Generation of Reduced Certificates in Abstraction-Carrying Code

Elvira Albert[1] Puri Arenas[1]
Germán Puebla[2] 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. In this paper, we first 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 then instrument a generic analysis algorithm with the necessary extensions in order to identify the information relevant to the checker.

Key words: Reduced certificates, abstraction-carrying code, abstract interpretation, mobile code safety, logic programming

1 Introduction

Proof-Carrying Code (PCC) [12] is a general framework for mobile code safety which proposes to associate safety information in the form of a *certificate*

⋆ An extended version of this paper will appear in [2]. This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* integrated project, and by the Spanish projects TIN2005-09207-C03 (*MERIT*) and CAM S-0505/TIC/0407 (*PROMESAS*). Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

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) [3] 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.

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 [8], 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* [8]. 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.

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 subset of the abstraction which must be sent within a certificate and which still guarantees a single pass checking process.

In the PCC scheme, the basic idea in order to reduce a certificate is to store only the analysis information which the checker is not able to reproduce by itself [9]. With this purpose, Necula and Lee [13] designed a variant of the

Edinburgh Logical Framework, called LF$_i$, in which certificates discard all the information that is redundant or that can be easily synthesized. Also, Oracle-based PCC [14] aims at minimizing the size of certificates by providing the checker with the minimal information it requires to perform a proof. Tactic-based PCC [4] 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 [16] 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 Generation of Full Certificates in ACC

This section introduces the notion of full certificate in the context of (C)LP [3]. We assume the reader is familiar with abstract interpretation (see [6]) and (Constraint) Logic Programming (C)LP (see, e.g., [11] and [10]). 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 [6] $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x$ and $\forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. 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.

Algorithm 1 has been presented in [8] 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` [7]. 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 $S_\alpha \in AAtom$ 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* [5] (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 the initial calls described by S_α in P. The program analysis graph is implicitly represented in the algorithm by means of two global data structures, the *answer table* and

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:     $W := vars(H_k, B_{k,1}, \ldots, B_{k,n_k})$; $CP_3$ := 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 Aglb($CP_1, AP_1$)
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($arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2), \Omega$)
```

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[1] and AP an abstract substitution. Its entries should be interpreted as "the answer pattern for calls to A satisfying precondition (or call pattern) CP meets postcondition (or answer pattern), AP."

[1] 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.

- *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. Different QHS may traverse the analysis graph in a depth-first, breadth-first fashion or any combination (see, e.g., [15] for different strategies). 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 k with H 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 five 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;
Aglb(CP_1, CP_2) performs the abstract conjunction of two descriptions;
Aadd(C, CP) performs the abstract operation of conjoining (i.e., computing the conjunction) the abstraction of the constraint C with the description CP;
Alub(CP_1, CP_2) performs the abstract disjunction of two descriptions.

More details on the algorithm can be found in [8,15]. 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. 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* (i.e., the conjunction of both descriptions is computed) 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 *certification* process carried out by the producer. First, an *abstraction* (written $Cert_\alpha$) is automatically generated which safely approximates the behaviour of the program by using a static analyzer. And, second, the verification condition is generated from this certificate and it can be proved only if the execution of the code does not violate the safety policy. In particular, we use an abstract *safety policy* $I_\alpha \in AInt$ in order to specify precisely the (abstract) conditions under which the execution of a program is considered safe. Then, the certifier checks whether the abstraction entails the safety policy, i.e., $Cert_\alpha \sqsubseteq \mathcal{I}_\alpha$.

Definition 2.1 We define function CERTIFIER_F:$Prog \times ADom \times AAtom \times AInt \times QHS \mapsto ACert$ which takes a program $P \in Prog$, $D_\alpha \in ADom$, $S_\alpha \in AAtom$, $I_\alpha \in AInt$, $\Omega \in QHS$ 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 $\sqsubseteq I_\alpha$.

3 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.

3.1 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 3.1 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, $DAT|_{A:CP} = \emptyset$.

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 3.2 In the conditions of Def. 3.1, 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 [15]. 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 3.3 In the conditions of Def. 3.1, 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 3.4 In the conditions of Def. 3.1 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.

Reduced certificates allow us to remove irrelevant entries from the answer table and produce a smaller certificate which can still be validated in one pass.

Definition 3.5 In the conditions of Def. 3.1, 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.

3.2 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. 3.5. The resulting analyzer ANALYZE_R is presented in Algorithm 2. It uses the same procedures of Algorithm 1 except for the new definitions of add_dependent_rules and insert_answer_info. The differences with respect to the original definition are:

(i) *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 (L3) and not when it is *introduced* in it (L13). 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.

(ii) *We do not generate redundant updates.* Our algorithm does not introduce redundant updated events (L13). 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 L11) than for initial updates ("0" in L10). Indeed, the value "0" just indicates that the initial updated event has been introduced in the priority queue but not yet processed. It will be increased to "1" once it is extracted from the queue. Therefore, in both cases the next updated event over the call pattern will increase the counter to "2" and will be relevant.

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 $A(u) : CP \mapsto AP$ .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 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 3.6 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 $\sqsubseteq I_\alpha$.

Algorithm 2 Analyze_r: Analyzer instrumented for Certificate Reduction

```
 1: procedure ADD_DEPENDENT_RULES($A : CP, \Omega$)
 2:     $(AP, u)$ =get_from_answer_table($A : CP$)
 3:     set_in_answer_table($A(u+1) : CP \mapsto AP$)
 4:     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
 5:         add_event($arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2), \Omega$)
 6: procedure INSERT_ANSWER_INFO($H : CP \mapsto AP, \Omega$)
 7:     $AP_0$ := lookup_answer($H : CP, \Omega$) ; $AP_1$ := Alub($AP, AP_0$)
 8:     if $AP_0 \neq AP_1$ then          % updated required
 9:         if $AP_0 = \bot$ then
10:             if $DAT|_{H:CP} \neq \emptyset$ then $u = 0$          % non redundant initial update
11:             else $u = 1$          % redundant initial update
12:         else  $(u, \_)$=get_from_answer_table($H : CP$)          % not initial update
13:         if $DAT|_{H:CP} \neq \emptyset$ then add_event(updated($H : CP$))
14:         set_in_answer_table($H(u) : CP \mapsto AP_1$)
```

We have demonstrated in [1] that a checking algorithm which uses the same QHS is able to reconstruct the full certificate from the reduced certificate in a single pass over the full abstraction. Our completeness results also ensure that all reduced certificates validated by the checker are indeed valid, regardless of the QHS upon which the checker is based.

4 Discussion

As we have pointed out 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 [15]. 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 [15]. In our experimental evaluation (see [2] for details) we have observed reductions in the size of certificates by a factor of over 3 on average using our reduced certificates across a set of benchmarks, with a very small variation in checking time (within 6% on average).

References

[1] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Reduced Certificates for Abstraction-Carrying Code. TR CLIP8/2005.0, Technical University of Madrid (UPM), School of Computer Science, UPM, October 2005.

[2] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Reduced Certificates for Abstraction-Carrying Code. In *Proc. of ICLP'06*, Springer LNCS, 2006. To appear.

[3] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, Springer LNAI 3452, pp. 380–397, 2005.

[4] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In *Proc. of CASSIS'04*, Springer LNCS, 2004.

[5] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[6] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pp. 238–252, 1977.

[7] 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.

[8] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.

[9] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[10] J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

[11] Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[12] G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pp. 106–119. ACM Press, 1997.

[13] G.C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proc. of LICS'98*, pp. 93. IEEE Computer Society, 1998.

[14] G.C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In *Proc. of POPL'01*, pp. 142–154. ACM Press, 2001.

[15] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *Proc. of SAS'96*, Springer LNCS 1145, pp. 270–284, 1996.

[16] E. Rose and K. Rose. Java access protection through typing. *Concurrency and Computation: Practice and Experience*, 13(13):1125–1132, 2001.