Abstract Interpretation-based Code Certification for Pervasive Systems: Preliminary Experiments

Elvira Albert
School of Computer Science
Complutense University of Madrid
elvira@sip.ucm.es

Germán Puebla
School of Computer Science
Technical University of Madrid
{german,herme}@fri.upm.es

Manuel Hermenegildo
Departments of Computer Science and Electrical and Computer Engineering
University of New Mexico

ABSTRACT
Proof carrying code is a general methodology for certifying that the execution of an untrusted mobile code is safe, according to a predefined safety policy. The basic idea is that the code supplier attaches a certificate (or proof) to the mobile code which, then, the consumer checks in order to ensure that the code is indeed safe. The potential benefit is that the consumer's task is reduced from the level of proving to the level of checking, a much simpler task. Recently, the abstract interpretation techniques developed in logic programming have been proposed as a basis for proof carrying code [1]. To this end, the certificate is generated from an abstract interpretation-based proof of safety. Intuitively, the verification condition is extracted from a set of assertions guaranteeing safety and the answer table generated during the analysis. Given this information, it is relatively simple and fast to verify that the code does meet this proof and so its execution is safe. This extended abstract reports on experiments which illustrate several issues involved in abstract interpretation-based code certification. First, we describe the implementation of our system in the context of CiaoPP: the preprocessor of the Ciao multi-paradigm (constraint) logic programming system. Then, by means of some experiments, we show how code certification is aided in the implementation of the framework. Finally, we discuss the application of our method within the area of pervasive systems which may lack the necessary computing resources to verify safety on their own. We herein illustrate the relevance of the information inferred by existing cost analysis to control resource usage in this context. Moreover, since the (rather complex) analysis phase is replaced by a simpler, efficient checking process at the code consumer side, we believe that our abstract interpretation-based approach to proof-carrying code becomes practically applicable to this kind of systems.

1. THE FRAMEWORK
Current approaches to mobile code safety, inspired by the technique of Proof-Carrying Code (PCC) [19], associate safety information in the form of a certificate to programs. The certificate (or proof) is created by the code supplier at compile time, and packaged along with the untrusted code. The consumer who receives the code+certificate package can then run a checker which by a straightforward 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 burden of ensuring compliance with the desired safety policy is shifted from the consumer to the supplier. Indeed the (proof) checker performs a task that should be much simpler, efficient, and automatic than generating the original certificate. For instance, in the first PCC system [19], the certificate is originally a proof in first-order logic of certain verification conditions and the checking process involves ensuring that the certificate is indeed a valid first-order proof.

The main practical difficulty of PCC techniques is in generating safety certificates which at the same time:

- allow expressing interesting safety properties,
- can be generated automatically and,
- are easy and efficient to check.

In [1], the abstract interpretation techniques [6] developed in logic programming1 are proposed as a basis for PCC. They offer a number of advantages for dealing with the aforementioned issues. In particular, the expressiveness of existing abstract domains will be implicitly available in abstract interpretation-based code certification to define a wide range of safety properties. Furthermore, the approach inherits the automation and inference power of the abstract interpretation engines used in (Constraint) Logic Programming, (C)LP.

1We refer to [2, 7, 15], and their references, for more details on analysis techniques developed in logic programming.
Fixpoint Analyzer...
2. SOME EXPERIMENTS IN CIAOPP

The above abstract interpretation-based code certification framework has been implemented in CiaoPP [14]: the preprocessor of the Ciao program development system [3]. Ciao is a multi-paradigm programming system, allowing programming in logic, constraint, and functional styles. At the heart of Ciao is an efficient logic programming-based kernel language. This allows the use of the very large body of approximation domains, inference techniques and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area (see, e.g., [18, 5, 12, 15] and their references). These techniques and systems can approximate in compile-time, always safely, and with a significant degree of precision, a wide range of properties which is much richer than, for example, traditional types. This includes data structure shape (including pointer sharing), independence, bounds on data structure sizes, and other operational variable instantiation properties as well as procedure-level properties such as determinacy, termination, non-failure and bounds on resource consumption (time or space cost). The latter tasks are performed in an integrated fashion in CiaoPP. The fundamental functionality behind CiaoPP is static global program analysis based on abstract interpretation. For this task CiaoPP uses the PLAI abstract interpreter including extensions for, e.g., incrementally [15], modularity [4, 21], analysis of constraints [13] and analysis of concurrency [17].

In the context of CiaoPP, the abstract interpretation-based certification system is implemented in Ciao 1.11#200 [3] with compilation to bytecode. In essence, we have used the efficient, highly optimized, state-of-the-art analysis system of CiaoPP (which is part of a working compiler) as a fixpoint analyzer for generating safety certificates. The checker has been implemented also as a simplification of such generic abstract interpreter. Our aim here is to present not the techniques used by CiaoPP for code certification (which are described in [1]) but its main functionalities by means of some examples.

Example 1. The next program mmultiply multiplies two matrices by using two auxiliary predicates: multiply which performs the multiplication of a matrix and an array and vmul which computes the vectorial product of two arrays (by multiplying all their elements):

\[
\text{mmultiply}([],[],[]).
\]
\[
\text{mmultiply}(\text{Rest}, \text{VI}, \text{Others}):-
\text{multiply}(\text{Rest}, \text{VI}, \text{Others}),
\text{mmultiply}([\text{VO}], \text{VI}, \text{Result}).
\]
\[
\text{multiply}([],[],[]).
\]
\[
\text{multiply}(\text{VO},\text{Rest}, \text{VI}, [\text{Result}|\text{Others}]):-
\text{multiply}(\text{Rest}, \text{VI}, \text{Others}),
\text{vmul}(\text{VO}, \text{VI}, \text{Result}).
\]
\[
\text{vmul}(\text{[]}, [], 0).
\]
\[
\text{vmul}([\text{H1}|\text{TI}], [\text{H2}|\text{T2}], \text{Result}):-
\text{vmul}([\text{T1}|\text{T2}], \text{Newresult}),
\text{Product} = \text{H1}\ast\text{H2},
\text{Result} = \text{Product}+\text{Newresult}.
\]

One of the distinguishing features of logic programming is that arguments to procedures can be uninstantiated variables. This, together with the search execution mechanism available (generally backtracking) makes it possible to have multi-directional procedures. i.e., rather than having fixed input and output arguments, execution can be "reversed". Thus, we may compute the "input" arguments from known "output" arguments. However, predicate is/2 (used as an infix binary operator) is mono-directional. It computes the arithmetic value of its second (right) argument and unifies it with its first (left) argument. The execution of is with an uninstantiation rightmost argument results in a run-time error. Therefore, a safety issue in this example is to ensure that calls to the built-in predicate is are performed with ground data in the right argument.

We can infer this safety information by analyzing the above program in CiaoPP using a mode and independence analysis ("sharing+freeness"). In the "sharing+freeness" domain, \text{var} denotes variables that do not point yet to any data structure, \text{mshare} denotes pointer sharing patterns between variables and \text{ground} variables which point to data structures which contain no pointers. The analysis is performed with the following \text{entry} assertion which allows specifying a restricted class of calls to the predicate:

\[
:- \text{entry} \text{mmultiply}(X,Y,Z)(: \text{var}(Z), \text{ground}(X), \text{ground}(Y)).
\]

It denotes that calls to mmultiply will be performed with ground terms in the first two arguments and a free variable in the last one.

For the above entry, the output of CiaoPP yields the following set of assertions which constitute our safety certificate:

\[
:- \text{true pred} \text{mmultiply}(A,B,C)
\]
\[
\text{(mshare([],[]),var(C),ground([A,B])})
\]
\[
\Rightarrow \text{(ground([A,B,C])}).
\]
\[
:- \text{true pred} \text{mmultiply}(A,B,C)
\]
\[
\text{(mshare([],[]),var(C),ground([A,B])})
\]
\[
\Rightarrow \text{(ground([A,B,C])}).
\]
\[
:- \text{true pred} \text{vmul}(A,B,C)
\]
\[
\text{(mshare([[]]),var(C),ground([A,B])})
\]
\[
\Rightarrow \text{(ground([A,B,C])}).
\]
\[
:- \text{true pred} \text{A is B+C}
\]
\[
\text{(mshare([[]]),var(A),ground([B,C])})
\]
\[
\Rightarrow \text{(ground([A,B,C])}).
\]
\[
:- \text{true pred} \text{A is B+C}
\]
\[
\text{(mshare([[]]),var(A),ground([B,C])})
\]
\[
\Rightarrow \text{(ground([A,B,C])}).
\]

The "true pred" assertions above specify in a combined...
way properties of both: "=" the entry (i.e., upon calling) and "=>" the exit (i.e., upon success) points of all calls to the predicate. For instance, the last two assertions for predicate is express that the leftmost argument is a free unaliased variable while the rightmost arguments are input values (i.e., ground on call) when is is called (\(=\)). Upon success, all three arguments will get instantiated. Given this information, we can verify that the safety condition is accomplished and thus the code is safe to run. Thus, the above analysis output can be used as a certificate to attest a safe use of predicate is.

The above experiment has been performed using a sharing-freeness domain. However, the whole method is domain-independent. This allows plugging in different abstract domains, provided suitable interfacing functions are defined. From the user point of view, it is sufficient to specify the particular abstract domain desired. For instance, CiaoPP can also infer (parametric) types for programs both at the predicate level and at the literal level \([10, 11, 23]\). Clearly, type information is very useful for program certification, verification, optimization, debugging (see, e.g., \([14]\)).

Example 2. Our next experiment uses the regular type domain eterns \([23]\) to analyze the same program of Ex. 1. We use in our examples term as the most general type (i.e., it corresponds to all possible terms), list to represent lists and num for numbers. We also allow parametric types such as list(T) which denotes lists whose elements are all of type T. Type list is clearly equivalent to list(term).

The program is analyzed w.r.t. the following entry assertion which specifies that calls to multiply are performed with matrices in the first two arguments:

\[
\texttt{:- entry multiply(X,Y,Z)} \rightarrow \texttt{list(A,num),list(Y,list(num))}.\]

CiaoPP output yields, among other, the following assertions for the built-in predicate is:

\[
\texttt{:- true pred A is B+C} \rightarrow \texttt{term(A),num(B),num(C)}.\]

\[
\texttt{:- true pred A is B+C} \rightarrow \texttt{num(A),num(B),num(C)}.\]

They indicate that calls to is will be performed with numbers in the rightmost argument (thus, ground terms) and will return, upon success, a number in the first argument. Therefore, they also constitute a valid (and more precise) certificate for the safety issue described in Ex. 1.

It is also interesting to note that properties natively understood by different analysis domains can be combined in the same assertion \([14]\) (c.f. Example 3).

### 3. APPLICATIONS IN PERVERSIVE COMPUTING

Pervasive computing platforms are becoming ever smaller and more powerful, and are embedded everywhere, even in living organisms. They can contain sophisticated models of our personal environment that help us to make everyday decisions; they have the power to do mathematical and logical reasoning in order to perform intelligent tasks. As a result, verification and validation techniques have to keep pace with the huge requirements for intelligent, user-oriented applications that must run on devices with a minimum of computing resources. In this context, there is a large number of computing devices which may range from personal computers to PDAs, mobile phones, dedicated processors, smart cards, wearable computers and such like. Such devices are often characterized by having a relatively small amount of computing resources \([24]\). As a result, time efficiency is an issue since often these devices have to operate on real-time tasks. Also, and possibly more importantly, memory efficiency is an issue. If either the software used is too large to fit in the device or needs too much memory to run, then it is simply not possible to use such software.

Abstract interpretation-based techniques are able to reason about computational properties which can be useful for controlling efficiency issues in the context of pervasive computing systems. For instance, CiaoPP can infer lower and upper bounds on the sizes of terms and the computational cost of predicates \([8, 9]\). Cost bounds are expressed as functions on the sizes of the input arguments and yield the number of resolution steps. Various measures can be used for the "size" of the input, such as list-length, term-size, term-depth, integer-value, etc. The idea is that the system can disregard code which makes requirement that are too large in terms of computing resources (in time and/or space). Let us see an example.

Example 3. The following program inc_all increments all elements of a list by adding one to each of them.

\[
\texttt{inc_all([],[]).} \\
\texttt{inc_all([H|T],[NH|NT]) :- NH is H+1, inc_all(T,NT).} \\
\]

The following assertions have been added by the user of the pervasive computing system:

\[
\texttt{:- entry inc_all(A,B) :- list(A,num),var(B).} \\
\texttt{:- check calls inc_all(A,B) :- list(A,num).} \\
\texttt{:- check success inc_all(A,B) :- list(B,num).} \\
\texttt{:- check comp inc_all(A,B) :- list(A,num),var(B) + steps_ub(length(A)+1).} \\
\]

The entry assertion specifies that calls to inc_all must be performed with a list of numbers in the first argument while the second one must be a free variable. The next three check
assertions express the intended semantics of the program. In particular, the second one is somehow trivial to verify for the given entry. The third one intends to check that, upon success, the second argument of calls to inc_all will be a list of numbers. Finally, the last computational (comp) assertion tries to verify that the upper bound of the predicate is the sum of the length of the first list and one. The idea is that the code will be accepted provided all assertions can be checked.

The cost analysis available in CiaoPP infers, among others, the following assertions for the above program and entries:

\[
\begin{align*}
\text{:- checked calls inc_all(A,B)} & \quad : \text{list(A,num)}.
\text{:- checked success inc_all(A,B)} \quad \Rightarrow \quad \text{list(B,num)}.
\text{:- checked comp inc_all(A,B)} & \quad : \left( \text{list(A,num)}, \text{var}(B) \right) \\
& \quad + \text{steps_ub}(\text{inc_all(A,B)},\text{length}(A)+1).
\text{:- true pred inc_all(A,B)} & \quad : \left( \text{list(A,num)}, \text{var}(B) \right) \\
& \quad \Rightarrow \quad \left( \text{list(A,num)}, \text{list(B,num)} \right)
\end{align*}
\]

Therefore, the status of the last three check assertions has become checked, which means that they have been validated and thus the program is safe to run (according to the intended meaning). The last procedure-level assertion merges them all and, additionally, indicates that calls to the predicate do not fail and their execution is deterministic by combining information available for other abstract domains.

Apart from expressing relevant properties, when developing software for deployment on Smart Cards (and similar ambient computing devices), two more important issues arise: 1) Pervasive computing is characterized by having a relatively large number of untrusted computing devices which interact. Thus, when modeling such a system, it is not realistic to consider one device in isolation: it will receive plenty of mobile data from the environment. In this context, the safety of the deployed software is crucial, as the cost of recalling untrusted devices can be prohibitive. 2) It is essential to simplify the (safety) verification process and reduce its resource usage. Indeed, Smart Cards typically provide less than 4Kb of RAM while it is possible to use only up to 128Kb for storing the application and static data. Such resource considerations tend to dominate the development process for pervasive systems, forcing developers to write low-level code from scratch, as mobile system developers have found in their own experience.

PCC techniques—based on certificates which are computed outside the device—constitute a good scenario for the certification of software deployed in pervasive systems. They compute tamper-proof certificates which simplify code verification and pass them along with the code. In our abstract interpretation-based context, although global analysis is now routinely used as a practical tool, it is still unacceptable to run the whole analyzer to validate the certificate as it involves considerable cost. One of the main reasons is that the fixpoint algorithm is an iterative process which often computes answers (repeatedly) for the same call due to possible updates introduced by further computations. At each iteration, the algorithm has to manipulate rather complex data structures—which involve performing updates, lookups, etc.—until the fixpoint is reached. Luckily, in abstract interpretation-based code certification, the burden on the consumer side is reduced by using a simple one-traversal checker, which is a very simplified and efficient abstract interpreter which does not need to compute a fixpoint. The benchmark results in [1] show that the speedup achieved by the checking is approximately 1.63 in just analysis time which, we believe, makes our approach practically applicable in pervasive contexts.

A similar proposal is presented in [22] to split the type-based bytecode verification of the KVM (an embedded variant of the JVM) in two phases, where the producer first computes the certificate by means of a type-based dataflow analyzer and then the consumer simply checks that the types provided in the code certificate are valid. This approach is extended in [16] to real world Java Software. As in our case, the validation can be done in a single, linear pass over the bytecode. However, these approaches are designed limited to types, whereas our approach supports a very rich set of domains especially well-suited for this purpose, including complex properties such as computational and memory cost, non-failure, determinacy, etc. (as we have seen in the examples in this section) and possibly even combining several of them.

Let us mention that, while our approach is general to other programming paradigms, we develop it for concreteness in the context of (Constraint) Logic Programming because this paradigm offers a good number of advantages, especially the maturity and sophistication of the analysis tools available.

4. CONCLUSIONS

Abstract interpretation-based verification forms the cornerstone of the safety model of CiaoPP: the preprocessor of the Ciao multi-paradigm programming system. It ensures the integrity of the runtime environment even in the presence of untrusted code. The framework uses modular, incremental, abstract interpretation as a fundamental tool to infer information about programs. This information is used to certify and validate programs, to detect bugs with respect to partial specifications written using program assertions, to generate and simplify run-time tests and to perform high-level optimizations such as multiple abstract specialization, parallelization, and resource usage control. Among these applications, we herein focus on the use of abstract interpretation-based verification for the purpose of mobile code safety by following the standard PCC methodology. We report on some experiments in CiaoPP at work which illustrate how the actual process of program certification is aided in an implementation of this framework. We also discuss the application of abstract interpretation-based code certification to the area of pervasive computing systems, which may lack computing resources to perform static analysis. We point out that computational properties inferred by CiaoPP can be useful for controlling resource usage and filtering out mobile code which does not meet certain cost requirements. Also, the fact that our approach follows PCC techniques—in
which the certificate is generated outside the device—makes it potentially applicable in this pervasive context. However, controlling it in a perfect way proves far from obvious, and a range of challenging open problems remain as topics for further research. For instance, we plan to study a more precise model of the memory requirements of small devices. The size of certificates needs to be minimized as much as possible to fit in such limited systems. We believe that they can be further reduced by omitting the information which has to be necessarily re-computed by the checker. This is the subject of ongoing research.

5. REFERENCES


in Logic-Based Program Development, number 3049 in
LNCS, pages 234–261. Springer-Verlag, Heidelberg,
Germany, August 2004.

In OOPSALA Workshop on Formal Underpinnings of

type inference for logic programs. In Proc. of SAS’02,