Electronic Notes in Theoretical Computer Science **20** (1999)
URL: http://www.elsevier.nl/locate/entcs/volume20.html 15 pages

Enforcing Security and Safety with Proof-Carrying Code

George C. Necula [1]

Department of Computer Science
University of California
Berkeley, USA

Abstract

In an environment where more and more code cannot be trusted to behave safety it is becoming necessary to employ mechanisms for detecting and preventing unsafe program behavior. This paper first reviews various such mechanisms and then focuses on static mechanisms with an emphasis on Proof-Carrying Code and its expressiveness.

Proof-Carrying Code is a technique that allows a code receiver to verify statically that the code has certain required properties, which are stated in the form of a safety policy. To make this possible the code is accompanied by a representation of an easily checkable formal proof of compliance with the safety policy. This paper discusses first the general properties of the Proof-Carrying Code technique and then explores a particular implementation of the idea using verification condition generators. As a surprising result we prove that by adopting such an implementation choice we limit ourselves to safety properties, which constitute but a subset (albeit a very important one) of all the interesting program properties. We further speculate on what it takes to extend Proof-Carrying Code to handle more that safety properties.

1 Introduction

With the advent of the Internet and mobile code it is becoming more and more important to ensure that untrusted code does not interfere in unintended ways with the operation of their host systems. Mobile code is an obvious example of untrusted code, with the lack of trust due mainly to the anonymity of the code authors. However, untrusted software has been around ever since we started writing programs. Even in the case a program of known origin we have to assume that it might be erroneous and, in certain instances, even malicious.

[1] A significant part of this work was done in collaboration with Peter Lee

This is especially important if the program is used in a context where a failure can have costly consequences.

Protecting against untrusted code is not a new concern. Various methods have been proposed and some have been used successfully for many years. We review some of these methods and compare them against each other based on their expressiveness and costs. The expressiveness of a safety mechanism is a measure of what kinds of undesired code behavior can be prevented. The cost of a protection mechanism can be looked at from different perspectives. One such perspective is the performance penalty imposed on applications in the process of enforcing security. Another cost aspect is the complexity of the trusted infrastructure. A security mechanism is better if it relies on a smaller body of trusted code for its correct operation. Such a trusted body of code is usually called the trusted computing base. Finally, another kind of cost, tightly linked to expressiveness, is the opportunity cost: what implementation strategies and programming paradigms cannot be used in conjunction with a given security mechanism.

The protection mechanisms used in present systems can be classified in three main categories. First, there are techniques that judge the safety of the code by the identity of the code author. Then, there are techniques where the untrusted code is executed in a "sandbox", so that it cannot access critical data and resources directly. Finally, there are the techniques that rely on interpreters or receiver-side compilers to enforce safe code behavior. These three classes of techniques, with their advantages and disadvantages, are reviewed next.

1.1 Cryptography: Security through Personal Authority

The aim of this class of techniques is not to prevent unsafe code from being executed, but to create accountability and thus a deterrent to the distribution of harmful programs. For this purpose, the code producers are required to sign digitally the code they produce. This allows the receiver to verify not only the identity of the producer but also that the code integrity was maintained from the producer to the receiver. The most commonly used technology for this purpose is that of public key cryptography, as used for example in Microsoft's Authenticode [6].

While this approach has its practical merits, it also has a number of disadvantages, the most important of which being that it does not prevent trusted producers from creating and distributing erroneous programs. The fact that the error can be traced to a concrete producer is of little consolation when significant damage has already occurred due to the error.

The personal authority approach to safety penalizes unfairly the small individual software producers in favor of larger and better-known producers. A receiver is unlikely to grant the benefit of doubt to a program that is received from an unknown or a lesser-known producer. While this might not be a

problem in small enclaves where the interacting entities know each other, it is definitely a drawback in environments where large anonymous masses of programmers share programs.

Because of the above reasons, personal authority *alone* is not a good way to ensure safety. Instead, we must look at approaches that consider the intrinsic, or semantic, properties of the untrusted code, independently of extrinsic properties such as who produced it or how it was produced. However, personal authority might be an important component of a safety policy for execution of mobile code and, in such cases, personal authority can be used in combination with one or more of the techniques discussed next.

1.2 Memory Protection: Safety through Run-Time Checking

One of the simplest methods to ensure the safe execution of untrusted code is to isolate it in a hardware-enforced address space so that all accesses to non-private data and resources can be intercepted and monitored by the server. This is also the main technique that operating system kernels use to protect themselves from misbehaving user-level applications.

The most important component of such a scheme for protection is the design of a secure system-call interface, which consists of system-provided functions that the untrusted code may invoke for accessing system data and resources. A proper implementation of the system-call interface must check that the invoking program has the necessary rights to perform the requested operation and that the provided arguments are valid. Extra care is required in a multiprocessing environment to ensure the atomicity of the argument checking procedure. Usually, this is accomplished by first copying the arguments in a memory area inaccessible to user applications.

The main advantage of the hardware-based solution is that it is relatively simple, thus easy to implement and to trust. The disadvantage is the high cost associated with switching between the protection domain of the untrusted program and the server, and back, along with the cost of copying the arguments and the results between address spaces.

Any pure run-time protection mechanism relies on its ability to monitor the execution of the untrusted software and perform two main operations: (1) *detecting* that an operation is about to violate the safety policy, and (2) *stopping* the execution of the untrusted code when such a situation is encountered. Most of the time the ability to perform these basic tasks is taken for granted. There are however instances when run-time checking falls short of an ideal protection mechanism.

Firstly, there are situations when it is hard to detect the "bad" operation by just watching the execution at run time. A classic example of such a case is when the property to be enforced is an instance of enforcing data abstraction. Data abstraction is a property that is best checked statically. It can also be checked at run-time by imposing various data-type representation

constraints. For example, it is theoretically impossible to check at run-time that a certain value is a pointer to a null-terminated sequence of characters. We can however conservatively require that the sequence have a maximum length so that we can check this property at run time. In general, heavy reliance on run-time checking constrains the design of the interface between the trusted and untrusted code.

Secondly, it is not always safe to terminate forcefully the execution of the untrusted code. The classic example of a safety policy that cannot possibly be enforced by pure run-time mechanisms requires that a certain critical action happens in a timely manner (e.g., "the reactor must be shutdown within 200ms after a certain event occurred"). While it is easy to detect the violation of such a safety policy immediately before it occurs, that does not help. This is because even though we can stop the execution we cannot stop the time. There are other, more mundane, situations when it is also undesirable to terminate forcefully the execution of a program, namely when the program has acquired critical resources. The classic solution in this case is to let the untrusted code acquire only revocable resources (such as processor cycles in a system with preemptive scheduling).

1.3 Type Checking: Safety through Static Checking

The discussion of the limitations of run-time checking enforcement mechanisms suggests that we should explore a combination of static and run-time checking techniques.

In general, a host using such a protection technique enforces the safe semantics in two stages. First, during the static checking stage, it performs a detailed inspection of the untrusted code to ensure that it is a valid program. This stage includes syntax checking and also type checking in the case of typed languages. The second stage takes place while the agent is executing. In this second stage, the server ensures that those operations that are potentially harmful and whose safety cannot be ascertained by the static checking pass, are preceded by run-time checks for safety.

The purpose of the static checking phase is to catch early many of the common programming errors and sources of harmful behavior. The amount and kind of static checking that can be performed ranges from simple control-flow checking to complicated type checking. For example, in the Berkeley Packet Filter (BPF) [5] architecture, the untrusted code is scanned to verify that all instructions belong to a restricted bytecode language, that all branches are forward, and that their targets are within the code boundaries. In contrast, for approaches based on type-safe languages, a full-fledged type checking pass is made over the agent's code.

After the static checking stage, the server must execute the untrusted code in such a way that potentially harmful operations are guarded by run-time checks for safety, such as array bounds checking or null-pointer checking. The

more complicated the static checking, the fewer run-time checks are required. In the case when static checking consists of type checking in an expressive type system, code properties such as memory safety and data abstraction are guaranteed with few or no run-time checks.

The easiest way to implement the dynamic checking stage is through *interpretation*. Basically, this means that the untrusted code is interpreted by a safe and trusted interpreter that performs all of the required dynamic checks mandated by the safety policy. For example, in order to enforce memory safety, an interpreter can verify, before each memory access, that the code accesses only memory areas that are permitted by the safety policy. Two examples of safe interpreters are the Berkeley Packet Filter interpreter [5] for operating system extensions and the Java Virtual Machine interpreter [4] for mobile code.

The major drawback of the interpreter-based approach is the reduced execution speed; it is not unusual to observe an order of magnitude slowdown due to interpretation. A natural solution to the interpretation overhead problem is to replace the interpreter with a trusted just-in-time compiler that, while compiling the untrusted code, inserts run-time checks equivalent to those that an interpreter would perform. Note that I use the generic term "compiler" for such a tool even in cases when the source and target language are the same. For example, a compiler can edit machine code agents by inserting bounds checks before memory operations. This approach, called Software Fault Isolation (SFI) [14], is used to enforce memory safety in the extensible operating system VINO [13].

The SFI compiler mentioned above analyzes and modifies machine code agents so as to enforce memory safety. However, it is easier to analyze agents written in high-level or intermediate-level languages, meaning that more complex safety properties can be enforced with fewer safety checks at the cost of a more complicated compiler. For example, a compiler for Java Virtual Machine bytecode produces machine code augmented with array bounds checks. The higher-level language allows a type checker to enforce statically properties like data abstraction, and it restricts the run-time checking for memory safety only to those memory operations that access arrays.

A compiler-based approach to safety leads to significantly faster execution of untrusted code than possible with the interpreted versions. However, this advantage comes at a high price. A compiler is significantly more complex than an interpreter and this means that a host system must rely on the correctness of a larger and more complicated body of code. This drawback is getting more pronounced as more optimizations are incorporated in the compiler.

2 Proof-Carrying Code

The discussion of traditional techniques for ensuring the safety of untrusted code suggests that an ideal enforcement method should be based as much as

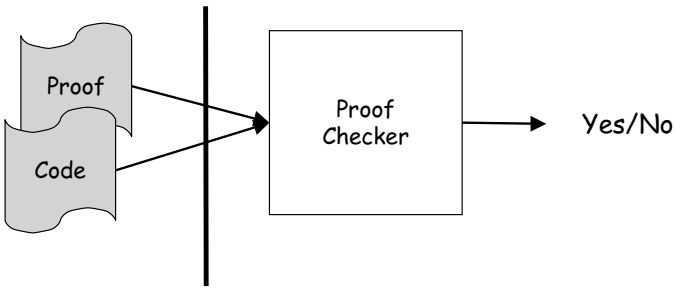

Fig. 1. A high-level view of Proof-Carrying Code. The grayed elements are untrusted while the proof checker is trusted.

possible on static checking. This way we avoid both the overhead of run-time checking and the complications associated with forcefully terminating agents. With static checking, the execution of the agent code is not even started unless it is guaranteed to be safe.

My thesis is that ideas from logic and programming languages can and should be used to ensure the safety of executing untrusted software agents by means of static checking, without sacrificing performance and without relying on personal authority. Furthermore, this can be achieved with a small trusted infrastructure on the code-receiver side. In order to minimize the complexity of the static checking, and therefore of the required infrastructure, the host system relies on easily checkable client-provided evidence attesting to the safety properties of the code. This technique is called *proof-carrying code* [7] and is depicted in Figure 1.

Although we will not discuss this here, for a large class of safety properties, the formal proof can be produced automatically by a certifying compiler. The particular safety properties that I have in mind are those that could also be enforced by the safe interpreter or just-in-time compiler approaches discussed in the previous section. The major difference is that the certifying compilation process takes place at the client's site, and the host does not have to incur the cost of compilation or interpretation nor does it have to trust the compiler or the interpreter. For more details, see [9].

Proof-carrying code has several key characteristics that, in combination, give it an advantage over previous approaches to safe execution of untrusted code. In addition, there are costs associated with using PCC.

For proof-carrying code several advantages can be claimed:

(i) *PCC is general.* PCC can be used to enforce more than memory safety, more even than type safety. At an extreme, PCC can be used to verify any code property for which there exists a logic capable of expressing it. This includes many code properties that would otherwise be undecidable to infer from the code alone. PCC has been tested with safety properties ranging from memory and type safety to bounded resource usage. However, any single implementation of PCC might be less general. In fact, starting with Section 3 we explore the limitations in expressiveness

introduced by a seemingly innocuous implementation choice.

(ii) *PCC trusted infrastructure is low-risk and automatic.* The proof-checking process used by the host to determine the safety of the code is completely automatic, and can be implemented by a program that is relatively simple and easy to trust. Thus, the safety-critical infrastructure that the code receiver must rely upon is reduced to a minimum.

(iii) *PCC is efficient.* In practice, the proof-checking process runs quickly. Furthermore, in contrast to some previous approaches, the host does not modify the code in order to insert costly run-time safety checks, nor does the server perform any other checking or interpretation once the proof itself has been validated and the code installed.

(iv) *PCC does not require trust relationships.* The host does not need to trust the code producer. In other words, the host does not have to know the identity of the producer, nor does it have to know anything about the process by which the agent code was produced. All of the information needed for determining the safety of the code is included in the untrusted code and its proof.

(v) *PCC is flexible.* The proof-checker does not require that a particular programming language be used. PCC can be used for a wide range of languages, even machine languages. Furthermore, a host can support multiple languages and safety policies with a minimal duplication of the infrastructure components.

(vi) *PCC clients can be automated in special cases.* If the safety properties can be decided statically or enforced through systematic run-time checks, a certifying compiler together with a suitable theorem prover can be used on the code-producer side to automate the process of producing the proofs.

The benefits of PCC and certifying compilation discussed above in this section do not come for free. The main costs of using PCC as a protection mechanism for untrusted code are:

(i) Because PCC is a cooperative process, the code producer must be involved in establishing the safety of the agent code, in contrast with other approaches where the safety enforcement is completely transparent for the producer. This poses difficulty in deploying PCC, as the potentially more numerous clients need to be made PCC-aware, not only the hosts. Note however that this cost only exists because PCC can be used to enforce a multitude of safety policies as opposed to other techniques where the safety policy is hardwired and so no negotiation is necessary or even possible.

(ii) Proof-carrying code prescribes a precise method by which code producers must cooperate with receiving hosts to assist in the verification of code properties. It is less well-defined the method by which producers can cooperate with the hosts for the purpose of establishing a mutually

convenient safety policy.

(iii) Finally, generating the proof of safety is a difficult task. The theorem prover that is part of our current system for generating PCC is powerful, but not complete. It is also modular, so new decision procedures can be easily added, but in the difficult cases the prover must be guided by the user. Theorem proving can be automated when the code is compiler generated, by extending the prover to handle the finite number of patterns of verification conditions that are possible. This is how automation is achieved for the Touchstone certifying compiler, described here briefly in Section 3 and in more detail in [8].

The costs enumerated above are important, but fortunately they are incurred only by code producers. By comparing the list of benefits and the list of costs, it becomes apparent the intentional design strategy for PCC: the burden of safety should lie on the producer and not on the receiving host. This enables PCC to work with a very small, easy-to-trust and automatic infrastructure, so that the difficult work is done by the client who is in a better position to understand the untrusted code or to use interactive tools for that purpose.

To conclude this overview section I note that the relative balance of advantages and costs of proof-carrying code demonstrate that this is a better and more general way to deal with the safety of untrusted code without having to pay the costs of high run-time overhead or of a complex trusted computing base.

3 An Implementation of Proof-Carrying Code

While the idea of Proof-Carrying Code can be described abstractly, as we did in the previous section, any practical implementation of it requires that certain choices be made. In this section we are going to describe at a high level the particular implementation choices that we made in out current prototype. For a more detailed discussion of the implementation issues the reader is referred to [8].

The key building block in our implementation of PCC is a *verification condition generator* (VCGen) that scans the untrusted code, performs simple syntax and control-flow checking and generates a list of *verification conditions*. Each verification condition reflects an atomic requirement that must be met at a given point in the program. Examples of verification conditions are "memory address A is readable" and "the value of register $\mathbf{r}_0$ is the same as on function entry". These statements are expressed as formulas in a suitable logic that is part of the safety policy. One way to view the verification condition generator is as a symbolic evaluator that collects a symbolic representation of the run-time checks that a safe interpreter would perform on the code. More about this analogy later.

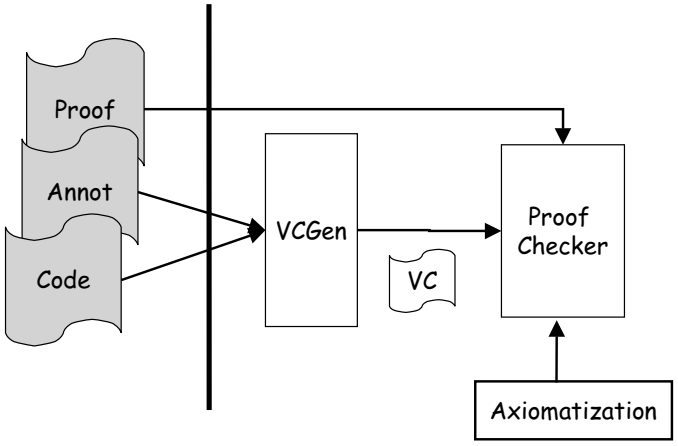

Fig. 2. A concrete implementation of the Proof-Carrying Code concept. The grayed elements are untrusted while the other are trusted.

The rest of the presentation refers to Figure 2, which shows the main building blocks of our implementation of Proof-Carrying Code. The decision of using VCGen has several consequences. We discuss here those constraining the rest of the implementation and we defer to Section 4 the expressiveness consequences.

In order to perform its symbolic evaluation in one scan of the code, VCGen requires that the untrusted code be annotated with loop invariants and with function specifications in the form of preconditions and postconditions. These are referred to as annotations in Figure 2.

The key benefit of using VCGen as part of the Proof-Carrying Code implementation is that the proof of the verification condition acts as a proof of compliance with the safety policy. The proof is checked for validity and at the same time it is verified to be a proof of the correct verification condition. Only if both of these tests succeed is the code accepted.

There are significant engineering advantages of using a VCGen-based approach for implementing Proof-Carrying Code. First, the trusted infrastructure (everything to the right of the vertical separator in Figure 2) is split into functionally independent modules. The VCGen module is highly dependent on the language used by the untrusted code but is largely independent of the particular safety policy being enforced. For this purpose, the logic that VCGen uses contains generic formulas such as "`saferd`(A)" to say that the symbolic expression "A" denotes an readable address. The actual meaning of the `saferd` predicate is defined as a series of trusted axioms and inference rules which are part of the safety policy. For example, one safety policy might define an inference rule that derives "`saferd`(A)" whenever "A" can be proved to be non-null and of a pointer type. Another safety policy might choose to not have any inference rules with "`saferd`" as a conclusion, thus effectively disallowing memory read operations.

The other component of the trusted infrastructure is the proof checker.

Our design choice here was to encode the proofs as expressions in a variant of typed-lambda calculus called *Edinburgh Logical Framework* (LF) [2]. At the same time, the verification conditions can be encoded as types in the same language and the inference rules as constant declarations. With this setup we can check that a proof is valid and that it proves a given verification condition by checking that the LF expression that encodes the proof has the LF type that encodes the verification condition [2,8]. Furthermore, with some further additions to the LF framework we can represent proof quite compactly [10].

The major advantage of the choice of using LF for proof representation and checking is that we are able to use a very simple proof checker (LF type checking can be described completely by 15 typing rules) yet extremely versatile. A vast number of logics, and hence safety policies, can be encoded in LF for use with the same implementation of the proof checker [2].

Thus neither the VCGen nor the proof checker need to be reimplemented when changing the safety policy. Only the logic must be changed but that is a simpler task since the description of the inference rules in LF is at a high level similar to the familiar mathematical notation. This opportunity for reuse is not only an engineering advantage but also a major reason for trusting the PCC infrastructure.

Before we move on to the implications that the VCGen-based design have on the expressiveness of PCC, we want to share some of our experience with using this prototype.

We have used the VCGen-based implementation of Proof-Carrying Code to enforce type safety and memory safety for untrusted optimized machine code produced by compiling a type-safe subset of C. Our Touchstone certifying compiler [9] is different from other C compilers in that it documents (in the form of loop invariants and function specifications) all the code transformations it performs. This allows a theorem prover to reconstruct automatically the proof of the verification condition.

The results of these experiments are promising. While code quality is not compromised (the quality of the output matches or surpasses that of commercial optimizing C compilers) the code is guaranteed (and proved) to be type safe and memory safe. Furthermore, the size of proofs ranges from 25% to 70% of the size of the machine code, and the time required for proof checking is a mere 2% of the time required for either compilation or theorem proving. These experiments also allow us to obtain a quantitative comparison between the sizes of the trusted infrastructure (the VCGen and the proof checker) and the untrusted software (the compiler and the theorem prover). We observed that the trusted code is about 25% the size of the untrusted code in our system. Furthermore, as more optimizations are incorporated in the compiler and the theorem prover this ratio is going to decrease further.

The choice of using a VCGen has effect on the expressiveness of the approach mainly due to the fact that VCGen is state based. To understand this it helps to imagine that we define a fictitious safe interpreter for untrusted

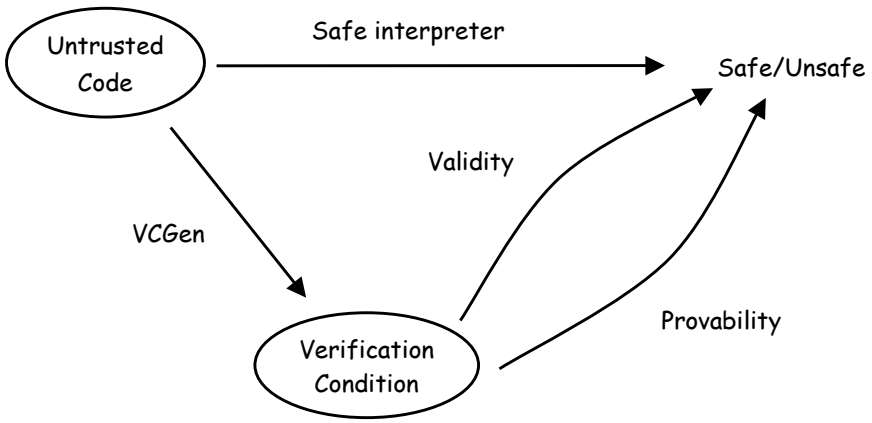

Fig. 3. The relationship between the safety policy (as represented by a fictitious safe interpreter), the verification condition, its validity and its provability

code and we make this interpreter perform enough run-time checks to enforce the safety policy. For example, when the interpreter would encounter a memory operation it performs the necessary check for memory safety. Similarly, when a function is invoked the interpreter checks certain relationships between the actual arguments. Essentially we can model our safety policy as such a fictitious interpreter or, more precisely, as the set of programs that execute in the safe interpreter without being forcefully terminated.

Of course it is impractical in general to construct and use such an interpreter, not only because of the potentially high run-time costs but also because some of the checks might be hard or impossible to perform at run time. With this view of the safety policy, VCGen is simply a symbolic evaluator that constructs a formula that is valid in a suitable logic only if all the checks that the symbolic interpreter would perform succeed. In fact, VCGen while scanning the code performs many of the same syntactic checks that the safe interpreter would perform and emits suitable verification conditions for all the other checks.

So, it seems that instead of having to build the safe interpreter we can just compute the verification condition and then verify its validity. This is depicted in Figure 3 as the closure of a commutative diagram. However, since checking the validity of a formula in most logics is impractical we resort to the sound but not necessarily complete method of checking the existence of a derivation of the formula with respect with a set of inference rules trusted to be sound. Along these exact lines is the proof of correctness of the VCGen-based implementation of Proof-Carrying Code. For more details and the complete proof see [8].

4 The Limitations of VCGen-based Proof-Carrying Code

The advantages of a VCGen-based implementation of Proof-Carrying Code do not come for free. We mentioned in the previous section the requirement

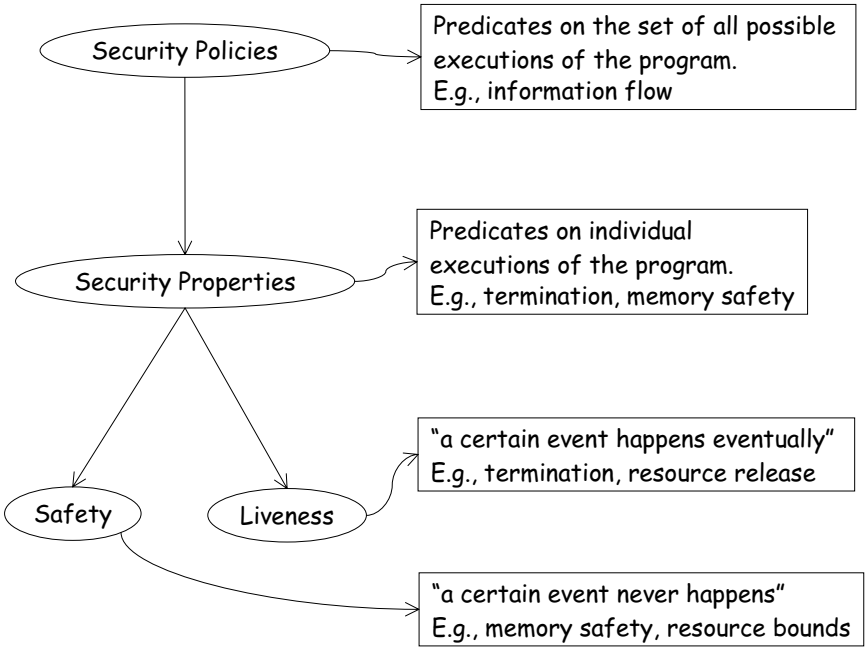

Fig. 4. A Taxonomy of Security Properties of Programs

for loop invariants. But the major, and somewhat surprising, limitation of a VCGen-based approach is that it restricts the Proof-Carrying Code implementation to enforcing only a small, albeit important, subset of all the interesting program properties. In order to understand this statement it helps to stand back and consider a taxonomy of program properties from a security point of view, as depicted in Figure 4.

At the root of the hierarchy are the *security policies*. Following Schneider [12], a security policy is a predicate on sets of program executions. The source of the program is a static encoding of the set of all its possible executions. A program is said to satisfy a security policy if the security policy predicate holds for the set of all possible executions of the program.

An important class of security policies are the *security properties*, which are defined in [1] as those security policies that can be specified by means of a predicate on individual executions, or equivalently by imposing constraints on each individual execution as opposed to constraints on the set of all possible executions. There are interesting security policies that are not properties. For example, an information flow security policy prohibits correlations between the values of state components with different secrecy status, so that principals cannot infer things about a state component considered classified by observing the values of other unclassified state components. It is obvious that information flow is not a property, because the very notion of correlation involves more than one execution. If we only look at a single execution or even at a subset of the possible executions we might notice correlations that do not exist when all executions are considered.

A further subclass of security properties are the *safety properties*, which

stipulate that no "bad thing" happens during the execution [3]. The safety properties are characterized by the fact that they hold for an execution only if they hold for all finite prefixes of the execution. This means that if a safety property fails, the point of failure is identifiable, which makes it possible to check safety properties at run time [12]. Not all security properties are safety properties. There are also *liveness properties* that stipulate that certain "good things" (e.g., termination, release of a resource) must happen [3]. In fact, it can be proven that any security property can be expressed as a combination of a safety and a liveness property [1].

5 Conclusion

We showed in the previous section that, in principle, Proof-Carrying Code implemented using a verification condition generator is not more expressive than run-time checking, even though is less expensive. Before we proceed to speculate what could be done to extend Proof-Carrying Code beyond the realm of safety properties we ought to point out that the vast majority of security policies are entirely or to a large extent safety policies. From a practical point of view the limitation of Proof-Carrying Code is not its expressiveness but our ability to generate the proofs that it requires.

The first natural extension of Proof-Carrying Code ought to handle liveness properties. VCGen, along with the fictitious safe interpreter that it models, is very good at checking assertions on the current and past execution states. For this purpose various forms of first-order logics usually suffice. In order to handle liveness properties we need to be able to make and check assertions about future execution states. This requires moving from a first-order logic to temporal logics. Furthermore, the structure of the program must be taken more seriously into consideration since, the only way to be able to assert something about the future of the execution is to consider the structure of the source program.

However, let us point out that in many practical instances liveness properties can be conveniently approximated with safety properties. The trick is to require not only that a certain event happens eventually but to impose fixed or variable deadlines. Once we have set deadlines the point of failure of compliance with the safety policy is precisely determined and detectable, meaning we are now dealing with a safety property. For further discussion of this approach see [11].

But what about extending the Proof-Carrying Code approach to non-properties? Obviously the safe interpreter is not anymore a good model of the safety policy. The whole idea of a run-time model of safety is futile since we would have to monitor multiple executions (or even all the possible executions) in order to be able to detect a violation of the security policy. The key idea here is that PCC has access to a concise representation of all the possible executions, namely the source of the program. What we need is a

way to collect from the program a set of structural and security constraints whose satisfying substitution constitutes a proof of compliance with the security policy.

Thus the key to extending Proof-Carrying Code is to exploit to a larger extent the availability of the source code and to derive from its structure liveness properties and general non-properties.

References

[1] Alpern, B. and F. B. Schneider, *Defining liveness*, Information Processing Letters **21** (1985), pp. 181–185.

[2] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, Journal of the Association for Computing Machinery **40** (1993), pp. 143–184.

[3] Lamport, L., *Proving the correctness of multiprocess programs*, IEEE Trans. Software Engineering **3** (1977), pp. 125–143.

[4] Lindholm, T. and F. Yellin, The Java Virtual Machine Specification, The Java Series, Addison-Wesley, Reading, MA, USA, 1997, xvi + 475 pp.
URL `http://www.aw.com/cp/javaseries.html`

[5] McCanne, S. and V. Jacobson, *The BSD packet filter: A new architecture for user-level packet capture*, in: The Winter 1993 USENIX Conference, USENIX Association, 1993.

[6] Microsoft Corporation, *Proposal for authenticating code via the Internet* (1996), http://www.microsoft.com/security/tech/authcode/authcode-f.htm.

[7] Necula, G. C., *Proof-carrying code*, in: The 24th Annual ACM Symposium on Principles of Programming Languages, ACM, 1997.

[8] Necula, G. C., Compiling with Proofs, Ph.D. thesis, Carnegie Mellon University (1998), also available as CMU-CS-98-154.

[9] Necula, G. C. and P. Lee, *The design and implementation of a certifying compiler*, in: ACM SIGPLAN'98 Conference on Programming Language Design and Implementation, 1998.

[10] Necula, G. C. and P. Lee, *Efficient representation and validation of proofs*, in: Thirteenth Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, Indianapolis, 1998.

[11] Necula, G. C. and P. Lee, *Safe, untrusted agents using proof-carrying code*, in: G. Vigna, editor, LNCS 1419: Special Issue on Mobile Agent Security, Springer-Verlag, 1998.

[12] Schneider, F. B., *Enforceable security policies*, Computer Science Technical Report TR98-1644, Cornell University, Computer Science Department (1998).

[13] Seltzer, M. I., Y. Endo, C. Small and K. A. Smith, *Dealing with disaster: Surviving misbehaved kernel extensions*, in: Second Symposium on Operating Systems Design and Implementations, Usenix, 1996.

[14] Wahbe, R., S. Lucco, T. E. Anderson and S. L. Graham, *Efficient software-based fault isolation*, in: 14th ACM Symposium on Operating Systems Principles, ACM, 1993.