Published in Proc. of Conference on Automated Deduction, 2002, pp. 47–62.

A Gradual Approach to a More Trustworthy, yet Scalable, Proof-Carrying Code

Robert R. Schneck[1] and George C. Necula[2,⋆]

[1] Group in Logic and the Methodology of Science
University of California, Berkeley
schneck@math.berkeley.edu
[2] Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
necula@cs.berkeley.edu

Abstract. Proof-carrying code (PCC) allows a code producer to associate to a program a machine-checkable proof of its safety. In the original approach to PCC, the safety policy includes proof rules which determine how various actions are to be proved safe. These proof rules have been considered part of the trusted code base (TCB) of the PCC system. We wish to remove the proof rules from the TCB by providing a formal proof of their soundness. This makes the PCC system more secure, by reducing the TCB; it also makes the system more flexible, by allowing code producers to provide their own safety-policy proof rules, if they can guarantee their soundness. Furthermore this security and flexibility are gained without any loss in the ability to handle large programs.

In this paper we discuss how to produce the necessary formal soundness theorem given a safety policy. As an application of the framework, we have used the Coq system to prove the soundness of the proof rules for a type-based safety policy for native machine code compiled from Java.

1 Introduction

Proof-carrying code (PCC) [7] is a technique that shifts the burden of certifying properties of a program or data from the consumer to the producer, with the main goal of keeping the consumer's trusted code base (TCB) as small and trustworthy as possible. However, in the existing implementations of proof-carrying code there seems to exist a tension between the minimality of the TCB and engineering considerations necessary for handling realistic safety policies and large programs.

⋆ This research was supported in part by National Science Foundation Career Grant No. CCR-9875171, ITR Grants No. CCR-0085949 and No. CCR-0081588, gifts from AT&T Research and Microsoft Research, and a National Science Foundation Graduate Research Fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

The system described by Necula [8] was engineered to scale to large programs (up to half a million lines of code) and to realistic safety policies (e.g. a type-safety policy for native machine code compiled from Java [3]) with a relatively modest investment (about two person-years).

The typical interaction taking place in this PCC system is depicted in Figure 1 as a negotiation between a code producer and a code consumer. Upon being presented with a code fragment, the code consumer uses a verification-condition generator (VCGen) which produces a set of verification conditions (VC), whose validity entails the safety of the code. The validity of the VC must be proved with respect to a set of proof rules that are provided (and trusted) by the code consumer. In the second stage, the code producer constructs a representation of a proof of the VC and presents that to the code consumer, who can now simply run a proof checker to satisfy itself that the VC is provable.

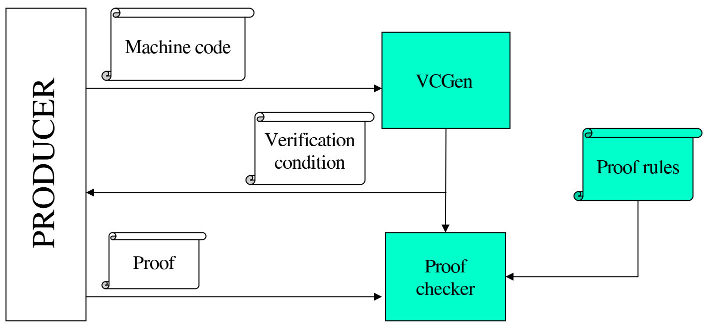

Fig. 1. The structure of a proof-carrying code system showing a "dialogue" between the code producer (on the left) and the code consumer (composed of the trusted elements shown shaded).

Thus, in addition to the proof checker, the TCB for this system includes a verification-condition generator (VCGen) and a list of proof rules, which together constitute the safety policy. Altogether the TCB requires about 15,000 to 25,000 lines of code, depending on configuration and underlying architecture. It is not inaccurate to describe this infrastructure as simple and small, and therefore easy to trust, at least when one considers possible alternatives such as trusting optimizing compilers.

However, it is reasonable to ask whether it is possible to create a working PCC where the TCB is smaller. We want to combine the low-cost and scalability benefits of the traditional implementation of PCC with the security of a reduced TCB. In particular, we observe that among the many lines of code that one must trust in a PCC implementation, most are independent of the safety policy. This category includes the proof checker and the VCGen (which essentially contains a machine-code decoder, and the handling of control-flow). These parts are shared between safety policies and change very rarely, which implicitly means that they are tested more extensively and thus are more trustworthy. On the other hand the proof rules of the safety policy change with every safety policy. In our experience these parts are more likely to contain errors. We consider it more cost-effective to focus heavy-weight tools such as theorem provers for the purpose of certifying policy-dependent extensions to PCC, while trusting the implementation of the core system.

We propose to remove the safety-policy proof rules from the TCB, by providing a framework in which the proof rules can be formally proven sound. This not only increases the security of the PCC system, but also its flexibility, as code producers would be able to supply their own safety policies, as long as the proof rules could be formally proven sound in our framework. One way to view this approach is as an attempt to extend the dialogue between the code producer and the code consumer to the level of the safety-policy rules: the code producer expresses the intention to install a new set of proof rules; the code consumer inspects the rules and replies with a soundness theorem which the code producer must prove. The question then is what is the counterpart of a VCGen for producing the soundness theorem for proof rules? And furthermore, what is the reference proof system against which the new rules are being judged?

This paper describes the prototype of a framework to take proof rules for a safety policy, and produce a formal theorem to guarantee that, relative to the still-trusted behavior of VCGen, the notion of safety given by the proof rules really does imply *memory safety*. Note that we do not show how to produce the formal *proof* of this theorem; in particular, although it is necessary to the feasibility of a PCC system that the safety proofs for programs can be automatically generated (e.g. by a certifying compiler), at this point the soundness proofs for safety policies still need to be interactively generated. Also note that our framework only ensures memory safety. Safety policies are often expected to handle more than memory safety: for instance, a type-safety policy will also enforce abstraction, or a safety policy may include other features such as instruction counting. Memory safety can be considered to play the role of a *safety meta-policy*: the system will not accept a new safety policy unless it can at least guarantee memory safety, regardless of what else it promises to do.

Using the specification language of the Coq system, we have implemented the framework in the case of typing rules for native machine code compiled from Java, providing the formal statement of the theorem that these rules guarantee memory safety. Moreover we have used the Coq proof system to produce a formal proof of this theorem.[3]

We begin with the development of our framework for how to take a safety policy and produce the statement of the necessary soundness theorem, using a

[3] Note that the handling of Java in PCC requires extensions to VCGen to handle e.g. dynamic dispatch and exceptions; our framework is concerned *only* with the proof rules of the safety policy, so our proof does not guarantee the soundness of these VCGen extensions. The soundness of VCGen extensions is the subject of our current research.

"toy" safety policy as a motivating example. Next we briefly consider the Java safety policy and our formal proof of the soundness of these proof rules. Finally we compare our work with other approaches for more secure proof-carrying code, and consider future directions for this research.

2 Stating Soundness for Safety Policies

Our intention, which is currently only partly realized, is to give an entirely modular framework which can take a description of a safety policy and produce the statement of the soundness theorem for the proof rules of the policy.

2.1 Safety Policies

First we must detail what constitutes a description of a safety policy. A safety policy comprises[4]

1. a list of proof rules;
2. trusted run-time support functions;
3. extensions to the VCGen mechanism.

The soundness theorem states only the soundness of the proof rules, so currently we require any extensions to VCGen to remain in the trusted code base.

The proof rules. Most importantly there is a formal description of the proof rules for the predicates `saferd` and `safewr`, which are used by VCGen as the proof obligations for memory reads and writes. The proof rules specify the meaning of these predicates—and thus the meaning of safety in the policy—by defining how they are to be proved. The proof rules may define various auxiliary types and predicates and the rules for them as well. For instance, a type-safety policy will typically define a (logical) type whose values are the various (computational) types of the type system considered, as well as a typing predicate with various rules; then the single rule for `safewr` will be that (`safewr` E_1 E_2) is provable given a proof that E_1 is a pointer to a type T and E_2 is of type T.

As a motivating example, consider the following "toy" safety policy. The basic idea is that there is a type of "boxes". Boxes encode some kind of value if odd; for instance, they might encode integers in all but the least significant bit—however for our purposes we ignore the coding and simply allow anything odd to be a box. Boxes which are even are pointers to other boxes. The proof rules are as follows. The typing predicate is `ofType`; `ofmem` is intended to hold of well-typed memory states.

[4] Certain safety requirements are imposed by the core VCGen (e.g., each instruction is valid, each jump is within the code block); we will not consider these general requirements when looking at any particular safety policy.

$\texttt{box} : \texttt{type},\ \texttt{ptr\_box} : \texttt{type}$

$$\frac{(\texttt{odd}\ E)}{(\texttt{ofType}\ E\ \texttt{box})} \qquad \frac{(\texttt{ofType}\ E\ \texttt{ptr\_box}) \quad (\texttt{even}\ E)}{(\texttt{ofType}\ E\ \texttt{box})}$$

$$\frac{(\texttt{ofType}\ E\ \texttt{box}) \quad (\texttt{even}\ E)}{(\texttt{ofType}\ E\ \texttt{ptr\_box})} \qquad \frac{(\texttt{ofType}\ E\ \texttt{ptr\_box}) \quad (\texttt{ofmem}\ M)}{(\texttt{ofType}\ (\texttt{sel}\ M\ E)\ \texttt{box})}$$

$$\frac{(\texttt{ofmem}\ M) \quad (\texttt{ofType}\ E_1\ \texttt{ptr\_box}) \quad (\texttt{ofType}\ E_2\ \texttt{box})}{(\texttt{ofmem}\ (\texttt{upd}\ M\ E_1\ E_2))}$$

$$\frac{(\texttt{ofType}\ E\ \texttt{ptr\_box})}{(\texttt{saferd}\ E)} \qquad \frac{(\texttt{ofType}\ E_1\ \texttt{ptr\_box}) \quad (\texttt{ofType}\ E_2\ \texttt{box})}{(\texttt{safewr}\ E_1\ E_2)}$$

These rules exhibit that anything odd can be a box; that even pointers and even boxes coincide; that reading a pointer in a well-typed memory produces a box; and that well-typedness of memory is preserved by memory writes of the appropriate type. Finally there are the rules for `saferd` and `safewr` based on the pointer type.

Observe that the proof rules alone provide no instances of the form (`ofType` E `ptr_box`) or (`ofmem` M). Such hypotheses will arise instead from trusted functions.

Trusted run-time functions. A safety policy may also specify trusted run-time functions. Each function is given as executable code accompanied by a precondition and postcondition, given in terms of the predicates specified by the proof rules of the safety policy.

The trusted run-time functions affect our handling of the soundness of the proof rules because of the extra assumptions created by the postconditions. The semantics of the predicates occurring in the postconditions changes, as they may hold for reasons other than those given by the proof rules. Consider again our box safety policy. It provides a trusted function which allocates memory for a box. The postcondition of the function is that, for the returned memory address RES,

$$(\texttt{ofType}\ RES\ \texttt{ptr\_box}).$$

This should not be taken to mean that this instance of `ofType` is provable using the proof rules: this is a new way of obtaining an instance of this predicate. Similarly, (`ofmem` M) holds for M the memory state after the most recent allocation.

When we try to prove soundness, we will need to consider the actual code of the allocation function as well. We will certainly want as an invariant that, whenever (`ofType` E `ptr_box`) is provable, then the memory at E has actually been allocated. As discussed in Section 2.2, the soundness theorem involves the execution of a hypothetical program on a certain abstract machine; we require encoding the trusted functions as state transitions on this machine.

VCGen extensions. A safety policy may provide extensions to VCGen. These extensions adjust the behavior of VCGen to handle situations expected by the safety policy. Examples include the handling of dynamic dispatch and exceptions in Java; another kind of example would be an instruction counter, which ensures (via emitted proof obligations) that a certain number of instructions intervene between two successive operations of a given type. Since our framework currently assumes the behavior of VCGen (and all extensions) is trusted, we can mostly ignore the exceptions in stating the soundness theorem.

However, an extension may change the meaning of a predicate from the proof rules by providing assumptions of that predicate based on static information obtained from the code. Essentially, new proof rules are added in a program-specific way. Instead of being formalized as an inductive type whose inductive constructors are the given proof rules, such a predicate needs to formalized as a parameter, where which instances hold is dependent on the particular program being considered. We call these predicates *trusted assumptions*. An example from the Java safety policy is the predicate `jifield`, where $(\texttt{jifield}\ C\ \mathit{OFF}\ T)$ is intended to mean that class C has a field of type T at offset OFF.

The extension which collects the trusted assumptions may emit proof obligations about them. These are *consistency guarantees*. An example of this is that any field of a class needs to fit within the specified size of an instance of a class, so

$$(\forall C, \mathit{OFF}, T)(\texttt{jifield}\ C\ \mathit{OFF}\ T) \rightarrow (\mathit{OFF} + 4) \leq (\texttt{size\_of\_instance}\ C)$$

(here, the type is assumed to require four spaces in memory). These consistency guarantees will typically be required in order to prove the soundness theorem. In fact, the extension will emit only the particular proof obligations for the particular instances of the assumptions which it creates; our meta-theorem will require the appropriate universal quantification to deal with the general case.

Finally, these extensions which collect static information may provide some assumptions from an initialization phase of the programs. In the Java policy the trusted assumptions include the locations of the the virtual-method tables and static members. So the soundness theorem must take into account that the assumed memory has in fact been allocated. To handle assumptions about the state of the memory before the program begins, we must specify a *start state* of the machine discussed in Section 2.2.

We collect here the various pieces of the safety policy which must be encoded in order to produce a statement of the soundness theorem:

1. the proof rules of the safety policy;
2. postconditions of trusted functions;
3. the trusted functions as state transitions;
4. trusted assumptions obtained about a given program;
5. consistency guarantees on the trusted assumptions; and
6. a start state.

In the current implementation of the PCC system, only the proof rules are encoded as an easily manipulable formal object (a file in LF format), such that it is easy to translate them into the formal language of the soundness theorem. In order to use our soundness approach as a modular framework where new safety policies can be introduced and one can easily construct the statement of the necessary soundness theorem, we must do more work to create an appropriate encoding of the necessary pieces of a safety policy.

2.2 The Reference Machine

The idea of the soundness theorem is to set up two abstract machines, modeled as state transition systems. The *reference* machine only allows memory reads and writes to memory which has in fact been allocated for the program; memory safety is guaranteed by run-time checks. The *auxiliary* machine only allows memory reads and writes when the appropriate instance of `saferd` or `safewr` is provable, using the proof rules of the safety policy. Soundness then amounts to showing that, among states reachable from the start state of the safety policy, any auxiliary transition can be effected by reference transitions.

Because we trust the core capabilities of VCGen, we do not need to model executions on a real architecture: VCGen is responsible for correctly determining the semantics of real instructions. We can instead model executions in a very simple way, as long as its computational power is equivalent. We assume a type `exp` of expressions to be used for both memory addresses and values. We need `exp` to be equipped with arithmetic operations, but it seems irrelevant what `exp` really is; in our formal development we use the Coq built-in type of natural numbers. We also assume a type `mem` of memory states, together with operations `upd`, `sel`, and `addr`, corresponding to memory update, memory select, and the test of whether an address has been allocated, respectively.

$$\begin{aligned} E &: \texttt{exp} \\ M &: \texttt{mem} \\ (\texttt{upd}\ M\ E_1\ E_2) &: \texttt{mem} \\ (\texttt{sel}\ M\ E) &: \texttt{exp} \\ (\texttt{addr}\ M\ E) &: \texttt{Prop} \end{aligned}$$

To keep the memory model generic we make only the following assumptions about `mem` and these operations.

$$\begin{array}{l} (\forall m : \texttt{mem})(\forall a, e : \texttt{exp})(\texttt{addr}\ m\ a) \to \\ \quad (\texttt{sel}\ (\texttt{upd}\ m\ a\ e)\ a) = e \\ (\forall m : \texttt{mem})(\forall a_1, a_2, e : \texttt{exp})(\texttt{addr}\ m\ a_1) \to \\ \quad \neg(a_1 = a_2) \to \\ \quad (\texttt{sel}\ (\texttt{upd}\ m\ a_1\ e)\ a_2) = (\texttt{sel}\ m\ a_2) \\ (\forall m : \texttt{mem})(\forall a_1, a_2, e : \texttt{exp})(\texttt{addr}\ m\ a_1) \to \\ \quad (\texttt{addr}\ m\ a_2) \to \\ \quad (\texttt{addr}\ (\texttt{upd}\ m\ a_1\ e)\ a_2) \end{array}$$

(We assume here for ease of presentation that all values stored in memory have a uniform size of one word and so there is no danger of aliasing. In our actual formal development we require a slightly more complicated picture of memory.)

The machine state is a quadruple with two registers a and b of type `exp`, a memory of type `mem`, and a history, which is intended as a list of reads and writes performed. The history will allow us to use just the final state of a sequence of state transitions, to determine whether there was any violation of memory-safety over the whole sequence, rather than having to prove the safety of each individual transition.

$$\begin{aligned} s &::= (E_a, E_b, M, H) \\ H &::= \texttt{nil} \mid (\texttt{read\_inst}\ M\ E) :: H \mid (\texttt{write\_inst}\ M\ E_1\ E_2) :: H \end{aligned}$$

Machine execution steps are modeled by a transition system on the machine state. Two features of the system should be noted. First, we ignore control flow. VCGen is responsible for collecting assumptions and emitting proof obligations in order to handle control flow; e.g. if a memory read occurs in a loop, the relevant proof obligation will be generalized to ensure the safety of the read in any particular iteration of the loop. For the purposes of the soundness theorem we will trust that VCGen does its job correctly, and so we are free to concentrate on straight-line code where all loops, branches, and function calls have been unfolded. Thus, in the example just mentioned, we will consider the particular proof obligation for each specific memory read in each iteration of the loop; because of the (here trusted) semantics of VCGen, each such proof obligation follows by instantiation from the general proof obligation actually emitted by VCGen.

Second, we allow a non-deterministic system. In essence, we work with a completely generic program rather than a specific program which dictates the next instruction at each step. We need only consider the safety of a given instruction in a given state.

The transitions includes a number of uninteresting register manipulations, as well as the memory operations: read, write, and allocate. Just as for the functions `upd` and `sel` for reading and writing memory, we define the function `allocate` axiomatically. For a state s and a natural number n, $(\texttt{allocate}\ n\ s) = (a, b, m, h)$ is intended to be a new state where n previously unallocated words in memory, beginning at location a, have been allocated (and initialized to zero), and the memory is otherwise unaltered. If the memory of the old state s is m_s, the axioms defining this are:

$$\begin{gathered} (\forall k)(k < n) \rightarrow (\texttt{addr}\ m\ (a + k)) \\ (\forall k)(k < n) \rightarrow \neg(\texttt{addr}\ m_s\ (a + k)) \\ (\forall e : \texttt{exp})(\texttt{addr}\ m_s\ e) \rightarrow (\texttt{addr}\ m\ e) \\ (\forall e : \texttt{exp})(\texttt{sel}\ m\ e) = (\texttt{sel}\ m_s\ e) \\ (\forall k)(k < n) \rightarrow (\texttt{sel}\ m\ (a + k)) = 0 \end{gathered}$$

The reference machine transitions are as follows. The salient feature by which the reference machine encodes a policy of memory soundness is that the memory read and write transitions are allowed only on allocated memory.

$$
\begin{array}{rll}
(a,b,m,h) \rightarrow_{\texttt{ref}} & (b,b,m,h) & \\
(a,b,m,h) \rightarrow_{\texttt{ref}} & (a,a,m,h) & \\
(a,b,m,h) \rightarrow_{\texttt{ref}} & (e,b,m,h) & \text{for } e : \texttt{exp} \\
(a,b,m,h) \rightarrow_{\texttt{ref}} & (a,e,m,h) & \text{for } e : \texttt{exp} \\
(a,b,m,h) \rightarrow_{\texttt{ref}} & (a+b,b,m,h) & \\
(a,b,m,h) \rightarrow_{\texttt{ref}} & ((\texttt{sel}\ m\ b),b,m,(\texttt{read\_inst}\ m\ b) :: h) & \text{if } (\texttt{addr}\ m\ b) \\
(a,b,m,h) \rightarrow_{\texttt{ref}} & (a,b,(\texttt{upd}\ m\ a\ b),(\texttt{write\_inst}\ m\ a\ b) :: h) & \text{if } (\texttt{addr}\ m\ a) \\
s \rightarrow_{\texttt{ref}} & (\texttt{allocate}\ n\ s) & \text{for } n \in \mathbb{N}
\end{array}
$$

2.3 The Auxiliary Machine

Now we need to show how to create a machine which reflects a particular safety policy. On this *auxiliary machine*, memory accesses are controlled by the provability of appropriate instances of `saferd` and `safewr`.

We have seen that certain hypotheses result from calls to the trusted functions rather than from the proof rules directly. To handle this, we define the states of the auxiliary machine by extending the machine state (a, b, m, h) with an additional *pseudo-state*; the pseudo-state encodes the information about which hypotheses are available as a result of calls to trusted functions. Consider again the box safety policy; in this instance, an appropriate pseudo-state consists of a list of allocated addresses, as well as the memory resulting from the most recent allocation. The auxiliary state for the box safety policy is thus given by:

$$
\begin{aligned}
s_{\texttt{aux}} &::= ((E_a, E_b, M, H), P) \\
A &::= \texttt{nil} \mid E :: A \\
P &::= (A, M)
\end{aligned}
$$

Now we take each predicate introduced in the proof rules of the safety policy, including `saferd` and `safewr`, and formalize it as an inductive type parameterized by the auxiliary state. The inductive constructors include all of the proof rules given (in which the state plays no role), as well as additional constructors for the postconditions of trusted functions, which examine the state to determine if the predicate holds in that state. In the example of the box safety policy, `ofType` will be an inductive predicate taking a state, an expression, and a type, with constructors including

$$\frac{(\texttt{ofType}\ s\ E\ \texttt{box}) \quad (\texttt{even}\ E)}{(\texttt{ofType}\ s\ E\ \texttt{ptr\_box})}$$

and similar versions of all the other proof rules; ofType will also have an additional constructor (here, let A be the first component of the pseudo-state of s):

$$\frac{E \in A}{(\texttt{ofType}\ s\ E\ \texttt{ptr\_box})}$$

This constructor reflects that in states s where an address E has been provided by the trusted allocator function, the extra hypothesis ($\texttt{ofType}\ s\ E\ \texttt{ptr\_box}$) will be available. Similarly, ofmem will have an additional constructor such that ($\texttt{ofmem}\ s\ M$) holds when M is the second component of the pseudo-state of s.

Thus, the predicates from the proof rules are formalized as inductive types, in such a way that the postconditions of trusted functions are accounted for. In a safety policy with trusted assumptions, such assumptions are formalized as parameters: we assume there is such a predicate, without making any assumption on when the predicate holds. This reflects the fact that we consider a generic program; when the trusted assumptions hold is program-specific. The consistency guarantees, which restrict what combinations of trusted assumptions are possible for legal programs, are formalized as axioms.

Finally we can describe the auxiliary transitions. We include all the register manipulations, which do not alter the pseudo-state; memory read and memory write also do not change the pseudo-state, but are restricted to be allowed only when saferd or safewr hold in the appropriate state. We forbid the general allocation, since type-safety policies will typically define their own typed allocators; and we add transitions for each trusted function. Each trusted function will need to be encoded as a state transition on the machine state; we also require the appropriate change in the pseudo-state. For a trusted function f, where we consider f itself to take reference states to reference states, let $f_{\texttt{pseudo}}$ be the corresponding operation taking the auxiliary state to the new pseudo-state.

$$\begin{array}{rll}
((a,b,m,h),p) \rightarrow_{\texttt{aux}} & ((b,b,m,h),p) & \\
((a,b,m,h),p) \rightarrow_{\texttt{aux}} & ((a,a,m,h),p) & \\
((a,b,m,h),p) \rightarrow_{\texttt{aux}} & ((e,b,m,h),p) & \text{for } e : \texttt{exp} \\
((a,b,m,h),p) \rightarrow_{\texttt{aux}} & ((a,e,m,h),p) & \text{for } e : \texttt{exp} \\
((a,b,m,h),p) \rightarrow_{\texttt{aux}} & ((a+b,b,m,h),p) & \\
((a,b,m,h),p) \rightarrow_{\texttt{aux}} & (((\texttt{sel}\ m\ b),b,m,(\texttt{read\_inst}\ m\ b) :: h),p) & \\
 & & \text{if } (\texttt{saferd}\ s\ m\ b) \\
((a,b,m,h),p) \rightarrow_{\texttt{aux}} & ((a,b,(\texttt{upd}\ m\ a\ b),(\texttt{write\_inst}\ m\ a\ b) :: h),p) & \\
 & & \text{if } (\texttt{safewr}\ s\ m\ a\ b) \\
(s,p) \rightarrow_{\texttt{aux}} & ((f\ s),(f_{\texttt{pseudo}}\ (s,p))) & \text{for trusted function } f
\end{array}$$

In the example of the box safety policy, the trusted allocator function (call it alloc_box) corresponds to the machine transition

$$(\texttt{alloc\_box}\ s) = (\texttt{allocate}\ 1\ s).$$

The effect on the pseudo-state is to add the newly allocated space (i.e. the a register of the new machine state) to the list of allocated addresses, and to replace the last-allocated memory with the newly allocated memory. In other words, if $(\texttt{alloc\_box}\ s) = (a, b, m, h)$, then

$$(\texttt{alloc\_box}_{\texttt{pseudo}}\ (s, (\mathit{addrs}, \mathit{last})) = (a :: \mathit{addrs}, m).$$

At this point in time, our handling of trusted functions is still at a prototype stage. It remains to be worked out, in the general case, exactly how to determine the appropriate pseudo-state, how the various functions alter the pseudo-state, and what the additional inductive constructors for the predicates must be.

2.4 Stating the Soundness Theorem

Now we can state the theorem which, for a given safety policy, will express soundness relative to memory safety. Let $s_{\texttt{start}}$ refer to the start state of the safety policy.

Theorem 1. *There is a pseudo-state $p_{\texttt{start}}$ such that the following holds: for any auxiliary state (s, p) reachable from $(s_{\texttt{start}}, p_{\texttt{start}})$, if $(s, p) \rightarrow_{\texttt{aux}} (s', p')$, then $s \rightarrow^{*}_{\texttt{ref}} s'$.*

In other words, the reference system can safely emulate the machine execution of any auxiliary transition. Because reads and writes are recorded in the machine history, all reads and writes allowed by the auxiliary transition would also be allowed in the reference system; in particular, if `saferd` or `safewr` is provable at a certain state, then the memory to be read or written has been allocated, so can, in fact, be safely read or written according to the reference system as well.

The idea of the general framework is that a code producer can provide a safety policy, appropriately encoded, and the code consumer will produce a formal statement of this theorem, together with all the necessary formal definitions corresponding to the reference and auxiliary machines, as well as the appropriate formalizations of the state-relativized proof rules. The code producer is then obligated to provide a formal proof of the theorem in order for the consumer to use the provided safety policy.

3 A Safety Policy for Java

We have applied the framework provided above to a PCC safety policy for Java, which is used for certifying that native machine code programs compiled from Java are indeed well-typed, according to the Java type system, and use properly various run-time mechanisms such as dynamic dispatch and exception handling. This policy is used in the system described in [3], and has been shown to scale to large programs. We want to be sure that it guarantees memory safety. As indicated we will not consider here the various extensions to VCGen (such as handling of dynamic dispatch), and prove only the soundness of the proof rules of the type system.

The soundness proof first requires generating the appropriate auxiliary machine, as was done with the "toy" safety policy above; then the reference and auxiliary machines are formalized in Coq, at which point it is easy to state the soundness theorem formally. Due to space limitations, here we can only very briefly introduce the general parameters of the safety policy and the soundness proof. The full Coq development, specifying the theorem as well as the proof, is available at `http://www.math.berkeley.edu/~schneck/javacade/`.

The proof rules of the safety policy introduce a set `type` along with constructors: `jint`, `jbool`, etc. for primitive types, (`jarray` T) for arrays of type T, (`jinstof` C) for instances of class (or interface) C. We also have types that do not correspond to source-level types but are instead necessary to describe implementation details: for instance, (`jvirtab` C) is the type of a virtual-method table for class C, and (`jimplof` SIG) is the type of implementations of functions with signature SIG.

The proof rules also define auxiliary predicates: (`ofType` E T) to express that E has type T; (`ptr` E `ro` T) and (`ptr` E `rw` T) to express that E is a (read-only or read-write, respectively) pointer to an element of type T; and (`ofmem` M) to express that the contents of the memory M is well-typed.

The Java safety policy provides certain trusted assumptions, declaring for each class provided either by the program or the trusted run-time system, the name of the class, the name of the superclass and that of the implemented interfaces, the layout of the class instances and that of the virtual-method table. There are consistency guarantees restricting these trusted assumptions; an example is that any field of a class needs to fit within the specified size of an instance of a class, so

$$(\forall C, OFF, T)(\texttt{jifield}\ C\ OFF\ T) \rightarrow (OFF + 4) \leq (\texttt{size\_of\_instance}\ C)$$

where (`jifield` C OFF T) is intended to mean that class C has a field of type T at offset OFF, and (`size_of_instance` C) is the memory space required to allocate an instance of class C.

The trusted assumptions include data from a trusted initialization phase of the program; for instance, the function (`loc_of_virtab` C) returns the location in memory where the virtual-method table for class C is allocated. The safety policy thus assumes a start state where the virtual-method tables and static class members have been allocated.

Finally, the safety policy specifies trusted allocation functions for objects, arrays, and primitive types. In the cases of objects and arrays, the functions also provide some very basic initialization, such as writing the location of the virtual-method table at the appropriate offset. The postcondition of an allocation for a type T is that the returned memory address RES satisfies (`ofType` RES T).

In order to formalize the auxiliary machine, we need an appropriate pseudo-state. In this case we use a *memory-type representation*, which is a list of memory addresses and associated types. The representation indicates which types have been allocated at which locations, and is used to produce the instances of `ofType` and `ptr` which follow from the postconditions of the allocations. The pseudo-state also includes the state of the memory after the most recent allocation, used to produce the correct instances of `ofmem`.

$$R ::= \texttt{nil} \mid (E, T) :: R$$
$$P ::= (R, M)$$

Then we can formalize `ofType`, `ptr`, and `ofmem` as mutually inductive predicates parameterized by the auxiliary state; as in the example of the box safety policy, the inductive constructors incorporate all the proof rules given by the safety policy (without considering the state parameter), as well as additional constructors to reflect which postconditions of the trusted allocators will be available. The predicates `saferd` and `safewr` are also specified inductively, with just one constructor each:[5]

$$(\texttt{ptr}\ s\ E\ \mathit{RWFLAG}\ T) \rightarrow (\texttt{saferd}\ s\ E)$$
$$(\texttt{ptr}\ s\ E_1\ \texttt{rw}\ T) \rightarrow (\texttt{ofType}\ s\ E_2\ T) \rightarrow (\texttt{safewr}\ s\ E_1\ E_2)$$

In outline, the soundness proof runs as follows. We develop an invariant on (auxiliary system) states, such that the invariant implies that whenever the `saferd` or `safewr` predicates hold, the appropriate memory has in fact been allocated. From this it follows that on all states satisfying the invariant, auxiliary transitions can be emulated by reference transitions. Next we establish that all auxiliary transitions preserve the invariant, and we define the start state and show that it satisfies the invariant. This entails that all reachable states satisfy the invariant, and the proof is complete. For more information, see the Coq development at `http://www.math.berkeley.edu/~schneck/javacade/`.

4 Aspects of Formalization

For a formalization tool we use the Coq system [4]. Coq has two main advantages over other choices. One is that Coq produces a proof object (a lambda-term in a typed lambda-calculus) for all proven theorems. This is vital for any PCC application, because the proof-checker is part of the trusted base, and so it is important not to require the full power of a complicated theorem prover just to check that a proof is correct. When the framework is sufficiently generalized to allow code producers to submit their own safety-policies, these proof objects will also be submitted, as the guarantees that the submitted policies ensure memory safety.

The second main advantage of Coq, and the one that makes it particularly suited to this application, is its strong handling of inductively defined types and predicates. The soundness proof requires defining typing predicates inductively,

[5] This is slightly simplified; the actual formal development has to take into account that the various types require different sizes in memory.

as derivations via inference rules, and then making arguments such as induction over the structure of the derivation, or inversion based on the form of the typing predicate (i.e., noting which inference rules allow a result of the given form, and then arguing that the hypotheses of one of those rules must hold). All of this is extremely easy in Coq.

There is, of course, some tension between these two advantages: Coq is far from a minimal formalism, having a rich type system (including inductive types as a primitive notion), and this makes its proof checker more complicated. It seems to us that Coq strikes the appropriate balance.

Coq's automation facility was quite useful. It uses a Prolog-like technique to try to prove the goal from the hypotheses and user-supplied hints. Unfortunately, only goal-directed hints are accepted; it many cases it would have been helpful to have a hint of the form, "if there is a hypothesis of a certain inductively defined predicate, proceed by performing inversion on that hypothesis". Also, reasoning based on equality seemed tricky to automatize. For these reasons, there were many steps of the proof that would be considered obvious in an informal presentation, but were not comfortably handled automatically by Coq. A large amount of the work of formalization was exploring the limits of automation.

5 Comparison with Foundational Proof-Carrying Code

Appel, Felty, and others have introduced a variant PCC framework which they call *foundational proof-carrying code* (FPCC) [2,1]. Instead of having a trusted VCGen to generate the theorem whose proof guarantees safety, foundational PCC defines, in some foundational logic, the semantics of machine instructions and when instructions are considered safe. The safety theorem is then directly expressible in the logic: simply that when the integers which compose the program are loaded into memory and the machine's program counter is set to the beginning of the program, the machine will never reach a state of attempting an unsafe instruction.

FPCC certainly requires a very small set of trusted components. One must still trust a proof checker, of course. There must indeed be code which, analogous to VCGen, looks at a program and produces the theorem, a proof of which witnesses the safety of the given program; however this code must do very little more than correctly produce the list of integers which constitutes the given program. The decoding of those integers into machine instructions is given as a set of logical definitions, rather than as a program which performs the decoding. (It is unclear whether it is actually *easier* to trust that a large set of logical definitions correctly specifies the decoding, than to trust a program which does the decoding; however it is surely not *harder* to trust.) Finally, one must trust logical axioms which encode the semantics of the machine instructions and the safety policy: both the effect of executing the instruction on the machine state, and what conditions must hold for the execution of a given instruction to be considered safe.

In general, it seems that a paradigm essentially similar to VCGen will be used to construct the FPCC proof of safety [6]. That is to say, the integers which constitute the program will be decoded into a list of machine language instructions; local invariants will be constructed for various points in the program in such a way that whenever an invariant holds, progress can be made in the execution of the program, and any other local invariant next reached will still hold; and finally, the local invariant corresponding to the initial state of the program is proven. In VCGen-based PCC, this initial invariant *is* the theorem that guarantees safety, and its proof is all that is transmitted to the code receiver. In foundational PCC, on the other hand, everything else that VCGen does, and the proof that it does it correctly, is also embedded in the proof sent to the code receiver.

An essential difference between FPCC and the work described in this paper is the approach to *scalability*. Current implementations of VCGen-based PCC can handle programs of half a million lines of Java code; making this possible required certain engineering innovations (see e.g. [9]) which may be much more difficult to adapt to FPCC, which necessarily produces more complicated proofs. Clearly, scalability is vitally important to the success of a PCC system. We hope to retain the engineering work which allows the scalability of current implementations, while adding support for greater security and extensibility bit by bit. Taking the approach of FPCC, one starts with a maximally secure and flexible PCC system, and gradually scales to larger and larger examples. This may be a very slow and difficult process.

It is also worth pointing out a difference in our approach to formal type-safety proofs. Appel and Felty in [2] advocate a semantic approach: typing judgments are assigned a semantic truth-value relative to the state, such that typing rules are to be proven as lemmas, and the safety of a well-typed machine state follows immediately from the semantic definition of well-typedness. In contrast we have found that the syntactic approach (where we work directly with the inductive definitions of derivations of typing judgments) to be successful, and almost certainly conceptually simpler. In this our work bears some similarity to the work of Hamid et al. [5], who also aim to develop a full-fledged FPCC system, but advocate the syntactic approach. We note that this group also uses the Coq system, which has very strong handling of inductive types.

6 Conclusions and Future Directions

We have created the prototype of a general framework for stating a soundness theorem for the proof rules of PCC safety policies. Using the framework, we have produced a machine-checkable soundness proof for the typing axioms of a real safety policy for PCC. This soundness proof increases the security of PCC by removing from the trusted base one of the most specific, and thus least well-tested, parts of the system. This is done without giving up trust in more well-tested parts of the system; and there is no reduction of the ability of PCC to scale to large programs. The potentially general nature of the framework allows a more flexible PCC where new safety policies can be introduced by any code producer.

There are still certain issues to be resolved. An important one is the handling of trusted functions, which is still very specific to the Java safety policy. In order to generalize the framework to axiom systems where the trusted functions have different postconditions, we need to specify precisely how the trusted functions are to affect the pseudo-state, and what the corresponding clauses in the inductive definitions of the predicates must be. In order to generalize to different trusted functions altogether, we must specify how the machine behavior of the trusted functions is to be translated onto our abstract machine state. More generally, we need to make clear exactly what is involved in specifying a safety policy as a formal object.

Also, our framework so far is limited to proving the soundness of the proof rules, which is only a part of a typical safety policy. The rest of the policy is implemented as extensions to VCGen. In the case of the Java module, VCGen extensions handle discovering the trusted assumptions and requiring proofs of the consistency guarantees as needed, including handling of the initialization phase of the program where the virtual-method tables are allocated; and they handle novel (to VCGen) programming constructs such as dynamic dispatch, which the core VCGen is unable to deal with. An important step in future research will be to find an approach to proving the soundness of such extensions.

References

1. Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.
2. Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, January 2000.
3. Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000.
4. Coq Development Team. The Coq proof assistant reference manual, version 7.2. January 2002.
5. Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. Submitted for publication, January 2002.
6. Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *Proceedings of the 17th International Conference on Automated Deduction*, pages 7–24. Springer-Verlag, June 2000.
7. George C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, January 1997.
8. George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Also available as CMU-CS-98-154.
9. George C. Necula. A scalable architecture for proof-carrying code. In *The 5th International Symposium of Functional and Logic Programming*, pages 21–39, March 2001.