Published in “Software Security – Theories and Systems”.
Proc. of International Software Security Symposium, 2002, pages 283–298.

Proof-Carrying Code with Untrusted Proof Rules

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

[1] Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
necula@cs.berkeley.edu

[2] Group in Logic and the Methodology of Science
University of California, Berkeley
schneck@math.berkeley.edu

Abstract. Proof-carrying code (PCC) allows a code producer to associate to a program a machine-checkable proof of its safety. In traditional implementations of PCC the producer negotiates beforehand, and in an unspecified way, with the consumer the permission to prove safety in whatever high-level way it chooses. In practice this has meant that high-level rules for type safety have been hard-wired into the system as part of the trusted code base. This limits the security and flexibility of the PCC system.

In this paper, we exhibit an approach to removing the safety proof rules from the trusted base, with a technique by which the producer can convince the consumer that a given set of high-level safety rules enforce a strong global invariant that entails the trusted low-level memory safety policy.

1 Introduction

Proof-carrying code (PCC) [Nec97] 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.

The system described by Colby et al. [CLN+00] was engineered to scale to large programs (e.g. half a million lines of code) and to realistic safety policies (e.g. a type-safety policy for native machine code compiled from Java [CLN+00]) with a relatively modest investment.

⋆ This research was supported in part by National Science Foundation Career Grant No. CCR-9875171, ITR Grants No. CCR-0085949, No. CCR-0081588, and No. INT98-15731, Air Force contract no. F33615-00-C-1693, and gifts from 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 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 verifier to produce a set of verification conditions (VC), whose validity entails the safety of the code. The verifier consists of two components: an instruction decoder that is responsible for interpreting the semantics of individual instructions, and a verification-condition generator (VCGen) that is responsible for handling the control-flow aspects 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.

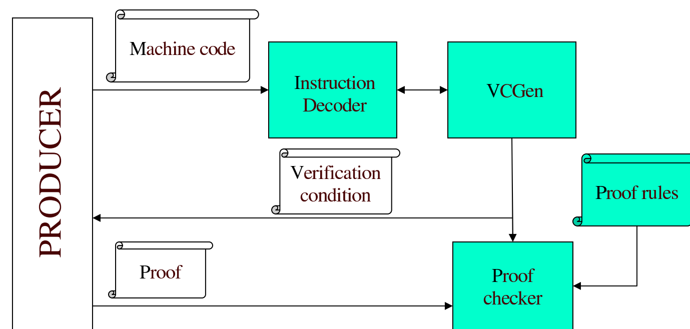

Fig. 1. The structure of a proof-carrying code system showing a dialogue between the code producer (on the left) and the code consumer (comprised of the trusted elements shown shaded).

Thus, in addition to the proof checker, the TCB for this system includes a decoder, 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 with a smaller 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, and change 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. The strategy we adopt in this paper is to use heavy-weight tools such as theorem provers for the purpose of certifying

the set of proof rules used in a PCC system, while trusting the implementation of the core system. In future work we will consider mechanisms for moving even components of the VCGen system out of the TCB.

In this paper we introduce a framework in which the proof rules can be formally proven sound (in the sense that they can be used to ensure safety). This increases the security of a PCC system, because it removes the proof rules from the trusted code base, and also increases its flexibility, because code producers would be able to supply their own safety policies without compromising security. 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 framework in which the receiver of a set of rules can state and verify a proof of their soundness? And furthermore, what is the reference proof system against which the new rules are being judged?

The supplied framework shows how 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 generated automatically (e.g. by a certifying compiler), at this point the soundness proofs for safety policies still need to be generated interactively. This is reasonable, as a single safety policy is expected to be used with a large number of individual programs. Also note that our framework ensures only memory safety. Safety policies are often expected to handle more than memory safety: for instance, a type-safety policy may 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 does not accept a new safety policy unless it can at least guarantee memory safety, regardless of what else it promises to do.

In a previous paper [NS02], we first introduced a method to prove a set of safety-policy proof rules sound with respect to a reference policy of memory safety; in that framework we used the specification language of the Coq [Coq02] system to prove the soundness of a set of typing rules for native machine code compiled from Java. This paper introduces a different method to accomplish the same tasks. The new method allows for a more precise description of how the new safety policy is integrated into the PCC system; additionally, we feel that the new method can be extended more easily with plans for removing parts of the VCGen itself from the trusted code base. Although the framework has changed, it seems that much of the formal proof construction done in the older framework can be easily adapted into this new setting.

In order to provide a concrete example for how our framework operates we discuss first an example of a set of safety policy rules. Then, in Section 4, we introduce the method used by the PCC system to enforce safety, and make precise

the reference memory-safety policy. In Section 5 we discuss how to incorporate a new custom safety policy, and what is required to show that the custom safety policy is at least as strong as the reference policy; we prove the soundness of the original motivating example in Section 6. Finally we discuss related work and conclude.

The results of this paper (including minor details which have been omitted due to reasons of space or presentation) have been formalized in the Coq [Coq02] system; this development is available on the web at the following URL: http://www.cs.berkeley.edu/~necula/ISSS02 .

2 An Example

Consider a code receiver that sets aside an area of accessible memory for the use of the untrusted code and wishes to ensure that the untrusted code accesses memory only within this area. We are going to refer to addresses that fall in the accessible area as valid addresses.

One producer of untrusted code chooses to use the accessible memory area to store *lists* that use the following representation invariant: (1) the value 1 is a list (the empty list), (2) a valid even address of a memory location containing a list is a list (a non-empty list), and (3) nothing else is a list.[3] We show in Figure 2 one possible fragment of untrusted code that uses this invariant. This code operates on two non-empty lists stored in registers $\mathtt{r}_a$ and $\mathtt{r}_b$; it first truncates the list $\mathtt{r}_a$ to one element and then sets the second element in the list $\mathtt{r}_b$ to point to $\mathtt{r}_a$.

0 $\mathtt{r}_t := 1$
1 $[\mathtt{r}_a] := \mathtt{r}_t$
2 $\mathtt{r}_s := [\mathtt{r}_b]$
3 $\mathtt{branch}\ \mathtt{odd}(\mathtt{r}_s), 5$
4 $[\mathtt{r}_s] := \mathtt{r}_a$
5 $\mathtt{halt}$

Fig. 2. An untrusted code fragment

The notation $[\mathtt{r}_1]$ means dereference of address in $\mathtt{r}_1$ and it is a read if appearing on the right-hand side of the assignment or a write otherwise. Notice that if $\mathtt{r}_a$ and $\mathtt{r}_b$ are aliases then $\mathtt{r}_s$ will be equal to 1 and the write at line 4 is skipped.

We can state informally the conditions that the code receiver wants to hold before this code is executed. We state these conditions on the initial values of registers:

– The value of $\mathtt{r}_a$ is a valid address (write in line 1 is safe),
– The value of $\mathtt{r}_b$ is a valid address (read in line 2 is safe),

[3] Such lists are of limited usefulness because there is no data in the list cells. However, they provide a simple but non-trivial example.

– If $\mathtt{r}_a \neq \mathtt{r}_b$ and the contents of address $\mathtt{r}_b$ is even, then the contents of address $\mathtt{r}_b$ is a valid address (write in line 4 is safe)

But as far as the code producer is concerned these conditions are too low-level. Instead, the code producer would prefer to use the following two simple conditions:

– The value of $\mathtt{r}_a$ is a non-empty list
– The value of $\mathtt{r}_b$ is a non-empty list

These conditions, along with a statement of the representation invariant, ensure that our example is memory safe. Not only have we reduced the number of the conditions to be proved but we have also eliminated the need to reason about memory aliasing: as long as all memory writes preserve the representation invariant we can assume that the result of memory reads satisfies the same invariant, without having to consider all of the possible writes that the read depends on. This is arguably one of the secrets that make type checking such a practical method to ensure memory safety.

In traditional implementations of PCC we have taken the view that the producer negotiates beforehand (and in an unspecified way) with the consumer the permission to prove safety at whatever high-level it chooses. In this paper, we show one possible way in which the producer can actually convince the consumer that a given set of high-level safety rules enforce a strong global invariant that entails the low-level memory safety policy.

In the next section we introduce notation necessary for stating the safety conditions formally, first from the low-level perspective of the code consumer and then from the high-level perspective of the code producer.

3 Preliminaries

We work in an underlying logic containing a type `val` to be used for values and addresses, and a type `state` of machine states. To keep our approach general we leave the machine state largely unspecified; however for the sake of examples we use a state that consists of various registers of type `val` together with a pseudo-register $\mathtt{r}_M$ of type `mem` (denoting the state of memory), which comes equipped with functions `upd` and `sel` for memory update and select.

A *symbolic expression* is a function of type $\mathtt{state} \to \mathtt{val}$, such that when given some values for the registers we obtain its value. Thus if e is a symbolic expression and ρ a state, then "$e\ \rho$" is the value of e in state ρ. For notational convenience we also write symbolic expressions as expressions involving variable names. For example, we write "$\mathtt{r}_1 + \mathtt{r}_2$" for the function that given a state yields the sum of the values of registers $\mathtt{r}_1$ and $\mathtt{r}_2$ in that state. Consequently we write "$\mathtt{r}_1\ \rho$" for the value of register $\mathtt{r}_1$ in state ρ.

We also use *symbolic predicates* (or simply predicates), which are functions of type $\mathtt{state} \to \mathtt{Prop}$, where `Prop` is the type of propositions; and we write $A\ \rho$ to say that the predicate A holds in state ρ.

For convenience, we overload the arithmetic operators and the boolean connectives to work on symbolic expressions and predicates. For example, the notation $\exists x.\mathtt{r}_1 + \mathtt{r}_2 \geq x$ denotes a symbolic predicate whose de-sugared form is $\lambda\rho.\exists x.\mathtt{r}_1\ \rho + \mathtt{r}_2\ \rho \geq x$. We write $\rho[\mathtt{r}_i \mapsto v]$ for the state ρ altered by setting the value of $\mathtt{r}_i$ to v. We write $A[\mathtt{r}_i \mapsto v]$ to denote the symbolic predicate obtained from A by replacing references to $\mathtt{r}_i$ by v, namely $\lambda\rho.\big(A\ (\rho[\mathtt{r}_i \mapsto v])\big)$. Finally, in order to simplify the handling of assignment, we use $\mathtt{r}_i :=_x e$ as shorthand for $\mathtt{r}_i = e[\mathtt{r}_i \mapsto x]$. With this notation we can say, for example, that after an assignment $\mathtt{r}_1 := \mathtt{r}_1 + \mathtt{r}_1$ the predicate $\exists x.\mathtt{r}_1 :=_x \mathtt{r}_1 + \mathtt{r}_1$ holds, where x is the prior value of $\mathtt{r}_1$.

VCGen works with local invariants, which are a subtype of symbolic predicates (type `state` $\rightarrow$ `Prop`). For the purposes of this paper, we assume that all local invariants have the form $(\mathtt{r}_{\mathtt{pc}} = n) \wedge A$, where n is a literal of type `val` denoting the value of the program counter, and A does not depend on the program counter; that is, each local invariant specifies a literal program counter, and (possibly) other facts about the state except the program counter. We use the notation (n, A) to represent this local invariant; we use the type `locinv` for local invariants.

4 A Formalization of Memory Safety

In this section we are going to describe the precise mechanism that a code consumer can use to enforce memory safety. First, we assume that the logic contains a predicate `addr` to indicate when a value is a valid address: $(\mathtt{addr}\ \rho\ E)$ means that E is a valid address in state ρ. The exact definition of `addr` depends on the particular machine and safety policy. The dependence on the machine state is useful in the presence of dynamic allocation, where ρ contains the allocation state; as with other symbolic predicates, we often suppress the dependence on ρ for convenience.

The Decoder. The code consumer uses a decoder to interpret individual instructions. In this paper we will define a couple of different decoders. The code consumer specifies a trusted *reference* decoder, and the code producer introduces the untrusted *custom* decoder for its custom safety policy. The result of any decoder for an individual instruction consists of two elements:

- The safety condition, which is a predicate that holds for states in which execution of the instruction meets the safety policy, and
- A set of possible machine states resulting from the execution of the instruction.

For example, the safety condition for memory access instructions specifies that the addresses involved are valid. For all but the branching instructions the decoder returns a single result state.

For convenience we state the decoder as a function of our type `locinv` of local invariants:

$$\mathtt{decode}_{\mathtt{ref}} : \mathtt{locinv} \rightarrow (\mathtt{state} \rightarrow \mathtt{Prop}) \times (\mathtt{set}\ \mathtt{locinv}).$$

The decoder takes a local invariant (p, A) and produces a pair $(P, \mathcal{D})$; P is a local safety condition, which must hold for a machine state satisfying (p, A) to make safe progress, while $\mathcal{D}$ is a list of local invariants, one of which is guaranteed to hold of the new machine state if safe progress is made.

A possible definition of the decoder for the language used in the example from Figure 2 is shown in the Figure 3. We use $\mathtt{r}_1$, $\mathtt{r}_2$, and $\mathtt{r}_3$ as meta-variables which need not be distinct. Notice that for the assignment instructions the resulting invariants are constructed using strongest postconditions.

Input state (p, A) with $Instr(p) = \ldots$	Local safety condition	Result states
$\mathtt{r}_1 := n$	`True`	$(p+1, \mathtt{r}_1 = n \wedge \exists x. A[\mathtt{r}_1 \mapsto x])$
$\mathtt{r}_1 := \mathtt{r}_2 + \mathtt{r}_3$	`True`	$(p+1, \exists x. \mathtt{r}_1 :=_x (\mathtt{r}_2 + \mathtt{r}_3) \wedge A[\mathtt{r}_1 \mapsto x])$
$\mathtt{r}_1 := [\mathtt{r}_2]$	$\lambda\rho.\mathtt{addr}\ \rho\ (\mathtt{r}_2\ \rho)$	$(p+1, \exists x. \mathtt{r}_1 :=_x (\mathtt{sel}\ \mathtt{r}_M\ \mathtt{r}_2) \wedge A[\mathtt{r}_1 \mapsto x])$
$[\mathtt{r}_1] := \mathtt{r}_2$	$\lambda\rho.\mathtt{addr}\ \rho\ (\mathtt{r}_1\ \rho)$	$(p+1, \exists m. \mathtt{r}_M = (\mathtt{upd}\ m\ \mathtt{r}_1\ \mathtt{r}_2) \wedge A[\mathtt{r}_M \mapsto m])$
`branch` $Cond, n$	`True`	$(n, A \wedge Cond), (p+1, A \wedge \neg Cond)$
`halt`	`True`	none

Fig. 3. The definition of the reference decoder

The decoder interprets the semantics of an individual instruction. It is the responsibility of the VCGen to reflect in the final verification condition all of the local safe-progress conditions identified by the decoder. It is also the responsibility of VCGen to consider the safety of all of the "next" states specified by the decoder. To do this properly VCGen must recognize loops (possibly helped by some required annotations in the untrusted code that specify loop invariants) and must ensure that the decoder is invoked only once for each loop body.

Details of a possible definition of VCGen can be found in [Nec97,Nec98]. Here by way of example, we show in Figure 4 the sequence of invocations of the decoder that VCGen makes for the untrusted code from Figure 2, where P_0 is the local invariant at the beginning of the code fragment (the precondition of the code fragment). We assume that $\mathtt{r}_a$, $\mathtt{r}_b$, $\mathtt{r}_s$, and $\mathtt{r}_t$ are distinct registers. Notice how each of the states mentioned in the column labeled "Result states" is eventually scanned; this is why the instruction at program counter 5 is scanned twice: it is in the result of the instructions at program counters 3 and 4.

To assemble the final verification condition, VCGen states that for each row with a non-trivial safety condition in Figure 4 the local invariant (shown in the column labeled A) implies the local safety condition. By rewriting the equalities generated by the assignments, we find that the resulting verification condition

p	A	Local safety condition	Result states
0	P_0	`True`	$\{(1, \exists x_t.P_1)\}$ where $P_1 \equiv \mathtt{r}_t = 1 \land P_0[\mathtt{r}_t \mapsto x_t]$
1	P_1	$\mathtt{addr}\ \mathtt{r}_a$	$\{(2, \exists x_t \exists x_M.P_2)\}$ where $P_2 \equiv \mathtt{r}_M = (\mathtt{upd}\ x_M\ \mathtt{r}_a\ 1) \land \mathtt{r}_t = 1 \land$ $P_0[\mathtt{r}_t \mapsto x_t, \mathtt{r}_M \mapsto x_M]$
2	P_2	$\mathtt{addr}\ \mathtt{r}_b$	$\{(3, \exists x_t \exists x_M \exists x_s.P_3)\}$ where $P_3 \equiv \mathtt{r}_s = (\mathtt{sel}\ (\mathtt{upd}\ x_M\ \mathtt{r}_a\ 1)\ \mathtt{r}_b) \land$ $\mathtt{r}_M = (\mathtt{upd}\ x_M\ \mathtt{r}_a\ 1) \land \mathtt{r}_t = 1 \land$ $P_0[\mathtt{r}_t \mapsto x_t, \mathtt{r}_M \mapsto x_M, \mathtt{r}_s \mapsto x_s]$
3	P_3	`True`	$\{(4, \exists x_t \exists x_M \exists x_s.P_4), (5, \exists x_t \exists x_M \exists x_s.P_5)\}$ where $P_4 \equiv P_3 \land \mathtt{even}\ (\mathtt{sel}\ (\mathtt{upd}\ x_M\ \mathtt{r}_a\ 1)\ \mathtt{r}_b)$ and $P_5 \equiv P_3 \land \mathtt{odd}\ (\mathtt{sel}\ (\mathtt{upd}\ x_M\ \mathtt{r}_a\ 1)\ \mathtt{r}_b)$
4	P_4	$\mathtt{addr}\ (\mathtt{sel}\ (\mathtt{upd}\ x_M\ \mathtt{r}_a\ 1)\ \mathtt{r}_b)$	$\{(5, \exists x_t \exists x_M \exists x_s \exists y_M.P_5')\}$ where $P_5' \equiv \mathtt{r}_M = (\mathtt{upd}\ y_M\ \mathtt{r}_s\ \mathtt{r}_a) \land P_4[\mathtt{r}_M \mapsto y_M]$
5	P_5	`True`	$\{\}$
5	P_5'	`True`	$\{\}$

Fig. 4. The operation of the reference decoder on the example code.

is essentially:

$$P_0 \Rightarrow \mathtt{addr}\ \mathtt{r}_a \land \mathtt{addr}\ \mathtt{r}_b \land \Big(\mathtt{even}\ \big(\mathtt{sel}\ (\mathtt{upd}\ \mathtt{r}_M\ \mathtt{r}_a\ 1)\ \mathtt{r}_b\big) \Rightarrow \mathtt{addr}\ \big(\mathtt{sel}\ (\mathtt{upd}\ \mathtt{r}_M\ \mathtt{r}_a\ 1)\ \mathtt{r}_b\big)\Big)$$

If the code producer is willing to prove such safety predicates then we have an effective PCC system. However, one can see that the size of the predicates grows exponentially with the size of the program, and that proving them requires very low-level reasoning about aliasing relationships (which is necessary for reasoning about constructs such as "$\mathtt{sel}\ (\mathtt{upd}\ \mathtt{r}_M\ \mathtt{r}_a\ 1)\ \mathtt{r}_b$"). In the next section we show how a code producer can on one hand construct these proofs at a higher level, and on the other hand convince the code consumer to accept them instead of the low-level proofs.

First, we must state precisely the property that we assume to hold of the reference decoder. Let represent the state transition relation of the machine, restricted so that only *safe* transitions are allowed; thus for example memory operations will be executed only if they are memory-safe. We assume that the machine is deterministic; if any transition is possible from a state, only one transition is possible. As usual $^+$ indicates a sequence of one or more transitions. The following correctness property is assumed to hold:

Property 1 (Reference Decoder Correctness). Let C be a local invariant. Then

$$\mathtt{decode}_{\mathtt{ref}}\ C = (P, \mathcal{D})$$

where $P : \texttt{state} \to \texttt{Prop}$ and $\mathcal{D}$ is a set of local invariants, such that

$$\forall \rho : \texttt{state}.(C\ \rho) \wedge (P\ \rho) \Rightarrow \bigvee_{D \in \mathcal{D}} \exists \rho' : \texttt{state}.(\rho^{+} \rho') \wedge (D\ \rho').$$

This means that if the local safety condition holds then the machine can make safe progress to a state that satisfies one of the "next" states identified by the decoder.

5 Custom Safety Policies

In the safety policy enforced by the code consumer, memory operations require the provability of particular instances of `addr`. We shall call this the *reference* safety policy. A code producer may find it easier to prove a stronger property, such as type safety, and prove separately that any program considered safe under this new policy also meets the conditions of the reference policy. For our example with lists, the code producer could add the typing predicates `list` L (meaning L is a list) and `nelist` L (meaning L is a non-empty list). For these predicates, the code producer specifies the proof rules shown in Figure 5.

$$\frac{}{\texttt{list}\ 1} \qquad \frac{\texttt{nelist}\ L}{\texttt{list}\ L} \qquad \frac{\texttt{list}\ L \quad \texttt{even}\ L}{\texttt{nelist}\ L}$$

Fig. 5. Proof rules for a type system with lists

These proof rules do not need to be a complete description of the representation invariant. The code producer can choose to publish only those proof rules that it knows are necessary for proving the verification condition. Observe that the proof rules alone provide no instances of the form `nelist` L. Instead, recall that the VC consists of conjuncts that a local invariant implies a local safety condition; so via these implications the local invariants can provide certain `nelist` assumptions to be used along with the typing rules. For example, we would assume that the code fragment of Figure 2 would be called with the precondition that `nelist` $\texttt{r}_a$ and `nelist` $\texttt{r}_b$.

These rules are not immediately useful for proving the verification conditions that we produced in the previous section. We would very much like to intervene in the process of producing the verification condition in order to produce smaller verification conditions that refer to the newly added predicates. It is apparent from the way in which the verification conditions are created that to accomplish this we must use a different decoder.

The Custom Decoder. The code producer achieves its desired form of the verification condition by changing the instruction decoding rules as shown in Figure 6. There are quite a few differences between this custom decoder and

the reference one. The `addr` address validity safety conditions are replaced with stronger conditions, specifically that all memory reads are from non-empty lists and that all memory writes are storing lists to addresses that are non-empty lists. Also, the custom decoder gives only an approximate description of the next state. The value with which a register is initialized is forgotten, unless that value is 1, indicating the empty list. An addition is always safe but since the code producer knows that it never generates code that mixes lists with arithmetic, it chooses to ignore the actual result of the addition. In memory operations the deliberate loss of information is significant in terms of efficiency. For a memory read the decoder does not specify what value was read but only that it must be a list. And for a memory write the new contents of the memory is kept completely abstract. This makes sense because in a subsequent memory read all the decoder cares about is that a read from a non-empty list yields a list value. This last form of abstraction makes sure that terms such as ($\mathtt{sel}$ ($\mathtt{upd}$ $\ldots$) $\ldots$) do not arise in the verification condition. Finally, the custom decoder abstracts the state following a conditional except in the case of the parity conditional, which it knows is used to test whether a list is a non-empty list in preparation for a memory operation.

Input state (p, A) with $\mathit{Instr}(p) = \ldots$	Local safety condition	Result states
$\mathtt{r}_1 := 1$	True	$(p+1, \mathtt{r}_1 = 1 \wedge \exists x.A[\mathtt{r}_1 \mapsto x])$
$\mathtt{r}_1 := n$	True	$(p+1, \exists x.A[\mathtt{r}_1 \mapsto x])$
$\mathtt{r}_1 := \mathtt{r}_2 + \mathtt{r}_3$	True	$(p+1, \exists x.A[\mathtt{r}_1 \mapsto x])$
$\mathtt{r}_1 := [\mathtt{r}_2]$	$\mathtt{nelist}\,\mathtt{r}_2$	$(p+1, \mathtt{list}\,\mathtt{r}_1 \wedge \exists x.A[\mathtt{r}_1 \mapsto x])$
$[\mathtt{r}_1] := \mathtt{r}_2$	$\mathtt{nelist}\,\mathtt{r}_1 \wedge \mathtt{list}\,\mathtt{r}_2$	$(p+1, \exists x.A[\mathtt{r}_M \mapsto x])$
$\mathtt{branch}\ \mathtt{odd}(\mathtt{r}_1), n$	True	$(n, A), (p+1, A \wedge \mathtt{even}\,\mathtt{r}_1)$
$\mathtt{branch}\ \mathit{Cond}, n$	True	$(n, A), (p+1, A)$
halt	True	none

Fig. 6. The definition of the custom decoder

Consider again the program of Figure 2. With the custom decoder the VCGen produces the following verification condition:

$$P_0 \Rightarrow \mathtt{nelist}\,\mathtt{r}_a \wedge \mathtt{list}\,1 \wedge \mathtt{nelist}\,\mathtt{r}_b \wedge \big(\forall x.\mathtt{list}\,x \wedge \mathtt{even}\,x \Rightarrow \mathtt{nelist}\,x \wedge \mathtt{list}\,\mathtt{r}_a\big)$$

Here, $\mathtt{nelist}\,\mathtt{r}_a \wedge \mathtt{list}\,1$ arises from the write in line 1; $\mathtt{nelist}\,\mathtt{r}_b$ arises from the read in line 2. In the remaining part, the quantified variable x refers to the result of the memory read in line 2; all we know is that, because it is a result of a well-typed memory read, then it is a `list`. We also have $\mathtt{even}\,x$ as a result of the branch, and finally we must show $\mathtt{nelist}\,x \wedge \mathtt{list}\,\mathtt{r}_a$ for the memory write in line 4.

It is easy to see how this predicate can be proved using the typing rules for lists, provided that the precondition of the code ensures that $\mathtt{r}_a$ and $\mathtt{r}_b$

are both non-empty lists. Notice also the similarity between this proof and a typing derivation. This is precisely the situation that the code producer wants to achieve.

The question now is how can the code producer convince the code consumer to use the custom decoder instead of the reference one, and to use the specified typing rules while checking the proof of safety.

6 Checking the Soundness of the Custom Decoder

In this section we describe the steps that the code producer must take to convince the code consumer that the combination of the custom decoder and the custom proof rules is sound with respect to the reference decoder. The overall approach is to prove that there exists a global invariant that is preserved by all instructions whenever their execution is deemed safe by the custom decoder, and furthermore that this global invariant, along with the local safety conditions produced by the custom decoder, implies the local safety conditions of the reference decoder. We use the example of the `list` safety policy, but will show the general case of the statement of the custom decoder theorem that the code producer must provide to the consumer.

First, observe that the code consumer manipulates the predicates `list` and `nelist` (when running the custom decoder) and their associated proof rules (when checking the proof of the *VC*), without actually having a complete definition of these predicates. The proof of the *VC* is parametric in the actual definition of the custom predicates, with only the assumption that they satisfy the proof rules. A proof of the *VC* then applies to *any instantiation of* `list` *and* `nelist` *that satisfies the proof rules.*

To formalize this detail, let `preds` be the type of the tuple of custom predicates used in a particular custom safety policy. (For the example with lists, $\texttt{preds} = (\texttt{val} \rightarrow \texttt{Prop}) \times (\texttt{val} \rightarrow \texttt{Prop})$.) The custom proof rules, the custom decoder and the resulting verification condition (i.e., all elements that mention the custom predicates) are parameterized by an actual instantiation $\psi : \texttt{preds}$ for the custom predicates. In all of these elements we attach the subscript ψ to all occurrences of the custom predicates, to denote their particular instantiation under ψ. (For example, $\texttt{list}_\psi = \texttt{fst}\ \psi$ and $\texttt{nelist}_\psi = \texttt{snd}\ \psi$.) In general, we write A_ψ to refer to the predicate A in which all occurrences of custom predicate symbols use the instantiation ψ.

Next, we can write the set of proof rules as a predicate `Rules` parameterized by an instantiation ψ. In this predicate each of the rules contributes a conjunct. For the example with lists (the rules shown in Figure 5):

$$\begin{aligned}\texttt{Rules}_\psi = &\texttt{list}_\psi\ 1 \wedge \\ &(\forall L. \texttt{nelist}_\psi\ L \Rightarrow \texttt{list}_\psi\ L) \wedge \\ &(\forall L. \texttt{list}_\psi\ L \wedge \texttt{even}\ L \Rightarrow \texttt{nelist}_\psi\ L)\end{aligned}$$

The custom predicates in our example are unary and appear to be independent of the execution state. Yet, their instantiations implicitly refer to the state of the memory since a value can be a non-empty list only in a state in which the memory location it denotes contains a list. In general, the instantiations of the custom predicates depend on the state, and we might need to change the instantiation when the state changes.

The next major element in the proof of soundness of the custom decoder is a generalized form of decoder correctness property that allows for a global invariant:

Property 2 (Generalized Decoder Correctness). Let `preds` be the type of the custom predicates and `Rules` : `preds` $\rightarrow$ `Prop` be the custom proof rules. Let I : `preds` $\rightarrow$ `state` $\rightarrow$ `Prop`, the *global invariant*, be some predicate of states involving the custom predicates. Then we say that a decoder d satisfies the *correctness property with invariant* I if, for any C with $d\,C = (P, \mathcal{D})$, we have

$$\forall \rho : \texttt{state}.\, \forall \psi : \texttt{preds}.\, (\texttt{Rules}_{\psi}) \wedge (I_{\psi}\ \rho) \wedge (C_{\psi}\ \rho) \wedge (P_{\psi}\ \rho) \Rightarrow$$
$$\exists \rho' : \texttt{state}.\, (\rho^{+}\rho') \wedge \bigvee_{E \in \mathcal{D}} \exists \psi' : \texttt{preds}.\, (\texttt{Rules}_{\psi'}) \wedge (I_{\psi'}\ \rho') \wedge (E_{\psi'}\ \rho')$$

This is worth restating: consider a state ρ, and an instantiation ψ of the custom predicates such that ψ obeys the custom proof rules. If ρ satisfies the global invariant, the input local invariant, and the output proof obligation (all instantiated at ψ), then the machine can make safe progress to some new state ρ'. Furthermore, there is a (possibly) new instantiation ψ' of the custom predicates corresponding to the new state, which obeys the custom proof rules, such that ρ' satisfies the global invariant and one of the output local invariants, each instantiated at ψ'.

We use without proof the following fact about the trusted VCGen. VCGen functions correctly—in that the provability of the *VC* implies safety of the program—using any custom predicates, custom proof rules, and custom decoder, as long as there is a global invariant I such that

1. the decoder satisfies the correctness property with invariant I, and
2. there is some instantiation ψ of the custom predicates, obeying the custom proof rules, such that I_{ψ} holds of the initial state of the machine upon execution of the program.

The second fact depends depends on the particular machine and safety policy, and is typically much less interesting, and so we will not consider this requirement in this paper.

We now consider how to go about proving the correctness of a custom decoder. Since we already have access to the reference decoder, which is known to satisfy the correctness property (with invariant `True`), we can use the reference decoder to prove correctness for the custom decoder. In particular, we have the following lemma that allows us to replace all consideration of the transition relation with consideration instead of the reference decoder.

Lemma 1. *Let* I : $\texttt{preds} \rightarrow \texttt{state} \rightarrow \texttt{Prop}$. *The following conditions ensure that* $\texttt{decode}_{\texttt{cus}}$ *satisfies the correctness property with invariant* I*: let* C *be any local invariant (possibly dependent on* $\texttt{preds}$*) and let* ψ *be any instantiation of* $\texttt{preds}$ *that obeys the custom proof rules* $\texttt{Rules}$. *Let* $\texttt{decode}_{\texttt{cus}}\ C = (P, \mathcal{D}_{\texttt{cus}})$*; let* $\texttt{decode}_{\texttt{ref}}\ (I_\psi \wedge C_\psi \wedge P_\psi) = (Q, \mathcal{D}_{\texttt{ref}})$. *The conditions are*

1. $\forall \rho.(I_\psi\ \rho) \wedge (P_\psi\ \rho) \Rightarrow (Q\ \rho)$;
2. $\bigwedge_{D \in \mathcal{D}_{\texttt{ref}}} \forall \rho'.(D\ \rho') \Rightarrow \bigvee_{E \in \mathcal{D}_{\texttt{cus}}} \exists \psi' : \texttt{preds}.\,(\texttt{Rules}_{\psi'}) \wedge (I_{\psi'}\ \rho') \wedge (E_{\psi'}\ \rho')$.

Proof. We are going to show that the custom decoder satisfies the generalized decoder correctness property. Let ρ be any state that satisfies all of I_ψ, C_ψ, and P_ψ. By the first condition, we have that $(Q\ \rho)$, and therefore, by the correctness of the reference decoder, safe progress can be made to a new state ρ' which satisfies one of the local invariants D in $\mathcal{D}_{\texttt{ref}}$. But then the second condition ensures that there is some local invariant E in $\mathcal{D}_{\texttt{cus}}$, and a (possibly) new instantiation ψ' that obeys the custom proof rules, such that ρ' satisfies $E_{\psi'}$ as well as $I_{\psi'}$. This suffices to establish the correctness of the custom decoder. ☐

A decoder can be specified as a table indexed by the instruction kind, as we have seen in previous sections. The code consumer verifies the safety of the custom decoder by applying Lemma 1. Thus, the code producer must supply a global invariant I and a number of proofs, as follows. For each row in the definition of the custom decoder, let P and $\mathcal{D}_{\texttt{cus}}$ be respectively the local safety condition and the resulting states corresponding to an arbitrary local invariant C. Find the row in the definition of the reference decoder that corresponds to the same instruction kind; let Q and $\mathcal{D}_{\texttt{ref}}$ be respectively the local safety condition and the resulting states of the reference decoder for the local invariant $(I_\psi \wedge C_\psi \wedge P_\psi)$, where ψ is some instantiation of the custom predicates that obeys the custom proof rules. Observe that since the reference decoder output is parametric in the input local invariant (except the program counter), we can consider Q and each $D \in \mathcal{D}_{\texttt{ref}}$ to be parameterized by ψ; thus, we write Q_ψ and D_ψ. The following items must be provided:

1. A proof that, for any instantiation ψ such that ψ satisfies the custom proof rules, $I_\psi \wedge P_\psi \Rightarrow Q_\psi$;
2. for each $D \in \mathcal{D}_{\texttt{ref}}$, and for any instantiation ψ obeying the custom proof rules, a new instantiation ψ' obeying the proof rules together with a proof that $D_\psi \Rightarrow I_{\psi'} \wedge E_{\psi'}$, for some $E \in \mathcal{D}_{\texttt{cus}}$.

In the absence of any sort of allocation operation, the second condition will often be established with $\psi' = \psi$ as we shall see.

Once the code consumer receives the above elements and checks the included proofs, it knows that all of the conditions required by Lemma 1 are satisfied and thus the combination of the custom proof rules and the custom decoder is safe to use instead of the reference decoder. In the next section we give concrete examples of these elements for the list-based safety policy.

The Soundness of the Example Safety Policy. First we must introduce an appropriate global invariant I with which to prove the correctness of the custom decoder for the `list` safety policy. We will use

$$(I_\psi\ \rho) \stackrel{def}{=} \forall x.(\mathtt{nelist}_\psi\ x) \Rightarrow \Big((\mathtt{addr}\ \rho\ x) \wedge \big(\mathtt{list}_\psi\ (\mathtt{sel}\ (\mathtt{r}_M\ \rho)\ x)\big)\Big).$$

Thus every non-empty list must be a valid memory address containing a list. For a more complicated safety policy, one can expect some work in determining an invariant which has the correct strength, neither too strong nor too weak, to be preserved. For our example this invariant suffices.

Next we have to provide the necessary proofs for each row in the definition of the custom decoder (shown in Figure 6). We show below only some representative cases.

For an addition, Q is `True`, and thus condition 1 is trivial. For condition 2, there is only one state in $\mathcal{D}_{\mathtt{ref}}$ and also in $\mathcal{D}_{\mathtt{cus}}$, and given an instantiation ψ of the predicates obeying the proof rules, we have to find a new instantiation ψ' obeying the proof rules, such that:

$$(\exists x.\mathtt{r}_1 :=_x (\mathtt{r}_2 + \mathtt{r}_3) \wedge (I_\psi \wedge C_\psi)[\mathtt{r}_1 \mapsto x]) \Rightarrow I_{\psi'} \wedge \exists x.C_{\psi'}[\mathtt{r}_1 \mapsto x]$$

Since the global invariant depends only on the memory, and in particular not on the changed register $\mathtt{r}_1$, we simply choose $\psi' = \psi$ and the required condition follows.

For the case of a memory read we have

$$\begin{aligned} P_\psi &= (\mathtt{nelist}_\psi\ \mathtt{r}_2);\\ \mathcal{D}_{\mathtt{cus}} &= \{(p+1, (\mathtt{list}_\psi\ \mathtt{r}_1) \wedge \exists x.C_\psi[\mathtt{r}_1 \mapsto x])\};\\ Q_\psi &= (\mathtt{addr}\ \rho\ \mathtt{r}_2);\\ \mathcal{D}_{\mathtt{ref}} &= \{(p+1, \exists x.\,\mathtt{r}_1 :=_x (\mathtt{sel}\ \mathtt{r}_M\ \mathtt{r}_2) \wedge (I_\psi \wedge C_\psi \wedge P_\psi)[\mathtt{r}_1 \mapsto x])\}. \end{aligned}$$

Condition 1 ($I_\psi \wedge P_\psi \Rightarrow Q_\psi$) follows from the definition of the global invariant. Condition 2 requires showing that

$$\exists x.\big(\mathtt{r}_1 :=_x (\mathtt{sel}\ \mathtt{r}_M\ \mathtt{r}_2) \wedge (I_\psi \wedge C_\psi \wedge P_\psi)[\mathtt{r}_1 \mapsto x]\big) \Rightarrow \\ I_{\psi'} \wedge (\mathtt{list}_{\psi'}\ \mathtt{r}_1) \wedge \exists x.C_{\psi'}[\mathtt{r}_1 \mapsto x].$$

Since a memory read does not change which values denote lists or non-empty lists, we choose $\psi' = \psi$. This mostly follows as for the addition; to show that $(\mathtt{list}_{\psi'}\ \mathtt{r}_1)$, observe that it is equivalent to $\big(\mathtt{list}_{\psi'}\ (\mathtt{sel}\ \mathtt{r}_M\ \mathtt{r}_2)\big)[\mathtt{r}_1 \mapsto x]$, which follows from $(I_\psi \wedge P_\psi)[\mathtt{r}_1 \mapsto x]$, where x is the value of $\mathtt{r}_1$ prior to the assignment. (The x is only important in the case where $\mathtt{r}_1$ and $\mathtt{r}_2$ are the same register.)

For the case of a memory update, the proof of condition 1 follows in a similar manner. Condition 2 is more interesting, since we must show that the global invariant still holds in the state after the memory update. However, since a

memory update does not change which values denote lists or non-empty lists, we can still choose $\psi' = \psi$. In this case we have:

$$
\begin{aligned}
P_\psi &= (\texttt{nelist}_\psi\ \texttt{r}_1) \wedge (\texttt{list}_\psi\ \texttt{r}_2);\\
\mathcal{D}_{\texttt{cus}} &= \{(p+1, \exists m. C_\psi[\texttt{r}_M \mapsto m])\};\\
Q_\psi &= (\texttt{addr}\ \rho\ \texttt{r}_1);\\
\mathcal{D}_{\texttt{ref}} &= \{(p+1, \exists m. \texttt{r}_M = (\texttt{upd}\ m\ \texttt{r}_1\ \texttt{r}_2) \wedge (I_\psi \wedge C_\psi \wedge P_\psi)[\texttt{r}_M \mapsto m])\}.
\end{aligned}
$$

The difficult part is to show that, given the fact that $\forall x.(\texttt{nelist}_\psi\ x) \Rightarrow \big(\texttt{list}_\psi\ (\texttt{sel}\ m\ x)\big)$ for the old memory state m, we still have $\forall x.(\texttt{nelist}_\psi\ x) \Rightarrow \big(\texttt{list}_\psi\ (\texttt{sel}\ (\texttt{upd}\ m\ \texttt{r}_1\ \texttt{r}_2)\ x)\big)$. If $x \neq \texttt{r}_1$ then we use the fact that $(\texttt{sel}\ m\ x) = (\texttt{sel}\ (\texttt{upd}\ m\ \texttt{r}_1\ \texttt{r}_2)\ x)$; and if $x = \texttt{r}_1$, then $(\texttt{sel}\ (\texttt{upd}\ m\ \texttt{r}_1\ \texttt{r}_2)\ x) = \texttt{r}_2$, and we can use P, which guarantees that $(\texttt{list}_\psi\ \texttt{r}_2)$.

The remaining rows in the definition of the custom decoder are relatively simple proofs that we omit here. This proof of generalized decoder correctness has been formalized in Coq and is available on the web at the following URL: http://www.cs.berkeley.edu/~necula/ISSS02 . The formal development also includes formalizations of the notions of local invariants and decoders, and the proof of Lemma 1.

Notice that the proofs that the code producer must supply with the custom safety policy are not likely to be produced completely automatically. However, we have found that these proofs can be done with relatively modest effort using a proof assistant such as Coq. Coq has support for the inductive definitions of instantiation predicates and for reasoning with them. Ideally, the code consumer would have a (small, more easily trusted) proof checker that is also able to verify such reasoning.

7 Conclusion and Future Directions

Appel, Felty, and others have introduced *foundational proof-carrying code* (FPCC) [AF00,App01], a variant PCC framework in which the trusted computing base contains only a definition of the semantics of machine instructions and the notion of safety in some foundational logic. The safety theorem is then directly expressible in the logic: simply that when the integers that 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. Finally, the code is accompanied by a proof of the safety theorem.

It is worth pointing out a difference in our approach to formal type-safety proofs. Appel and Felty in [AF00] 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 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 respect, our work bears some similarity to the work of Hamid et al. [HST+02], who also aim to develop a full-fledged FPCC system, but advocate the syntactic approach.

While we think that the FPCC approach is quite a promising research direction it is already apparent that the cost of implementing such a system that operates on large programs is much higher than the cost of implementing a traditional PCC system (such as Touchstone [CLN+00]).

Touchstone and FPCC constitute two extremes in the PCC design spectrum. We propose in this paper one incremental step that will take a traditional PCC system closer to the ideal goal of FPCC. In future work we plan to make more steps in the same direction and more specifically to address the issue of the trusted VCGen. Ideally, we imagine a completely generic PCC system in which the code producer first uploads a custom safety policy consisting of a specialized decoder and VCGen, along with new trusted proof rules. Then, the code producer can upload programs whose proofs can be very short and easy to generate, since the bulk of the safety argument is embodied in the custom safety policy elements.

References

AF00. 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.

App01. 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.

CLN+00. 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.

Coq02. Coq Development Team. The Coq proof assistant reference manual, version 7.3. May 2002.

HST+02. Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

Nec97. George C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, January 1997.

Nec98. George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Also available as CMU-CS-98-154.

NS02. George C. Necula and Robert R. Schneck. A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 47–62. Springer-Verlag, July 27-30 2002.