Proof Carrying Code: A Survey

Anupam Gupta

Abstract

Proof Carrying Code (PCC) is a concept by which the users of a program can verify the safety of a program which is provided by an untrusted source. This is done by attaching a proof of safe behaviour of the program to the code, which can be verified by the user before execution. In this survey, we attempt to look into the issues involved in this mechanism. Further, we try to look at techniques to avoid sending the entire proof (which can be extremely large). These techniques reduce the communication cost at the expense of certainty of safety — i.e. the user can now only be sure of the safety with a high probability.

1 Introduction

The advent of the Internet has changed the lives of most computer users: most of the software one needs nowadays is not bought in shrink-wrapped packages but instead obtained off the web: downloaded from the literally millions of web-pages that litter cyberspace. This helps to get results much faster: already-written code can be found for most of the tasks that we need to perform. However, this raises the very important question: How can we be sure that the code behaves the way we believe it does ? What can prevent it from deleting some of our files, sending classified data to someone else or stealthily using up valuable CPU time ?

The ideal solution would be to check the code to see if it is safe or not; but this can be very time consuming. A better idea is to have the code prove itself safe. The concept of Proof Carrying Code (PCC) is based on this idea: a program carries with itself a proof of its adherence to certain safety policies.

The rest of this survey is structured thus: in section 2, we look at some of the other approaches to this problem and examine their pros and cons. In section 3, we dive into the idea of PCC and attempt to give an flavour of the technologies involved in it. In section 4, we look at an possible method to reduce the communication costs involved, and finally, we wrap up in section 5.

2 Approaches

The most obvious solution to our problem is to extend the conventional idea of trust into this domain as well. Note that similar problems existed earlier, though on a smaller scale — there were the problems with viruses and Trojan horse programs which would spread by passing floppy-disks around. The way to take care of these problems was to use trusted software, which would come with the guarantee of a brand-name with it. We can do similar things here: programs can be digitally signed by the code producer. This idea has been put into use by many vendors, including Sun and Microsoft, who sign their applet class files to convince users that they do not contain spurious or malicious code. However, this is not really the approach we are looking for since this approach really tries to define away the problem: The producers of the programs in question are not always major companies; and an unknown programmer in some remote part of the world guaranteeing that his code is safe may not mean much to us.

Another approach is to "sandbox" the suspect code, as is done by Netscape. This prevents the code from accessing the file systems, making network connections to arbitrary hosts and performing other potentially dangerous actions. Unfortunately, this has the detrimental effect of crippling the power of the programs. A way to remove such restrictions is to allow the programs more breathing space and perform run-time checks — which, of course, is very expensive in terms of performance.

A more elegant way is to use a type-safe programming language, such as Java or the safe subset of Modula-3. Programs which type-check are "guaranteed" not to cause any malicious effects. However, this may not solve all the problems. Code producers may not want to give out source code but only binaries, and then these can be manipulated to produce unwanted effects. There are ways to handle this; the Java complier outputs an encoding (called the Java Bytecode) which is send to the user. He then checks it for consistency before executing it. Typically these checks are computationally expensive and have to be performed at every execution of the code, if we want to run the program a number of times. Possible ways of getting round this problem is to have another compiler at the user end which converts the Bytecode to a binary executable, and these are avenues that can be explored.

The above approach suffers from another drawback: the set of properties we can ensure using the above strategy is limited by the programming language. In case the qualities we demand of the program are not easily expressible in the language of our choice, we may have to live with unsafe programs or inefficient implementations.

3 Proof Carrying Code

The concept of PCC is very simple: each program carries with itself a proof. This proof certifies that the program does not violate certain clearly-defined safety properties. On receiving the code, the consumer validates the program against the proof. If the check is successful, he then can be convinced that the program is safe, and can run it repeatedly without further checking.

In the following discussion, we often allude to the program as being "correct". This should be read as the statement that the program is safe; we are not dealing with the computational correctness of programs in this survey. Also, for brevity, we shall refer to the code-producer as Alice and the code-consumer as Bob.

3.1 Overview

The whole process can be considered to form three stages. The first stage can be called the *negotiation* stage. During this time, Alice and Bob decide upon the safety policy — which consists of the conditions that the program produced by Alice will be invoked and those restrictions which it must satisfy, and a format to encode the proof in before transferring.

After the negotiations are over, Alice will write the program and prove it correct. This forms the *certification* stage. The proof is then encoded and passed along with the code to the user. The consumer then checks the code against the proof, which forms the *validation* stage. All these stages are described in further detail in the following subsections. An outline of the process is presented in figure 3.1.

3.2 The Negotiation

Before anything else is done, Alice and Bob must make sure that they speak the same language. So they get together and agree on the *safety policy*, which consists of several points listed here:

- The calling convention that defines how Bob will invoke the program, which is called the *precondition*. Also, the properties of the system which the code must establish before returning, which forms the *postcondition*.
- The restrictions on the execution of the program. This is specified by means of a *verification condition generator*, which is a procedure that computes the *safety predicate* (SP) after looking at the code. This safety predicate must be such that a proof for this implies the safe behaviour of the program.
- A set of axioms and rules of inference that are used to prove the predicate. This can typically be given by specifying the rules that are used in addition to the standard rules of first-order logic.

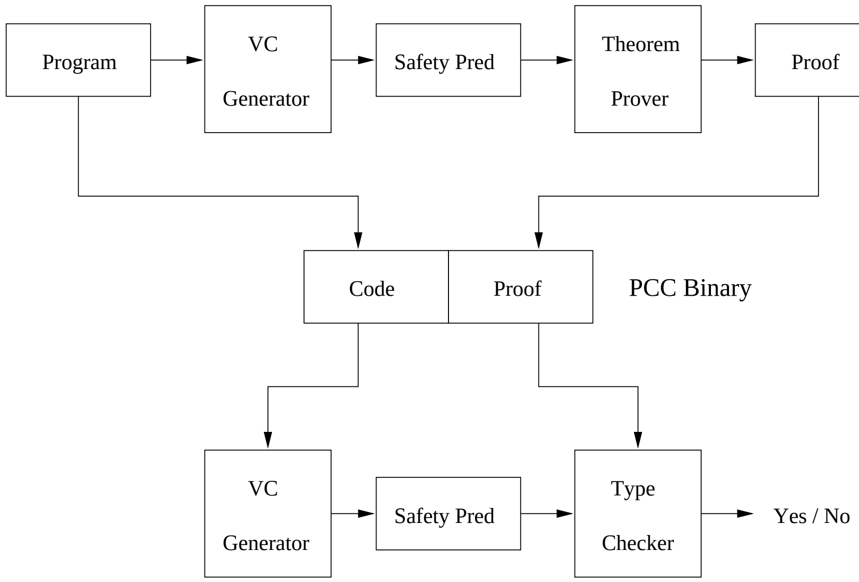

Figure 1: Overview of Proof Carrying Code

- A representation format for the proof. This is the form in which Bob expects the proof, and this defines an encoding function $\lceil \cdot \rceil$ from formulae in first-order logic to this format.

Clearly, the policy will depend on the required of the program and hence would have to be tailored specifically. However, it may be possible to have some standard sets of reasonable policies that can be used by developers on the Internet, where there is no clearly specified user.

3.2.1 The Questions

There are questions regarding this negotiation stage that have to be looked into more carefully — it is this stage that forms the backbone of the entire process and any weaknesses in this may cause our edifice to collapse or be totally ineffective.

- It may be a very complicated procedure to ensure that the safety policy is water tight: there are generally many ways of allowing any operation to be performed in practical systems, and most of them have certain quirks that a malicious user may take advantage of. Making sure we plug all the holes may not be easy at all.

- Further, there is the question of the potency of the proof system. First order logic may not be powerful enough to encapsulate all the constraints that are desired of the system. e.g. we may be having reactive programs and may want to prove certain temporal properties for the system.

- Another question is that of proving the *safety theorem* (see section 3.4.3). Though this may be well understood, it has to be performed with care to prevent buggy programs to be certified as "safe". It may not be very easy for casual programmers to perform and may limit the usefulness of PCC to mostly academic cases.

3.3 The Certification

Certification is the part of the process where Alice generates the "safety proof" for the program under consideration. The questions

3.3.1 Producing the Safety Predicate

To get the safety condition, we must encode the semantic meaning of the program Π as a first-order logic predicate. This predicate is a formal statement that the program, if executed under the conditions guaranteed by Pre will not violate any safety conditions and return in a state which respects the postcondition $Post$.

The technique currently used is based on Floyd's verification conditions which were the precursors of Hoare conditions. The basic idea is this: suppose we want the condition B to be satisfied after the instruction c has been executed. Then the precondition A could be roughly thought of as :

$$A = \text{ (condition required for } c\text{) } \wedge [\text{ effect of } c]B.$$

For example, if c was $X := a$, then A would be $\textbf{writeable}(X) \wedge [a/X]B$.

It is generally the case that these rules can be mechanically written down for most of the programming language constructs; the difficult parts are the iterative constructs for which we have to guess the invariants. The technique used in PCC is to ask the programmer to provide a table $\overline{Inv}$ which contains the required invariants for the program. By ensuring that every loop has an invariant specified for it, we can write the verification condition (in the style of Winskel section **7.4**) easily.

3.3.2 Proving the Predicate

This safety predicate SP_a is then proved using a theorem prover, which also gives us the derivation $\mathcal{D}$. This seems to be computationally the most difficult part of Alice's task. Subsequently, this derivation is encoded into the format specified in the safety policy, and the triple $(\Pi, \overline{Inv}, \lceil\mathcal{D}\rceil)$ is sent to Bob.

3.3.3 The Questions

The certification of the programs may be the hardest problem that PCC has to solve. The use of Floyd-style verification conditions requires that the user provide the loop invariants, which may be rather complicated for non-trivial programs. However, most of the programs written are in some high level language, and these programs contain enough structure to glean some of (if not all of) the invariant conditions. The programmer may also annotate his code to help this process, if required.

The other potential thorn in the side is the inherent complexity of theorem proving, which is known to be undecidable in general. Although experiments performed on practical programs give some hope, it may be nicer to try and get some theoretical guarantees as well. It may be useful to look at the types of predicates that actually come up in real life programs and try to find efficient theorem proving techniques for some of them.

3.4 Validation

After Bob gets the program, he has to *validate* the code to see if it indeed preserves the safety properties. If he actually had to prove the safety predicate of the program himself, he would have little cause for happiness. However, in this case, he has but to verify that the proof provided by Alice shows that the program is correct.

3.4.1 An Aside : Logical Frameworks

This problem could be very difficult by itself if Alice and Bob did not know of efficient ways of encoding and verifying proofs. However, this problem has been well studied, and one of the ways to take care of this problem is through the delightful theory of *logical frameworks* [Pfe96]. These are meta-languages in which one can specify and reason about deductive systems, and are thus applicable to the study of of programming languages, especially of type systems and operational semantics.

Logical frameworks are means of specifying deductive systems, like the ones used to prove programs correct in PCC. To specify a deductive system, we have to specify its abstract syntax and its rules of inference, and the language must provide concise ways of representing these. But that is not all: it should be able to check deductions for their validity and detect meaningless expressions statically. And all this should be done in a computationally efficient manner.

The logical framework we shall look at is the Edinburgh Logical Framework (LF) [HHP93]. This is refinement of the simply-typed λ-calculus that uses the *judgments-as-proofs* idea. The valid terms in this language are in bijective correspondence with the syntactic entities (terms, formulae, proofs) of the deductive system, such that the judgments are identified with the type of the objects represented by their proofs. Hence *proof-verification is reduced to type-checking* in

LF. Since we know that type-checking in LF is decidable, we have a way to encode the proofs so that they can verified efficiently.

The above description is correct in spirit but not in detail: there are several points to be taken care of. Firstly, all objects in LF are considered modulo $\beta\eta$-conversion — hence there may be many objects which correspond to an entity in the logic, but all of them must be equivalent under the above notion. Also, a simple bijection does not suffice: we must ensure that the bijection be *compositional* (encoding must commute with substitution) and ensure that it is *adequate*, i.e. it is "full" (does not include any additional entities) and "complete" (encodes all entities correctly). There are other details like variable bindings, variable occurrence and discharge to handle in the logic, which are handled by the type system without much effort, but have to be kept in mind.

3.4.2 How PCC uses LF

The specification of the logic is done by constructing the *signature* Σ, which assigns kinds and types to a finite set of constants that represent its syntax and its inference rules. This signature Σ is a concrete representation of the logic and could have been given to Alice to describe the safety policy in the first place. Given Σ, Alice can encode the proof $\mathcal{D}$ of the safety predicate SP_a as an LF object $M = \lceil \mathcal{D} \rceil$.

On receiving the program Π and the LF object M representing the proof, Bob computes the safety predicate SP_b on his own. Note that he needs Π, the table of invariants $\overline{Inv}$ and Pre, all of which are available to him. He then checks to see that the type of the object M corresponds to the safety predicate computed by him, i.e. $\vdash_\Sigma M : \mathtt{pf}\lceil SP_b \rceil$. If it is indeed so, then Bob knows that there is a proof of the safety predicate, and hence the program obeys the safety properties. Note that the above test is safe against any altering of the code or the proof.

3.4.3 Tying up loose ends

In the above description, there were few steps that were explicit or implicitly mentioned but not really fleshed out. These are issues which involve proving the correctness of the entire approach. For the approach outlined above, we would have to prove the following theorems:

- **A soundness theorem:** This proves that if the safety predicate is derivable, then the program is "safe". To define this formally, we define an operational semantics in which an unsafe program would be 'stuck', and show that the derivability of the safety predicate implies that the program is never stuck.
- **An adequacy theorem:** This proves the compositional bijection between entities of the deductive system (in our case, a variation on first

order logic) and objects in the LF, which justifies the use of type-checking as a sufficient procedure for verification of proofs.

These proofs are long but largely mechanical; they are obtained by standard methods like induction on the structure of derivations. We do not give the details here: they can be found in Necula and Lee [NL96a].

4 Further Ideas

One of the major problems encountered is the large bandwidth required to send the proof across the network. In general the proofs can be very large, and so this may be a major bottleneck in an efficient implementation of the idea of PCC. To counter this, we propose using ideas from the field of Probabilistically Checkable Proofs (PCP) in complexity theory. A very readable survey of PCP can be found in [Joh92].

The idea that started all this was that of Interactive Proof Systems. These involve two individuals, the prover P and the verifier V. Consider the case that P claims that for some predicate ψ and input x, there is a y such that $\psi(x, y)$ is true. Such a y is called a *witness* for x, and W_x is the set of all witnesses for x.

An interactive proof is a protocol in which V flips some coins and asks P some questions; it guarantees that if there is indeed such a y, then there is a way for P to convince V. However, if P is lying and there is no such y (i.e. there is no witness), then with high probability, V is not convinced.

PCP grew out of these ideas, and describes a way to remove the element of interaction. So the prover initially writes down a proof π, which it sends to the prover. The prover then flips some coins and queries this proof at some locations (whose identity depends on the outcomes of the coin flips). We want the verifier to behave like in the case of Interactive Proofs. More formally :

- **[Completeness]** If $W_x \neq \phi$, then there exists a proof π such that the verifier accepts with probability 1. The probability is, of course, over the coin tosses of the verifier.

- **[Soundness]** If $W_x = \phi$, then clearly the prover is trying to prove an incorrect statement. So for all proofs π', the verifier V rejects with probability at least $\frac{1}{2}$.

How is this applicable here ? In our application, Bob is the verifier. After he gets the code Π, he computes the safety predicate SP_b for it. He now wants to find if there exists a derivation $\mathcal{D}$ for SP_b. More formally, let ψ represent the predicate that is true if $\mathcal{D}$ is the derivation for SP_b and false otherwise. In this scenario, Bob wants to decide the question $\exists \mathcal{D}\ \psi(SP_b, \mathcal{D})$. Further, Alice is the prover who wants to convince Bob that this is the case. Now that we can cast our scenario in the language of PCPs, we can ask the question “Do there exists efficient PCPs for the problem described above ?” i.e. what is the number of bits of the proof we have to read to get the above behaviour ?

The answer depends on what the complexity of proof-checking is. A famous result by Babai, Fortnow and Lund [BFL90] shows that given y, if we can verify $\phi(x, y)$ holds in exponential time in the length of x (say, n), then we can write a PCP π that can be verified by flipping poly(n) coins and reading only poly(n) bits of π.

So a positive answer to the question “Can we verify whether $\mathcal{D}$ is a derivation for SP_b in time exponential in the length of SP_b ?” will imply that we have probabilistically checkable proofs which only require us to read only polynomially many bits of a suitable encoding of $\mathcal{D}$ (instead of the entire derivation $\mathcal{D}$). Thus we would be sacrificing certainty about the safety of Π for faster certification and fewer bits sent over the network.

It should be noted that even if we cannot bound the time for verification to be exponential, it may be interesting to find out the complexity classes within which we can bound the proof checking. Once this is done, we can investigate the efficiency of providing PCPs for those classes.

5 Finally...

The idea that the proof of correctness of a program accompany it to engender trust in the program is a rather useful idea; in fact, it seems to be among the cleanest solutions to the problem of trying to identify safe code. Unfortunately, there are many bottlenecks, as indicated at the end of various sections above, and ironing out some of them may help a lot. Otherwise, this idea may end up being too theoretical to have mass appeal.

References

[BFL90] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. In *31st Annual Symposium on Foundations of Computer Science*, volume I, pages 16–25, St. Louis, Missouri, 22–24 October 1990. IEEE.

[FL97] Joan Feigenbaum and Peter Lee. Trust management and proof-carrying code in secure mobile-code applications. In *DARPA Workshop on Foundations for Secure Mobile Code*, 26–28 March 1997.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[Joh92] David S. Johnson. The NP-completeness column: An ongoing guide: The tale of the second prover. *Journal of Algorithms*, 13(3):502–524, September 1992.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 15–17 January 1997. ACM.

[NL96a] George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, 1996.

[NL96b] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 28–31 October 1996.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.