Proof Carrying Code

Manish Mahajan
mahajan_manish@rediffmail.com
Department of Computer Science and Engineering
Indraprastha Engineering College, Ghaziabad, India

Abstract - Proof-Carrying Code (PCC) is a technique that can be used for safe execution of untrusted code. In a typical instance of PCC, a code receiver establishes a set of safety rules that guarantee safe behavior of programs, and the code producer creates a formal safety proof that proves, for the untrusted code, adherence to the safety rules. Then, the receiver is able to use a simple and fast proof validator to check, with certainty that the proof is valid and hence the untrusted code is safe to execute.

Keywords - Proof-Carrying Code, Verification Condition, VCGen, Safety-Policy, code consumer, code producer, Proof Producer

(Received July 21, 2006 / Accepted October 19, 2006)

1. Introduction

High level programming languages usually assume a closed environment where the entire program or project will be constructed using the same language, thereby ensuring that the safety precautions laid down by the language like the type safety rules are followed by all the program components. But what practically happens is entirely different, with a project using modules written in different languages like C, ALP etc. thus we lose the guarantee provided by the programming language unless we use costly measures like sockets and processes. The problem is increased manifold when we consider distributed and web computing particularly when mobile code is allowed.[2]

The problem is not limited to the realms of programming languages. If we delve deeper at the operating system level we again encounter a similar paradigm. While executing application programs in an operating system, we may need to run the code in the same address space as the operating system kernel. This may cause serious problems unless the kernel can be sure that the application (which is usually from an untrusted source) will respect the kernels internal constraints.[12]

A code consumer must be convinced that the code provided by the (often untrusted) code producer has some (previously agreed upon) set of properties that makes it safe to be executed at the code consumer. This is called "*establishing trust*" between code consumer and code producer.[1] This can also be achieved by using cryptography to ensure that a trusted person has developed the code. But this system has the weakness that it depends on personal authorization. Even trusted persons or compilers can make errors either accidentally or with malicious intent.[7]

Proof-Carrying Code (PCC) is a technique by which a *code consumer* (e.g., host) can verify statically that code provided by an untrusted *code producer* adheres to a predefined set of safety rules. This is done by certifying the compilation process. The code consumer in such a way chooses these rules, also referred to as the safety policy, that they are sufficient guarantees for safe behavior of programs. Using this safety policy the code producer can provide binaries in a predefined format called *the "Proof Carrying Code"* or the PCC that contains, in addition to the native code, a formal proof that this binary satisfies the safety policy. The code consumer can easily verify the proof and be sure that this application, although from an untrusted source, is safe to use. [1,7]

PCC has many uses in systems whose trusted computing base is dynamic, either because of mobile code or because or regular bug fixes or updates. Examples include, but are not limited, to extensible operating systems, Internet browsers capable of downloading code, active network nodes and safety-critical embedded controllers. For mobile code the code consumer would be an Internet host (e.g., a web browser) and the code producer a server that sends applets. In operating systems, one can have the kernel act as the host, with untrusted applications acting as code producers that download and execute code in the

kernel's address space.[11]

The rest of the paper is organized as follows : In section 2 I have listed the various features of PCC. Section 3 discusses the architecture of PCC. The five steps in the process of creation and application of PCC are described in 4. The performance considerations are discussed in 5 and 6 provides the concluding remarks.

2. Features of PCC

- PCC does not rely on the usual methods of authentication using cryptography and does not need trusted third parties.[3]
- It requires the application or source to generate the binaries in a predefined format- PCC
- No need for program analysis, code editing, compilation or interpretation.
- PCC is quiet Fast as the safety check is done only once and there is no need for any further run time checking.
- The proof is linked to the native code so it is difficult to tamper the code or proof without rendering the resulting binary non verifiable.[1]
- In the few cases where the code or the proof is modified in such a manner that validation still succeeds, the new code is still safe, it may not give the expected results but it is safe to execute. So PCC is intrinsically safe without requiring external authentication.[3]
- The proof-checking algorithm is fast and simple.
- The code consumer can easily validate the proof without using cryptography and without consulting any external trusted entity.
- The main practical difficulty in PCC is in generating the proofs and this is the one roadblock to its widespread use.[3]

3. PCC Architecture

Any implementation of PCC must contain at least four elements: (1) a formal specification language used to express the safety policy, (2) a formal semantics of the language used by the untrusted code, usually in the form of a logic relating programs to specifications, (3) a language used to express the proofs, and (4) an algorithm for validating proofs. A typical architecture is shown in figure 1. The central component of any PCC implementation is the safety policy which represents the set of rules that define unambiguously whether a given agent program is safe to execute. Before a code consumer can accept PCC binaries, it must establish a safety policy, which defines the actions that the binary is allowed to perform and also the circumstances when these actions are allowed. [3] The safety policy is defined in advance by the code consumer and is a trusted component of the infrastructure. The safety policy defines what is meant by safety and the interface that may exist between the code consumer and any binary from the code producer. The policy lays down explicit conditions under which the code consumer considers the execution of an external program safe.[10]

The safety policy comprises of two components –

Safety Rules – all the authorized operations and the various preconditions associated with them.

Interface – the calling convention between the code consumer and the foreign program ,i.e., the signature against which the external program has been compiled.[4]

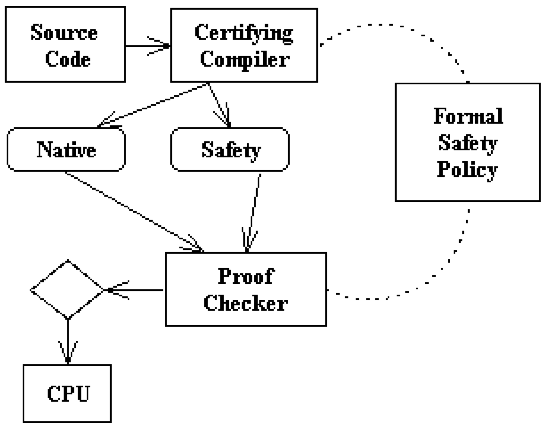

Figure 1. The basic PCC architecture

The PCC binary, in its life cycle, undergoes three phases :

In the first phase, called the *certification* phase, the code producer generates the source code and ensures that it adheres to the safety policy laid down by the code consumer. The source program is then compiled and a proof that this program confirms to

the safety policy is generated. This verifies the program with respect to the safety policy generated by the code consumer. A proof of the successful verification is generated and is encoded to avoid tampering thereby producing the safety proof. This

safety proof, along with the native code, forms the PCC binary that is eventually delivered to the code consumer for use or is stored at the code producer for further use.

In the second phase – *validation*, the code consumer, upon receiving the PCC binary from an (often un-trusted) source, validates the proof part of the binary. If the validation succeeds, the native code is loaded for execution. A fast and straightforward algorithm does the validation making the process fast and inexpensive.

As the proof is in-built with the native code in the PCC binary, it is possible to carry out the validation off-line. Moreover we need to validate the binary only once irrespective of how many times it is intended to be used. This becomes very significant in cases where the validation of programs are complex, time consuming and involves users as repeated verifications may introduce a lot of overhead. [5]

In the last phase, the code consumer eventually loads the validated and verified native code from the PCC binary to be executed as many times as needed. No additional run-time checks are needed as the validation stage ensures that the program confirms to the safety policy. The basic PCC Protocol is shown in figure 2.

4. The Process

4.1 Step One – Defining the Safety Policy

PCC places no restrictions on the languages in which the binaries can be generated. PCC can be adapted to both high and low level languages to maximize the performance of the binary. A given code consumer may also receive binaries written in multiple programming languages, thus the safety policy must be adapted to each language. [4] The safety policy has three components:

- A mathematical logic that defines the preconditions under which the specific operations are allowed. The logic also defines the Verification Conditions (VC). This logic is made available to the proof producers. This logic is the language used by the PCC to define and verify the preconditions.
- Safety policy also includes the specifications of all the functions that the agent is supposed to provide to the code consumer and also for the functions of the code consumer that the agent is allowed to invoke. The specifications are defined as a pair of pre and post conditions, which define the state of variables, the values of invariables, relationship between variables and actual arguments and return values.[3]
- Finally the safety policy contains a method for inspecting the agent code and for discovering the actions that an agent might perform and under which circumstances. This is accomplished by the VC generator which scans the agent code and collects the set of all the actions that might be performed during execution, along with a partial description of the program state when such actions would be attempted. This information is expressed as a predicate in the logic (the verification condition).

An Example Safety Policy [8]

- All memory locations after a designated location are readable.
- All memory locations after a designated location are writable.
- Exiting by jumping to a designated address is safe.
- The program counter is initially set to the first instruction of the code.

4.2 Step Two – Generating the Annotated Agent Code

The first stage of interaction between code consumer and code producer through the PCC is the preparation of binary as per the specifications laid down by the safety policy or more specifically, by the VCGen component of the safety policy. [2]

VCGen requires that the agent code is syntactically well formed in the selected language, all functions defined and used internally by the agent code are annotated with a precondition and a post-condition, and also that each loop have an associated loop invariant. The loop invariants and the specifications for the internal functions are referred to as annotations. If the annotations are well formed predicates in the selected logic, this will make the VCGen accept the code, but just that.

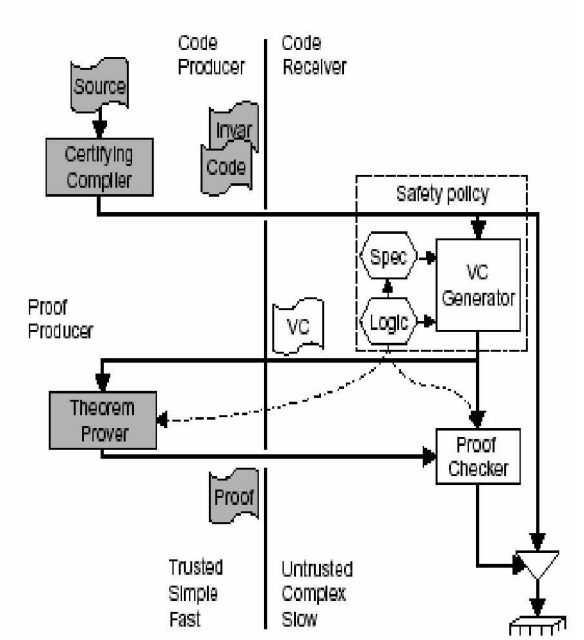

Figure 2: The basic proof-carrying code protocol. The wavy boxes represent data and code and the rectangular boxes represent system components. The white elements are trusted, while the grey elements are not trusted. [2]

But it is not sufficient to guarantee that the code will ultimately make it through all of the steps of the PCC protocol. For this to happen, the annotations must also be correct and sufficiently strong. [4]

- A loop invariant is a correct annotation if it is indeed a valid predicate every time the execution reaches the beginning of the loop.
- A function specification is correct if the precondition holds every time the function is invoked, and if the post-condition holds every time the function returns.
- The weaker the annotation the easier it is to ensure its correctness. For example, the weakest loop invariant "true" is evidently a correct invariant.
- A function precondition is sufficiently strong if, by assuming that it holds at the beginning of the function, one can prove that the body of the function is safe to execute.
- The stronger the annotation the easier it is to satisfy the sufficiently requirement.
- We see, thus, that annotations must be not too strong and not too weak. This is what makes the task of annotating the code a delicate one.

4.3 Step Three – Generating the Verification Condition

The code consumer carries out a fast, detailed and automatic inspection of the code it receives by using a program, called the verification condition generator (VCGen), which is one of the components of the safety policy.[5]

The verification condition generator (VCGen) is implemented as a symbolic evaluator for the program being checked. It scans the program in a forward direction and at each program point it maintains a symbolic value for each register in the program. These symbolic values are then used at certain program points (e.g. memory operations, function calls and returns) to formulate checking goals for the Checker module. VCGen checks the syntax of the received code and emits a verification-condition predicate for all agent instructions that might violate the safety policy.[6]

The syntactic checks that VCGen performs depend on the particular safety policy. In addition, VCGen can enforce restrictions on the set of instructions or operations that might occur in the agent code. The safety policy may choose to restrict the syntax of the language so as to implement the desired safety properties syntactically. For example, VCGen could enforce termination if the safety policy disallows function calls and backward branches. But this method has the shortcoming that it can not be extended without crippling the expressiveness of the language. Thus most safety policies do allow potentially dangerous actions but impose restrictions on their use. [5, 6]

The specific conditions under which an action is considered safe is called the action precondition. In practice, most action preconditions denote properties that are difficult or even impossible to verify directly.

*J*These preconditions are expressed as a predicate in the selected logic. The VCGen collects all the preconditions and combines them with control flow information and with the specification part of the safety policy to create the verification condition for the entire agent code.

The VCGen must also have an good understanding of the semantics of the language and the code in order to detect the potentially hazardous instructions and to construct meaningful verification conditions for them.

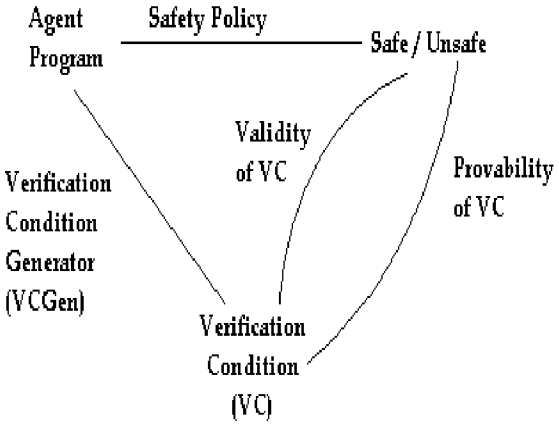

Figure 3: The relationship between the safety policy, the verification-condition generator, validity of verification conditions and provability of verification conditions. [3

In order to achieve this efficiently, we adopt the general design rule that whenever some information about the behavior of the agent is difficult to discover, the code producer must provide it in the form of code annotations. However, in order to prevent mistakes in the verification process due to erroneous annotations, VCGen must take special care when using them.

4.4 Step four – Proving the Verification Condition

Once the verification conditions are received, the proof producer attempts to prove them according to the logic specified in the safety policy. Because the code receiver does not have to trust the proof producer, any system (even the code producer) can be the proof producer. Generally the proof generator is a general-purpose theorem prover for predicate logic.

[11] A typical proof producer in a PCC system has three requirements

- It must be able to prove verification conditions efficiently.
- It should be able to generate detailed proofs of the verification conditions.
- It must specify these proofs using the axioms and inference rules specified as part of the safety policy.

4.5 Step Five – Verifying the Proof

The last step in a PCC session is the validation and verification by the code consumer of the proof generated by the proof producer and contained in the PCC binary. This is done using a proof checker that verifies that the various inferences in the proof are in fact valid instances of one of the axioms or inference rules specified as part of the logic in the safety policy. Also, the proof checker verifies that the proof
proves the same verification condition that was generated in Step 3 and not some other predicate.[3]

A good technique for representing and validating proofs must have the following desirable attributes:

- The representation of proofs and the proof checking algorithm should be logic independent so that the implementation can be reused for multiple applications of proof carrying code. The proof checking algorithm must be simple so that it can be trusted easily.
- Proof checking must be relatively fast and inexpensive.
- Proofs and predicates must be represented in a compact form in order to minimize the cost of communication between the code receiver and the proof producer.

5. Performance Considerations

The complete process of generating the proof, attaching it to the binary and subsequent verification by the code consumer has a significant impact on the performance of the binary.

- The development life cycle of the binary is bound to be elongated due to the complex process of generating the proof and attaching it to the binary. This overhead can be reduced to some extent by using a generic method to describe the verification conditions and to separate the process of proof generation by making it a generalized one.
- The effort of generating the binary and subsequently the cost will be more, though with appropriate measures this cost increase can be reduced.
- There is added responsibility on the code consumer as he has to explicitly specify the safety policy and publish it beforehand.
- The speed of execution of the binary at the code consumer end may suffer due to the added overhead of verification of the proof. Using fast and efficient proof checkers, this delay can be greatly minimized. This is also offset by the fact that the validation and

verification needs to be carried out only once by the receiver.

- Maintainability could also become a serious issue if not given proper consideration at the beginning. As the safety proof is tightly linked to the native code, any modification to the later may have serious implications for the former and may render the binary non-verifiable.
- Reliability of a PCC binary is significantly higher as not only is the binary fully tested for bugs, but it is also ensured to follow the restraints laid down by the code consumer.
- The size of binary will be increased due to the inclusion of the safety proof. This is one of the major areas where further work needs to be done to reduce the size of the proof and consequently that of the binary.

6. Conclusion

This paper discusses a methodology of ensuring the code consumer that the binary received from the code producer is safe to work with. Although the PCC methodology has, at this time, a number of limitations but it promises to provide a better solution to the "safe un-trusted code" issue. The method provides significant benefits over the contemporary techniques being used for the purpose and provides marked benefits in establishing trust.

7. References

[1] Thomas A. Henzinger1 Ranjit Jhala1 Rupak Majumdar, George C. Necula1 Gr´egoire Sutre2 Westley Weimer1, Temporal-Safety Proofs for System Code - Proc. of Conference on Computer Aided Verification, 2002.

[2] Andrew W. Appel, Foundational Proof-Carrying Code., in 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01), June 2001

[3] George Necula. Proof-carrying code – Design and Implementation. In Twenty-Fourth Annual ACM Symp. on Principles of Prog. Languages, New York, Jan 1997. ACM Press

[4] George Necula and Peter Lee. Safe, un-trusted agents using proof-carrying code. In Special Issue on Mobile Agent Security, volume 1419 of Lecture Notes in Computer Science. Springer- Verlag, October 1997.

[5] J. Feigenbaum and P. Lee. Trust management and proof carrying code in secure mobile code applications (A position paper). Submitted to the DARPA Workshop on Foundations for Secure Mobile Code, Monterey, California, March, 1997.

[6] George Necula and Peter Lee. Efficient representation and validation of logical proofs. Technical Report CMU-CS-97-xxx, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1997

[7] Andrew W. Appel, Protection against un-trusted code IBM Developer Works, September 1999.

[8] Andrew W. Appel and Amy P. Felty, A Semantic Model of Types and Machine Instructions for Proof-Carrying Code, 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00), pp. 243-253, January 2000.

[9] Evans D., and A. Twyman. Policy-directed code safety. In Proceedings of the 1999 IEEE Symposium on Security an Privacy, Oakland, CA, May, 1999.

[10] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga A Trustworthy Proof Checker, Princeton University CS TR-648-02, April 2002

[11] George Necula, Peter Lee Safe Kernel extension without run time checking ,Second Symposium on Operating Systems Design and Implementation (OSDI '96), Seattle, Washington, October 28-31, 1996

[12] Andrew W. Appel and David McAllester, An Indexed Model of Recursive Types for Foundational Proof-Carrying Code..ACM Transactions on Programming Languages and Systems 23 (5) 657-683, September 2001.

[13] Yasuyuki Tsukada, Proof-based Approach to Safe Software Distribution, Ph.D. Thesis, Department of Computer Science, Tokyo Institute of Technology (March 2006).