Secure Mobile Code and Control Flow Analysis [*]

Francisco Bavera, Jorge Aguirre, Martín Nordio
Universidad Nacional de Río Cuarto,
Departamento de Computación
Río Cuarto, Argentina
pancho,jaguirre,nordio@dc.exa.unrc.edu.ar

Abstract

The interaction between software systems by means of mobile code is a powerful and truly effective method, particularly useful for installing and executing code dynamically. However, for this mechanism to be applicable safely, especially in industrial or critical applications, techniques that guarantee foreign code execution safety for the consumer or host will be necessary. Of course, tool support for automating, at least partially, the application of these techniques is essential. The importance of guarantee code execution safety originates numerous active research lines, among which Proof-Carrying Code (PCC) is one of the most successful. One of the problems to overcome for the PCC industrial use is to obtain lineal methods of safeness certification and verification.

A framework for the generation and execution of safe mobile code based on PCC together with techniques for static analysis of control and data-flow, called PCC-SA, was developed later by the authors.

The results of the group that allowed proving the hypothesis that the PCC-SA complexity in practice is lineal respect to the input programs length, as for certification as for verification processes are also presented. To achieve this, a C-program family, whose elements are referred to as *lineally annotative*, is defined. Parameters statically measured over their source code determine whether a program belongs to this family or not. Different properties of this family are demonstrated in this work, which allows formally showing that for all the programs of this family, the PCC-SA presents a lineal behavior. The parameters required for a large sample of programs keeping of standard packages, are calculated. This calculation finally determines that all the programs of the sample are *lineally annotative*, which validates the hypothesis previously stated.

Keywords: Mobile Code, Proof-Carrying Code, Certifying Compilation, Static Analysis, Automated Program Verification.

1 Introduction

Along the last decade, the use of *Information and Telecommunication Technologies* (ITTs) irrupted in all the areas of human activity. The ITTs have been used in a wide range of applications and devices. At the same time, the use of mobile code (generated by one producer and used by either one or numerous consumers) has also increased. The distribution of software via internet is the most evident example of it. Besides, the use of mobile code to transmit updates and new versions of programs reaches even the cellular phone system, intelligent cards, and all those areas in which the incorporation of new functions to devices controlled by software are important. The migration of code offers an advantageous automatic low-cost way of solving the distribution or substitution of code for great masses of users. However, this technique presents disadvantages as well, since the migrated software can put the consumer's security in risk, as for natural as for intentional program faults. Attempts to reach solutions to overcome this disadvantage gave birth to active research lines, among which are those related to certifying compilers generation, which started with *Proof-Carrying Code* (PCC) by G. Necula and P. Lee in 1996. A survey about research works on PCC was published by the authors [2].

Proof-Carrying Code is a technique specially developed to guarantee and statically demonstrate that the software exhibits certain qualities. In particular, this technique focuses on the analysis of critical qualities

[*] This work was supported by grants from the SECyT-UNRC, the Agencia Córdoba Ciencia, and the CONICET

such as type and memory safety. PCC mainly requires that the code producer carries out a formal proof (certificate) that proof that his code satisfies the required properties. The resulting code may either be executed locally by the producer, who is also the consumer in this particular case, or migrate and be executed by other consumers. For the first case, both the compilation and execution environments are reliable. For the second case on the contrary, the only reliable environment for the consumer is its own, since the received code may have been modified intentionally or even not really belong to the producer. For the first case, proving that the generated code satisfies all the security policies is enough proof to certificate. On the contrary, the second case must be considered and treated as a security problem of mobile code. Among the techniques to guarantee security to a consumer of mobile code we can mention all the *Language-Based Security* variants [17, 15, 11, 12, 19].

The *Language-Based Security* approach mainly consists of preserving the relevant information obtained from a high-level language program version inside the compiled code. The extra information, called the certificate, is obtained during the compilation process and is included in the output code. The user can then carry out a safe verification of the code by analyzing it as well as its certificate to confirm that it fits the security policy requirements. If the certificate proves to be secure then the code is and it can be safely executed. The main advantage of this procedure lies on the fact that code producer must assume the costs of guaranteeing the code security (by generating the certificate) whereas the user has only to verify whether this certificate fits the security requirements [12].

Most bibliography about *Language-Based Security* introduces logic frameworks and type systems that guarantee security. The main disadvantage of this approaches is that it lacks offering an efficient and good-sized safety proof.

In previous works [19, 20, 3], the authors presented a secure execution environment, called *Proof-Carrying Code based-on Static Analysis* (PCC-SA). PCC-SA is based on the *Language-Based Security* principles. And uses techniques of static analysis of control and data-flow, commonly used in optimizing compilers. This particular focus was expected to offer lineal solutions respect to the source program length, as for verification of proofs as for their generation. PCC-SA generates an abstract syntactic tree with all the information needed to verify security conditions. The verification process operates on this abstract syntactic tree using static analysis techniques, particularly, those of control and data-flow. In order to enlarge the range of secure programs, dynamic verifications are used in those cases in which the state of the program can not be determined by static analysis at a certain point.

Once the PCC-SA framework was designed, a PCC-SA prototype for a C sub-set was built [19]. This prototype was used to study the behavior of a set of programs for classic algorithms written as Mini programs ad hoc by volunteers [20]. These experiments confirm the hypothesis about the linearity of the temporal framework behavior respect to the program length. However, in order to be able to carry out significant tests, huge code programs banks were necessary. That is how the idea that motivated the present work was born: to characterize a C-program family through certain parameters, statically measured on these programs, for which the certification and verification-time is lineal respect to the code length. And then calculate these parameters for a great variety of programs to see whether they belong to that family.

The present paper describes this work. It also confirms the hypothesis about the PCC-SA complexity, lineal in practice respect to the length of the input programs, as for certification as for verification processes. To achieve this, a C-Program family is defined, whose elements are *lineally annotative*. Parameters statically measured determine whether a program belongs to this family or not. Different properties of this family are demonstrated which allows proving that for all the programs of the family, the PCC-SA shows a lineal behavior. At last, experiments carried out with more than 90.000 functions and more than 4.000.000 statements from standard library or standard software packages are described in the present work. The experiments consisted of measuring parameters and proving that they were it lineally annotative. From this, it was possible to confirm the hypothesis about the practical linearity of PCC-SA. Results strengthen the idea of using PCC-SA, since other current techniques are of higher complexity order, generally exponential.

The present paper is organized as follows: In section 2 the PCC-SA framework is presented and the main advantages and disadvantages of it discussed. Relevant features of the implemented prototype are explained in section 3. In section 4, is devoted to corroborate that the certification and verification processes have linear temporal-complexity in practice. Related works are discussed in section 5. Finally, conclusions and some proposals for future work are given in section 6.

2 The PCC-SA Framework

Figure 1 shows the structure of the proposed framework. Following the conventions used in [17], the undulated boxes represent the code, and the rectangular ones represent the modules that manipulate such code. Moreover, the shadowed boxes represent untrusted entities, while the white ones represent trusted entities belonging to the *Trusted Computing Base* (TCB).

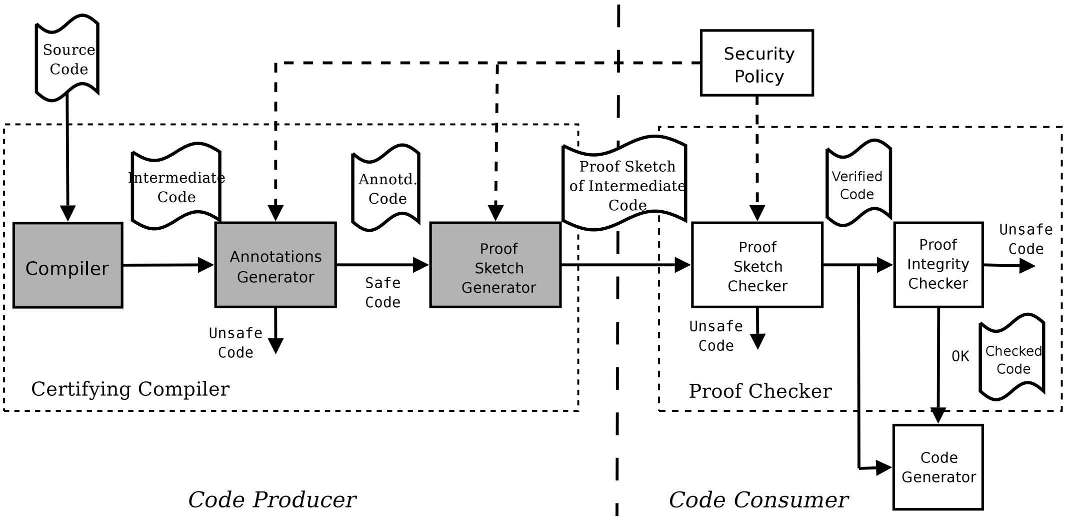

Figure 1: Structure of the framework *Proof-Carrying Code based on Static Analysis*.

Modules *Compiler*, *Annotations Generator*, and *Proof Sketch Generator* constitute the *Certifying Compiler* used by the code producer.

The *Compiler* is a traditional compiler that takes the source code, analyzes it lexically and syntactically to verify the expressions, and produces an intermediate code. This generated code is an abstract representation of the source code, and it could be used independently of the source language and the security policy.

The *Annotations Generator* (GenAnot) applies several static analyses in order to generate the information required to annotate the intermediate code, based on the security policy. If at some point of the program appears that the security policy is not satisfied, then the program is rejected. At those points of the program at which the static analysis techniques cannot assure security, a run-time check is inserted. Thus, if a program succeeds in passing across the GenAnot module, then we can certify that it is safe.

The last process applied by the code producer is the *Proof Sketch Generator*. This process uses all the annotations and the security policy to generate a *Proof Sketch* taking the critical program points and their dependencies into account. This information is stored in the intermediate code. A *Proof Sketch* is the minimal path that the code consumer must check in the intermediate code.

The code consumer uses the *Proof Sketch Checker* to analyze both the annotated intermediate code and the *Proof Sketch* provided by the code producer. After that, the module *Proof Integrity Checker* checks if the proof sketch is strong enough as to prove that the code satisfies the security policy. Its task is to verify that every program critical point either was checked by the *Proof Sketch Checker* or contains a run-time check. From this process, the code consumer can detect modifications in the mobile code or/and even weaknesses in the generation of the *Proof Sketch*.

Both, *Proof Sketch Checker* and *Proof Integrity Checker* belong to the *Proof Checker* that is included in the *Trusted Computing Base* (TCB) of the code consumer.

2.1 Advantages and Disadvantages of PCC-SA

Proof-Carrying Code based on Static Analysis has the same advantages that PCC. Due to the fact that the code consumer has only to provide a fast and simple proof verification, the host infrastructure is automatic and low-risk. On the contrary, the harder task is on the side of the code producer, who must provide the proof. Moreover, trust between producer and consumer is not required. PCC-SA, such as PCC, is a flexible framework since it can be used with different languages and security policies. What's more, it is not only applicable to security. Other advantages are that the code generation can be automated, it can be statically verified, any modification (accidental or malicious) easily detected (even in those cases in which security is guaranteed), and it can be combined with other techniques.

Moreover, PCC-SA has an new set of advantages, produced by the combination of the static analysis and PCC techniques. The most important advantage is that the verification process has a temporal linear complexity w.r.t. the size of the programs. Another important advantage is that the size of the generated proofs is linear w.r.t. the size of the programs (in fact, most of the time the proofs are smaller than the programs). Moreover, a broader range of security policies can be automatically verified using PCC-SA, more platform independence is obtained using a representation of the source code, and less runtime checks are necessary compared with PCC.

Though the idea of PCC-SA is simple, its efficient implementation requires solving some problems. Some weakness points of PCC-SA are inherited from PCC. For example, PCC frameworks are very sensitive to changes in the security policies. More important, establishing and formalizing a security policy is expensive, and both, the code producer and consumer, must be involved in the process.

3 The Prototype Developed

In order to provide a *proof of concept* of our framework, a prototype was developed as follow. First, a source language was defined. To do that, we choose a C-language subset, with some notation added. Second, the security policies were established. In this case, we decided to check for variable initialization and out-of-bound array accesses. Third, the intermediate code was chosen. The prototype uses *abstract syntax trees* as intermediate code. Finally, the module *Annotations Generator*,*Proof Sketch Generator*, *Proof Sketch Checker*, *Proof Integrity Checker* and *Code Generator* were implemented.

The following paragraphs show the main features of the prototype developed.

3.1 The Mini Language

The source language of our prototype is called Mini and it is an extended subset of the C-language programming. Since the security policy is based on the access to arrays, Mini includes array manipulation operations. Almost all the remaining features of the C-language were discarded in order to keep Mini simple.

A Mini program is a function that takes at least one argument and returns a value. The type of arguments can be integer or boolean. One-dimensional arrays, with basic-type elements, can be defined. Note that the complexity of including one-dimensional arrays is equivalent to including more complex data structures, such as matrices.

We say that Mini is an *extended* subset of the C-language because we include a notation to define allowed limits for integer parameters. The declaration of an integer parameter requires the definition of the lower and upper values that such parameter can take. For example, the function profile `int func(int a(0,10))` indicates that when the function `func` is called, the value of its argument should satisfy the condition $0 \leq \mathtt{a} \leq 10$. An explanation of the prototype is given using the example of Figure 2.

3.2 The Security Policy

A security policy is a set of rules that define the conditions under which a program is safe to be executed. A program is a codification of a set of possible runs; thus, a program satisfies a security policy if the security predicate is true for every possible execution path of the program [23].

We chose a security policy that guarantees type and memory safety, whose non-initialized variables are not read, and do not have out-of-bound array accesses.

3.3 Intermediate Code: Abstract Syntax Tree

The intermediate code is an *abstract syntax tree* (ASA). It is an abstract representation of the source code that enables us to apply several static analysis, such as control and data flow analysis. It also can be used to apply code optimizations.

```
int ArraySum ( int index(0,0) ) {

      int [10] data;     /* Define an array */
int value = 1;     /* Define an initialization
                               variable */
int sum = 0;       /* Define the summatory
                                           variable */

                while (index<10) { /* Initialize the array*/
                         data[index] = value;
                         value = value+1;
                         index = index+1;
                    }
                    while (index>0) {  /* Calculate the
                                          summatory */
                          sum   = sum+data[index-1];
                          index = index-1;
                    }
                    return sum;
                }
```

Figure 2: Example of a Mini program.

The abstract syntax trees of the prototype are similar to a traditional ASA, but trees include code annotations. These annotations show the status of the program objects, and contain information about variable initializations, loop invariants, and variable ranges. Figure 3 shows the ASA of the previous example.

Each statement of a program is represented by an ASA. The nodes in an ASA contain a label, information or references to the sub-statements that compose the statement, and a reference to the next statement.

Each expression is represented by a graph. Two different labels are used when an array is accessed: unsafe and safe. These labels mean that it is not safe to access such element of that array and that it is safe to access it, respectively. By modifying the node label, we avoid including run-time checks.

The circles in Figure 3 represent statements, the hexagons represent variables, and the rectangles represent expressions. Arrows show the control flow, and straight lines join statements with their attributes. The label DECL is used for declarations, ASSIGN for assign statements, UNSAFE ASSIG ARRAY for array assign statements, WHILE for loops, and RETURN for function return statements. For example, note that the ASA of the first loop includes the logic condition ($index < 10$) and the body of the loop. The ASA of the body includes three assign statements; the first assigns *value* to the *index* element of the array *data*. So, it is labeled UNSAFE ASSIGN ARRAY.

3.4 The Certifying Compiler CCMini

The certifying compiler CCMini (Certifying Compiler for Mini) is composed by a traditional compiler, an annotation generator, and a proof sketch generator. The *Compiler* takes a program, written in the source language, Mini, and returns an *abstract syntax tree* (ASA). The *Annotation Generator* (GenAnot) applies several control-flow and data-flow static analysis on the ASA and generates an *annotated abstract syntax tree*. Moreover, GenAnot checks the code security requirements by using the information about variable ranges and loop invariants included in the annotations. If the code does not fit the security policies, then it is rejected. If GenAnot cannot determine the security status at any point of the program, then a code for a run-time check is added at this point. Finally, the *Proof Sketch Generator* takes the annotated ASA and generates a *Proof Sketch* by taking the minimal path that the code consumer must follow to verify the code safety.

3.4.1 Code Annotations

The code annotations include loop invariants and the range of each variable, together with the program pre- and post-conditions. In order to obtain this information, GenAnot applies control-flow and data-flow analysis on ASA. These analysis allows obtaining information about the initialization and range of variables and, sometimes, pre- and post-conditions. To know the range of a variable is useful to determine if a variable can be used to access some element of an array.

In order to make the analysis efficient and scalable, the implemented GenAnot module only applies the analysis to the body of the functions. Moreover, our GenAnot is at a middle point between *flow-sensitive analysis*, that considers all the possible paths of a program, and *flow-insensitive analysis*, which does not consider the program flow. GenAnot analyzes control flow in general; but, for example, in loops it only recognizes some patterns in the code.

In order to generate the required information, GenAnot applies the following processes:

1. *Identification of initialized variables.* By analyzing all the possible run-time paths of the program, the access to uninitialized variables is detected. In such case the program is rejected.

2. *Identification of ranges of variables not modified in body loops.* The range of each variable is obtained by taking the range of the function parameters into account and analyzing the execution flow (without considering loops). Analyzing each operation and statement assures that the values of each variable fit their range.

3. *Identification of ranges of induction variables.* A variable can be considered an *induction variable* if its value is increased or decreased by a constant value at each loop iteration. If the loop condition depends upon an induction variable, then it is possible to determine the number of iterations of such loop. Thus, it is possible to determine the ranges of all the induction variables included in the loop and their output values.

4. *Identification of valid array accesses.* The information obtained in the previous phases can be used to determine if most of the array accesses are either valid or out-of-bounds. In particular, it is easy to do this when induction variables are used to access array elements.

The *Proof Sketch Generator* identifies the critical variables (those used as array indexes) and the program points where they are used. This information is used to create a *Proof Sketch*. A *Proof Sketch* is the minimal path on the ASA that the code consumer must check in order to verify the code safety. The *Proof Sketch* for the program in the example is shown in Figure 3.

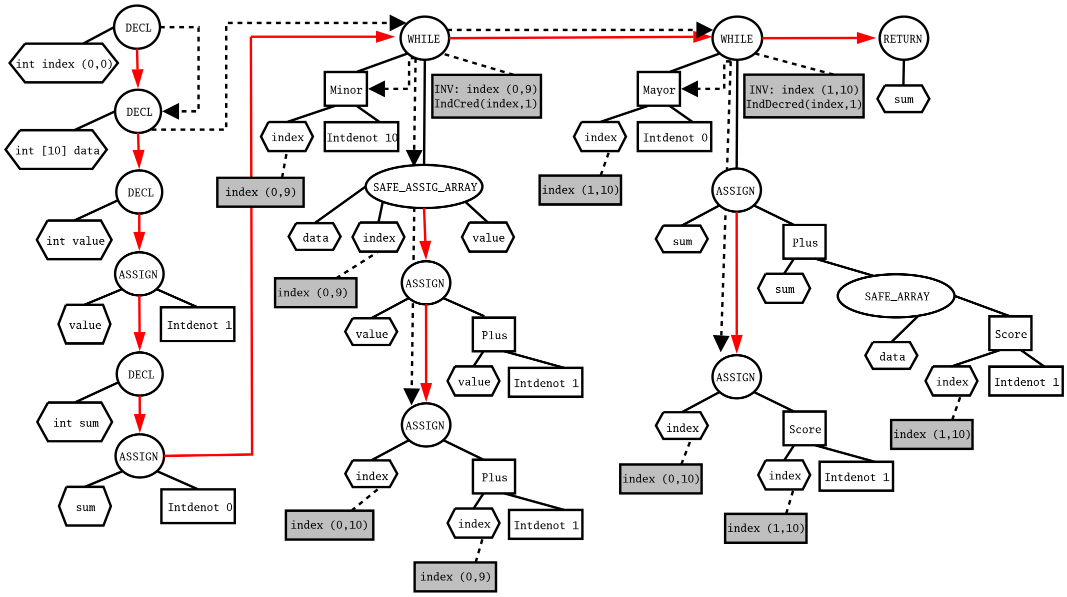

Figure 3: Proof Sketch on the Annotated ASA for the example in Figure 2.

The shadowed rectangles represent annotations and the dotted lines are the proof minimal path. For example, the annotation $index(0, 9)$ means that the value of the $index$ variable is between 0 and 9. The annotations labeled INV : represent loop invariants. For example, $INV : index(0, 9)$ means that the loop invariant is $0 \leq index \leq 9$. The predicate $IndCred(index, 1)$ indicates that the variable $index$ is an induction variable.

Note that in figure 3 the labels of nodes that refer to array accesses are considered safe, but before applying GenAnot, these accesses are considered unsafe. Since each access in the annotated ASA is safe, no run-time checking is needed.

3.5 The Proof Checker

On the side of the code consumer, the *Proof Checker* is composed by the *Proof Sketch Checker* and the *Proof Integrity Verifier*, and its output is sent to the *Code Generator*.

The *Proof Sketch Checker* checks the well-formedness of the received ASA. After that, the path provided by the *Proof Sketch* is analyzed by checking that each variable is initialized and each array access in the path is safe. Each visited node is tagged. For example, first, the *Proof Sketch Checker* follows the path signaled by the pointed arrows in Figure 3.

After that, the *Proof Integrity Checker* analyzes the ASA as a whole, checking that the *Proof Sketch* includes all the critical points in the program. If some critical points, not included in the *Proof Sketch*, is found, then the code becomes unsafe and consequently must be rejected.

Finally, if the code is accepted as safe, the *Code Generator* uses the ASA to generate the object code. Our implementation produces x86 assembly code. By having a separate module as code generator, we can use different modules to generate code in several assembly languages or even binary code. Moreover, an interpreter can be used instead of the code generator module.

4 Code Certification Temporal Complexity Analysis

This section is devoted to demonstrate that for a large family of programs (it seems to include most current programs), the code certification and verification processes, carried out by the PCC-SA prototype, is lineal respect to the source programs length.

Theorem 1 *The quantity of nodes of a ASA is limited by a lineal function dependant on the source code length.*

Proof 1

A- The quantity of steps followed by LR parser is lineal respect to the input chain. This demonstration can be found at A. Aho y J. Ullman ([1], p. 395).

B- The nodes quantity in a LR parsing tree is limited by a lineal function dependant on the input program length. This statement results valid since from the previous true statement, the quantity of steps for a LR parser is lineal respect to the input chain. In addition, to each step correspond at least one node of the LR parsing tree.

C- The ASA used by CCMini is built from the parsing tree taking out all the nodes required by concrete syntaxes and considering only those required by abstract syntaxes.

It can be demonstrated through C that the ASA is the result of parsing tree trimming process and assured through B that the quantity of ASA nodes is limited by a lineal function that limits the quantity of nodes in the syntactic tree. As a conclusion, the quantity of nodes of ASA is limited by a lineal function which depends on the input program length. □

Definition 1 *An A ASA is lineally annotative, given by*
$t = (c_{exp}, c_{exp\_op}, c_{assing}, c_{return}, c_{if}, c_{while}, V, M) \in \mathbb{N}^8$ *if A satisfies the following conditions:*

- *c_l is an upper limit for the necessary computations to generate the annotations for the type l of A ($l \in \{EXP, EXP\_OP, ASSING, RETURN, IF, WHILE\}$).*
- *V is also an upper limit for the times each A node is visited.*
- *M is an upper limit for the variables used in A.*

Where, EXP represents the leaf nodes of the ASA (constant identifiers), EXP P the nodes expressions, ASSING represents the assignments, RETURN the return order, IF the conditional order and WHILE represents all the nodes corresponding to iterative orders.

Definition 2 *Given a tuple $t \in \mathtt{N}^8$, we will name $\mathcal{F}(t)$ the ASA family lineally annotative by t.*

$$\mathcal{F}(t) = \{A : \mathsf{ASA} \mid A \ \textit{is} \ \textit{lineally annotative by } t\}$$

Theorem 2 *Given $t = (c_{exp}, c_{exp\_op}, c_{assing}, c_{return}, c_{if}, c_{while}, V, M)$, for every ASA $\in \mathcal{F}(t)$, the annotation generation process shows a temporal lineal behavior respect to the quantity of nodes of ASA.*

Proof 2 *Let's Γ be the number of computations of the process of generating annotations at each visit to a node.*

Let's also A:ASA be so that $A \in \mathcal{F}(t)$, if $\{node_1, ..., node_n\}$ is the set of A and v_i the number of cycles for visit to `nodei` *during annotation generation process, then the number of computation the process will need is:*

$$\textstyle\sum_{i=1}^{n} v_i * \Gamma(node_i)$$

Let $k = max\{c_{exp}, c_{exp\_op}, c_{assing}, c_{return}, c_{if}, c_{while}\}$ it result:

$$\textstyle\sum_{i=1}^{n} v_i * \Gamma(nodo_i) \leq \sum_{i=1}^{n} V * k,$$

Moreover, it can be observed that:

$$\textstyle\sum_{i=1}^{n} V * k = n * (V * k), \quad \textit{where} \quad (V * k) \in \mathtt{N}$$

Then, the temporal complexity to generate annotations is of $\mathrm{O}(n * V * k) = \mathrm{O}(n)$ order as it was stated. □

Theorem 3 *For each and every program whose A ASA (generated by PCC-SA) is lineally annotative by $t \in \mathtt{N}^8$, the annotation generation process presents a temporal lineal behavior respect to the input code length.*

Proof 3 *This demonstration is quite trivial. Let's $\#A$ be the number of A nodes, then, from theorem 2, it can be assured that annotation generation is $\mathrm{O}(\#A)$ order and from theorem 1, that $\#A$ is the lineal order respect to the input program length.* □

Theorem 4 *For each program whose A ASA (generated by CCMini) is lineally annotative by $t \in \mathtt{N}^8$, then the code verification process shows a lineal order temporal behavior respect to the input program length.*

Proof 4 *Given a ASA A, the A verification process has two steps: (1) annotation generation, whose temporal complexity is lineal respect to the input (statement validated by theorem 1); and (2) annotation verification process, which consists of carefully verifying that all the annotations generated in A satisfy the security policy requirements. Only once each A node is visited in this process. Therefore, its complexity is lineal respect to the input program length.*

In conclusion, the temporal complexity of the verification process is O (input program length). □

Table 1: Observation of C Aplications.

Program	Files	Statements	Depth Max	Amount of Depth Nesting					
			Nested	1	2	3	4	5	6
Mozilla	1365	782.275	6	4186	825	102	20	0	4
gcc	3435	827.385	4	3381	512	46	7	0	0
AbiWord	1601	861.335	6	4749	930	115	24	0	4
Kernel	5196	3.618.436	4	16994	1989	155	14	0	0

4.1 Worst Case Complexity Analysis

The worst case occurs when:

a - The nested cycles can be as many as the whole program. Then, each node is visited as many times as it is possible

$$2*(n-1) \quad = \quad 2*n-2$$

where n is the number of nodes. It must be noted that each nested cycle is only one node smaller that the cycle to which it belongs.

It was demonstrated through Theorem 2 that temporal complexity is $O(n*V*k)$.

Then, given $V = 2*n-2$, the maximum number of times a node can be visited, then, temporal complexity is $O(n*(2*n-2)*k) = O(n^2)$. That is, in this particular case, complexity is a second order polynomial, which is still lower than exponential complexity of most certification processes based on theorem demonstration such as `Touchstone`.

b - k, the maximum number of steps to analyze each node, depends on the number of nodes. It may occur, for instance, when the number of variables is equal to the number of nodes (n), which implies that the cost to recuperate an element from the symbol table depends on n. Then $k = n + c$ and temporal complexity for this particular case is $O(n*V*k) = O(n*V*n+c) = O(n^2)$ (with c and V constant values).

4.2 Measurement of Tuples for Sampling C-Programs

With the purpose of analyzing whether a program belongs to a lineally-limited program family, the tuples associated to all the programs of a huge standard C-program sampling were computed. The sampling consists of: (1) Internet navigator `Mozilla 1.7.8`, (2) `C compiler gcc 3.3.5`, (3) word processor `AbiWord 2.5.5` and (4) `kernel 2.6.11` operative system `Linux`, used by `Gentoo`. All these applications have their own free code under `GNU` license.

Table 1 shows some results from the observation carried out.

These results allow inferring that for real applications, it is quite exceptional to find more than six nested cycles in a function code. It is important to highlight that six nested cycles were found in one file of each analyzed application (the same code in both applications). For most files the nested level usually varies around zero and three (the last column of table 1). It can also be observed that for most cases, the number of variables used in a function body does not exceed the 50 variables. It is important to say that there are some methods of around 3000 lines that use less than 70 variables. Some other results, not displayed in this table, show that the maximum nested level does not exceed level 10. They also show that 2/3 of the cycles correspond to `for` commands. Although the `for` commands in `C` can be widely used, most of them (about 80%) belong to the pattern of inductive variables used to limit cycles.

4.3 Linearity Hypothesis Validation in Practice

From the results obtained in Section 4.2, it was possible to determine that the ASAs of the sampling programs are *lineally annotative* by the tuple $t = (73, 6, 74, 1, 240, 240, 7, 70)$. Therefore, all the sampling programs are lineally limited.

5 Related Work

Language-Based Security: *Proof-Carrying Code* (PCC) [17], *Type Assembly Language* (TAL) [15], *Efficient Code Certification* (ECC) [11, 12] and PCC-SA [19] have something in common: they all use the information generated during the compilation process so that the consumer can be able to verify the code security efficiently. However, they differ in expressiveness, flexibility and efficiency.

The certificate format for each procedure presents different characteristics. For instance, for PCC, it is a first-order-logic sampling of certain verification conditions. For this case, the verification process consists of verifying that the certificate is valid for the adopted logic system and that all the sampling are carried out on a low level mobile code (as for instance, a type assembly language). For TAL on the other hand, the certificate consists of annotations of types and the verification process is a type check. For ECC, the certificate consists of annotations in the code, which provides information about structure and purpose of the code as well as basic information about types.

PCC-SA improves one of the main disadvantages of PCC: the proof size. For most cases, the PCC proof is exponential whereas for PCC-SA, the proof is lineal respect to the code size. This is due to the use of proof schemes.

Java bytecode verification [7, 14] consists of an abstract execution of the class code to check if the type of the values is maintained. In particular, Leroy [13] reduces the verifier in order to apply it on Java-Cards. Nevertheless, some assertions that can be checked in PCC-SA, such as out-of-bounds array accesses, cannot be done by the bytecode verifier.

Control-Flow Analysis: Traditionally, the control-flow and data-flow static analysis techniques to guarantee security were applied locally. That is, the information generated by the analysis is not kept in the generated code. Splint [6] is a certificate compiler that uses control-flow static analysis. For the required analysis and the subsequent certification to be carried out, annotations must be included into the source code by the programmer. Splint guarantees a safe code if he compiled it. But, Splint does not provide any evidence to prove such safety assertion.

J. Bergeron et al. [4] proposes decompiling non-trusted binaries in order to be able to build a control-flow and data-flow graph among others. Then, different static analyses on these graphs are carried out with the purpose of verifying the binaries safeties. This particular point of view does not assume that the code consumer is in a framework, for which the verification process takes place with no information from the producer side.

Well-formed Encoding at the Language Level (WELL)[8] uses a similar approach to PCC-SA. In this case, compressed abstract syntax trees (CASTs) are transmitted to the code consumer. CASTs are safe by construction. That is, a program which does not satisfy the security policy cannot be expressed by a CAST. However, the policies presented in the mentioned work include only escape analysis.

Ccured [18] is a tool that processes C applications and analyzes them for type safety. Where the analysis fails, Ccured adds run-time checks to guarantee the application safety. Because Ccured can access the source code, it can detect more errors than other compilers and at an order-of-magnitude lower run-time code. But, he does not provide any evidence to prove such safety assertion.

Certifying Compilers: Necula and Lee developed the first certifying compiler, Touchstone [17]. After that, several advances in certifying compilation techniques were introduced by the certifying compilers Special J [5], Cyclone [10, 9], TIL [24], FLINT/ML [22], and Popcorn [15]. Furthermore, the well-know javac compiler of Sun, which produces Java bytecode, is considered a certifying compiler. The purpose of this compiler is similar to that of Special J's, but its output is simpler because the bytecode language is more abstract than the generated by Special J. TIL and FLINT/ML compilers maintain type information through the compilation process, but this information is eliminated after the code generation. On their sides, Popcorn and Cyclone are certifying compilers whose target language is the typed assembly language TAL [15], which allows including type information in the generated code.

6 Conclusions and Future Work

The hypothesis (about the linearity of the PCC-SA complexity respect to the input program lengths, as for certification as for verification processes) was confirmed. Evidence of profits using control-flow and data-flow static analysis to guarantee a safe execution of the mobile code was presented.

One of the most important characteristics of the framework is the lineal size of the proof (respect to the produced program). In most cases, the proof size is smaller than that of the programs. Only for the worst case, the proof and the program size are the same.

The source language of the PCC-SA prototype developed is simpler than that of Touchstone (a PCC prototype), but our prototype can be easily extended. Furthermore, the most relevant core of Touchstone is similar to that of the developed prototype. The advantages and disadvantages of our prototype over Touchstone are similar to those of PCC over PCC-SA.

This is a small first step but we believe that it is a very important contribution to build a prototype with lineal behavior that allows generating safe mobile applications based on static analysis techniques and PCC.

So, the task ahead is to develop a certifying compiler for a realistic programming language or, at least, a more comprehensive subset of a realistic programming Language. We are also interested in extending the security policy of the prototype including, for instance, the treatment of pointer arithmetic.

It would be interesting to include fix-point analysis to limit the cycles and analyze costs and efficiency of using this analysis results. This type of analysis allows limiting most of the cycles (sometimes all of them). In order for the consumer to reduce costs, it would be possible to include a fix-point candidate to the sampling scheme. The process of verifying if a candidate is fix-point is simple and it can be done in a single one pass.

Moreover, we intend to do some research on three directions: abstract interpretation, Type assembly languages and security policies specified by automata. Currently, we are studying Information Flow [16] and Declassification [21] with the objective of improving the prototype with these analyses.

Acknowledgments The authors would like to thank Ricardo Medel and Gabriel Baum by their collaboration in the development of PCC-SA in previous works.

References

[1] A. Aho, J. Ullman. The Theory of Parsing, Translation, and Compiling. Volumen I: Parsing. Prentice-Hall. 1972.

[2] F. Bavera, M. Nordio, R. Medel, J. Aguirre, G. Baum, M. Arroyo. "Un Survey sobre Proof-Carrying Code". *5to AST, JAIIO 2004*. University of Córdoba (Argentina). September 2004.

[3] F. Bavera. "Compilación y Certificación de Código mediante Análisis Estático de Flujo de Control y de Datos". Master Thesis. INCO, Universiy of Repúblic, Montevideo, Uruguay. December 2005.

[4] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, N. Tawbi. "Static Detection of malicious Code in Executable Programs". LSFM Research Group, Departament of Informatic, University of Laval, Canada. 2001.

[5] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, K. Cline. "A certifying compiler for Java".*Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'00), pp. 95–105, ACM Press, Vancouver (Canada), June 2000.

[6] D. Evans and D. Larochelle. "Improving Security Using Extensible Lightweight Static Analysis". IEEE Software, pp. 42-51, January-February 2002

[7] J. Gosling. "Java intermediate bytecodes". Proc. ACM SIGPLAN Workshop on Intermediate Representations, pages 111-118. ACM, 1995.

[8] V. Haldar, C. Stork, M. Franz. "Tamper-Proof Annotations by Construction". Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine, March 2002.

[9] L. Hornof, T. Jim. "Certifying Compilation and Run-Time Code Generation". *Proceedings of ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation* (PEPM), pp. 60–74, ACM Press, San Antonio, Texas (EE.UU.), January 1999.

[10] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang. "Cyclone: A safe dialect

[11] D. Kozen. "Efficient Code Certification". Tech. Report 98-1661, Cornell Univ., 1998.

[12] D. Kozen. “Language-Based Security”. Proc. Conf. Mathematical Foundations of Computer Science (MFCS’99), Lecture Notes in Computer Science v. 1672, pp. 284-298, Springer-Verlag, 1999.

[13] X. Leroy, “Bytecode Verification on Java smart cards”. *Proceedings Software Practice and Experience.* 2002.

[14] T. Lindholm, F. Yellin. “The Java Virtual Maquine Specification”. The Java Series. Addison-Wesley, 1999. Second Edition.

[15] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, S. Zdancewic. “TALx86: A Realistic Typed Assembly Language”. *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pp. 25–35, ACM Press, Atlanta, Georgia (EE.UU.). May 1999.

[16] A. Myers, A. Sabefeld. “Language-Based Information-Flow Security ”. IEEE Journal on Selected Areas in Communications, Vol. 21, No 1. January 2001.

[17] G. Necula. “Compiling with Proofs”. Ph.D. Thesis School of Computer Science, Carnegie Mellon University CMU-CS-98-154. 1998.

[18] G. Necula, J. Condit, M. Harren, S. McPeak, W. Weimer. “CCured: type-safe retrofitting of legacy software”. ACM Transaction on Programming Languages and Systems (TOPLAS). Vol. 27 - Num. 3 - pp. 477-526. 2005.

[19] M. Nordio, F. Bavera, R. Medel, J. Aguirre, G. Baum. “A Framework for Execution of Secure Mobile Code based on Static Analysis”. *XXIV International Conference of the Chilean Computer Science Society.* Universidad de Tarapaca, Arica (Chile), November 2004, pp. 59–66. IEEE Computer Society Press. 2004.

[20] M. Nordio. “Verificación de la Seguridad del Código Foráneo mediante Análisis Estático de Control de Flujo y de Datos ”. Master Thesis, INCO, University of Republic, Montevideo, Uruguay. April 2005.

[21] A. Sabelfeld, D. Sands. “Dimensions and Principles of Declassification”. IEEE Computer Security Foundations Workshop (CSFW 2005). Session 9 - pp. 255-269. 2005.

[22] Z. Shao. “An Overview of the FLINT/ML Compiler”. *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation* (TIC’97), ACM Press, Amsterdam (Holanda). June 1997.

[23] Fred B. Scheneider. “Enforceable security policies”. Computer Science Technical Report TR98-1644, Cornell University, Computer Science Department, September 1998.

[24] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, P. Lee. “TIL: A Type-Directed Optimizing Compiler for ML”. *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI’96), pp. 181–192, ACM Press, Philadelphia, Pennsylvania (EE.UU.), May 1996.