Experiments in Abstract Interpretation-based Code Certification for Pervasive Systems

Elvira Albert[1]
[1]Complutense University of Madrid
elvira@sip.ucm.es

Germán Puebla[2], Manuel Hermenegildo[2,3]
[2]Technical University of Madrid
[3]University of New Mexico (UNM)
{german,herme}@fi.upm.es

Abstract – Proof carrying code *(PCC) is a general methodology for certifying that the execution of an untrusted mobile code is safe. The basic idea is that the code supplier attaches a* certificate *to the mobile code which 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. Recently, the abstract interpretation techniques developed in logic programming have been proposed as a basis for PCC. This extended abstract reports on experiments which illustrate several issues involved in abstract interpretation-based certification. First, we describe the implementation of our system in the context of* `CiaoPP`*: the preprocessor of the* `Ciao` *multi-paradigm 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.*

Keywords: Logic Programming, Abstract Interpretation, Mobile Safety, Proof-Carrying Code.

1 The Framework

Current approaches to mobile code safety, inspired by the technique of *Proof-Carrying Code* (PCC) [15], 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 [15], 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: i) allow expressing interesting safety *properties*, ii) can be generated *automatically* and, iii) are easy and *efficient* to check. In [1], the *abstract interpretation* techniques [5] developed in logic programming[1] 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.

1.1 Certification in the Supplier

In Fig. 1, we illustrate the *certification* process of [1] carried out to generate a safety certificate by the code supplier. It is based on the idea that a *particular subset of the analysis results computed by abstract interpretation-based fixpoint algorithms can play the role of certificate for attesting program safety* [1]. The certification process consists in the next four steps.

Safety Policy. A subset of the high-level *assertion* language of [16] is used to define the safety policy in the context of *CLP programs*. Assertions are syntactic objects which allow us to express "abstract"—i.e. symbolic—properties over different *abstract domains*. Examples are assertions which state information on *entry points* to a program module, assertions which describe properties of built-ins, assertions which provide some type declarations, cost bounds, etc. The certification process starts from an initial program and an

[1]We refer to [2, 6, 12], and their references, for more details on analysis techniques developed in logic programming.

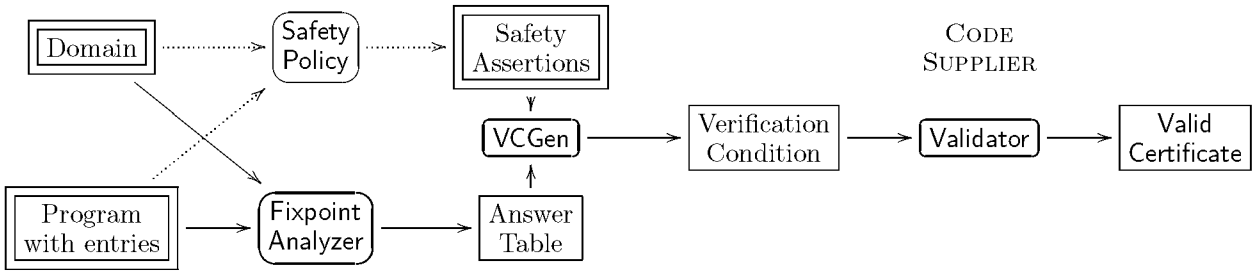

Figure 1: Abstract Interpretation-based Mobile Code Certification in CiaoPP

abstract domain and obtains a set of *safety assertions* from the predefined assertions for system predicates and those provided by the user. The Safety Policy consists in guaranteeing that safety assertions hold for the given program (and entries) in the context of the desired abstract domain.

Fixpoint Analyzer. A main idea in [1] is that the certificate is automatically generated by a fixpoint abstract interpretation-based analyzer. In particular, the *goal dependent* (a.k.a. goal oriented) analyzer of [12] plays the role of Fixpoint Analyzer. This analysis algorithm receives as input, in addition to the program and the abstract domain, a set of *calling patterns* (or entries). A calling pattern is a description of the calling modes into the program. For simplicity, we assume that the program comes enhanced with its entries. Due to space limitations, and given that it is now well understood, we do not describe here the fixpoint algorithm (details can be found in, e.g., [2, 12]). An interesting point to note is that analysis results in [12] are represented by means of two data structures in the output: the *answer table* and the *arc dependency table*. In [1], we show that a particular *subset* of the analysis results—namely the answer table—is sufficient for mobile code certification.

Verification condition generator. Then, a verification condition generator, VCGen, computes from the assertions and the answer table a *verification condition* in order to attest compliance of the program with respect to the safety policy. The formal definition of VCGen is outside the scope of this paper (it can be found in [1]). Intuitively, the verification condition is a conjunction of boolean expressions whose validity ensures the consistency of a set of assertions w.r.t. the answer table computed by the analyzer.

Validator. The condition is sent to an automatic Validator which attempts to check its validity w.r.t. the answer table. This validation may yield three different possible status: i) the verification condition is indeed checked and the answer table is considered a Valid Certificate, ii) it is disproved, and thus the certificate is not valid and the code is definitely not safe to run (we should obviously correct the program before continuing the process); iii) it cannot be proved nor disproved, which may be due to several circumstances. For instance, it can happen that the analysis is not able to infer precise enough information to verify the conditions. The user can then provide a more refined description of initial calling patterns or choose a different, finer-grained, domain. Although, it is not showed in the picture, in both the ii) and iii) cases, the certification process needs to be restarted until achieving a verification condition which meets i). If it succeeds, the answer table constitutes a valid certificate and can be sent to the consumer together with the program.

1.2 Validation in the Consumer

The *validation* process of [1] performed by the code consumer is similar to the certification process described in Fig. 1 by replacing the fixpoint analyzer by an Analysis Checker. Indeed, the supplier sends the program together with the certificate to the consumer and, to retain the safety guarantees, the consumer can trust neither the code nor the certificate. Thus, in the *validation* process, a code consumer not only checks the validity of the answer table but it also (re-)generates a trustworthy verification condition, as it is done by the supplier in the above figure.

The whole validation process is centered around the following observation: *the checking algorithm can be defined as a very simplified "one-pass" analyzer* [1]. Intuitively since the certification process already provides the fixpoint result as certificate, an additional analysis pass over it cannot change the result. Thus, as long as the answer table is valid, one single execution of the abstract interpreter validates the certificate. The definition of the checker can be found in [1].

2 Experiments in CiaoPP

The above abstract interpretation-based code certification framework has been implemented in CiaoPP [11]: 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., [14, 4, 9, 12] and their references). These techniques and systems can approximate at compile-time, always safely, and with a significance 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.

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 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 2.1 (sharing+freeness) *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):*

```
mmultiply([],_,[]).
mmultiply([V0|Rest], V1, [Result|Others]):-
        mmultiply(Rest, V1, Others),
        multiply(V1,V0,Result).

multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
        multiply(Rest, V1, Others),
        vmul(V0,V1,Result).

vmul([],[],0).
vmul([H1|T1], [H2|T2], Result):-
        vmul(T1,T2, Newresult),
        Product is H1*H2,
        Result is Product+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,* var *denotes variables that do not point yet to any data structure,* mshare *denotes pointer sharing patterns between variables and* ground *variables which point to data structures which contain no pointers. The analysis is performed with the following* entry *assertion which allows specifying a restricted class of calls to the predicate.*

```
:- entry mmultiply(X,Y,Z):( var(Z), ground(X),
                            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, among others, the following set of assertions which constitute our safety certificate:*

```
:- true pred A is B+C
       : ( mshare([[A]]),var(A),ground([B,C]) )
      => ( ground([A,B,C]) ).
:- true pred A is B*C
       : ( mshare([[A]]),var(A),ground([B,C]) )
      => ( 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. These 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 [9, 10, 18]. Clearly, type information is very useful for program certification, verification, optimization, debugging (see, e.g., [11]).

Example 2.2 (eterms) *Our next experiment uses the* regular type *domain* eterms *[18] to analyze the same program of Ex. 2.1. 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* `mmultiply` *are performed with matrices in the first two arguments:*

```
:- entry mmultiply(X,Y,Z): (var(Z),
         list(X,list(num)),list(Y,list(num))).
```

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

```
:- true pred A is B+C
         : ( term(A),num(B),num(C) )
        => ( num(A),num(B),num(C) ).

:- true pred A is B*C
         : ( term(A),num(B),num(C) )
        => ( 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. 2.1.*

3 Applications in Pervasive 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 [19]. 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 [7, 8]. 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 3.1 *The following program* `inc_all` *increments all elements of a list by adding one to each of them.*

```
inc_all([],[]).
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:

```
:- entry inc_all(A,B) : (list(A,num),var(B)).
:- check calls inc_all(A,B)
         :  list(A,num).
:- check success inc_all(A,B)
         =>  list(B,num).
:- 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. 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:*

```
:- checked calls inc_all(A,B)
        : list(A,num).
:- checked success inc_all(A,B)
       => list(B,num).
:- checked comp inc_all(A,B)
        : ( list(A,num), var(B) )
        + steps_ub(inc_all(A,B),length(A)+1).
:- true pred inc_all(A,B)
        : ( list(A,num), var(B) )
       => ( list(A,num), list(B,num) )
        + ( not_fails, is_det,
            steps_ub(length(A)+1) ).
```

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 unfit 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 [17] 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 [13] 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.

4 Conclusions

Abstract interpretation-based verification forms the corner stone 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.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. An Abstract Interpretation-based Approach to Mobile Code Safety. Technical Report CLIP8/2003.0, Technical University of Madrid, School of Computer Science, UPM, November 2003.

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

[3] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual (v1.8). Technical Report CLIP4/2002.1, School of Computer Science, UPM, 2002. Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[4] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[5] 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.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 and 3):103–179, 1992.

[7] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Proc. of SAS'94*, number 864 in LNCS, pages 255–265. Springer-Verlag, 1994. Invited Talk.

[8] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *Proc. of ILPS'97*, pages 291–305. MIT Press, Cambridge, MA, 1997.

[9] J. Gallagher and D. de Waal. Fast and Precise Regular Approximations of Logic Programs. In *Proc. of ICLP'94*, pages 599–613. MIT Press, 1994.

[10] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Proc. of PADL'02*, LNCS, pages 243–261, 2002.

[11] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *Proc. of SAS'03*, pages 127–152. Springer LNCS 2694, 2003.

[12] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[13] K. Klohs and U. Kastens. Memory Requirements of Java Bytecode Verification on Limited Devices. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, 2004.

[14] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.

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

[16] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.

[17] K. Rose, E. Rose. Lightweight bytecode verification. In *OOPSALA Workshop on Formal Underpinnings of Java*, 1998.

[18] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *Proc. of SAS'02*, pages 102–116. Springer LNCS 2477, 2002.

[19] M. Weiser. The computer for the twenty-first century. *Scientific American*, 3(265):94–104, September 1991.