A Framework for Execution of Secure Mobile Code based on Static Analysis*

Martín Nordio
Universidad Nacional de Río Cuarto,
Departamento de Computación
Río Cuarto, Argentina
nordio@dc.exa.unrc.edu.ar

Francisco Bavera
Universidad Nacional de Río Cuarto,
Departamento de Computación
Río Cuarto, Argentina
pancho@dc.exa.unrc.edu.ar

Ricardo Medel
Stevens Institute of Technology,
New Jersey, EE.UU.
rmedel@cs.stevens-tech.edu

Jorge Aguirre
Universidad Nacional de Río Cuarto,
Departamento de Computación
Río Cuarto, Argentina
jaguirre@dc.exa.unrc.edu.ar

Gabriel Baum
Universidad Nacional de La Plata, LIFIA
La Plata, Argentina
gbaum@sol.info.unlp.edu.ar

Abstract

Since its conception, Proof-Carrying Code *(PCC) woke up the interest of the research community and several methods based on this technique were developed. This technique guarantees that untrusted programs run safely in a host machine. In a PCC framework, the code producer equips the produced code with a formal proof establishing that the code satisfy the consumer's security policies. So, the code consumer only needs to verify such proof before the execution of the mobile code. On the other hand, static analysis is a technique useful for the production of the information required to construct the mentioned proof. Based on these two techniques, PCC and static analysis, we developed a framework that guarantees the safe execution of mobile code. This framework uses a high-level intermediate language to verify the security of the code. A control flow graph or an abstract syntax tree with type annotations could be used. Such intermediate representations of the code enable us to use static analysis techniques to generate and verify the type information needed. Moreover, we implemented a prototype as a proof of concept for our framework.*

* This work was supported by grants from the SECyT-UNRC, the Agencia Córdoba Ciencia, and the NSF project *CAREER:* #0093362.

Keywords: Mobile Code, Proof-Carrying Code, Certifying Compilation, Security Properties, Automated Program Verification.

1. Introduction

Sharing mobile code is a powerful method for the interaction between software systems. By using this method, a server can provide flexible access to its internal resources and services. However, since untrusted mobile code can be malicious, the use of this method puts the consumer's system at risk.

There exist several approaches to guarantee the safety of mobile code. In particular, *Proof-Carrying Code* (PCC) [11] woke up the interest of the research community. PCC is a technique developed by Necula and Lee [11] that guarantees the safety of untrusted mobile code. Since its inception in 1998, this technique generated several active lines of research [1, 2, 3, 4, 5, 11, **?**, 12, 13, 15], but there are some open problems. The effective use of PCC requires the formal specification of security properties and the construction of easily verifiable mathematical proofs.

In a PCC framework the code producer must pro-

vide a formal proof of the code safety together with the code. The code consumer's security policies is formalized by using a system with axioms and inference rules. The proof of safety of the mobile code is based on such formal system. Thus, the code consumer only needs to verify that the proof refers to the code and the validity of the proof. Finally, the mobile code can be executed safely only if both answers are positive.

The process of creation and use of PCC is centered in security policies. These policies are defined and made public by the code consumer. These policies establish under what conditions the execution of a program is considered safe.

PCC does not require producer authentication because the program will be executed by the host only if the proof is valid. So, this approach minimize the external trust required. Also, dynamic verification is not required because the verification is made before the execution of the code. PCC is a technique based on a static approach that does not add any run-time performance penalty, in that sense it is more efficient than the method provided by Java, that requires the monitoring of the execution of each operation.

In general, the frameworks for the production of secure software can be divided in two phases. The first phase consist of the translation of the source code to some object language (for example, Java bytecode, or assembly language) and the inclusion of security annotations in the produced code. In order to use PCC at an industrial scale, these annotations must be generated automatically. During the second phase, the annotated object code is verified to ensure that it does not violate the security policies. If this verification is passed, the code can be run safely.

Several methods derived from PCC are based on type systems. Unfortunately, several interesting security policies cannot be verified by using a type system, or even the verification process is too costly. For example, the initialization of variables and the control of out-of-bound array accesses cannot be verified efficiently with such approach. Fortunately, the information required to guarantee these security properties can be generated by using static analysis of the program's control flow [6, 8].

Our framework, called *Proof-Carrying Code based on Static Analysis* (PCC-SA), combines the PCC and static analysis techniques. The main objective is to provide an efficient and effective solution when the verification of the security policies can not be done by using an approach based on type systems. V. Haldar, C Stork, and M. Franz [8] argue that the efficiency problems of PCC are due to the semantic differences between the source code and the low-level object code. In contrast, PCC-SA uses a high-level intermediate language, such as a *control flow graph* or an *abstract syntax tree* with type annotations. These intermediate representations enable us to use static analysis techniques to generate and verify this type information. The main advantages of this technique is that the size of the generate proofs is linear w.r.t. the size of the programs, and it is possible to apply several code optimization techniques on the intermediate representation.

Static analysis is a very well-known technique in the field of compiler construction. Recently, this technique has being applied for verification and reengineering of software. Static analysis can be used for the verification of security properties because a concrete approximation to the dynamic behavior of a program can be obtained at compile-time. There exist several static analysis techniques that provide a good balance between the design costs and the solutions provided. For example, *array-bound checking* or *escape analysis* [6].

A traditional compiler provides type verification and other simple analysis. In contrast, the static analysis techniques that we use require more effort in order to be applied. However, these techniques are simpler than other methods, such as functional verification of programs. It is important to consider the trade off between precision and scalability, because static analysis techniques can reject some safe programs.

Even considering that static analysis is a powerful technique to guarantee security, it does not provide a solution for every security problem. Sometimes, dynamic verification is required because the security policies are not statically computable. For example, the general case of the problem of guarantee the safety of the elimination of all the *array-bound checkings* is equivalent to the halting problem. So, our framework enables us to insert dynamic verifications in the intermediate code in order to include, in the set of safe programs, some programs that cannot be verified completely by static analysis. In other words, the mechanism used to verify the code is a combination of compile-time static analysis and run-time dynamic control.

This paper is structured as follows: in section 2 we present the proposed framework and discuss the main advantages and disadvantages of the framework. Relevant features of the implemented prototype are explained in section 3. The last section is devoted to the conclusions of our work and some proposals of future work.

2. The PCC-SA Framework

Figure 1 shows the structure of the proposed framework. Following the conventions used in [11], the undulated boxes represent code, and the rectangular ones represent modules that manipulate such code. Moreover, the shadowed boxes represent untrusted entities, while the whites represent trusted entities belonging to the *Trusted Computing Base* (TCB).

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

The *Compiler* is a traditional compilar that takes the source code, applies lexical and syntactic analysis to verify the type of the expressions, and produces intermediate code. This produced 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 analysis in order to generate the information required to annotate the intermediate code, based on the security policy. If at some program point it can be determined that the security policy is not satisfied, then the program is rejected. At the program points where the static analysis techniques cannot determine the security status, a run-time check is inserted. Thus, if a program succed in passing across the GenAnot module, we can certify that it is safe.

The last process applied by the code producer is the *Proof Sketch Generator*. This module uses the annotations and the security policy to generate a proof sketch by taking in account the critical program points and their dependencies. This information is stored in the intermediate code. A proof sketch is the minimal path that the code consumer must to check in the intermediate code, together with the data structures to analyze.

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

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

2.1. Advantages and Disadvantages of PCC-SA

Since *Proof-Carrying Code based on Static Analysis* is based on *Proof-Carrying Code*, it has the same advantages than PCC. Because the code consumer only has to provide a fast and simple proof verification process, the host infrastructure is automatic and low-risk; while the harder task is on the side of the code producer, that must provide the proof. Moreover, trust between producer and consumer is not required. PCC-SA, such as PCC, is a flexible framework because it could be used with different languages and security policies, and even it is not only appliable to security. Another advantages are that the code generation can be automated, that the code is statically verified, that any modification (accidental or malicious) of the code can be detected (and even in such case the security is guaranteed), and that it can be combined with other techniques.

Moreover, PCC-SA has an all new set of advantages, produced by the combination of the static analysis and PCC techniques. The most important of these advantages is the fact 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, by using PCC-SA a broader range of security policies can be automatically verified, more platform independence is obtained by using a representation of the source code, and, compared with PCC, less run-time checks must be used.

Even when the idea of PCC-SA is simple, its efficient implementation requires to solve some problems. Some of the weakpoints of PCC-SA are inherited from PCC. For example, these frameworks are very sensitive to changes in the security policies. More important, to establish and formalize (that is, translate to a static analysis) a security policy is expensive, and both, 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. First, a source language was defined. We chose a subset of the language C, 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, we implemented the modules *Annotations Gen-*

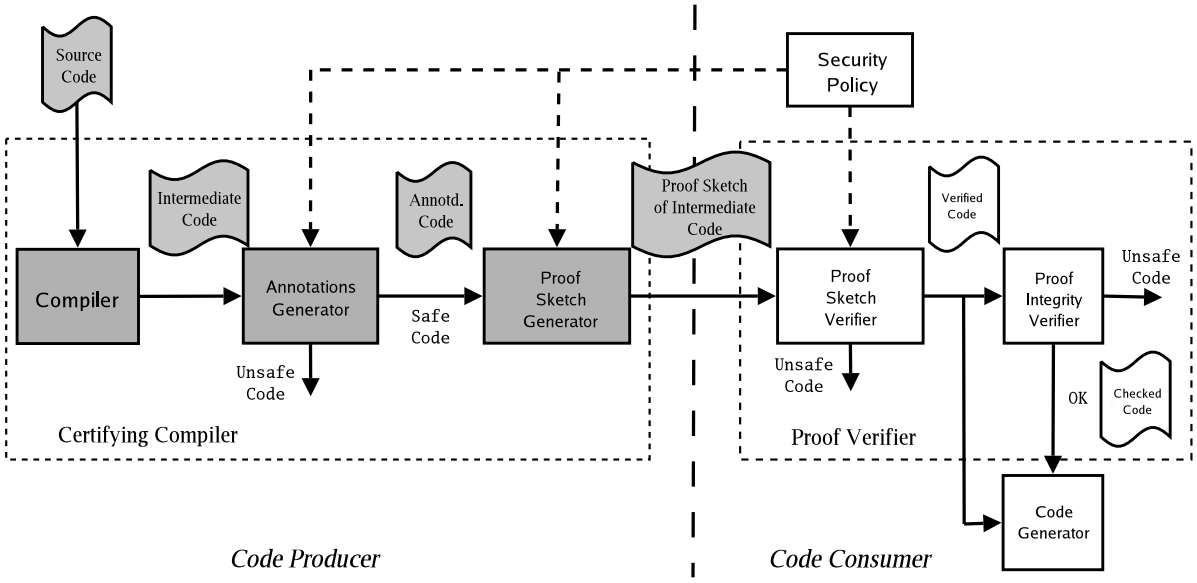

Figure 1. Structure of the framework Proof-Carrying Code based on Static Analysis.

erator, *Proof Sketch Generator*, *Proof Sketch Verifier*, *Proof Integrity Verifier*, and *Code Generator*.

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 programming language C. Since the security policy to check is based on the access to arrays, Mini includes array manipulation operations. Almost all the remaining features of the language C 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 the arguments can be integer or boolean. Unidimensional arrays, with elements of basic type, can be defined. Note that the complexity of including unidimensional arrays is equivalent to include more complex data structures, such as matrices.

We say that Mini is an *extended* subset of C 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 $0 \leq \mathtt{a} \leq 10$. Here, we will proceed with the explanation of the prototype by using the example of Figure 2.

```
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.

3.2. The Security Policy

A security policy is a set of rules that define the conditions under which is safe to execute a program. 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 [14].

We chose a security policy that guarantees type and memory safety, that non-initialized variables are not read, and that there is no out-of-bound array accesses.

3.3. Intermediate Code: Abstract Syntax Tree

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

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

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

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 to such element of that array and that it is safe to access, respectively. By modifying the node label we avoid the necessity of include run-time checks.

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

3.4. The Certifying Compiler CCMini

The certifying compiler CCMini is composed by a traditional compiler, an annotations generator, and a proof sketch generator. The **Compiler** takes a program written in the source language Mini and returns an *abstract syntax tree* (AST). The **Annotations Generator** (GenAnot) applies several control and data flow static analysis on the AST and it generates an *annotated abstract syntax tree*. Moreover, GenAnot checks if the code satisfies the security policies by using the information about variable ranges and loop invariants included in the annotations. If the code falsifies the security policies, it is rejected. If GenAnot cannot determine the security status at some program point, code for a run-time check is added at this point. Finally, the **Proof Sketch Generator** takes the annotated AST and it 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 pre-condition and the post-condition of the program. In order to obtain this information, GenAnot applies control and data flow analysis on the AST. These analysis can obtain information about the initialization and range of the variables and, sometimes, pre- and post-conditions are obtained. The range of a variable is useful to determine if the variable can be used to access some element of an array.

In order to make it efficient and scalable, the implemented GenAnot module only applies the analysis to the body of the functions. Moreover, our GenAnot is at a middlepoint between *flow-sensitive analysis*, that considers all the possible paths of a program, and *flow-insensitive analysis*, that does not consider the flow of the program. GenAnot analyzes the control flow in some cases, for example, in loops; but, in general 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, uninitialized variables access are detected. In such case the program is rejected.
2. *Identification of ranges of variables not modified in body loops.* By taking into account the range of the function parameters and analyzing the execution flow (without considering loops), the range of each variable is obtained. By analyzing each operation and sentence, it is ensured that the values of each variable respect their range.
3. *Identification of ranges of induction variables.* A variable is an *induction variable* if its value is increased or decreased by a constant value at each iteration of a loop. If the loop condition depends upon a 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 valid or out-of-bounds. In particular, it is easy to do this

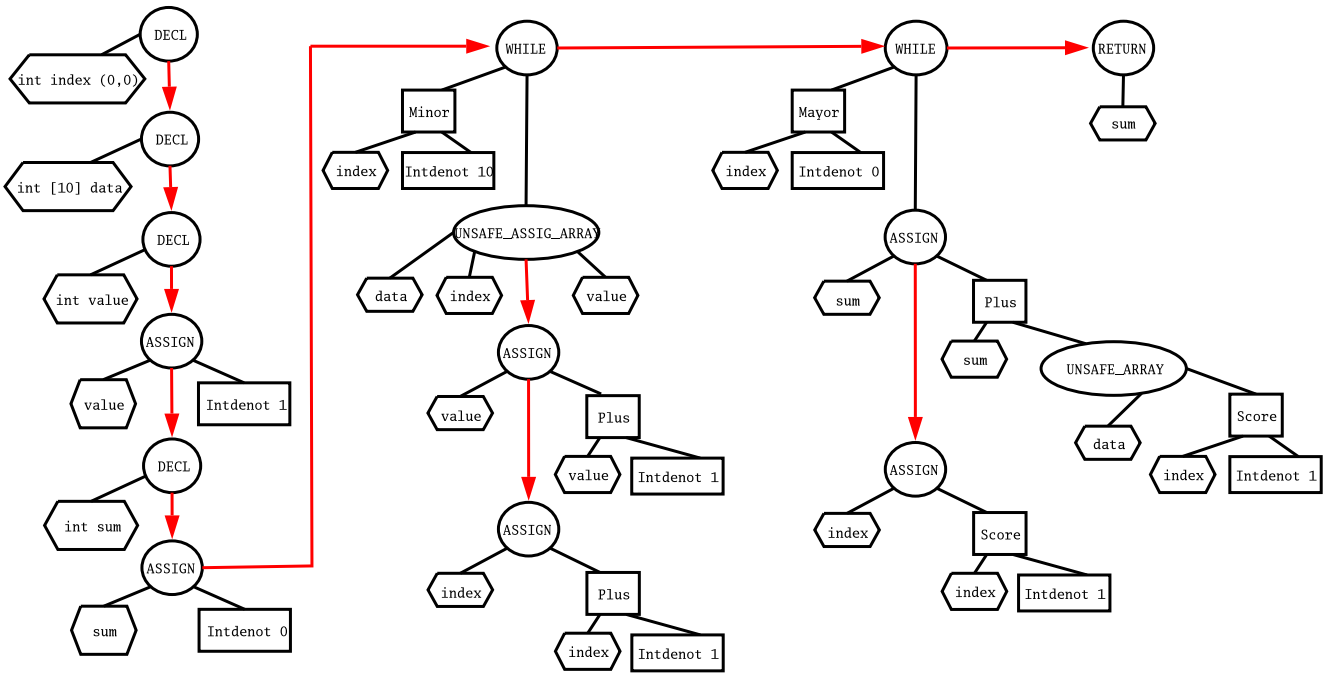

Figure 3. AST of the Mini program from Figure 2.

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 AST 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 4.

The shadowed rectangles represent annotations and the dotted lines are the proof's minimal path. For example, the annotation $index(0, 9)$ means that the value of the variable *index* 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 Figures 3 and 4 the labels of nodes that refer to array accesses are different. In Figure 3 the accesses are considered unsafe, but after applying GenAnot these accesses are considered safe in Figure 4. Since each access in the annotated AST is safe no runtime checking is needed.

3.5. The Proof Verifier

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

The Proof Sketch Verifier checks the well-formedness of the received AST. 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, the Proof Sketch Verifier follows the path signaled by the pointed arrows in Figure 4.

After that, the Proof Integrity Verifier analyzes the AST as a whole, checking if the proof sketch includes all the critical points in the program. If some critical point not included in the proof sketch is found, the code is rejected as unsafe.

Finally, if the code was accepted as safe, the Code Generator uses the AST to generate 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.

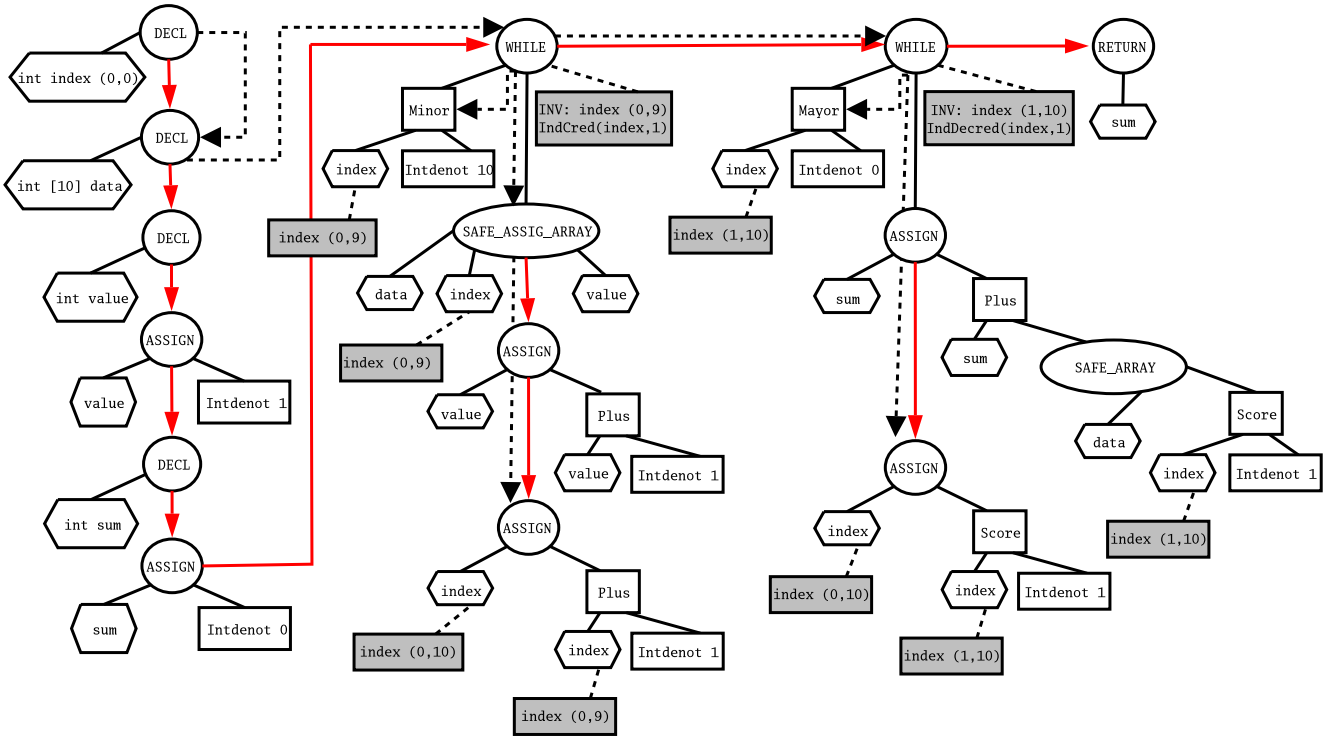

Figure 4. Proof Sketch on the Annotated AST for the example in Figure 2.

4. Related Work

In order to apply the PCC technique in an industrial setting, the creation of the *certificate* must be done automatically. The original PCC implementation [11] required human intervention in the proof-making process. On the contrary, in our PCC-SA framework the certificates are automatically generated by the verifying compiler. Moreover, the size of the generated proofs is linear w.r.t. the size of the program, while in PCC the size of the proofs is exponential w.r.t. the size of the programs.

As mentioned before, the main difference between our framework and PCC is based on the properties that can be included in the security policies of PCC-SA but not in the policies of PCC. For example, the checking of out-of-bounds array accesses.

In order to overcome the limitations induced by the *type-specialization* of PCC, Appel [2] developed the more flexible Foundational Proof-Carrying Code (FPCC). However, the definition of the semantic models for the type systems used within this framework requires a hard task and a very skill code generator.

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

Another related work is the Java bytecode verification [7, 10]. This consists of an abstract execution of the code of each class in order to check if the type of the values is respected. In particular, Leroy [9] 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.

5. Conclusions and Future Work

In this paper we present a framework for safe execution of mobile code, called *Proof-Carrying Code based-on Static Analysis* (PCC-SA). This framework, based on PCC and static analysis techniques, provides safety, platform independence, a simple verification process, small formal proofs, and provides the information required to apply code optimization.

The prototype that has been implemented validates

this approach. CCMini is a reduced, small, simple certifying compiler that provides an envirounment for the safe execution of mobile code. Moreover, CCMini can also be used in applications that doesnot require code mobility.

The design of the prototype ensures that there will be only discarted such programs that are certainly unsafe. If CCMini cannot determine the safety of a program, run-time checks are inserted in the critical points. By hand examination of a reasonable number of programs allows that most of array accesses use as indexes, expressions including only inductive variables. This suggests that the proposed approach solves for safety on accesses to arrays on most situations.

In order to conduct a series of small experiments, we asked several programmers, not involved in the development of CCMini, to provide us with Mini programs that include array accesses. In most of the cases, the compiler CCMini determined the safety of such programs without include any run-time check. However, to confirm this hypothesis, it would be nesessary to perform the test on a large number of programs, piched out from the existing libraries. This can only be done if the CCMini compiler is extended to a standard language.

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.

References

[1] A. Appel, A. Felty, "A Semantic Model of Types and Machine Instructions for Proof-Carrying Code", in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL'00), pp. 243–253, ACM Press, Boston, Massachusetts (USA), January 2000.

[2] A. Appel, "Foundational Proof-Carrying Code", in *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pp. 247–256, IEEE Computer Society Press, 2001.

[3] A. Appel, E. Felten, "Models for Security Policies in Proof-Carrying Code". Princeton University Computer Science Technical Report TR-636-01, March 2001.

[4] A. Bernard, P. Lee, "Temporal Logic for Proof-Carrying Code", in *Proceedings of Automated Deduction* (CADE-18), Lectures Notes in Computer Science 2392, pp. 31–46, Springer-Verlag, 2002.

[5] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, K. Cline, "A certifying compiler for Java", in *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". In 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] X. Leroy, "Bytecode Verification on Java smart cards". in *Proceedings Software Practice and Experience*. 2002.

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

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

bibitemNEC02 G. Necula, R. Schneck, "Proof-Carrying Code with Unstrusted Proof Rules", in *Proceedings of the 2nd International Software Security Symposium*, November 2002.

[12] G. Necula, R. Schneck, "A Sound Framework for Untrusted Verification-Condition Generators", in *Proceedings of IEEE Symposium on Logic in Computer Science* (LICS'03), July 2003.

[13] M. Plesko, F. Pfenning, "A Formalization of the Proof-Carrying Code Architecture in a Linear Logical Framework", in *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento (Italy), 1999.

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

[15] R. Schneck, G. Necula, "A Gradual Approach to a More Trustworthy, yet Scalable, Proof-Carrying Code", in *Proceedings of International Conference on Automated Deduction* (CADE'02), pp. 47–62, Copenhagen, July 2002.