Reduced Certificates for Abstraction-Carrying Code

Elvira Albert[1], Puri Arenas[1], Germán Puebla[2], and Manuel Hermenegildo[2,3]

[1] Complutense University of Madrid, {elvira,puri}@sip.ucm.es
[2] Technical University of Madrid, {german,herme}@fi.upm.es
[3] University of New Mexico, herme@unm.edu

Abstract. *Abstraction-Carrying Code* (ACC) has recently been proposed as a framework for mobile code safety in which the code supplier provides a program together with an *abstraction* 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. The advantage of providing a (fixed-point) abstraction to the code consumer is that its validity is checked in a *single pass* of an abstract interpretation-based checker. A main challenge is to reduce the size of certificates as much as possible while at the same time not increasing checking time. We introduce the notion of *reduced certificate* which characterizes the subset of the abstraction which a checker needs in order to validate (and re-construct) the *full certificate* in a single pass. Based on this notion, we instrument a generic analysis algorithm with the necessary extensions in order to identify the information relevant to the checker. We also provide a correct checking algorithm together with sufficient conditions for ensuring its completeness. The experimental results within the CiaoPP system show that our proposal is able to greatly reduce the size of certificates in practice.

1 Introduction

Proof-Carrying Code (PCC) [16] 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 which 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 approach is that the task of the consumer is reduced to checking, a procedure that should be much simpler, efficient, and automatic than generating the original certificate. Abstraction-Carrying Code (ACC) [2] 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 carried out automatically, by a fixed-point analyzer. Both the analysis and checking algorithms are always parametric on the abstract domain, with the resulting genericity. This allows proving a wide variety of properties by using the large set of abstract domains that are available, well understood, and with already developed proofs for the correctness of the corresponding abstract operations. This is one of the fundamental advantages of ACC.[4]

In this paper, we consider analyzers which construct a *program analysis graph* which is an abstraction of the (possibly infinite) set of states explored by the concrete execution. To capture the different graph traversal strategies used in different fixed-point algorithms we use the *generic* description of [10], which generalizes the algorithms used in state-of-the-art analysis engines. 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 analysis information inferred for each call is stored in the *answer table* [10]. In the original ACC framework, the final *full* answer table constitutes the certificate. Since this certificate contains the fixpoint, a single pass over the analysis graph is sufficient to validate it on the consumer side. It should be noted that while the ACC framework and our work here are applied at the source-level, and 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 the ACC approach and in our current proposal can also be applied to bytecode.

One of the main challenges for the practical uptake of ACC (and related methods) is to produce certificates which are reasonably small. This is important since the certificate is transmitted together with the untrusted code and, hence, reducing its size will presumably contribute to a smaller transmission time. Also, this reduces the storage cost for the certificate. Nevertheless, a main concern when reducing the size of the certificate is that checking time is not increased as a consequence. In principle, the consumer could use an analyzer for the purpose of generating the whole fixpoint from scratch, which is still feasible since analysis is automatic. However, this would defeat one of the main purposes of ACC, which is to reduce checking time. The objective of this paper is to characterize the smallest subset of the abstraction which must be sent within a certificate –and which still guarantees a single pass checking process– and to design an ACC scheme which generates and validates such reduced certificates.

[4] The coexistence of several abstract domains in our framework is somewhat related to the notion of *models* to capture the security-relevant properties of code, as addressed in the work on Model-Carrying Code [22].However, their combination has not been studied which differs from our idea of using combinations of (high-level) abstract domains, which is already well understood.

Fixpoint compression is being used in different contexts and tools. For instance, in the Astrée analyzer [8], only one abstract element by head of loop is kept for memory usage purposes. In the PCC scheme, the basic idea in order to compress a certificate is to store only the analysis information which the checker is not able to reproduce by itself [12]. With this purpose, Necula and Lee [17] designed a variant of LF, called LF_i, in which certificates discard all the information that is redundant or that can be easily synthesized. Also, Oracle-based PCC [18] aims at minimizing the size of certificates by providing the checker with the minimal information it requires to perform a proof. Tactic-based PCC [3] aims at minimizing the size of certificates by relying on large reasoning steps, or tactics, that are understood by the checker. Finally, this general idea has also been deployed in lightweight bytecode verification [20] where the certificate, rather than being the whole set of frame types (FT) associated to each program point is reduced by omitting those (local) program point FTs which correspond to instructions without branching *and* which are lesser than the final FT (fixpoint). Our proposal for ACC is at the same time more general (because of the parametricity of the ACC approach) and carries the reduction further because it includes only in the certificate those calls in the analysis graph (including both branching an non branching instructions) required by the checker to re-generate the certificate in one pass.

2 A General View of Abstraction-Carrying Code

We assume the reader is familiar with abstract interpretation (see [7]) and (Constraint) Logic Programming (C)LP (see, e.g., [14] and [13]). A 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 a safety policy $I_\alpha \in \mathit{AInt}$ generates a 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 six points and Equations:

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 $\alpha : 2^D \rightarrow D_\alpha$, and a *concretization* function $\gamma : D_\alpha \rightarrow 2^D$. 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 [7] $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. 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 $[\![P]\!]_\alpha = \text{lfp}(S_P^\alpha)$. The operator S_P^α is the abstract counterpart of S_P.

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

Correctness of analysis ensures that $[\![P]\!] \in \gamma([\![P]\!]_\alpha)$.

Verification Condition. 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:

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

Certifier. 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 $\mathsf{checker} : \textit{Prog} \times \textit{ADom} \times \textit{ACert} \mapsto \textit{bool}$ which for a program $P \in \textit{Prog}$, an abstract domain $D_\alpha \in \textit{ADom}$ and a certificate $Cert_\alpha \in \textit{ACert}$ 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 Generation of Certificates in ACC

This section recalls ACC and the notion of full certificate in the context of (C)LP [2]. For concreteness, we build on the algorithms of `CiaoPP` [9].

Algorithm 1 has been presented in [10] as a generic description of a fixed-point algorithm which generalizes those used in state-of-the-art analysis engines, such as the one in `CiaoPP` [9]. In order to analyze a program, traditional (goal dependent) abstract interpreters for (C)LP programs receive as input, in addition to the program P and the abstract domain D_α, a set $S_\alpha \in \textit{AAtom}$ of Abstract Atoms (or *call patterns*). Such call patterns 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 $CP \in D_\alpha$. For brevity, we sometimes omit the subscript α in the algorithms. The analyzer of Algorithm 1 constructs an *and–or graph* [4] (or analysis graph) for S_α which is an abstraction of the (possibly infinite) set of (possibly infinite) execution paths (and-or trees) explored by the concrete execution of initial the calls described by S_α in P. The

Algorithm 1 Generic Analyzer for Abstraction-Carrying Code

```
 1: function ANALYZE_F($S, \Omega$)
 2:     for $A : CP \in S$ do
 3:         add_event($newcall(A : CP), \Omega$)
 4:     while $E :=$ next_event($\Omega$) do
 5:         if $E := newcall(A : CP)$ then new_call_pattern($A : CP, \Omega$)
 6:         else if $E := updated(A : CP)$ then add_dependent_rules($A : CP, \Omega$)
 7:         else if $E := arc(R)$ then process_arc($R, \Omega$)
 8:     return answer table
 9: procedure NEW_CALL_PATTERN($A : CP, \Omega$)
10:     for all rule $A_k : -B_{k,1}, \ldots, B_{k,n_k}$ do
11:         $CP_0$ :=Aextend($CP, vars(\ldots, B_{k,i}, \ldots)$); $CP_1$ := Arestrict($CP_0, vars(B_{k,1})$)
12:         add_event($arc(A_k : CP \Rightarrow [CP_0]\ B_{k,1} : CP_1)$,$\Omega$)
13:     add $A : CP \mapsto \bot$ to answer table
14: procedure PROCESS_ARC($H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$,$\Omega$)
15:     if $B_{k,i}$ is not a constraint then
16:         add $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ to dependency arc table
17:     $W := vars(A_k, B_{k,1}, \ldots, B_{k,n_k})$; $CP_3$ := get_answer($B_{k,i} : CP_2, CP_1, W, \Omega$)
18:     if $CP_3 \neq \bot$ and $i \neq n_k$ then
19:         $CP_4$ := Arestrict($CP_3, vars(B_{k,i+1})$);
20:         add_event( $arc(H_k : CP_0 \Rightarrow [CP_3]\ B_{k,i+1} : CP_4)$,$\Omega$)
21:     else if $CP_3 \neq \bot$ and $i = n_k$ then
22:         $AP_1$ := Arestrict($CP_3, vars(H_k)$); insert_answer_info($H : CP_0 \mapsto AP_1, \Omega$)
23: function GET_ANSWER($L : CP_2, CP_1, W, \Omega$)
24:     if $L$ is a constraint then return Aadd($L, CP_1$)
25:     else $AP_0$ := lookup_answer($L : CP_2, \Omega$); $AP_1$ := Aextend($AP_0, W$)
26:          return Aconj($CP_1, AP_1$)
27: function LOOKUP_ANSWER($A : CP, \Omega$)
28:     if there exists a renaming $\sigma$ s.t.$\sigma(A : CP) \mapsto AP$ in answer table then
29:         return $\sigma^{-1}(AP)$
30:     else add_event($newcall(\sigma(A : CP)), \Omega$) where $\sigma$ is renaming s.t. $\sigma(A)$ in base
            form; return $\bot$
31: procedure INSERT_ANSWER_INFO($H : CP \mapsto AP, \Omega$)
32:     $AP_0$ := lookup_answer($H : CP$); $AP_1$ := Alub($AP, AP_0$)
33:     if $AP_0 \neq AP_1$ then
34:         add ($H : CP \mapsto AP_1$) to answer table ;
35:         add_event($updated(H : CP), \Omega$)
36: procedure ADD_DEPENDENT_RULES($A : CP, \Omega$)
37:     for all arc of the form $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ in graph where there
            exists renaming $\sigma$ s.t. $A : CP = (B_{k,i} : CP_2)\sigma$ do
38:         add_event($arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2), \Omega$)
```

program analysis graph is implicitly represented in the algorithm by means of two global data structures, the *answer table* and the *dependency arc table*, both initially empty.

– The answer table contains entries of the form $A : CP \mapsto AP$ where A is always a base form.[5] Informally, its entries should be interpreted as "the

[5] 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

answer pattern for calls to A satisfying precondition (or call pattern) CP meets postcondition (or answer pattern), AP."

- A dependency arc is of the form $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$. This is interpreted as follows: if the rule with H_k as head is called with description CP_0 then this causes the i-th literal $B_{k,i}$ to be called with description CP_2. The remaining part CP_1 is the program annotation just before $B_{k,i}$ is reached and contains information about all variables in rule k.

Intuitively, the analysis algorithm is a graph traversal algorithm which places entries in the answer table and dependency arc table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, a *prioritized event queue* is used. We use $\Omega \in QHS$ to refer to a *Queue Handling Strategy* which a particular instance of the generic algorithm may use. Events are of three forms:

- *newcall*$(A : CP)$ which indicates that a new call pattern for literal A with description CP has been encountered.
- *arc*$(H_k : \_ \Rightarrow [\_]\ B_{k,i} : \_)$ which indicates that the rule with H_k as head needs to be (re)computed from the position k, i.
- *updated*$(A : CP)$ which indicates that the answer description to call pattern A with description CP has been changed.

The functions add_event and next_event respectively push an event to the priority queue and pop the event of highest priority, according to Ω. The algorithm is defined in terms of four abstract operations on the domain D_α:

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

More details on the algorithm can be found in [10, 19]. Let us briefly explain its main procedures. The algorithm centers around the processing of events on the priority queue, which repeatedly removes the highest priority event (Line 4) and calls the appropriate event-handling function (L5-7). The function new_call_pattern initiates processing of all the rules for the definition of the internal literal A, by adding arc events for each of the first literals of these rules (L12). Initially, the answer for the call pattern is set to $\perp$ (L13). The procedure process_arc performs the core of the analysis. It performs a single step of the left-to-right traversal of a rule body. If the literal $B_{k,i}$ is not a constraint (L15), the arc is added to the dependency arc table (L16). Atoms are processed by function get_answer.

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.

Constraints are simply added to the current description (L24). In the case of literals, the function lookup_answer first looks up an answer for the given call pattern in the answer table (L28) and if it is not found, it places a *newcall* event (L30). When it finds one, then this answer is extended to the variables in the rule the literal occurs in (L25) and *conjoined* with the current description (L26). The resulting answer (L17) is either used to generate a new arc event to process the next literal in the rule, if $B_{k,i}$ is not the last one (L18); otherwise, the new answer is computed by insert_answer_info. This is the part of the algorithm more relevant to the generation of reduced certificates. The new answer for the rule is *combined* with the current answer in the table (L32). If the fixpoint for such call has not been reached, then the answer table entry is updated with the combined answer (L34) and an updated event is added to the queue (L35). The purpose of such an update is that the function add_dependent_rules (re)processes those calls which depend on the call pattern $A : CP$ whose answer has been updated (L37). This effect is achieved by adding the arc events for each of its dependencies (L38). Note that dependency arcs are used for efficiency: they allow us to start the reprocessing of a rule from the body atom which actually needs to be recomputed due to an update rather than from the leftmost atom.

The following definition corresponds to the essential idea in the ACC framework –equations (1) and (2)– of using a static analyzer to generate the certificates. The analyzer corresponds to Algorithm 1 and the certificate is the *full* answer table.

Definition 1 (full certificate). *We define function* CERTIFIER_F*:$Prog \times ADom \times AAtom \times AInt \times QHS \mapsto ACert$ which takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$, $\Omega \in QHS$ and returns as* full certificate, FCert $\in ACert$, *the answer table computed by* ANALYZE_F(S_α, Ω) *for P in D_α if* FCert $\sqsubseteq I_\alpha$.

4 Abstraction-Carrying Code with Reduced Certificates

The key observation in order to reduce the size of certificates is that certain entries in a certificate may be *irrelevant*, in the sense that the checker is able to reproduce them by itself in a single pass. The notion of *relevance* is directly related to the idea of recomputation in the program analysis graph. Intuitively, given an entry in the answer table $A : CP \mapsto AP$, its fixpoint may have been computed in several iterations from $\bot$, AP_0, $AP_1, \ldots$ until AP. For each change in the answer, an event updated($A : CP$) is generated during the analysis. The above entry is relevant in a certificate (under some strategy) when its updates force the recomputation of other arcs in the graph which *depend* on $A : CP$ (i.e., there is a dependency from it in the table). Thus, unless $A : CP \mapsto AP$ is included in the (reduced) certificate, a single-pass checker which uses the same strategy as the code producer will not be able to validate the certificate.

4.1 The Notion of Reduced Certificate

According to the above intuition, we are interested in determining when an entry in the answer table has been "updated" during the analysis and such changes affect other entries. However, there are two special types of updated events which can be considered "irrelevant". The first one is called a *redundant update* and corresponds to the kind of updates which force a redundant computation. We write $DAT|_{A:CP}$ to denote the set of arcs of the form $H : CP_0 \Rightarrow [CP_1]B : CP_2$ in the current dependency arc table such that they depend on $A : CP$ with $A : CP = (B : CP_2)\sigma$ for some renaming σ.

Definition 2 (redundant update). *Let $P \in Prog$, $S_\alpha \in AAtom$ and $\Omega \in QHS$. We say that an event* updated*$(A : CP)$ which appears in the event queue during the analysis of P for S_α is* redundant *w.r.t. Ω if, when it is generated, $DAT|_{A:CP} = \emptyset$.*

In the following section we propose a slight modification to the analysis algorithm in which redundant updates are never introduced in the priority queue, and thus they never enforce redundant recomputation. The proof of correctness for this modification can be found in [1].

The second type of updates which can be considered irrelevant are *initial updates* which, under certain circumstances, are generated in the first pass over an arc. In particular, we do not take into account updated events which are generated when the answer table contains $\bot$ for the updated entry. Note that this case still corresponds to the first traversal of any arc and should not be considered as a reprocessing.

Definition 3 (initial update). *In the conditions of Def. 2, we say that an event* updated*$(A : CP)$ which appears in the event queue during the analysis of P for S_α is* initial *for Ω if, when it is generated, the answer table contains $A : CP \mapsto \bot$.*

Initial updates do not occur in certain very optimized algorithms, like the one in [19]. However, they are necessary in order to model generic graph traversal strategies. In particular, they are intended to *resume* arcs whose evaluation has been *suspended*.

Definition 4 (relevant update). *In the conditions of Def. 2, we say that an event* updated*$(A : CP)$ is* relevant *iff it is not initial nor redundant.*

The key idea is that those answer patterns whose computation has introduced relevant updates should be available in the certificate.

Definition 5 (relevant entry). *In the conditions of Def. 2 we say that the entry $A : CP \mapsto AP$ in the answer table is* relevant *for Ω iff there has been at least one relevant event* updated*$(A : CP)$ during the analysis of P for S_α.*

The notion of *reduced certificate* allows us to remove irrelevant entries from the answer table and produce a smaller certificate which can still be validated in one pass.

Definition 6 (reduced certificate). *In the conditions of Def. 2, let* FCert= ANALYZE_F*(S_α, Ω) for P and S_α. We define the* reduced certificate, RCert, *as the set of relevant entries in* FCert *for Ω.*

Example 1. Consider the `Ciao` version of procedure `rectoy`, taken from [21]:

```
rectoy(N,M) :- N = 0, M = 0.
rectoy(N,M) :- N1 is N-1, rectoy(N1,R), M is N1+R.
```

Assume the call pattern `rectoy(N,M)` : {`N/int`} which indicates that external calls to `rectoy` are performed with an integer value, `int`, in the first argument `N`. It holds that FCert = `rectoy(N,M)` : {`N/int`} $\mapsto$ {`N/int`, `M/int`} (the steps performed by ANALYZE_F are detailed in [1]). Assume now that we use a strategy $\Omega \in QHS$ which assigns the highest priority to redundant updates and selects the rules for `rectoy` in the textual order. For this strategy, the unique entry in FCert is not relevant as there has been no relevant updated event in the queue. Therefore, the reduced certificate for our example is empty (and, with the techniques of the next section, our checker is able to reconstruct the fixpoint in a single pass from this empty certificate). In contrast, lightweight bytecode verification [21] sends, together with the program, the reduced *non-empty* certificate $cert = (\{30 \mapsto (\epsilon, rectoy\cdot int\cdot int\cdot int\cdot \bot)\}, \epsilon)$, which states that at program point 30 the stack does not contain information (first occurrence of ϵ),[6] and variables N, M and R have type *int*, *int* and $\bot$. The need for sending this information is because `rectoy`, implemented in Java, contains an *if*-branch (equivalent to the branching for selecting one of our two clauses for *rectoy*). Thus *cert* has to inform the checker that it is possible for variable R at point 30 to be undefined, if the *if* condition does not hold. Note that this program is therefore an example of how our approach improves on state-of-the-art PCC techniques by reducing the certificate further while still keeping the checking process one-pass. □

4.2 Generation of Certificates without Irrelevant Entries

In this section, we proceed to instrument the analyzer of Algorithm 1 with the extensions necessary for producing reduced certificates, as defined in Def. 6. The resulting analyzer ANALYZE_R is presented in Algorithm 2. It uses the same procedures of Algorithm 1 except for the new definitions of process_arc, add_dependent_rules and insert_answer_info. The differences with respect to the original definition are:

1. *We annotate suspended arcs in the dependency arc table.* A dependency $A : CP_A \Rightarrow [\ \_\ ]\ B : CP_B$ in the dependency arc table is said to be *suspended* if when analysing the corresponding arc, the answer table did not contain an entry for $B : CP_B$ or it contained $\bot$ as answer. Suspended arcs are marked in L10 of function process_arc. A not suspended arc in the dependency arc table will be named *relevant*. Suspended arcs do not cause reprocessing in the sense that its continuation is not inserted in the queue (see L18 in Algorithm 1). These suspended arcs are processed when an updated event (possibly initial) for $B : CP_B$ is extracted from the queue.

[6] The second occurrence of ϵ indicates that there are no backwards jumps.

2. *We count the number of relevant updates for each call pattern.* To do this, we associate with each entry in the answer table a new field "u" whose purpose is to identify relevant entries. Concretely, u indicates the number of updated events processed for the entry. u is initialized when the (unique and first) initial updated event occurs for a call pattern. The initialization of u is different for redundant and initial updates as explained in the next point. When the analysis finishes, if $u > 1$, we know that at least one reprocessing has occurred and the entry is thus relevant. The essential point to note is that u has to be increased when the event is actually *extracted* from the queue (L14) and not when it is *introduced* in it (L26). The reason for this is that when a non-redundant, updated event is introduced, if the priority queue contains an identical event, then the processing is performed only once. Therefore, our counter must not be increased.
3. *We do not generate redundant updates.* Our algorithm does not introduce redundant updated events (L26). However, if they are initial (and redundant) they have to be counted as if they had been introduced and processed and, thus, the next update over them has to be considered always relevant. This effect is achieved by initializing the u-value with a higher value ("1" in L23) than for initial updates ("0" in L22). Indeed, the value "0" just indicates that the initial updated event has been introduced in the priority queue but not yet processed and that all arcs in the dependency arc table associated to it are *suspended arcs.* The u value will be increased to "1" either once it is extracted from the queue or if another updated event occurs before processing the initial updated event and a relevant arc appears in the set of dependencies associated to the call pattern at hand (L27). The function $relevant(Dep)$, where Dep is a set of dependencies, returns the set of relevant dependencies (not suspended) occurring in Dep. This distinction is required because it is possible to generate new updated events before the corresponding initial and non redundant updated event be processed. In such case, if the set of dependencies associated to the call pattern only contains suspended arcs, this updated event must be considered also initial. Otherwise the updated event is relevant, and thus u must be increased to 1. Therefore, when this updated event be processed, u will be increased to "2" (relevant entry).

In Algorithm 2, a call (u, AP)=get_from_answer_table($A : CP$) looks up in the answer table the entry for $A : CP$ and returns its u-value and its answer AP. A call set_in_answer_table($A(u) : CP \mapsto AP$) replaces the entry for $A : CP$ with the new one $A(u) : CP \mapsto AP$.

Proposition 1. *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $\Omega \in QHS$. Let* FCert *be the answer table computed by* ANALYZE_R*(S_α, Ω) for P in D_α. Then, an entry $A(u) : CP_A \mapsto AP \in$* FCert *is relevant iff $u > 1$.*

Note that, except for the control of relevant entries, ANALYZE_F(S_α, Ω) and ANALYZE_R(S_α, Ω) have the same behavior and they compute the same answer table (see [1] for details). We use function remove_irrelevant_answers which takes

Algorithm 2 ANALYZE_R: Analyzer instrumented for Certificate Reduction

```
1: procedure PROCESS_ARC($H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2, \Omega$)
2:     $W := vars(A_k, B_{k,1}, \ldots, B_{k,n_k})$; $CP_3 :=$ get_answer($B_{k,i} : CP_2, CP_1, W, \Omega$)
3:     if $CP_3 \neq \bot$ and $i \neq n_k$ then
4:         $CP_4 :=$ Arestrict($CP_3, vars(B_{k,i+1})$);
5:         add_event( $arc(H_k : CP_0 \Rightarrow [CP_3]\ B_{k,i+1} : CP_4)$,$\Omega$)
6:     else if $CP_3 \neq \bot$ and $i = n_k$ then
7:         $AP_1 :=$ Arestrict($CP_3, vars(H_k)$); insert_answer_info($H : CP_0 \mapsto AP_1, \Omega$)
8:     if $B_{k,i}$ is not a constraint then
9:         if $CP_3 = \bot$ then
10:            add $suspended(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$ to dependency arc table
11:        else add $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ to dependency arc table
12: procedure ADD_DEPENDENT_RULES($A : CP, \Omega$)
13:     $(AP, u)$ =get_from_answer_table($A : CP$)
14:     set_in_answer_table($A(u+1) : CP \mapsto AP$)
15:     for all arc of the form $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ in graph where there
            exists renaming $\sigma$ s.t. $A : CP = (B_{k,i} : CP_2)\sigma$ do
16:         add_event($arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2), \Omega$)
17: procedure INSERT_ANSWER_INFO($H : CP \mapsto AP, \Omega$)
18:     $AP_0 :=$ lookup_answer($H : CP, \Omega$)
19:     $AP_1 :=$ Alub($AP, AP_0$)
20:     if $AP_0 \neq AP_1$ then          % updated required
21:         if $AP_0 = \bot$ then
22:             if $DAT|_{H:CP} \neq \emptyset$ then $u = 0$          % non redundant initial update
23:             else $u = 1$          % redundant initial update
24:         else $(u, \_)$=get_from_answer_table($H : CP$)          % not initial update
25:         if $DAT|_{H:CP} \neq \emptyset$ then
26:             add_event(updated($H : CP$))
27:             if $u = 0$ and relevant($DAT|_{H:CP}$) $\neq \emptyset$ then $u = 1$
28:         set_in_answer_table($H(u) : CP \mapsto AP_1$)
```

a set of answers of the form $A(u) : CP \mapsto AP \in$ FCert and returns the set of answers $A : CP \mapsto AP$ such that $u > 1$.

Definition 7 (certifier). *We define the function* CERTIFIER_R*: $Prog \times ADom \times AAtom \times AInt \times QHS \mapsto ACert$, which takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$, $\Omega \in QHS$. It returns as certificate,* RCert=remove_irrelevant_answers(FCert)*, where* FCert=ANALYZE_R*(S_α, Ω), if* FCert $\sqsubseteq I_\alpha$.

5 Checking Reduced Certificates

In the ACC framework for full certificates the checking algorithm [2] uses a specific graph traversal strategy, say Ω_C. This checker has been shown to be very efficient but in turn its design is not generic with respect to this issue (in contrast to the analysis design). This is not problematic in the context of full certificates since, even if the certifier uses a strategy Ω_A which is different from Ω_C, it is ensured that all valid certificates get validated in one pass by that specific checker. This result does not hold any more in the case of reduced certificates. In particular, *completeness* of checking is not guaranteed if $\Omega_A \neq \Omega_C$. This occurs because though the answer table is identical for all strategies, the subset of redundant entries depends on the particular strategy used. The problem is that, if there is an entry $A : CP \mapsto AP$ in FCert such that it is relevant w.r.t. Ω_C but it is not w.r.t. Ω_A, then a single pass checker will fail to validate the RCert generated using Ω_A. Therefore, it is essential in this context to design generic checkers which are not tied to a particular graph traversal strategy. Upon agreeing on the appropriate parameters,[7] the consumer uses the particular instance of the generic checker resulting from application of such parameters.

It should be noted that the design of generic checkers is also relevant in light of current trends in verified analyzers (e.g., [11, 6]), which could be transferred directly to the checking end. In particular, since the design of the checking process is generic, it becomes feasible in ACC to use automatic program transformers to specialize a certified (specific) analysis algorithm in order to obtain a certified checker with the same strategy while preserving correctness and completeness.

The following definition presents a generic checker for validating reduced certificates. In addition to the genericity issue discussed above, an important difference with the checker for full certificates [2] is that there are certain entries which are not available in the certificate and that we want to reconstruct and output in checking. The reason for this is that the safety policy has to be tested w.r.t. the full answer table –Equation (2). Therefore, the checker must reconstruct, from RCert, the answer table returned by ANALYZE_F, FCert, in order to test for adherence to the safety policy –Equation (4). Note that reconstructing the answer table does not add any additional cost compared to the checker in [2], since the full answer table also has to be created in [2].

Definition 8 (checker for reduced certificates). *Function* CHECKING_R *is defined as function* ANALYZE_R *with the following modifications:*

1. *It receives* RCert *as an additional input parameter.*
2. *It may fail to produce an answer table. In that case it issues an* Error.
3. *Function* insert_answer_info *is replaced by the new one in Algorithm 3.*

Function CHECKER_R *takes $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $\Omega \in QHS$,* RCert *$\in ACert$ and returns the result of* CHECKING_R($S_\alpha, \Omega,$ RCert) *for P in D_α.*

Let us briefly explain the differences between Algorithms 2 and 3. First, the checker has to detect (and issue) two sources of errors:

a) The answer in the certificate is more precise than the one obtained by the checker (L4). This is the traditional error in ACC and means that the certificate and program at hand do not correspond to each other.

[7] In a particular application of our framework, we expect that the graph traversal strategy is agreed a priori between consumer and producer. But, if necessary (e.g., the consumer does not implement this strategy), then it could be sent along with the transmitted package.

Algorithm 3 Generic Checker for Reduced Certificates CHECKING_R

```
1: procedure INSERT_ANSWER_INFO($H : CP \mapsto AP, \Omega$)
2:     $AP_0 :=$ lookup_answer($H : CP, \Omega$); $AP_1 :=$ Alub($AP, AP_0$)
3:     (IsIn,$AP'$)=look_fixpoint($H : CP$,RCert)
4:     if IsIn and Alub($AP, AP'$) $\neq AP'$ then return Error          % error of type a)
5:     if $AP_0 \neq AP_1$ then          % updated required
6:         if $AP_0 = \perp$ then
7:             if $DAT|_{H:CP} \neq \emptyset$ then $u = 0$; add_event($updated(H : CP), \Omega$)
8:             else $u = 1$
9:         else ($u$, _)=get_from_answer_table($H : CP$)
10:        if $DAT|_{H:CP} \neq \emptyset$ and ($u = 1$ or ($u = 0$ and $relevant(DAT|_{H:CP}) \neq \emptyset$))
               then return Error          % error of type b)
11:        if IsIn and $AP_0 = \perp$ then $AP_1 = AP'$
12:        set_in_answer_table($H(u) : CP \mapsto AP_1$)

13: function LOOK_FIXPOINT($A : CP$,RCert)
14:     if $\exists$ a renaming $\sigma$ such that $\sigma(A : CP \mapsto AP) \in$ RCert then
15:         return (True,$\sigma^{-1}(AP)$)
16:     else return (False,$\perp$)
```

b) Recomputation is required. This should not occur during checking, i.e., only initial updates should be generated (L7) by the checker.[8] This second type of error corresponds to situations in which a relevant update is needed in order to obtain an answer (it cannot be obtained in one pass). This is detected in L10 prior to introducing the (non redundant) update if either u is already 1 or $u = 0$ but its associated dependencies contain a relevant arc.

The second difference is that the entries $A : CP \mapsto AP'$ stored in RCert have to be added to the answer table, after the initial updated event for $A : CP$ occurs, in order to detect errors of type a) above. In particular, L11 and L12 add the fixpoint AP' stored in RCert to the answer table together with the corresponding u-value (same value as in Algorithm 2).

The following theorem ensures that if CHECKER_R validates a certificate (i.e., it does not return Error), then the re-constructed answer table is a fixpoint. This implies that any certificate which gets validated by the checker is indeed a valid one.

Theorem 1 (correctness). *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$ and $\Omega_A, \Omega_C \in QHS$. Let* FCert= CERTIFIER_F*$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_A)$ and* RCert= CERTIFIER_R*$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_A)$. If* CHECKER_R*$(P, D_\alpha, S_\alpha, I_\alpha,$* RCert, *$\Omega_C)$ does not issue an* Error, *then it returns* FCert.

The following theorem (completeness) provides sufficient conditions under which a checker is guaranteed to validate reduced certificates which are actually valid.

Theorem 2 (completeness). *Let $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$ and $\Omega_A \in QHS$. Let* FCert= CERTIFIER_F*$(P, D_\alpha, S_\alpha, I_\alpha, \Omega_A)$ and* RCert$_{\Omega_A}$

[8] Initial updates are not needed in the particular instance of the checker of [2] because the strategy is fixed. They are needed to model a generic checker though.

	Size	**Certificate Size**			**Checking Time**		
Program	**Source**	**FCert**	**RCert**	**F/R**	**C$_F$**	**C$_R$**	**C$_F$/C$_R$**
aiakl	1555	3090	1616	1.912	85	86	0.991
bid	4945	5939	883	6.726	46	49	0.943
browse	2589	1661	941	1.765	18	20	0.929
deriv	957	288	288	1.000	50	28	1.806
grammar	1598	1259	40	31.475	15	14	1.042
hanoiapp	1172	2325	880	2.642	30	28	1.049
occur	1367	1098	666	1.649	20	19	1.085
progeom	1619	2148	40	53.700	20	15	1.351
qsortapp	664	2355	650	3.623	20	21	0.990
query	2090	531	40	13.275	18	12	1.436
rdtok	13704	6533	2659	2.457	57	58	0.986
rectoy	154	167	40	4.175	8	8	1.079
serialize	987	1779	1129	1.576	27	27	1.022
zebra	2284	4058	40	101.450	123	125	0.979
Overall				3.35			1.06

Table 1. Size of Reduced Certificates and Checking Time

= CERTIFIER_R*($P, D_\alpha, S_\alpha, I_\alpha, \Omega_A$). Let $\Omega_C \in$ QHS be such that* $\mathsf{RCert}_{\Omega_C}$ = CERTIFIER_R*($P, D_\alpha, S_\alpha, I_\alpha, \Omega_C$) and* $\mathsf{RCert}_{\Omega_A} \supseteq \mathsf{RCert}_{\Omega_C}$*. Then,* CHECKER_R*($P$, $D_\alpha, S_\alpha, I_\alpha$,* $\mathsf{RCert}_{\Omega_A}, \Omega_C$*) returns* FCert *and does not issue an* Error*.*

Obviously, if $\Omega_C = \Omega_A$ then the checker is guaranteed to be complete. Additionally, a checker using a different strategy Ω_C is also guaranteed to be complete as long as the certificate reduced w.r.t Ω_C is equal to or smaller than the certificate reduced w.r.t Ω_A. Furthermore, if the certificate used is full, the checker is complete for any strategy. Note that if $\mathsf{RCert}_{\Omega_A} \not\supseteq \mathsf{RCert}_{\Omega_C}$, CHECKER_R with the strategy Ω_C may fail to validate $\mathsf{RCert}_{\Omega_A}$, which is indeed valid for the program under Ω_A.

6 Discussion and Experimental Evaluation

As we have illustrated throughout the paper, the gain of the reduction is directly related to the number of *updates* (or iterations) performed during analysis. Clearly, depending on the graph traversal strategy used, different instances of the generic analyzer will generate reduced certificates of different sizes. Significant and successful efforts have been made during recent years towards improving the efficiency of analysis. The most optimized analyzers actually aim at reducing the number of updates necessary to reach the final fixpoint [19]. Interestingly, our framework greatly benefits from all these advances, since the more efficient analysis is, the smaller the corresponding reduced certificates are. We have implemented a generator and a checker of reduced certificates in `CiaoPP`. Both the analysis and checker use the optimized depth-first new-calling QHS of [19].

In Table 1 we study two crucial points for the practicality of our proposal: the size of the reduced vs. full certificates and the relative efficiency of checking reduced certificates. As mentioned before, the algorithms are parametric w.r.t. the abstract domain. In our experiments we use the *sharing+freeness* [15] abstract domain, that is very useful for reasoning about instantiation errors, a crucial aspect for the safety of logic programs. The system is implemented in Ciao 1.13 [5] with compilation to bytecode. The experiments have been performed on a Pentium 4 (Xeon) at 2 Ghz and 4 Gb RAM, running GNU Linux FC-2, 2.6.9.

The set of benchmarks used is the same as in [10, 2], where they are described in more detail. The column **Source** shows the size in bytes of the source code. The size in bytes of the certificates is showed in the next set of columns. **FCert** and **RCert** contain the size of the full and reduced certificate, respectively, for each benchmark and they are compared in column (**F/R**). Our results show that the reduction in size is very significant in all cases. It ranges from 101.45 in *zebra* (RCert is indeed empty –the size of an empty certificate is, in this case, 40 bytes since it includes information about the abstract domain used for generating the certificate– whereas FCert is 4058) to 1 for *deriv* (both certificates have the same size). The final part of the table compares the checking time both when full and reduced certificates are used. Execution times are given in milliseconds and measure *runtime*. They are computed as the arithmetic mean of five runs. For each benchmark, $\mathbf{C}_F$ and $\mathbf{C}_R$ are the times for executing CHECKER_F and CHECKER_R, respectively. The column $\mathbf{C}_F/\mathbf{C}_R$ compares both checking times. It can be seen that the efficiency of CHECKER_R is very similar to that of CHECKER_F in most cases. The last row (Overall) summarizes the results for the different benchmarks using a weighted mean, where the weight is the actual checking time for each benchmark. Overall, certificates are reduced by a factor of 3.35 and the checker for reduced certificates is slightly faster, with an overall speedup of 1.06.

References

1. E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Reduced Certificates for Abstraction-Carrying Code. Technical Report CLIP8/2005.0, Technical University of Madrid (UPM), School of Computer Science, UPM, October 2005.
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. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, LNCS. Springer, 2004. To appear.
4. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
5. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.13). Technical report, School of Computer Science (UPM), 2006. Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

6. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. In *Proc. of ESOP 2004*, volume LNCS 2986, pages 385 – 400, 2004.
7. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
8. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyser. In *Proc. ESOP 2005*, pages 21–30. Springer LNCS 3444, 2005.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
10. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.
11. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 3(298):583–626, 2003.
12. Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.
13. J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.
14. Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.
15. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
16. G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.
17. G.C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proceedings of LICS'98*, page 93. IEEE Computer Society, 1998.
18. G.C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of POPL'01*, pages 142–154. ACM Press, 2001.
19. 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.
20. E. Rose and K. Rose. Java access protection through typing. *Concurrency and Computation: Practice and Experience*, 13(13):1125–1132, 2001.
21. K. Rose, E. Rose. Lightweight bytecode verification. In *OOPSLA Workshop on Formal Underpinnings of Java*, 1998.
22. R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proc. of SOSP'03*, pages 15–28. ACM, 2003.