like the one in [2]) requires a whole traversal of the analysis graph where the en-

tire extended program is checked against the full certificate. However, it is now possible to define an *incremental checking* algorithm which, given the program increment and its incremental certificate, only rechecks the part of the analysis graph for the procedures which have been affected by the extension 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 stored by the consumers, together with the original certificate. With this, the checking process is carried out in a single pass over the subgraph affected by the extension. Thus, the second advantage of our incremental approach is that checking time is further reduced. We believe that the incremental ACC proposed in this work contributes to the practical uptake of PCC systems since it can significantly reduce certificate size and checking time in the context of program extensions.

2 A General View of Abstraction-Carrying Code

We assume some familiarity with abstract interpretation (see [5]), (Constraint) Logic Programming (C)LP (see, e.g., [10] and [9]) and PCC [11].

An abstract interpretation-based certifier is a function $\mathsf{certifier} : \mathit{Prog} \times \mathit{ADom} \times \mathit{AInt} \mapsto \mathit{ACert}$ which for a given program $P \in \mathit{Prog}$, an abstract domain $D_\alpha \in \mathit{ADom}$ and an abstract safety policy $I_\alpha \in \mathit{AInt}$ generates an abstract certificate $\mathit{Cert}_\alpha \in \mathit{ACert}$, by using an abstract interpreter for D_α, which entails that P satisfies I_α. In the following, we denote that I_α and Cert_α are specifications given as abstract semantic values of D_α by using the same α.

The basics for defining such certifiers (and their corresponding checkers) in ACC are summarized in the following five points:

Approximation. We consider an *abstract domain* $\langle D_\alpha, \sqsubseteq \rangle$ and its corresponding *concrete domain* $\langle 2^D, \subseteq \rangle$, both with a complete lattice structure. Abstract values and sets of concrete values are related by an *abstraction* function $\alpha : 2^D \to D_\alpha$, and a *concretization* function $\gamma : D_\alpha \to 2^D$. An abstract value $y \in D_\alpha$ is a *safe approximation* of a concrete value $x \in D$ iff $x \in \gamma(y)$. The concrete and abstract domains must be related in such a way that the following holds [5] $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. 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. 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:

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

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 $\mathsf{checker} : Prog \times ADom \times ACert \mapsto bool$ which for a program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and certificate $Cert_\alpha \in ACert$ checks whether $Cert_\alpha$ is a fixpoint of S_P^α or not:

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

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 Incremental Certificates

Our proposal will be illustrated through a running example, implemented 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*, 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 algorithm.

Example 1 (running example). Our running example is borrowed from [7] and shows a piece of a code which contains the following (normalized) implementation for the naive reversal of a list, while the definition of `app/3` is not yet available in it. This example illustrates the situation where some procedures which appear in the code are not really operational since they rely on other procedures not yet implemented, but which may become available in further extensions.

```
rev(X,Y) :- X = [ ], Y = [ ].
rev(X,Y) :- X = [U|V], rev(V,W), T = [U], app(W,T,Y).
```

We assume that the module contains other procedures which will not be "affected" by our next extension, hence, we do not need to show them here. □

3.1 The Original Certificate

Example 2 (abstract domain and call pattern). The description domain that we use in our examples is the *definite Boolean functions* [3], 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 $\mathtt{X} \wedge (\mathtt{Y} \leftrightarrow (\mathtt{U} \wedge \mathtt{V}))$. Groundness information is useful for many program optimizations and is also of great importance as a safety property, in order to verify that (C)LP programs are "well moded". The most general description $\top$ does not provide information about any variables in $\mathtt{V}$. 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 $\mathtt{X}$ nor $\mathtt{Y}$. □

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 ANALYZE, given a program P and abstract domain D_α, receives a set $S_\alpha \in AAtom$ of Abstract Atoms (or *call patterns*) and constructs an *analysis graph* [4] for S_α. The elements of S_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 $CP \in D_\alpha$.[3] 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 S_α in P. 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* and the *dependency arc table* (which are in fact the result of the analysis algorithm).

- The answer table contains entries of the form $A : CP \mapsto AP$ where A is always an atom in base form. Informally, its entries should be interpreted as "the answer pattern for calls to A satisfying precondition (or call pattern), CP, accomplishes postcondition (or answer pattern), AP."
- The intended meaning of a *dependency* $A : CP \Rightarrow [\_]\ B : CP_1$ in the dependency arc table is that the answer for $A : CP$ depends on the answer for $B : CP_1$, say AP_1. Thus, if the value of AP_1 increases during analysis, then the arc $A : CP \Rightarrow [\_]\ B : CP_1$ must be relaunched in order to compute the new answer for $A : CP$. I.e., the rule for A is processed again starting from its body atom B.

All the details and the formalization of the algorithm can be found in [7]. Certification in ACC [2] consists on using the *complete* set of entries stored in the answer table as certificate. Therefore, the elements in $ACert$ are set of entries of the form $A : CP \mapsto AP$. The information in the dependency arc table is not included in the certificate, thought it will be used later for incremental checking.

[3] We sometimes omit the subscript α from S_α when it is clear from the context.

Definition 1 (original certificate). *Let $P \in Prog$, $D_\alpha \in ADom$ and $S_\alpha \in AAtom$. We define* Orig_Cert $\in ACert$, *the* original certificate *for P and S_α, as the set of entries $A : CP_A \mapsto AP_A$ stored in the answer table computed by* ANALYZE(S_α) *[7] for P in D_α.*

Example 3 (original certificate). For our running example and the abstract domain and call pattern of Ex. 2, the analysis algorithm of [7] returns the following answers (a detailed description of the steps performed by the analysis algorithm in order to infer the next entries can be found in [7]):

$\texttt{rev}(\texttt{X},\texttt{Y}) : \top \mapsto \texttt{X} \wedge \texttt{Y} \qquad \texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) : \texttt{X} \mapsto \bot$

Intuitively, the answer for $\texttt{rev}(\texttt{X},\texttt{Y}) : \top$ is inferred from the analysis of the first rule. This clause binds the two variables to the empty list, hence, they are trivially ground. The analysis of the second rule for `rev` does not provide any further information since the code for `app` is not available, and the analyzer has to assume the answer $\bot$. It should be noted that the entry for $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z})$ has as call pattern `X` since, if we assume the answer $\texttt{X} \wedge \texttt{Y}$ for $\texttt{rev}(\texttt{X},\texttt{Y}) : \top$, procedure `app` is called from the second rule of `rev` with the first variable being a ground term.

Additionally, the analysis algorithm stores (in the *dependency arc table*) all the existing dependencies between call patterns. For the example at hand, such dependencies are:

(1) $\texttt{rev}(\texttt{X},\texttt{Y}) : \top \Rightarrow [\texttt{X} \leftrightarrow (\texttt{U} \wedge \texttt{V})]\ \texttt{rev}(\texttt{V},\texttt{W}) : \top$

(2) $\texttt{rev}(\texttt{X},\texttt{Y}) : \top \Rightarrow [(\texttt{X} \leftrightarrow (\texttt{U} \wedge \texttt{V})) \wedge \texttt{V} \wedge \texttt{W} \wedge (\texttt{T} \leftrightarrow \texttt{U})]\ \texttt{app}(\texttt{W},\texttt{T},\texttt{Y}) : \texttt{W}$

(2) means that the answer for $\texttt{rev}(\texttt{X},\texttt{Y}) : \top$ may change if the answer for $\texttt{app}(\texttt{W},\texttt{T},\texttt{Y}) : \texttt{W}$ changes. In such a case, the second rule for `rev` must be processed again starting from atom $\texttt{app}(\texttt{W},\texttt{T},\texttt{Y})$ in order to recompute the fixpoint for $\texttt{rev}(\texttt{X},\texttt{Y}) : \top$. (1) reflects the recursivity of $\texttt{rev}(\texttt{X},\texttt{Y}) : \top$, since it depends on itself. □

3.2 Incremental Certificates and Incremental Certifiers

Given a program P, an *increment* of P (written as $\nabla_P \in EProg$) is a new set of rules which are added to P resulting in an *extended* program $P' = P \odot \nabla_P$. As already mentioned, the increment ∇_P may include both the addition of new procedures and data as well as the addition of new rules (or functionality) for already existing procedures. Note that a program increment ∇_P contains both the new code to be added to the program and instructions on where to place such new code. On an implementation perspective, this can easily be done by using a program in the spirit of the traditional Unix *patch* command as $\odot$ operator and by using a *diff* format for coding program increments.

Consider the case where a program P with a certificate Orig_Cert has already been validated on a consumer. If now, P is incremented with ∇_P, it appears inefficient to generate, transmit, and check a certificate Ext_Cert for the extended program P' defined as $P' = P \odot \nabla_P$. Our proposal is that it is possible to submit only the new program extension ∇_P together with the *incremental certificate* Inc_Cert, i.e., the *difference* of both abstractions (Inc_Cert = Ext_Cert − Orig-_Cert). The bottom line is that the global fixpoint Ext_Cert for the extended program differs from the original fixpoint Orig_Cert in 1) the new fixpoint for

each procedure affected by the extension and 2) the update of other fixpoints possibly affected by the propagation of the effect of 1. However, there may be large parts of Orig_Cert which have not been affected by the changes and which do not need to be submitted nor checked again. In order to materialize this idea, it is necessary that the consumer stores Orig_Cert and properly extends it with the upcoming incremental certificates for achieving a compositional approach to incremental PCC. Some storage vs time trade-offs are discussed in Sect. 5.

Definition 2 (incremental certificate). *In the conditions of Def. 1, let* $\nabla_P \in EProg$ *be an increment for* P*. Let* Orig_Cert *be the original certificate for* P *and* S_α*. Let* Ext_Cert *be the certificate for* $P \odot \nabla_P$ *and* S_α*. We define* Inc_Cert*, the* incremental certificate *for* ∇_P *w.r.t.* Orig_Cert*, as the difference of certificates* Ext_Cert − Orig_Cert*.*

The difference *of two certificates,* Ext_Cert − Orig_Cert*, is defined as the set of entries* $B : CP_B \mapsto AP_B \in$ Ext_Cert *such that:*

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

Intuitively, Inc_Cert contains the subset of Ext_Cert which corresponds to the extensions and modifications w.r.t. Orig_Cert. The first obvious advantage of the incremental approach is that Inc_Cert can be much smaller than Ext_Cert. On the other, the following example illustrates that incrementing a program with new clauses can require the change in the analysis information previously computed for other procedures whose fixpoint is affected by the increment (although their definitions have not been directly extended).

Example 4 (incremental certificate). We now add the following rules for app/3:

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

Again, the steps performed by an (incremental) analysis algorithm and the associated analysis graph can be found in [7]. As result, the answers (Ext_Cert) obtained for the extended program are:

(a) $\mathtt{rev(X,Y)} : \top \mapsto \mathtt{X} \leftrightarrow \mathtt{Y}$ (b) $\mathtt{app(X,Y,Z)} : \mathtt{X} \mapsto \mathtt{X} \wedge (\mathtt{Y} \leftrightarrow \mathtt{Z})$
(c) $\mathtt{app(X,Y,Z)} : \top \mapsto (\mathtt{X} \wedge \mathtt{Y}) \leftrightarrow \mathtt{Z}$

which will necessarily be part of Inc_Cert (all fixpoints have changed w.r.t. Orig_Cert and a new entry has been added). □

The next definition introduces the notion of incremental certifier which, given the original certificate and an increment of the program, returns the incremental certificate iff the safety policy can still be entailed from the extended program.

Definition 3 (incremental certifier). *We define function* Inc_Certifier*:* $Prog \times EProg \times ADom \times AAtom \times AInt \times ACert \mapsto ACert$ *which takes* $P \in Prog$*,* $\nabla_P \in EProg$*,* $D_\alpha \in ADom$*,* $S_\alpha \in AAtom$*,* $I_\alpha \in AInt$*, a certificate* Orig_Cert $\in ACert$ *for* P *and* S_α*. Let* Ext_Cert $\in ACert$ *be the full certificate for* $P \odot \nabla_P$*. The function* Inc_Certifier *returns* Ext_Cert − Orig_Cert*, i.e., the* incremental certificate *for* ∇_P *w.r.t.* Orig_Cert*, iff* Ext_Cert $\sqsubseteq I_\alpha$*.*

Note that the above definition does not depend on the particular analysis algorithm used to generate Ext_Cert. Incremental analysis algorithms (like the ones in [7, 12, 14]) are very well suited to do this task. They reanalyze only the part of the analysis graph affected by the increment. Thus, the time required to generate the certificate on the producer side can be reduced. Although this optimization on the producer side is always desirable, it is not as critical within the PCC scheme as the reduction of the package transmission time or the checking time, which take place on the consumer side and that we discuss in the next section.

4 Incremental Checking

We present an incremental checking algorithm in two steps. We first define a checking algorithm for full certificates which is instrumented with a *Dependency Arc Table* (*DAT* for short) which stores the dependencies among atoms in the analysis graph. This structure is not required by existing checkers of full certificates [2] but it is instrumental in the design of our incremental checker. In the second step, we introduce the extensions required for validating an incremental certificate w.r.t. a program increment.

4.1 A Checking Algorithm with Dependencies

In essence, our abstract interpretation-based checking algorithm receives an original certificate Orig_Cert, an answer table AT_{mem} (initially empty) and a set of dependencies DAT_{mem} (initially empty), and constructs a program analysis graph in a single iteration by assuming the fixpoint information in Orig_Cert. While the graph is being constructed, the obtained answers are compared with the corresponding fixpoints stored in Orig_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, Orig_Cert gets checked.

Algorithm 1 presents our checker which is parametric w.r.t. the abstract domain of interest. It is hence defined in terms of five abstract operations on a selected abstract domain 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 actual 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 5. For the domain *Def* in Example 2, these abstract operations are defined as follows:

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

Algorithm 1 Checking with Support for Incrementality

```
1: function check(P, S, Cert, AT_mem, DAT_mem)
2:     AT_mem := ∅;  DAT_mem := ∅
3:     for all A : CP ∈ S do process_node(P, A : CP, Cert, AT_mem, DAT_mem)
4:     return Valid
5: function process_node(P, A : CP, Cert, AT_mem, DAT_mem)
6:     if (∃ a renaming σ s.t. σ(A : CP ↦ AP) in Cert) then
7:         add (A : CP ↦ σ^{-1}(AP)) to AT_mem
8:     else return Error
9:     process_set_of_rules(P, P|_A, A : CP ↦ σ^{-1}(AP), Cert, AT_mem, DAT_mem)
10: function process_set_of_rules(P, R, A : CP ↦ AP, Cert, AT_mem, DAT_mem)
11:     for all rule A_k ← B_{k,1}, ..., B_{k,n_k} in R do
12:         W := vars(A_k, B_{k,1}, ..., B_{k,n_k})
13:         CP_b :=Aextend(CP, vars(B_{k,1}, ..., B_{k,n_k})) ; CPR_b := Arestrict(CP_b, B_{k,1})
14:         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)
15:         AP_1 := Arestrict(CP_a, vars(A_k)) ; AP_2 := Alub(AP_1, AP)
16:         if (AP <> AP_2) then return Error
17: function process_rule(P, A : CP, A_k ← B_{k,j}, ..., B_{k,n_k}, W, CP_b, CPR_b, Cert,
                          AT_mem, DAT_mem)
18:     for all B_{k,i} in the rule body i = j, ..., n_k do
19:         CP_a := process_arc(P, A : CP, B_{k,i} : CPR_b, CP_b, W, Cert, AT_mem, DAT_mem)
20:         if (i <> n_k) then CPR_a := Arestrict(CP_a, var(B_{k,i+1}))
21:         CP_b := CP_a
22:         CPR_b := CPR_a
23:     return CP_a
24: function process_arc(P, A : CP, B_{k,i} : CPR_b, CP_b, W, Cert, AT_mem, DAT_mem)
25:     if (B_{k,i} is a constraint) then CP_a := Aadd(B_{k,i}, CP_b)
26:     else
27:         add A : CP ⇒ [CP_b]B_{k,i} : CPR_b to DAT_mem
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)
30:         AP_1 := Aextend (ρ^{-1}(AP), W) where ρ is a renaming s.t.
31:                                         ρ(B_{k,i} : CPR_b ↦ AP) in AT_mem
32:         CP_a := Aconj (CP_b, AP_1)
33:     return CP_a
```

where $\exists_{-V}CP$ represents $\exists v_1, \ldots, v_n F$, where $\{v_1, \ldots, v_n\} = vars(F) - V$, and $\sqcup$ is the least upper bound (lub) operation over the *Def* lattice. For instance, $\mathsf{Aconj}(X, Y \leftrightarrow (X \wedge Z))$=$X \wedge (Y \leftrightarrow Z)$. $\mathsf{Aadd}(X = [U|V], Y)$=$(X \leftrightarrow (U \wedge V)) \wedge Y$. $\mathsf{Alub}(X, Y)$=$X \vee Y$. □

Let us briefly explain the main functions of Algorithm 1.

1. Function check(P,S,Orig_Cert,AT_{mem},DAT_{mem}), receives as parameters a program P, a set S of call patterns, the original certificate Orig_Cert returned by ANALYZE, and two output variables AT_{mem} and DAT_{mem} which are initialized to $\emptyset$ (Line 2 of function check). If it succeeds (L4), AT_{mem} coincides with Orig_Cert, and DAT_{mem} stores all dependencies generated during the checking process.

2. For each abstract atom $A : CP \in S$, check calls process_node (L3). This function checks if there exists an answer for $A : CP$ in Orig_Cert and stores it in AT_{mem} (L6-7). All calls must have an entry in Orig_Cert or the certificate is not valid, and an error is issued (L8). It then proceeds to compute an answer for $A : CP$ by processing all rules (L9) defining A in a depth-first, left-to-right fashion.
3. This is done in function process_set_of_rules where the answers obtained for each rule in (L14) are lubbed (L15) with those stored in AT_{mem} (the fixpoint) to check that they are less or equal than the fixpoint. Otherwise an error is issued (L16).[4] As notation, given the set of rules (or arcs) R, $R|_A$ denotes the set of rules (or arcs) in R applicable to atom A.
4. Each particular rule is dealt with by function process_rule, which traverses the rule body (L18) and processes its corresponding atoms from left-to-right. In the algorithm, function process_arc looks up for an answer (L28) in AT_{mem} (i.e., the answer copied from Orig_Cert). If it does not exist, then a (recursive) call to process_node (L29) computes a solution for the atom.

The main difference between the checking Algorithm 1 and the one in [2] is that our checker stores the dependencies. This is done in function process_arc (L27). Dependencies are used in analyzers for achieving efficient implementations of fixed-point re-computations. Naturally, they are not used in non incremental checking algorithms since recomputation should never happen when one assumes the fixpoint. In contrast, DAT_{mem} will be fundamental in the design of our incremental checker, as we will see in the next section.

Definition 4 (checker). *We define function* CHECKER*:*$Prog \times ACert \times AAtom \times ADom \mapsto$ *boolean which takes a program* $P \in Prog$ *and its original certificate* Orig_Cert $\in ACert$ *for* $S_\alpha \in AAtom$ *in* $D_\alpha \in ADom$ *and*

1. *It returns the result of* check$(P, S_\alpha,$ Orig_Cert$, AT_{mem}, DAT_{mem})$.
2. *If it does not issue an* Error*, then it stores in memory* $AT_{persist} := AT_{mem}$, $DAT_{persist} := DAT_{mem}$ *and* $P_{persist} := P$.

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.

4.2 Incremental Checking

Intuitively, the task performed by an incremental checking algorithm has to be optimized such that it only: 1) rechecks the part of the analysis graph for the procedures which have been affected by the extension and, 2) propagates and rechecks the effect of these changes. In order to do this, we will take as

[4] This is the main difference with an analyzer. The latter needs to iterate if the new lub is different from the previously stored one (i.e., the fixpoint has not been reached). To do this iteration efficiently, dependencies detect the parts of the analysis graph which need to be reprocessed.

Algorithm 2 Incremental Checking

```
1: function incremental_checking(∇_P,Inc_Cert,AT_mem,DAT_mem)
2:     CP_update:=update_entries(AT_mem,Inc_Cert)
3:     DAT_check:=dependencies_to_check(P,DAT_mem,CP_update)
4:     process_new_rules(P,∇_P,Inc_Cert,AT_mem, DAT_mem)
5:     process_dependencies(P,Inc_Cert,AT_mem, DAT_mem, DAT_check)
6: function update_entries(AT_mem,Inc_Cert)
7:     CP_update := ∅
8:     for all entry A : CP ↦ AP in AT_mem do
9:         if (there exists a renaming σ such that A : CP = σ(A_1 : CP_1) and A_1 :
               CP_1 ↦ AP_1 belongs to Inc_Cert) then
10:            replace A : CP ↦ AP in AT_mem by A : CP ↦ σ(AP_1)
11:            add A : CP to CP_update
12:    return CP_update
13: function process_new_rules(P,∇_P,Inc_Cert,AT_mem, DAT_mem)
14:    for all entry A : CP ↦ AP in AT_mem do
15:        process_set_of_rules(P,∇_P|_A, A : CP ↦ AP, Inc_Cert,AT_mem,DAT_mem)
16: function process_dependencies(P,Inc_Cert,AT_mem, DAT_mem,DAT_check)
17:    for all arc A_k : CP_0 ⇒ [CP_1]B_{k,i} : CP_2 ∈ DAT_check do
18:        let A_k : −B_{k,1}, ..., B_{k,n_k} its associated rule in P
19:        W := vars(A_k : −B_{k,1}, ..., B_{k,n_k})
20:        CP_a:=process_rule(P,A_k : CP_0,A_k : −B_{k,i}, ..., B_{k,n_k},W,CP_1,CP_2,
21:                           Inc_Cert,AT_mem,DAT_mem)
22:        AP_1 := Arestrict(CP_a, vars(A_k))
23:        let AP the answer for A_k : CP_0 in AT_mem (modulo renaming)
24:        AP_2 := Alub(AP_1, AP)
25:        if (AP <> AP_2) then return Error
```

starting point the checker in Algorithm 1. This allows the incremental checker to propagate the changes and carry out the process in a single pass over the subgraph affected by the extension. Algorithm 2 presents our implementation of this intuition. The new checker is defined as follows: replace the function check in Algorithm 1 by the new function incremental_checking, use the remaining functions defined in Algorithm 1 and add the new functions update_entries, process_new_rules and process_dependencies. Essentially, the additional tasks that the incremental checker has to perform w.r.t. the non incremental one in Algorithm 1 are the following:

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. Our checker retrieves such stored data and initializes, respectively, the parameters AT_{mem}, DAT_{mem} and P with them.
2. *Update entries.* Prior to proceeding with the checking proper, we need to update the answers for those call patterns in AT_{mem} which have a different (updated) answer in Inc_Cert (L7-11). The function update_entries performs this task. Furthermore, it returns in CP_{update} the set of call patterns whose answer has been updated (L12).
3. *Dependencies to check.* Not all dependencies stored in DAT_{mem} must be revisited. We use function dependencies_to_check, which receives as input

parameters the program P, DAT_{mem} and CP_{update}, and selects the dependencies from DAT_{mem} which need to be processed for achieving the effect 2) mentioned above. We first have to remove those dependencies $A : CP \Rightarrow \_\ B : CP_1$ such that $B : CP_1$ does not belong to CP_{update}. In addition to them, we get rid also of those dependencies $H_k : CP \Rightarrow \_\ B_{kj} : \_$ such that there is $H_k : CP \Rightarrow \_\ B_{ki} : \_$ with $i < j$, i.e., for a given atom H_k (head of clause), we only maintain the dependency corresponding to the body atom at a leftmost position. The reason for this is that the processing of the leftmost atom will then launch the required arcs to its right. The code of function dependencies_to_check can be found in an extended version of this paper [1].

4. *Check new procedures.* The checking starts by processing, as described in the previous section, the new rules of the extension (∇_P) which have some associated entry in AT_{mem}. This process is carried out by function process_new_rules (L14-15) . Note that a new 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.
5. *Propagate effects.* After processing the new rules, function process_dependencies launches all arcs in DAT_{check} which are affected by the extension. This is done by using function process_rule (L20-21) from the non incremental checker.
6. *Store data.* Upon return, the checker has to store the computed AT_{mem}, DAT_{mem} and $P \odot \nabla_P$, respectively, in $AT_{persist}$, $DAT_{persist}$, and $P_{persist}$ for achieving a compositional design of our incremental approach.

Example 6. In our running example, function update_entries returns as updated answer table, AT_{mem}, the entries (a) and (b) of Example 4. CP_{update} contains the call patterns $\texttt{rev}(\texttt{X},\texttt{Y}) : \top$ and $\texttt{app}(\texttt{X},\texttt{Y},\texttt{Z}) : \texttt{X}$. After executing function dependencies_to_check, DAT_{check} contains dependency (1) of Example 3 only. □.

Definition 5 (incremental checker). *We define function* INCR_CHECKER*:* $EProg \times ACert \times \mapsto$ *boolean which takes an extension* $\nabla_P \in EProg$ *of* $P_{persist}$ *and its incremental certificate* Inc_Cert $\in ACert$ *and*

1. *It retrieves from memory* $AT_{mem} := AT_{persist}$, $DAT_{mem} := DAT_{persist}$ *and* $P = \nabla_P \odot P_{persist}$.
2. *It returns the result of* incremental_checking(∇_P, Inc_Cert, AT_{mem}, DAT_{mem}) *for* P.
3. *If it does not issue an* Error*, then it stores* $AT_{persist} := AT_{mem}$, $DAT_{persist} := DAT_{mem}$ *and* $P_{persist} := P$.

An important difference between the above definition and the checker in [2] is that the safety policy has to be tested w.r.t. the answer table for the extended program –Equation (2). Therefore, the checker must reconstruct, from Inc_Cert, the answer table returned by ANALYZE for the extended program, Ext_Cert, in order to test for adherence to the safety policy –Equation (4).

Example 7. Consider the extended program in Example 4 and its incremental certificate Inc_Cert. A call to incremental_checking(∇_P, Orig_Cert, AT_{mem}, DAT_{mem}), where $\nabla_P := \{\mathtt{app}_1, \mathtt{app}_2\}$, proceeds[5] as follows:

- AT_{mem}, DAT_{mem}, CP_{update} and DAT_{check} are initialized as in Example 6.
- A call to process_new_rules is generated. Since AT_{mem} contains an entry for $\mathtt{app(X, Y, Z) : X}$, a call to process_set_of_rules(∇_P,$\mathtt{app(X, Y, Z) : X} \mapsto \mathtt{X} \wedge (\mathtt{Y} \leftrightarrow \mathtt{Z})$) is made. In the following, we use AP to name $\mathtt{X} \wedge (\mathtt{Y} \leftrightarrow \mathtt{Z})$. The execution of this call produces the following two new calls:
  - process_rule($\mathtt{app(X, Y, Z) : X} \mapsto \mathtt{AP}, \mathtt{app}_1, \mathtt{W}_1, \mathtt{X}, \mathtt{X}$), where $\mathtt{W}_1 = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$. Next, a call to process_arc($\mathtt{app(X, Y, Z) : X}, \mathtt{X} = [\,] : \mathtt{X}, \mathtt{X}, \mathtt{W}_1$) is made, and X is returned as result (L25) since we are processing a constraint.
    Similarly a call to process_arc($\mathtt{app(X, Y, Z) : X}, \mathtt{Y} = \mathtt{Z} : \top, \mathtt{X}, \mathtt{W}_1$), returning $\mathtt{X} \wedge (\mathtt{Y} \leftrightarrow \mathtt{Z})$ as solution (L25) is made. Since we are in the last body literal, process_rule returns $CP_a = \mathtt{X} \wedge (\mathtt{Y} \leftrightarrow \mathtt{Z})$ (L23). Because CP_a and AP are equal, the algorithm does not issue Error (L16) and $\mathtt{app}_2$ is checked.
  - process_rule($\mathtt{app(X, Y, Z) : X} \mapsto \mathtt{AP}, \mathtt{app}_2, \mathtt{W}_2, \mathtt{X}, \mathtt{X}$), where $\mathtt{W}_2 = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{U}, \mathtt{V}, \mathtt{W}\}$. After computing the solutions for the first two constraints (by calling to process_arc as in $\mathtt{app}_1$), a call to process_arc($\mathtt{app\ (X, Y, Z) : X}, \mathtt{app(V, Y, W) : V}, \mathtt{CP}, \mathtt{W}_2$), where $CP = \mathtt{X} \wedge (\mathtt{X} \leftrightarrow (\mathtt{U} \wedge \mathtt{V})) \wedge (\mathtt{Z} \leftrightarrow (\mathtt{U} \wedge \mathtt{W}))$, occurs. Since we are processing an atom (L27), the dependency (3) $\mathtt{app(X, Y, Z) : X} \Rightarrow [\mathtt{CP}]\ \mathtt{app(V, Y, W) : V}$ is added to DAT_{mem}. The answer table contains $\rho(AP)$ as answer for $\mathtt{app(V, Y, W) : V}$, where $\rho = \{\mathtt{X/V}, \mathtt{Z/W}\}$. Thus, process_arc returns as answer the abstract conjunction between CP and $\rho(AP)$ (L32), which is $CP_a = \mathtt{X} \wedge \mathtt{U} \wedge \mathtt{V} \wedge (\mathtt{Z} \leftrightarrow \mathtt{W}) \wedge (\mathtt{Y} \leftrightarrow \mathtt{W})$. The restriction of CP_a to variables $\{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$ (L15) returns AP as solution for the rule. Hence the rule has been checked without issuing an Error (L15).
- Now, process_dependencies is executed and the unique arc (1) from DAT_{check} is launched (L20). The call to process_rule, for the call pattern $\mathtt{rev(X, Y)} : \top$ and the arc $\mathtt{rev(X, Y)}$:- $\mathtt{rev(V, W)}, \mathtt{T} = [\mathtt{U}], \mathtt{app(W, T, Y)}$ is executed. After two calls to process_arc for the call patterns $\mathtt{rev(V\ , W)} : \top$ and $\mathtt{T} = [\mathtt{U}] : \top$, the call process_arc ($\mathtt{rev(X, Y)} : \top, \mathtt{app(W, T, Y)} : \top, \mathtt{X} \leftrightarrow (\mathtt{U} \wedge \mathtt{W}) \wedge (\mathtt{V} \leftrightarrow \mathtt{W}) \wedge (\mathtt{T} \leftrightarrow \mathtt{U}), \{\mathtt{X}, \mathtt{Y}, \mathtt{V}, \mathtt{W\ T}, \mathtt{U}\}$) occurs and the dependency
  (2_{new}) $\mathtt{rev(X, Y)} : \top \Rightarrow [(\mathtt{X} \leftrightarrow (\mathtt{U} \wedge \mathtt{V})) \wedge (\mathtt{V} \leftrightarrow \mathtt{W}) \wedge (\mathtt{T} \leftrightarrow \mathtt{U})]\ \mathtt{app(W, T, Y)} : \top$
  is added to DAT_{mem}, replacing the old one (2). Since AT_{mem} does not contain an entry for $\mathtt{app(W, T, Y)} : \top$ (L28), a call to process_node ($\mathtt{app(W, T, Y)} : \top$) is generated and the entry $\mathtt{app(X, Y, Z)} : \top \mapsto (\mathtt{X} \wedge \mathtt{Y}) \leftrightarrow \mathtt{Z}$ is added to AT_{mem} (L7). Now, the algorithm checks if $(\mathtt{X} \wedge \mathtt{Y}) \leftrightarrow \mathtt{Z}$ is a fixpoint for $\mathtt{app(X, Y, Z)} : \top$ by processing all rules for app. The process finishes without Error and the dependency (associated to the second rule for app) is added to DAT_{mem}.
  (4) $\mathtt{app(X, Y, Z)} : \top \Rightarrow [(\mathtt{X} \leftrightarrow (\mathtt{U} \wedge \mathtt{V})) \wedge (\mathtt{Z} \leftrightarrow (\mathtt{U} \wedge \mathtt{W}))]\ \mathtt{app(V, Y, W)} : \top$

Once Inc_Cert has been validated, the consumer stores the answer table AT_{mem} (which is in fact Ext_Cert), DAT_{mem} (composed of the arcs (1), (2_{new}), (3) and (4)) and $P \odot \nabla_P$. □

[5] For clarity, we omit parameters P, Inc_Cert, AT_{mem} and DAT_{mem} of function calls.

Correctness of the checking process is ensured by demonstrating that our incremental checker is able to reconstruct the extended certificate and the dependencies in a single pass over (part of) the analysis graph.

Theorem 1 (correctness). *Let* $P \in Prog$, $\nabla_P \in EProg$, $D_\alpha \in ADom$ *and* $S_\alpha \in AAtom$. *Let* Orig_Cert $\in ACert$ *be the original certificate for* P *and* S_α, Ext_Cert $\in ACert$ *be the full certificate for* $P \odot \nabla_P$ *and* S_α, *and* Inc_Cert *be the incremental certificate for* ∇_P *w.r.t.* Orig_Cert. *If* INCR_CHECKER$(\nabla_P$, Inc_Cert*) does not issue an* Error, *then it holds that:*

- $AT_{persist} \equiv AT_{mem} \equiv$ Ext_Cert*;*
- $DAT_{persist} \equiv DAT_{mem}$

where AT_{mem} *and* DAT_{mem} *are, respectively, the answer table and dependencies returned by* check$(P \odot \nabla_p, S_\alpha,$ Ext_Cert$, AT_{mem}, DAT_{mem})$*, and the validation of* Inc_Cert *is done in a single pass.*

The proof of this theorem can be found in [1].

5 Conclusions

Incremental certificates and incremental checkers for ACC aim at reducing, respectively, the size of certificates and the checking time when a supplier provides an increment (or extension) of a previously validated package. Essentially, when a program is extended with new procedures or extended functionality for existing ones, the incremental certificate we propose contains only the *difference* between the certificate for the original program and the full certificate for the extended one. Checking time is reduced by traversing only those parts of the analysis graph which are affected by the changes, rather than traversing the whole graph. 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 in order to have it available for upcoming extensions. 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. In this work, we have considered the extension of a program with new procedures and possible additions of new functionalities over an already existing procedure. As future work, we plan to study whether our proposal applies to other possible (untrusted) modifications over a program, like the deletion and arbitrary changes of procedures, which affect certificates in different ways.

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.

References

1. E. Albert, P. Arenas, and G. Puebla. Incremental Certificates and Checkers for Abstraction-Carrying Code. Technical Report CLIP3/2006, Technical University of Madrid (UPM), School of Computer Science, UPM, February 2006.
2. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.
3. T. Armstrong, K. Marriott, P. Schachte, and g H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 266–280, Namur, Belgium, September 1994.
4. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
5. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
6. M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *Proc. of PPDP'05*. ACM Press, July 2005.
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. Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.
9. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
10. Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
11. G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.
12. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *Proc. of SAS'96*, pages 270–284. Springer LNCS 1145, 1996.
13. K. Rose, E. Rose. Lightweight bytecode verification. In *OOPSLA Workshop on Formal Underpinnings of Java*, 1998.
14. B. Ryder. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, 1988.