An Abstract Interpretation-based Approach to Mobile Code Safety

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

[1] *Complutense University of Madrid,* elvira@sip.ucm.es

[2] *Technical University of Madrid,* {german,herme}@fi.upm.es

[3] *University of New Mexico,* herme@unm.edu

Abstract

Recent approaches to mobile code safety, like *proof-carrying code*, involve associating safety information to programs. The code supplier provides a program and also includes with it a certificate (or *proof*) whose validity entails compliance with a predefined safety policy. The intended benefit is that the program consumer can locally validate the certificate w.r.t. the "untrusted" program by means of a certificate checker—a process which should be much simpler, efficient, and automatic than generating the original proof. We herein introduce a novel approach to mobile code safety which follows a similar scheme, but which is based throughout on the use of abstract interpretation techniques. In our framework the safety policy is specified by using an expressive assertion language defined over abstract domains. We identify a particular *slice* of the abstract interpretation-based static analysis results which is especially useful as a certificate. The validity of the certificate on the consumer side is checked by a very simplified and efficient specialized abstract-interpreter. Our ideas are illustrated through an example implemented in the context of constraint logic programs, using the CiaoPP system. Though further experimentation is still required, we believe the proposed approach is of interest for bringing the automation and expressiveness which is inherent in the abstract interpretation techniques to the area of mobile code safety.

Key words: Mobile Code Safety, Certifying Compilation, Proof-Carrying Code, Abstract Interpretation, Static Analysis.

1 Introduction

One of the most important challenges which computing research faces today is the development of security techniques for verifying that the execution of a program (possibly) supplied by an untrusted source is *safe*, i.e., it meets certain properties according to a predefined *safety policy*. Recent approaches to mobile code safety involve associating safety information in the form of

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

a *certificate* to programs [13,11,12]. The certificate (or proof) is created at compile time, and packaged along with the untrusted code. The consumer who receives or downloads the code+certificate package can then run a *verifier* 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 "certificate-based" approach to mobile code safety is that the burden of ensuring compliance with the desired safety policy is shifted from the consumer to the supplier. The consumer's task is reduced from the level of proving to the level of checking. Indeed the verifier, or proof checker, performs a task that should be much simpler, efficient, and automatic than generating the original certificate. Well-known methods following this approach are, among others, Proof-Carrying Code (PCC) [13] and Typed Assembly Languages (TAL) [12]. An interesting point to note is that the certificate may take different forms. For instance, in PCC the certificate is originally a proof in first-order logic of certain verification conditions and the verification process involves checking that the certificate is indeed a valid first-order proof. A recent proposal [1] uses temporal logic to specify security policies in PCC. In TAL, the certificate is a type annotation of the assembly language program and the verification process involves a form of type checking. Nevertheless, the design of mobile code safety systems based on certificates shares the same, fundamental, challenges:

(i) defining *expressive safety policies* covering a wide range of properties,

(ii) solving the problem of how to *automatically generate the certificates* and,

(iii) designing *simple, reliable, and efficient checkers* for the certificates.

The various approaches differ in expressiveness, flexibility, and efficiency, but they all share the common goal of using safety information to make the local execution of untrusted mobile code by the consumer safe and efficient. Our main contribution is to introduce a novel approach to certificate-based mobile code safety which follows the overall scheme, but which is based throughout on the use of the technique of *abstract interpretation* [3] in order to handle the fundamental and difficult issues mentioned above.

A starting point of our work is the observation that the now well established technique of abstract interpretation has allowed the development of very sophisticated global static program analyses which are at the same time automatic, provably correct, and practical. The basic idea of abstract interpretation is to infer information on programs by interpreting ("running") them using abstract values rather than concrete ones, thus, obtaining safe approximations of programs behavior. The technique allows inferring much richer information than, for example, traditional types. This includes data structure shape (like pointer sharing), 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). `CiaoPP` [7] is the abstract interpretation-based preprocessor of the `Ciao` multi-paradigm constraint logic programming system. It uses modular, incremental abstract interpretation as a fundamental tool to obtain information about the program. In `CiaoPP`, the semantic approximations produced by the analysis have been applied to high- and low-level optimizations during program compilation, including transformations such as multiple abstract specialization, parallelization, and resource usage control. More recently, novel and promising applications of such semantic approximations have been proposed in the more general context of program development.

In the context of the `CiaoPP` system, we herein introduce a novel approach to mobile code safety which follows a certificate-based scheme, but which is based throughout on the technique of abstract interpretation. The design of our abstract interpretation-based system is made up of three main elements:

(i) An expressive assertion language used to define the safety policy. Assertions allow us to express "abstract"—i.e. symbolic—properties over different abstract domains. Our framework is parametric w.r.t. the abstract domain of interest, which gives us generality and expressiveness.

(ii) A fixpoint static analyzer is used to automatically infer information about the mobile code which can then be used to prove that the code is safe w.r.t. the given assertions in a straightforward way. We identify the particular *slice* of the analysis results which is sufficient for this purpose.

(iii) A simple, easy-to-trust analysis checker verifies the validity of the information on the mobile code. It is indeed a specialized abstract interpreter which does not need to iterate in order to reach a fixpoint (in contrast to standard analyzers). Efficiency is achieved by taking advantage of the analysis information gathered in a previous analysis phase.

The resulting scheme has been incorporated in the `CiaoPP` preprocessor and its efficiency is now in the process of being experimentally evaluated.

The paper is organized as follows. Section 2 describes the assertion language which is used to define our safety policy. Section 3 presents the certification process together with the generation of the verification condition to attest compliance with the safety policy. In Section 4, we outline the process of checking the validity of the safety information. Finally, Section 5 discusses the work presented in this paper together with related work.

2 An Assertion Language to Specify the Safety Policy

The purpose of a *safety policy* is to specify precisely the conditions under which the execution of a program is considered safe. In existing approaches, safety policies usually correspond to some variants of type safety (which may also control the correct access of memory or array bounds [14]). We propose the use of (a subset of) the high-level *assertion* language [15] available in `CiaoPP` to define the safety policy in the context of *constraint logic programs*.

```
create_streams([],[]).
create_streams([N|NL],[F|FL]):-
        number_codes(N,ChInN), generate(ChInN,Fname),
        safe_open(Fname,write,F), create_streams(NL,FL).

generate(ChInN,Fname):- app("/tmp/",ChInN,Fname).

safe_open(Fname,Mode,Stream):-
        atom_codes(File,Fname), open(File,Mode,Stream).
```

Fig. 1. Example mobile code

2.1 Preliminaries

We start by introducing some notation and preliminary concepts on constraint logic programming [9] (CLP). Terms are constructed from variables (e.g., X), functors (e.g., f) and predicates (e.g., p). We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for all $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$) and $\sigma(X) = X$ for any other variable X, where t_i are terms. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} \equiv \rho^{-1}\rho \equiv id$.

A *constraint* is essentially a conjunction of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H :- B where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of rules.

Example 2.1 Let us consider the CLP program in Figure 1. The main predicate, `create_streams/2`, receives a list of numbers as first argument and returns in the second argument the list of file handlers (*streams*) associated to the opened files. Predicates `number_codes/2`, `atom_codes/2`, and `open/3` are ISO-standard Prolog predicates, and thus they are available in `CiaoPP`. In our example, the call `number_codes(N, ChInN)` receives the number `N` and returns in `ChInN` the list of the ASCII codes of the characters comprising a representation of `N`. Also, the call `atom_codes(File, Fname)` receives in `Fname` a list of ASCII codes and returns the atom `File` made up of the corresponding characters. A call `open(File, Mode, Stream)` opens the file named `File` and returns in `Stream` the stream associated with the file. The argument `Mode` can have any of the values: `read`, `write`, or `append`.

The auxiliary predicate `generate` concatenates the prefix "`/tmp/`" to the number which receives as first parameter by using the well-known list concatenation predicate `app/3`. Note that predicate `create_streams` does not call the system predicate `open` directly, but instead calls the auxiliary predicate safe_open. The reason for this will be discussed in Example 2.3.

2.2 Abstract Properties

Assertions are syntactic objects which allow expressing a wide variety of high-level properties of (in our case CLP-) programs. 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. A distinguishing feature of our approach is that safety properties are expressed as *substitutions* in the context of an *abstract domain* (D_α) which is simpler than the *concrete domain* (D). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain. Our approach relies on the abstract interpretation theory [3], where the set of all possible abstract semantic values which represents D_α is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets, both for the concrete $\langle 2^D, \subseteq\rangle$ and abstract $\langle D_\alpha, \sqsubseteq\rangle$ domains. Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle\alpha, \gamma\rangle$: *abstraction* $\alpha : 2^D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow 2^D$, such that $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x$ 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.

In this framework an *abstract property* is defined as an abstract substitution which allows us to express properties, in terms of an abstract domain, that the execution of a program must satisfy.

The description domain that we use in our examples is a *regular type* domain [4]. We will often refer to this domain as *eterms* [19] since it is the name it has in CiaoPP. A regular type is a set of terms which can be described by a regular term grammar or, equivalently, by a finite tree automaton. In order to define a regular type, one can choose *Regular Unary Logic* programs as a representation of tree automata (like [5,6]). We also adopt this representation as Ex. 2.2 will illustrate. Abstract substitutions in the *eterms* domain over a set of variables V assign a *regular type* to each variable in V. Apart from the user's defined regular types, in the *eterms* domain, we consider a number of distinguished symbols which correspond to predefined types. For instance, we will use in our examples `term`, which is the most general type, i.e., it corresponds to all possible terms. The type `constant` denotes functors with zero arguments, `num`, the set of all possible numbers, `string`, lists of characters, `list`, any possible list, `io_term` the modes of accessing files (i.e., `write`, `read` or `append`), and `stream`, handlers for sequential files. We allow parametric types such as `list(T)` which denotes lists whose elements are all of type `T`. Note that the type `list` is equivalent to `list(term)`. Clearly, `list(T)` $\sqsubseteq$ `list` $\sqsubseteq$ `term` for any type `T`. In *eterms*, the most general substitution $\top$ assigns `term` to all variables in V. The least general substitution $\perp$ assigns the

```
:- regtype safe_name/1.
safe_name("/tmp/"||L) :- list(L,alphanum_code).

:- regtype alphanum_code/1.
alphanum_code(X):- alpha_code(X).
alphanum_code(X):- num_code(X).

:- regtype alpha_code/1.
alpha_code(A):- member(A,"abcdefghijklmnopqrstuvwzyz").
alpha_code(A):- member(A,"ABCDEFGHIJKLMNOPQRSTUVWXYZ").

:- regtype num_code/1.
num_code(C):- member(C,"0123456789").
```

Fig. 2. Regular types for the example

empty set of values to each variable. For brevity, in the examples we often skip variables whose type is the most general substitution (i.e., `term`).

Example 2.2 In the context of mobile code, it is a safety issue whether the code tries to access files which are not related to the application in the machine consuming the code. A very simple safety policy can be to enforce that the mobile code only accesses temporary files. For example, in a UNIX system this can be controlled (under some assumptions) by ensuring that the file resides in the directory `/tmp/`.

The Regular Unary Logic program `safe_name` in Figure 2 defines a regular type such that all its values satisfy this very simple notion of safety. The following abstract property made up of the abstract substitution $\{\mathtt{X} \mapsto \mathtt{safe\_name}\}$ expresses that `X` be bound to a string which starts by the prefix "`/tmp/`" followed by a list of alpha-numerical characters. In the following, we write simply `safe_name(X)` to represent the previous abstract substitution. The `regtype` declarations are used to define new regular types in `CiaoPP`. In fact, auxiliary predicates used to define a regular type, like `alphanum_code`, `alpha_code`, or `num_code` must be declared using `regtype` as well. The construction `member(C,"0123456789")` is a shortcut for expressing that `C` can correspond to any of the codes in the list from character 0 to 9.

2.3 The Safety Policy

The original assertion language [15] available in `CiaoPP` is composed of several assertion schemes. Among them, we simply consider the two following schemes for the purpose of this paper, which intuitively correspond to the traditional pre- and postcondition on procedures.

calls$(B, \{\lambda^1_{Pre}; \ldots; \lambda^n_{Pre}\})$**:** They express properties which should hold in *any* call to a given predicate similarly to the traditional precondition. B is a *predicate descriptor*, i.e., it has a predicate symbol as main functor and all

`calls(number_codes(X,Y), {(num(X);list(Y,numcodes))})`
`calls(atom_codes(X,Y), {(constant(X);string(Y))})`
`calls(open(X,Y,_Z), {constant(X),io_mode(Y)})`
`calls(safe_open(Fname,_,_), {safe_name(Fname)})`
`success(number_codes(X,Y), ⊤, {num(X),list(Y,numcodes)})`
`success(atom_codes(X,Y),⊤, {constant(X),string(Y)})`
`success(open(X,Y,Z), ⊤, {constant(X),io_mode(Y),stream(Z)})`

Fig. 3. Assertions for the example

arguments are distinct free variables, and λ^i_{Pre}, $i = 1, \ldots, n$, are abstract properties about execution states. The resulting assertion should be interpreted as "in all activations of B at least one property λ^i_{Pre} should hold in the calling state."

success$(B, [\lambda_{Pre},]\lambda_{Post})$**:** This assertion schema is used to describe a *post-condition* which must hold on all success states for a given predicate. In the assertion, B is a predicate descriptor, and λ_{Pre} and λ_{Post} are abstract properties about execution states. λ_{Pre} is optional and must be evaluated w.r.t. the store at the calling state to the predicate. However, the condition λ_{Post} must be evaluated w.r.t. the store at the success state of the predicate. If the optional λ_{Pre} is present, then λ_{Post} is only required to hold in those success states which correspond to call states satisfying λ_{Pre}. Note that several success assertions with different λ_{Pre} may be given.

Therefore, abstract properties λ_{Pre} and λ_{Post} in assertions allow us to express conditions, in terms of an abstract domain, that the execution of a program must satisfy. Each condition is an abstract substitution corresponding to the variables in some atom.

In general, it is the task of the compiler designer to define the safety policy associated to the system. In the `CiaoPP` precompiler, the above assertion language allows us to define the safety policy for the run-time system in the presence of foreign functions, built-ins, etc.

Example 2.3 Figure 3 shows the assertions which are relevant to the program in our running example. The first four rows correspond to `calls` assertions, whereas the last three are `success` assertions. Out of the four `calls`, the first three are predefined in the system. The last user-defined assertion for predicate `safe_open` provides a simple way to guarantee that all calls to `open` are safe. It can be read as "the calling conventions for predicate `safe_open` require that the first argument be a `safe_name`". Let us note that the actual implementation in the `CiaoPP` system also includes *program point* assertions [15] which avoid the use of auxiliary predicates such as this one. For simplicity, we do not discuss program point assertions here. The safety policy in our example corresponds to guaranteeing that the program satisfies all the seven assertions in the figure.

The coexistence of different domains in `CiaoPP` [7] allows expressing a wide range of properties using the assertion language. They include modes, types, non-failure, termination, determinacy, non-suspension, non-floundering and cost bounds and their combinations. This idea is related to the notion of *models* addressed in the work on Model-Carrying Code (MCC) [18] to capture the security-relevant properties of code. MCC enables code consumers to try out different security policies of interest and select one that can be statically proved to be consistent with the model associated to the untrusted code. In our framework, the coexistence of different domains recalls the existence of various models in MCC. However, models are intended to describe low-level properties and their combination has not been approached, which differs from our idea of combining (high-level) abstract domains.

In contrast to other approaches, assertions are not compulsory for every predicate. This is important since assertions have to be provided manually. Thus, the user can decide how much effort to put into writing assertions: the more of them there are, the more complete the partial correctness of the program is described and more possibilities to detect problems. However, pre- and post-conditions are often provided by programmers since they are often easy to write and very useful for generating program documentation. Furthermore, assertions are helpful but not actually required in order to obtain information about the program: the analysis algorithm is able to obtain safe approximations of the program behavior even if no assertions are given. This is not always the case in other approaches such as classical program verification, in which loop invariants are actually required. Such invariants are hard to find and existing automated techniques are generally not sufficient to infer them, so that often they have to be provided by hand.

3 Certifying Programs by Static Analysis

This section describes the *certification* process, i.e., the generation of a certificate to attest the adherence of the program to the safety policy. The whole certification method is based on the following idea: *a particular slice of the analysis results computed by abstract interpretation-based fixpoint algorithms can play the role of certificate for attesting program safety.* Intuitively, our certification process performs the following steps. We start from a set, AS, of assertions which establishes the safety policy associated to a program, P, in the context of an abstract domain, D_α, as defined in Sect. 2. Firstly, a standard program analyzer is run, which returns, among other data structures, an *answer table*, AT, encoding relevant information about P's execution (in terms of the abstract domain D_α). Secondly, a verification condition, $VC(AS, AT)$, is generated from AS and AT in order to attest compliance of P with respect to the safety policy. The condition $VC(AS, AT)$ is sent to an automatic verifier which attempts to validate it. If it succeeds, AT constitutes the certificate and can be sent to the consumer together with the program P. Sections 3.1 and 3.2 give further details on elements AT and $VC(AS, AT)$, respectively.

3.1 Using Analysis Results as Certificates

A main idea in our certification process is that the certificate is automatically generated by a fixpoint abstract interpretation-based analyzer. In particular, we rely on the *goal dependent* (a.k.a. goal oriented) analyzer of [8] which is the one implemented in the `CiaoPP` system. This analysis algorithm (we simply write *Analysis* for short in the following) receives as input, in addition to the program P, a set of *calling patterns*. A calling pattern is a description of the calling modes (or entries) into the program. In particular, for an abstract domain D_α, a set of calling patterns Q consists of a set of pairs of the form $\langle A : CP \rangle$ where A is a predicate descriptor and CP is an abstract substitution (i.e., a condition of the run-time bindings) of A expressed as $CP \in D_\alpha$. In principle, calling patterns are only required for exported predicates. The analysis algorithm is able to generate them automatically for the remaining internal predicates. Nevertheless, they can still be automatically generated by assuming $\top$ as the abstract substitution (i.e., no initial information is given) for all exported predicates. Although the idea is to improve this information by means of the initial calling patterns.

In order to compute $Analysis(P, Q, D_\alpha)$, traditional (goal dependent) abstract interpreters for (C)LP programs construct an *and–or graph* (or analysis graph for short) which corresponds to (or approximates) the abstract semantics of the program [2]. The graph has two sorts of nodes: *or–nodes* and *and–nodes*. Or–nodes correspond to literals whilst and–nodes to rules. Both kinds of nodes are connected as follows. Or–nodes have arcs to those and–nodes which correspond to the rules whose head unifies with the literal. An and–node for a rule H :- $B_1, \ldots, B_n$ has n arcs to the or–nodes which corresponds to the literals in the body of the rule. Due to space limitations, and given that it is now well understood, we do not describe here how to compute the and–or graph, or equivalently, $Analysis(P, Q, D_\alpha)$. More details can be found in, e.g., [2,8,16].

The analysis graph computed by `CiaoPP`'s abstract interpreter is represented by means of two data structures in the output: the *answer table* and the *arc dependency table*. The following definition introduces the notion of analysis table (similar definitions can be found, e.g., in [2,8,16]). Informally, it says that its entries are of the form $\langle A : CP \mapsto AP \rangle$ which should be interpreted as "the answer pattern for calls satisfying precondition (or call substitution), CP, to A accomplishes postcondition (or success substitution), AP."

Definition 3.1 [Analysis answer table] Let P be a program. Let Q be a set of calling patterns expressed in the abstract domain D_α. We define an *analysis answer table*, AT, as the set of entries $\langle A_j : CP_j \mapsto AP_j \rangle$, $\forall j = 1..n$ computed by $Analysis(P, Q, D_\alpha)$[8] where, in each entry, A_j is an atom and CP_j and AP_j are, respectively, the abstract call and success substitutions.

Intuitively, the answer table contains the answer patterns for all literals in the or–nodes of the graph while the arc dependency table keeps detailed information about dependencies among or–nodes in the graph. A central idea in this work is that, for certifying program safety, it suffices to send the information stored in the analysis answer table since, in contrast to the original generic algorithm [8], a simple analysis checker can be designed for validating the answer table without requiring the use of the arc dependency table at all (as we show in Sect. 4). The theory of abstract interpretation guarantees that the answer table is a safe approximation of the runtime behavior (see [2,8,16] for details).

Example 3.2 Reconsider the program of Example 2.1 and the abstract domain *eterms* enhanced with the regular type declaration `safe_name` of Example 2.2. Take the calling pattern $\langle$`create_streams(X,Y)`$, \{$`list(X,num)`$\}\rangle$, which indicates that initial calls to `create_streams` are performed with a list of numbers in the first argument. `CiaoPP` computes this answer table for it:

Predicate	Calling Pattern	Success Pattern
`create_streams(A,B)`	`list(A,num)`	`list(A,num),list(B,stream)`
`number_codes(A,B)`	`num(A)`	`num(A),list(B,numcodes)`
`generate(A,B)`	`list(A,numcodes)`	`list(A,numcodes),sf(B)`
`app(A,B,C)`	`A="/tmp/",` `list(B,numcodes)`	`A="/tmp/",` `list(B,numcodes),sf(C)`
`safe_open(A,B,C)`	`sf(A),B=write`	`sf(A),B=write,stream(B)`
`atom_codes(A,B)`	`sf(B)`	`constant(A),sf(B)`
`open(A,B,C)`	`constant(A),` `B=write`	`constant(A),B=write,` `stream(C)`

For instance, the first entry should be interpreted as: all calls to predicate `create_streams` provide as input a list of numbers in the first argument and, upon success, they yield lists of numbers and streams, respectively, in each of its two arguments. It is interesting to note that `CiaoPP` generates the auxiliary type `sf("/tmp/"||A) :- list(A,numcodes).` to represent lists of numbers starting by the prefix `"/tmp/"`. Clearly, `sf` $\sqsubseteq$ `safe_name`. This will allow `CiaoPP` to infer that calls to `open` performed within this program satisfy the simple safety policy discussed in Ex. 2.2. Moreover, we use the notation $Var = constant$ to denote that the system generates a new type whose only element is this constant, as it happens: for `write`, in the entries for `safe_open` and `open` and, for `"/tmp/"`, in the entry for `app`.

In order to increase accuracy, analyzers are usually *multivariant* on calls (see, e.g., [8]). Indeed, though not visible in this example, `CiaoPP` incorporates a multivariant analysis, i.e., more than one triple $\langle A : CP_1 \mapsto AP_1\rangle, \ldots,$ $\langle A : CP_n \mapsto AP_n\rangle$ $n > 1$ with $CP_i \neq AP_i$ for some i, j may be computed for the same predicate descriptor A.

3.2 The Verification Condition

In the next step, the code supplier extracts a *Verification Condition* (VC) which can be proved only if the execution of the code does not violate the safety policy. For an initial set of assertions, we define our VC as follows.

Definition 3.3 [Verification Condition] Let P be a program, Q a set of calling patterns in the abstract domain D_α and AT its analysis answer table. Let S be an assertion. Then, the verification condition, $VC(S, AT)$, for S w.r.t. AT is defined as follows:

$$VC(S,AT) ::= \begin{cases} \displaystyle\bigwedge_{\langle A:CP \mapsto AP\rangle \in AT} (\rho(CP) \sqsubseteq \lambda^1_{Prec} \vee \ldots \vee \rho(CP) \sqsubseteq \lambda^n_{Prec}) \\ \qquad\qquad \text{if } S = \mathit{calls}(B, \{\lambda^1_{Prec}; \ldots ; \lambda^n_{Prec}\}) \\ \displaystyle\bigwedge_{\langle A:CP \mapsto AP\rangle \in AT} \rho(CP) \sqcap \lambda_{Prec} = \bot \vee \rho(AP) \sqsubseteq \lambda_{Post} \\ \qquad\qquad \text{if } S = \mathit{success}(B, \lambda_{Prec}, \lambda_{Post}) \end{cases}$$

where ρ is a variable renaming substitution of A w.r.t. B.

If AS is a finite set of assertions, then its verification condition, $V(AS, AT)$, is the conjunction of the verification conditions of the elements of AS.

Roughly speaking, the VC generated according to Def. 3.3 is a conjunction of boolean expressions (possibly containing disjunctions) whose validity ensures the consistency of a set of assertions w.r.t. the answer table computed by *Analysis*. It distinguishes two different cases depending on the kind of assertion. For *calls* assertions, the VC requires that at least one precondition λ^i_{Prec} be a safe approximation of all existing abstract calling patterns for the atom B. In the case of *success* assertions, there are two cases for them to hold. The first one indicates that the precondition is never satisfied and, thus, the assertion trivially holds (and the postcondition does not need to be tested). The second corresponds to the case in which the success substitutions computed by analysis for the predicate are more particular than the one required by the assertion. Let us illustrate this definition by means of an example.

Example 3.4 Consider the answer table generated in Example 3.2 and the *calls* and *success* assertions of Figure 3. According to Def. 3.3, the VC is:

$$
\begin{aligned}
(\mathtt{num(X)} &\sqsubseteq (\mathtt{num(X)};\mathtt{list(Y,numcodes)})\wedge\\
\mathtt{sf(Y)} &\sqsubseteq (\mathtt{constant(X)};\mathtt{string(Y)})\wedge\\
\mathtt{constant(X)},\mathtt{Y}=\mathtt{write} &\sqsubseteq \mathtt{constant(X)},\mathtt{io\_mode(Y)}\wedge\\
\mathtt{sf(X)} &\sqsubseteq \mathtt{safe\_name(X)}\wedge\\
\mathtt{num(X)},\mathtt{list(Y,numcodes)} &\sqsubseteq \mathtt{num(X)},\mathtt{list(Y,numcodes)}\wedge\\
\mathtt{constant(X)},\mathtt{sf(Y)} &\sqsubseteq \mathtt{constant(X)},\mathtt{string(Y)}\wedge\\
\mathtt{constant(X)},\mathtt{Y}=\mathtt{write},\mathtt{stream(Z)} &\sqsubseteq \mathtt{constant(X)},\mathtt{io\_mode(Y)},\mathtt{stream(Z)})
\end{aligned}
$$

Each conjunct corresponds to an assertion in Fig. 3 in the same order they appear there. Thus, the first four conjuncts are for the *calls* assertions and the last three for the *success* assertions. The validity of the whole conjunction can be easily proved by taking into account the following (trivial) relations between the elements in the domain:

$$
\begin{aligned}
\mathtt{sf(X)} &\sqsubseteq \mathtt{string(X)}\\
\mathtt{X}=\mathtt{write} &\sqsubseteq \mathtt{io\_mode(X)}
\end{aligned}
$$

Note that the first two conjuncts contain a disjunction in the right condition. In the second one, the condition $\mathtt{sf(Y)} \sqsubseteq (\mathtt{constant(X)};\mathtt{string(Y)})$ holds because $\mathtt{sf(Y)} \sqsubseteq \mathtt{string(Y)}$.

Therefore, upon creating the answer table and generating the VC, the validity of the whole boolean condition is checked by resolving each conjunct separately. Note that each conjunct consists of comparisons of pairs of abstract substitutions, which simply return either true or false but do not compute any substitution. This validation may yield three different possible status: i) the VC is indeed checked, as it happens in the above example; 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. In both the ii) and iii) cases the certification process needs to be restarted until achieving a VC which meets i).

4 Checking Safety in the Consumer

After certifying the safety of the code, the supplier sends the program together with the certificate to the consumer. 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 certificate w.r.t. the program but it also (re-)generates a trustworthy VC. This section describes only the former part of the validation process, since the latter is identical to that already discussed in the previous section.

There are at least three reasons for requiring the validation process to be efficient and driven by a simple algorithm. First, the implementation of the checking algorithm is part of the safety-critical infrastructure and we want to minimize it. Second, the local host could be a small embedded system that lacks computing resources to run large and complex programs. Third, the checking will be performed by every consumer (whilst the certification generation is done only once by the supplier).

As already mentioned, *Analysis* plays the role of the certificate generator in our approach. Although global analysis is now routinely used as a practical tool, it is still unacceptable to run the whole *Analysis* to validate the certificate since it still involves considerable cost. One of the main reasons for this is that the fixpoint algorithm is an iterative process which often recomputes 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. The whole validation process is centered around the following observation: *the checking algorithm can be defined as a very simplified "one-traversal" analyzer*. Intuitively, the computation of a fixpoint algorithm, such as *Analysis*, can be understood as: $Analysis = fixpoint(analysis\_step)$. We write explicitly $fixpoint$ to highlight that the analysis can be seen as an iterative process which repeatedly performs a traversal of the analysis graph (denoted by *analysis_step*) until the computed information does not change, i.e., it reaches a fixpoint. The idea is that the simple, non-iterative, *analysis_step* process can play the role of abstract interpretation-based checker. In other words, $\texttt{check} \equiv analysis\_step$. This is justified by the assumption that the certification process already provides the fixpoint result in the form of certificate. Thus, as long as the answer table is valid, an additional analysis pass over it—or equivalently one single execution of *analysis_step*—cannot change the result.

A similar approach is also taken in the verification of Java bytecode [11,10] in the context of embedded systems. As it is done in PCC, the work in [17] proposes to split bytecode verification of the KVM (an embedded variant of the JVM) in two phases: 1) the producer computes the certificate by means of a type-based dataflow analyzer and 2) then the consumer simply checks that the types provided in the code certificate are valid. As it happens in our case, the second phase can be done in a single, linear pass over the bytecode.

5 Discussion

The idea of using the results of abstract interpretation for program verification and debugging is not new. For instance, a dataflow analysis applied to a type-level abstract interpretation of the Java virtual machine is the basis of all existing bytecode verifiers [11,10]. Analysis results allow proving that the program is correct w.r.t. non-trivial correctness conditions. This is also the case in `CiaoPP`, whose combination of abstract interpretation with a flexible assertion language opens the door to many uses of abstract interpretation for program development.

In this paper, we have introduced a novel approach to mobile code safety which follows the standard strategy of associating safety certificates to programs, proposed by PCC and related techniques, but which is based throughout on the use of abstract interpretation. In particular, it differs from PCC in the following aspects. In our case, the burden on the consumer side is reduced by replacing an *analysis* phase with a simple one-*traversal* abstract interpretation-based checker. The certificate takes the form of a particular *slice* of the analysis results generated by an abstract interpreter. The certificate checker on the consumer side is itself in fact a very simplified and efficient specialized abstract-interpreter. The importance of our definition of the checker comes from the fact that, while abstract interpretation is a powerful technique, in return it is not without cost: the results it provides are guaranteed to be correct and often sufficiently precise in order to be useful, but obtaining analysis results is a costly task, mainly due to the fact that an analysis fixpoint has to be reached. The checker that we have proposed, on the other hand, greatly reduces the cost on the receiving side.

Another notable difference is that that our scheme is completely defined at the source-level, whereas in PCC and related approaches the code supplier typically packages the certificate with the untrusted *object* code rather than with the *source* code. From our point of view these two approaches are of interest. In many cases the source code is simply not available to the consumer. Even when there is a choice between object and source code, using object code has the clear advantage that the trusted computing base in the consumer is reduced since there is no need for a compiler. However, open-source code is getting much more relevant these days. As a result, it is now realistic to expect that a relatively large amount of untrusted source code is available to the consumer. Part of our interest in open-source is due to the fact that `Ciao` is itself a GNU-Licensed Prolog System based on the availability of the source code for its reviewing and modification. The advantages of open-source with respect to safety are important since it allows inspecting the code and applying powerful techniques for program analysis and validation which allow inferring information which may be difficult to observe at low-level, compiled code. This enables handling more involved properties which in turn allow more expressive safety policies. Therefore, we share with PCC the idea of reducing the load in the consumer but our method is somehow applied in a different manner.

Acknowledgments

This work was funded in part by projects ASAP (EU IST FET Programme Project Number IST-2001-38059) and CUBICO (MCYT TIC 2002-0055). Part of this work was performed during a research stay of Elvira Albert and Germán Puebla at UNM supported by respective grants from the Secretaría de Estado de Educación y Universidades. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

References

[1] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In *Proc. of CADE'02*, pages 31–46. Springer LNCS, 2002.

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

[3] 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.

[4] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[5] T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. LICS'91*, pages 300–309, 1991.

[6] John P. Gallagher and Julio C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher Order and Symbolic Computation*, 14(2,3):143–172, 2001.

[7] 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.

[8] 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.

[9] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

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

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[12] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

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

[14] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proc. of PLDI'98*. ACM Press, 1998.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.