A Scalable Architecture for Proof-Carrying Code

George C. Necula

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720, USA
necula@cs.berkeley.edu

Abstract. Proof-Carrying Code (PCC) is a general mechanism for verifying that a code fragment can be executed safely on a host system. The key technical detail that makes PCC simple yet very powerful is that the code fragment is required to be accompanied by a detailed and precise explanation of how it satisfies the safety policy. This leaves the code receiver with the simple task of verifying that the explanation is correct and that it matches the code in question.
Previous implementations of PCC used safety explanations in the form of explicit formal proofs of code safety, thus gaining leverage from a substantial amount of previous research in the area of proof representation and checking, but at the expense of poor scalability due to large proof sizes. In this paper we describe a series of changes that are necessary to achieve a truly scalable architecture for PCC. These include a new proof representation form along with a better integration of the various components of a PCC checker. We also present experimental results that show this architecture to be effective for checking the type safety of even very large programs expressed as machine code.

1 Introduction

More an more software systems are designed and build to be extensible and configurable dynamically. The proportion of extensions can range from a software upgrade, to a third-party add-on or component, to an applet. On another dimension, the trust relationship with the producer of the extension code can range from completely trusted, to believed-not-malicious, to completely unknown and untrusted. In such a diverse environment there is a need for a general mechanism that can be used to allow even untrusted system extensions to be integrated into an existing software system without compromising the stability and security of the host system.

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

* An invited paper at the Fifth International Symposium of Functional and Logic Programming, Waseda University, Tokyo, March 2001.

Proof-Carrying Code (PCC) [NL96, Nec97] was designed to be a *general* mechanism that allows the receiver of code (referred to as the **host**) to check *quickly* and *easily* that the code (referred to as the **agent**) has certain safety properties. The key technical detail that makes PCC very general yet very simple is a requirement that the agent producer cooperates with the host by attaching to the agent code an "explanation" of why the agent complies with the safety policy. Then all that the host has to do to ensure the safe execution of the agent is to define a framework in which the "explanation" must be conducted, along with a simple yet sufficiently strong mechanism for checking that (a) the explanation is acceptable (i.e., is within the established framework), that (b) the explanation pertains to the safety policy that the host wishes to enforce, and (c) that the explanation matches the actual code of the agent.

Below is a list of the most important ways in which PCC improves over other existing techniques for enforcing safe execution of untrusted code:

- PCC operates at **load time** before the agent code is installed in the host system. This is in contrast with techniques that enforce the safety policy by relying on extensive run-time checking or even interpretation. As a result PCC agents run at native-code speed, which can be ten times faster than interpreted agents (written for example using Java bytecode) or 30% faster than agents whose memory operations are checked at run time.

  Additionally, by doing the checking at load time it becomes possible to enforce certain safety policies that are hard or impossible to enforce at run time. For example, by examining the code of the agent and the associated "explanation" PCC can verify that a certain interrupt routine terminates within a given number of instructions executed or that a video frame rendering agent can keep up with a given frame rate. Run-time enforcement of timing properties of such fine granularity is hard.
- PCC is **small**. PCC is simple and small because it has to do a relatively simple task. In particular, PCC does not have to discover on its own whether and why the agent meets the safety policy.
- For the same reason, PCC can operate even on agents expressed in native-code form. And because PCC can verify the code after compilation and optimization, the checked code is ready to run without needing an additional interpreter or compiler on the host. This has serious software engineering advantages since it reduces the amount of security critical code and it is also a benefit when the host environment is too small to contain an interpreter or a compiler, such as is the case for many embedded software systems.
- PCC is **general.** All PCC has to do is to verify safety explanations and to match them with the code and the safety policy. By standardizing a language for expressing the explanations and a formalism for expressing the safety policies it is possible to implement a single algorithm that can perform the required check, for any agent code, any valid explanation and a large class of safety policies. In this sense a single implementation of PCC can be used for checking a variety of safety policies.

The combination of benefits that PCC offers is unique among the techniques for safe execution of untrusted code. Previously one had to sacrifice one or more of these benefits because it is impossible to achieve them all in a system that examines just the agent code and has to discover on its own why the code is safe.

The PCC infrastructure is designed to complement a cryptographic authentication infrastructure. While cryptographic techniques such as digital signatures can be used by the host to verify external properties of the agent program, such as freshness and authenticity, or the author's identity, the PCC infrastructure checks internal semantic properties of the code such as what the code does and what it does not do. This enables the host to prevent safety breaches due to either malicious intent (for agents originating from untrusted sources) or due to programming errors (for agents originating from trusted sources).

The description of PCC so far has been in abstract terms without referring to a particular form of safety explanations. There are a number of possible forms of explanations each with its own advantages and disadvantages. In any implementation the safety explanations must be precise and comprehensive, just like formal proofs. In fact, in the first realization of a PCC architecture [Nec98] the explanations were precisely formal proofs represented as terms in a variant of the dependently-typed λ-calculus called the Edinburgh Logical Framework (LF) [HHP93]. The major advantage of this approach is that proof checking can be accomplished using a relatively simple and well understood LF type checker.

The proof-based realization of PCC was very useful for gaining initial experience with the PCC technique in various applications. However, when we attempted to try out PCC for medium and large examples it quickly became apparent that the size of the LF representation of proofs grows roughly quadratically with the size of the program. (All experiments referred to in this paper are in the context of type-safety policies, for which the proof certifies that the machine code is well typed in a given, usually first-order, type system.) In an attempt to address this issue we first devised a simple extension of the LF type checker that enables it to reconstruct certain small but numerous fragments of proofs while type checking [NL98]. This variant of LF, which we call LF_i (for implicit LF), allows us to omit those fragments from the representation of proofs with the major benefit that proof sizes and the checking times are now growing linearly with the size of the program. This allows us to process examples up to several thousands of line of code with proof sizes averaging 2.5 to 5 times the size of the code. But even LF_i-based proof representations are too large for the hundred-thousand line examples that we want to process.

In this paper we describe a series of improvements to the PCC architecture that together allow us to process even the largest examples. These improvements range from major changes in the representation and checking of proofs to changes in the way different components of the PCC system communicate. We start in Section 2 with a description, in general terms, of the new proof representation technique that we use. Then, in Section 3 we demonstrate in the context of an example both the new proof representation technique and the way in which the various checker components communicate with each other. Although most of this paper focuses on the code-receiving end, we observed that for better scaling the code producer must also cooperate. We describe in Section 4 one way in which a compiler that produces PCC agents can speed up the checking process. Finally, in Section 5 we discuss experimental evidence that substantiates the claim that this combination of techniques does indeed scale to very large examples.

2 A Hint-Based Representation of Proofs

One of the main impediments to scalability in the original proof-based realization of PCC is that proofs can be very large. Our solution to this problem is motivated by the observation that PCC proof checking is not able to exploit domain-specific knowledge in order to reduce the size of the proofs. For example, the PCC proofs of type safety are an order of magnitude larger than the size of the typing annotations that the Typed Assembly Language (TAL) system [MWCG99] uses. The overhead in TAL is smaller because TAL is less general than PCC and targets a specific type-safety policy, for a specific language and type system. The TAL type checker can be viewed as a proof checker specialized and optimized for a specific logic.

What we need is a different PCC implementation strategy that allows the size of PCC proofs to adapt automatically to the complexity of the property being checked. As a result, we should not have to pay a proof-size price for the generality of PCC in those instances when we check relatively simple properties. There are several components to our new strategy. First is a slightly different view on how the proofs in PCC can assist the verification on the host side. We assume that the host uses a non-deterministic checker for the safety policy. Then, a proof can essentially be replaced by an oracle guiding the non-deterministic checker. Every time the checker must make a choice between N possible ways to proceed, it consults the next $\lceil \log_2 N \rceil$ bits from the oracle. There are several important points that this new view of PCC exposes:

- The possibility of using non-determinism simplifies the design of the checker and enables the code receiver to use a simple checker even for checking a complex property.
- This view of verification exposes a three-way tradeoff between the complexity of the safety policy, the complexity and "smartness" of the checker, and the oracle size. If the verification problem is highly directed, as is the case with typical type-checking problems, the number of non-deterministic choices is usually small, and thus the required oracles are small. If the checker is "smart" and can narrow down further the choices, the oracle becomes even smaller. At an extreme, the checker might explore by itself small portions of the search space and require guidance from the oracle only in those situations when the search would be either too costly or not guaranteed to terminate.
- In the particular case of type-checking, even for assembly language, the checking problem is so directed that in many situations there is only one applicable choice, meaning that no hand-holding from the oracle is needed.

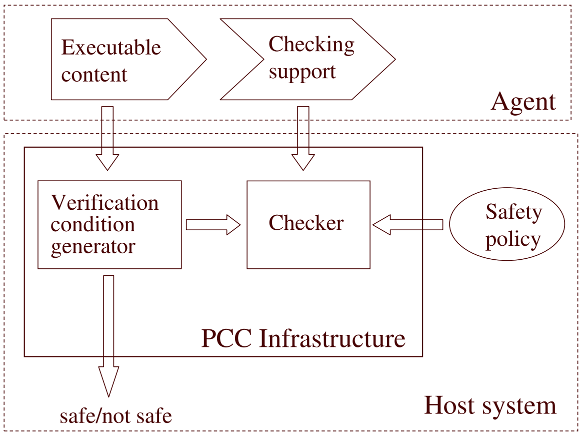

Fig. 1. The high-level structure of the PCC architecture.

This explains the large difference between the size of the typing derivation (i.e. the size of the actual proof) and the size of the oracles in our experiments.

- This view of PCC makes direct use of the defining property of the complexity class NP. This suggests that one of the benefits of PCC is that it allows the checker to check the solutions of problems in NP in polynomial time (with help from the oracle). But PCC can also help with checking even of solutions for semi-decidable problems, provided the checker and the oracle negotiate before hand a limit on the number of inference steps to be carried during verification.
- Since oracles are just streams of bits and no lookahead is necessary, they can be used in an online fashion, which is harder to do with syntactic representations of proofs. This leads to a smaller memory footprint for the checker, which is important in certain applications of PCC for embedded devices.

A high-level view of an oracle-based architecture for PCC is shown in Figure 1. The untrusted code is first inspected by a verification condition generator (VCGen) that checks simple syntactic conditions (e.g. that direct jumps are within the code boundary). Each time VCGen encounters an instruction whose execution could violate the safety policy, it asks the Checker module to verify that in the current context the instruction actually behaves safely. For this purpose, VCGen encodes the property to be checked as a simple symbolic formula. Our choice for the implementation of the Checker module is as a non-deterministic logic interpreter whose goals are the formulas produced by VCGen and whose logic program is an encoding of the host's safety policy. The next section describes in more details these components in the context of a simple example.

```
bool forall(bool *data, int dlast) {
  int i;
  for(i=dlast; i >= 0; i --)
    if(! data[i]) return false;
  return true;
}
```

Fig. 2. The C code for a function that computes the conjunction of all elements in an array of booleans.

3 An Extended Example

In this section we describe the operation of the PCC checking infrastructure in the context of verifying a simple type-safety policy for an agent written in a generic assembly language. Due to space limits and to keep the presentation concise we consider here only a trivial agent consisting of a function that computes the boolean value representing the conjunction of all elements in an array of booleans. The C source code for the function is shown in Figure 2. The inputs consist of a pointer to the start of an array of booleans along with the index of the last element. To simplify the presentation somewhat the array is scanned backwards.

```
                       /* r_d=data, r_l=dlast */
     r_i = r_l
L_0:
     r_t = ge r_i, 0
     jfalse r_t, L_1
     r_t = add r_d, r_i
     r_t = load t
     jfalse r_t, L_2
     r_i = sub r_i, 1
     jump L_0
L_1:  ret 1
L_2:  ret 0
```

Fig. 3. The assembly language code for the function shown in Figure 2.

.

In Figure 3 we show one possible compilation of the `forall` function into a generic assembly language. The resulting program uses four registers, r_d and r_l to hold respectively the start address and the index of the last element of the array, a register r_i to hold the value of the local variable i and a temporary register r_t.

3.1 The Safety Policy

Before the code producer and the host can exchange agents they must agree on a safety policy. Setting up the safety policy is mostly the host's responsibility. For our example, the safety policy is a variant of type safety and requires that all memory accesses be contained in a memory range whose starting value and last-element index are being passed by the host in the input argument registers r_d and r_l. Furthermore, only boolean values may be written to that memory range and the agent may assume that the values read from that memory range are themselves boolean values. The safety policy further specifies that the function's return value must be a boolean value. Finally, the safety policy specifies that only the values 0 and 1 are valid boolean values.

It is obvious that this safety policy is tailored to our example and thus is not very general. To alleviate this problem we allow the host to define a language of types along with their meaning and the agent to customize the safety policy by declaring the type of the arguments that it expects and the type of the return value. This constitutes the interface of the agent and in our system is expressed as a pair of a function precondition and postcondition formulas constructed by the code producer using a number of type constructors defined by the host's safety policy.

In the case of our example the agent declares the following specification:

$$\begin{aligned}\mathtt{Pre}_{\mathtt{forall}} &= r_d\colon \mathtt{array}(\mathtt{bool}, r_l)\\ \mathtt{Post}_{\mathtt{forall}} &= \mathtt{res} : \mathtt{bool}\end{aligned}$$

which is a straightforward transcription of the function type. This specification uses the infix binary predicate constructor ":" to denote that a certain register contains value of a certain type, along with two type constructors, `array` and `bool`. Somewhat unusual for a type, the `array` type constructor declares the index of the last element of the array along with the type of the array elements. In the postcondition, the name `res` refers to the result value.

These predicate and type constructors are declared as part of the trusted safety policy along with inference rules (or typing rules) that specify how the constructors can be used. We show in Figure 4 the rules of inference that are necessary for our example. In order to understand these rules we preview briefly the operation of the VCGen. We do so here with only enough details to motivate the form of the rules. More details on VCGen follow in Section 3.3.

When VCGen encounters a memory read at an address E_a, it asks the Checker module to verify that the predicate $\mathtt{saferd}(E_a)$ holds according to the trusted rules of inference. The first rule of inference from Figure 4 says that one instance in which a memory read is considered safe is when it falls within the boundaries of an array. The rule `mem` goes even further and says that the value read from an array has the same type as the declared array element type.[1]

[1] Since the contents of memory does not change in our example we can introduce a constructor `mem` such that $\mathtt{mem}(E)$ denotes the contents of memory at address E. In general, a different mechanism must be used to keep track of the memory contents in the presence of memory writes. The reader is invited to consult [Nec98] for details.

$$\frac{A : \mathtt{array}(T,\ L) \quad I \geq 0 \quad I \leq L}{\mathtt{saferd}(\mathtt{add}(A,\ I))}\ \mathtt{rd}$$

$$\frac{A : \mathtt{array}(T,\ L) \quad I \geq 0 \quad I \leq L}{\mathtt{mem}(\mathtt{add}(A,\ I)) : T}\ \mathtt{mem} \qquad \frac{A : \mathtt{array}(T,\ L) \quad I \geq 0 \quad I \leq L \quad E : T}{\mathtt{safewr}(\mathtt{add}(A,\ I),\ E)}\ \mathtt{wr}$$

$$\frac{}{0 : \mathtt{bool}}\ \mathtt{bool0} \qquad \frac{}{1 : \mathtt{bool}}\ \mathtt{bool1}$$

$$\frac{}{E = E}\ \mathtt{eqid} \qquad \frac{}{E \leq E}\ \mathtt{leqid} \qquad \frac{I \leq E \quad \mathtt{ge}(I,\ 0)}{\mathtt{sub}(I,\ 1) \leq E}\ \mathtt{dec} \qquad \frac{\mathtt{ge}(E, 0)}{E \geq 0}\ \mathtt{geq0}$$

Fig. 4. The rules of inference that constitute the safety policy for our example.

Similarly, the predicate $\mathtt{safewr}(E_a, E)$ is generated by VCGen to request a verification that a memory write of value E at address E_a is safe. The third rule of inference in our safety policy can prove such a predicate if E_a falls inside an array and if E has the array element type. Notice that in all of these rule the function `add` has been used instead of the usual mathematical function "+". This is because we want to preserve the distinction between the mathematical functions and their approximate implementation in a processor.

The rules `bool0` and `bool1` specify the representation of booleans. The last row of inference rules in Figure 4 are a sample of the rules that encode the semantics of arithmetic and logic machine instructions. Consider for example the rule `dec` rule. Here `ge` and `sub` are the predicate constructors that encode the result of the machine instructions with the same name. VCGen uses such predicate constructors to encode the result of the corresponding instructions, leaving their interpretation to the safety policy. The rule `dec` says that the result of performing a machine subtraction of 1 from the value I is less or equal to some other value E if I itself is known to be less-or-equal to E and also if the test $\mathtt{ge}(I, 0)$ is successful. This rule deserves an explanation. If we could assume that the machine version of subtraction is identical in behavior to the standard subtraction operation then the rule would be sound even without the second hypothesis. However, the second hypothesis must be added to prevent the case when I is the smallest representable integer value and the `sub` operation underflows. (A weaker hypothesis would also work but this one is sufficient for our example.) The rule `geq0` say that in comparisons with zero the meaning of the machine `ge` operation is the same as that of the mathematical greater-or-equal comparison.

The rules of Figure 4 constitute a representative subset of a realistic safety policy. The safety policy used for the experiments discussed in Section 5 for the type system of the Java language consists of about 140 such inference rules.

3.2 The Role of Program Annotations

The VCGen module attempts to execute the untrusted program symbolically in order to signal all potentially unsafe operations. To make this execution possible in finite time and without the need for conservative approximations on the part of VCGen, we require that the program be annotated with invariant predicates. At least one such invariant must be specified for each cycle in the program's control-flow graph. To further simplify the work of VCGen each such invariant must also declare those registers that are live at the invariant point and are guaranteed to preserve their values between two consecutive times that the execution reaches the invariant point. We call these registers the *invariant registers.*

For our example, at least one invariant must be specified for some point inside the loop that starts at label L_0. We place the following invariant at label L_0:

$$\mathtt{L}_0: \;\; \mathtt{INV} = r_i \leq r_l \;\; \mathtt{REGS} = \{r_d, r_l\}$$

The invariant annotation says that whenever the execution reaches the label L_0 the contents of register r_i is less-or-equal to the contents of register r_l and also that in between two consecutive hits of the program point the register r_d and r_l are the only ones among the live registers that are guaranteed to preserve their values.

A valid question at this point is who discovers this annotation and how. There are several possibilities. First, annotations can be inserted by hand by the programmer. This is the only alternative when the agent code is programmed in assembly language or when the programmer wants to hand-optimize the output of a compiler. It is true that this method does not scale well, but it is nevertheless a feature of PCC that the checker does not care whether the code is produced by a trusted compiler and will gladly accept code that was written or optimized by hand.

Another possibility is that the annotations can be produced automatically by a certifying compiler. Such a compiler first inserts bounds checks for all array accesses. In the presence of these checks the invariant predicates must include only type declarations for those live registers that are not declared invariant. An exception are those registers that contain integers, for which no declaration is necessary or useful since the integer data type contains all representable values. In our example, the reader can verify that in the presence of bounds checks preceding the memory read the invariant predicate `true` is sufficient. In general, the invariant predicates that are required in the absence of optimizations are very easy to generate by a compiler.

An optimizing compiler might analyze the program and discover that the lower bound check is entailed by the loop termination test and the upper bound check is entailed by a loop invariant "$r_i \leq r_l$". With this information the optimizing compiler can eliminate the bounds checks but it must communicate through the invariant predicate what are the loop invariants that it discovered and used in optimization. This is the process by which the code and annotations of Figure 3 could have been produced automatically. The reader can consult [Nec98] for the detailed description of a certifying compiler for a safe subset of C and [CLN+00] for the description of a certifying compiler for Java.

Finally, note that the invariant annotations are required but cannot be trusted to be correct as they originate from the same possibly untrusted source as the code itself. Nevertheless, VCGen can still use them safely, as described in the next section.

3.3 The Verification Condition Generator

The verification condition generator (VCGen) is implemented as a symbolic evaluator for the program being checked. It scans the program in a forward direction and at each program point it maintains a symbolic value for each register in the program. These symbolic values are then used at certain program points (e.g. memory operations, function calls and returns) to formulate checking goals for the Checker module. To assist the checker in verifying these goals VCGen also records for each goal that is submitted to the Checker a number of assumptions that the Checker is allowed to make. These assumptions are generated from the control-flow instructions (essentially informing the Checker about the location of the current program point) or from the program-supplied invariants. Figure 5 shows a summary of the sequence of actions performed by VCGen for our example.

First, VCGen initializes the symbolic values of all four registers with fresh new symbolic values to denote unknown initial values in all registers. But the initial values of the registers r_d and r_l are constrained by the precondition. To account for this, VCGen takes the specified precondition and, without trying to interpret its meaning, substitutes in it the current symbolic values of the registers. The result is the symbolic predicate formula "$d_0 : \mathtt{array}(\mathtt{bool}, l_0)$" that is added to a stack of assumptions. (These assumptions are shown underlined in Figure 5 and with an indentation level that corresponds to their position in the stack.)

After these initial steps VCGen proceeds to consider each instruction in turn. The assignment instruction of line 2 is modeled as a assignment of symbolic values. On line 3, VCGen notices that it hit an invariant. Since the invariant is not trusted VCGen asks the Checker to verify first that the invariant predicate holds at this point. To do this VCGen substitutes the current symbolic register values in the invariant predicate and the resulting predicate "$l_0 \leq l_0$" is submitted to the Checker for verification. (Such verification goals are shown in Figure 5 right-justified and boxed.)

Then, VCGen simulates symbolically an *arbitrary* iteration through the loop. To achieve this VCGen generates fresh new symbolic values for all registers, except the invariant ones. Next VCGen adds to the assumption stack the predicate "$i_1 \leq l_0$" obtained from the invariant predicate after substitution of the new symbolic values for registers.

When VCGen encounters a conditional branch it simulates both possible outcomes. First, the branch is assumed to be taken and a corresponding assumption is placed on the assumption stack. When that symbolic execution of that branch

1: Generate fresh values $r_d = d_0$, $r_l = l_0$, $r_i = i_0$ and $r_t = t_0$

1: Assume Precondition $\underline{d_0 \texttt{ : array(bool, } l_0\texttt{)}}$

2: Set $r_i = l_0$

3: Invariant (first hit) $\boxed{l_0 \leq l_0}$

3: Generate fresh values $r_i = i_1$, $r_t = t_1$

3: Assume invariant $\underline{i_1 \leq l_0}$

4: Set $r_t = \texttt{ge}(i_1, 0)$

5: Branch 5 taken $\underline{\texttt{not (ge(}i_1\texttt{, 0))}}$

11: Check postcondition $\boxed{\texttt{1 : bool}}$

5: Branch 5 not taken $\underline{\texttt{ge(}i_1\texttt{, 0)}}$

6: Set $r_t = \texttt{add}(d_0, i_1)$

7: Check load $\boxed{\texttt{saferd(add(}d_0\texttt{, }i_1\texttt{))}}$

7: Set $r_t = \texttt{mem}(\texttt{add}(d_0, i_1))$

8: Branch 8 taken $\underline{\texttt{not (mem(add(}d_0\texttt{, }i_1\texttt{)))}}$

10: Check postcondition $\boxed{\texttt{0 : bool}}$

8: Branch 8 not-taken $\underline{\texttt{mem(add(}d_0\texttt{, }i_1\texttt{))}}$

9: Set $r_i = \texttt{sub}(i_1, 1)$

3: Invariant (second hit) $\boxed{\texttt{sub(}i_1\texttt{, 1)} \leq l_0\texttt{, } d_0 \texttt{ = } d_0\texttt{, } l_0 \texttt{ = } l_0}$

4: Done

Fig. 5. The sequence of actions taken by VCGen. We show on the left the program points and a brief description of each action. Some actions result in extending the stack of assumptions that the Checker is allowed to make. These assumptions are shown underlined and with an indentation level that encodes the position in the stack of each assumption. Thus an assumption at a given indentation level implicitly discards all previously occurring assumptions at the same or larger indentation level. Finally, we show right-justified and boxed the checking goals submitted to the Checker.

terminates (e.g. when hitting a return instruction or an invariant for the second time), VCGen restores the state of the assumption stack and processes in a similar way the case when the branch is not taken.

Consider for example the branch of line 5. In the case when the branch is taken, VCGen pushes the assumption "$\texttt{not}\ (\texttt{ge}(i_1, 0))$" and continues the execution from line 11. There a return instruction is encountered and VCGen asks the Checker to verify that the postcondition is verified. The precise verification goal is produced by substituting the actual symbolic return value (the literal 1 in this case) for the name `res` in the postcondition $\texttt{Post}_{\texttt{forall}}$. Once the Checker module verifies this goal, VCGen restores the symbolic values of registers and the state of the assumption stack to their states from before the branch was taken and then simulates the case when the branch is not taken. In this case, the memory read instruction is encountered and VCGen produces a `saferd` predicate using the symbolic value of the registers to construct a symbolic representation of the address being read.

The final notable item in Figure 5 is what happens when VCGen encounters the invariant for the second time. In this case VCGen instructs the Checker to verify that the invariant predicate still holds. At this point VCGen also asks the Checker to verify that the symbolic values of those registers that were declared invariant are equal to their symbolic values at the start of the arbitrary iteration thorough the loop. In our case, since the registers declared invariant were not assigned at all, their symbolic values before and after the iteration are not only equal but identical.

This completes our simplified account of the operation of VCGen. For details on how VCGen deals with more complex issues such as function calls and memory updates, the reader is invited to consult [Nec98]. Note that in the VCGen used in previous versions of PCC (such as that described in [Nec98]) the result of the verification condition generator is a single large predicate that encodes all of the goals (using conjunctions) and all of the assumptions (using implications). This means that the VCGen runs first to completion and produces this predicate which is then consumed by the Checker. This approach, while natural, turned out to be too wasteful. For large examples it became quite common for this monolithic predicate to require hundreds of megabytes for storage slowing down the checking process considerably. In some of the largest examples not even a virtual address space of 1Gb could hold the whole predicate. In the architecture that we propose here VCGen produces one small goal at a time and then passes the control to the Checker. Once a goal is validated by the Checker, it is discarded and the symbolic evaluation continues. This optimization might not seem interesting from a scientific point of view but it is illustrative of a number of purely engineering details that we had to address to make PCC scalable.

3.4 The Checker Module

The Checker module in the proposed architecture is simply a non-deterministic logic interpreter whose logic program consists of the safety policy rules of inference formulated as Horn clauses and whose goals are the formulas produced by VCGen. For example, the `rd` inference rule of Figure 4 is expressed as the following clause written in Prolog notation:

$$\texttt{saferd}(\texttt{add}(A,\ I)) \quad {:}{-} \quad \texttt{of}(A,\ \texttt{array}(T,\ L)),\ \ I \geq 0,\ \ I \leq L.$$

There are two major differences between a traditional logic interpreter and our Checker. One is that in PCC the logic program is dynamic since assumptions (represented as logic facts) are added and retracted from the system as the symbolic execution follows different paths through the agent. However, this happens only in between two separate invocations of the Checker. The second and more important difference is that while a traditional interpreter selects clauses by trying them in order and backtracking on failure, the Checker is a non-deterministic logic interpreter meaning that it "guesses" the right clause to use at each step. This means that the Checker can avoid backtracking and it is thus significantly simpler and generally faster than a traditional interpreter. In essence the Checking contains a first-order unification engine and a simple control-flow mechanism that records and processes all the subgoals in a depth-first manner. The last element of the picture is that the "guesses" that the Checker makes are actually specified as a sequence of clause names as part of the "explanation" of safety that accompanies the code. In this sense the proof is replaced by an oracle that guides the non-deterministic interpreter to success.

As an example of how this works, consider the invocation of the Checker on the goal "$\texttt{saferd}(\texttt{add}(d_0,\ i_1))$". For this invocation the active assumptions are:

$$\begin{aligned} A_0 &= \texttt{of}(d_0,\ \texttt{array}(\texttt{bool},\ l_0)) \\ A_1 &= i_1\ \leq l_0 \\ A_2 &= \texttt{ge}(i_1,\ 0) \end{aligned}$$

The fragment of oracle for this checking goal is the following sequence of clause names "$\texttt{rd},\ A_0,\ \texttt{geq0},\ A_2,\ A_1$". To verify the current goal the Checker obtains the name of the next clause to be used (`rd`) from the oracle and unifies its head with the current goal. This leads to the following subgoals, where T and L are not-yet-instantiated logic variables:

$$\begin{aligned} &\texttt{of}(d_0,\ \texttt{array}(\mathrm{T},\ \mathrm{L})) \\ &i_1\ \geq 0 \\ &i_1\ \leq \mathrm{L} \end{aligned}$$

To solve the first subgoal the Checker extracts the next clause name (A_0) from the oracle and unifies the subgoal with its head. The unification succeeds and it instantiates the logic variables T and L to `bool` and l_0 respectively. Then the oracle guides the interpreter to use the rule `geq0` followed by assumption A_2 to validate the subgoal "$i_1 \geq 0$" and assumption A_1 to validate the subgoal "i1 $\leq l_0$". The oracle for checking all of the goals pertaining to our example is:

$$\texttt{leqid},\ \texttt{bool1},\ \texttt{rd},\ A_0,\ \texttt{geq0},\ A_2,\ A_1,\ \texttt{bool0},\ \texttt{dec},\ A_1,\ A_2,\ \texttt{eqid},\ \texttt{eqid}$$

3.5 An Optimized Checker

So far we have not yet achieved a major reduction in the size of the "explanation". Since the oracle contains mentions of every single proof rule its size is comparable with that of a proof. What we need now is to make the Checker "smarter" and ask it to narrow down the number of clauses that could possibly match the current goal, before consulting the oracle. If this number turns out to be zero then the Checker can reject the goal thus terminating the verification. In the fortunate case when the number of usable clauses is exactly one then the Checker can proceed with that clause without needing assistance from the oracle. And in the case when the number of usable clauses is larger than one the Checker needs only enough bits from the oracle to address among these usable clauses.

Such an optimization is also useful for a traditional logic interpreter because it reduces the need for backtracking. By having phrased the checking problem as a logic interpretation problem we can simply use off-the-shelf logic program optimizations to reduce the amount of non-determinism and thus to reduce the size of the necessary oracle.

Among the available logic program optimizations we selected automata-driven term indexing(ATI) [RRW95] because it has a relatively small complexity and good clause selectivity. The basic idea behind ATI is that the conclusions of all the active clauses are scanned and a decision tree is build from them. Intuitively, the leaves of the decision tree are labeled with sets of clauses whose conclusions could match a goal whose structure is encoded by the path corresponding to the goal. This allows the quick computation of the set of clauses the could match a goal. For details on how the ATI technique is used in the implementation of the Checker the reader can consult [NR01].

In our example the ATI optimization works almost perfectly. For nearly all goals and subgoals it manages to infer that exactly one clause is usable. However, for three of the subgoals involving the predicate "$\leq$" the ATI technique confuses the correct clause (either A_1 or `dec`) with the `leqid` rule. This is because ATI is not able to encode exactly in the decision tree conclusions involving duplicate logical variables. Thus the conclusion of the `leqid` rule is encoded as "$E_1 \leq E_2$", which explains why ATI would think that this conclusion matches all subgoals involving the $\leq$ predicate.

Even with this minor lack of precision the oracle for our example is reduced to just 3 bits. This constitutes a major saving. The original oracle consisted of 13 clause names and since we never had more than 16 active clauses in our example, we could use 52 bits to encode the entire oracle. In practice we observe savings of more than 30 times over the uncompressed oracles. This is because it is not uncommon to have even 1000 clauses active at one point. This would happen in deeply nested portions of the code where there are many assumptions active. And the ATI technique is so selective that even in these cases it is able to filter out most of the clauses.

4 Compiler Support for Scalability

So far we have described two techniques that proved essential for obtaining a scalable architecture for PCC: an oracle-based encoding of proofs and an interleaved execution of the VCGen and the Checker modules. Both of these techniques are implemented on the code-receiver end. But it would be incorrect to draw the conclusion that scalability can be achieved without cooperation from the code-producer end.

To see how the code producer can and should help, recall that VCGen considers both branches for each conditional and that symbolic execution stops when a return instruction is encountered or when an invariant is encountered for the second time. VCGen does verify that each cycle through the code has at least one invariant thus ensuring the termination of the symbolic evaluation.

But consider a program without loops and with a long sequence of conditionals, such as the following:

$$\begin{array}{l} \mathrm{if}(\mathrm{c}_1)\ \mathrm{s}_1\ \mathrm{else}\ \mathrm{s}'_1; \\ \cdots \\ \mathrm{if}(\mathrm{c}_n)\ \mathrm{s}_n\ \mathrm{else}\ \mathrm{s}'_n; \\ \mathrm{return\ x} \end{array}$$

VCGen considers each of the 2^n paths through this function because it might actually be the case that each such path has a different reason (and proof) to be safe. In most cases (and in all cases involving type safety) the verification can be modularized by placing an invariant annotation after each conditional. Then VCGen will have to verify only $2n$ path fragments from one invariant to another. Thus the code producer should place more than the minimum necessary number of invariants. If it fails to do so then the number of verification goals, and consequently the size of the oracle and the duration of the verification process can quickly become very large. We have verified in our experiments that this optimization is indeed very important for scalability.

5 Experimental Results

The experiments discussed in this section are in the context of a PCC system that checks Intel x86 binaries for compliance with the Java type-safety policy. The binaries are produced using a bytecode-to-native optimizing compiler [CLN+00]. This compiler is responsible for generating the invariant annotations. The oracle is produced on the code producer side in a manner similar to oracle checking. The VCGen is used as usual to produce a sequence of goals and assumptions. Each of these goals is submitted to a modified Checker engine that first uses the ATI engine to compute a small set of usable clauses and then tries all such clauses in order, using backtracking. Then this modified Checker emits an oracle that encodes the sequence of clauses that led to success for each goal.

We carried out experiments on a set of nearly 300 Java packages of varying sizes. Some of the larger ones are described in Figure 6. The running times

Program	Description	Source Size (LOC)	Code Size (bytes)
gnu-getopt	Command-line parser	1588	12644
linpack	Linear algebra routines	1050	17408
jal	SGI's Java Algorithm Library	3812	27080
nbody†	N-body simulation	3700	44064
lexgen	Lexical analyzer generator	7656	109196
ocaml†	Ocaml byte code interpreter	9400	112060
raja	Raytracer	8633	126364
kopi†	Java development tools	71200	760548
hotjava	Web browser	229853	2747548

Fig. 6. Description of our test cases, along with the size of the Java source code and the machine-code size. † indicates that source code was not available and the Java source size is estimated based on the size of the bytecode.

Program	LF_i Size (bytes)	LF_i Checking Time (ms)	Oracle Size (bytes)	Logic Interpretation Time (ms)
gnu-getopt	49804	82	1936	223
linpack	65008	117	2360	319
jal	53328	84	1698	314
nbody†	187026	373	7259	814
lexgen	413538	655	15726	1948
ocaml†	415218	641	13607	1837
raja	371276	747	11854	2030
kopi†	3380054	5321	96378	14693
hotjava	10813894	19757	354034	53491

Fig. 7. For each of the test cases, the size and time to check the LF_i representation of proofs, the size of the oracles and the logic interpretation time using oracles.

were averaged over a sufficient number of iterations of the checker on a 400MHz Pentium II machine with 128MB RAM.

Based on the data shown in Figure 6 we computed, for each example, how much smaller are the oracles compared to LF_i proofs (shown in Figure 8), what percentage of the code size are the oracles (shown in Figure 9) and how much slower is the logic interpreter compared to the LF_i checker (shown in Figure 10). These figures also show the geometric means for the corresponding ratios over these examples.

We observe that oracles are on average nearly 30 times smaller than LF_i proofs, and about 12% of the size of the machine code. While the binary representation of oracles is straightforward, for LF_i it is more complicated. In particular, one has to decide how to represent various syntactic entities. For the purpose of computing the size of LF_i proofs, we streamline LF_i terms as 16-bit tokens, each containing tag and data bits. The data can be the deBruijn index [DeB72] for a

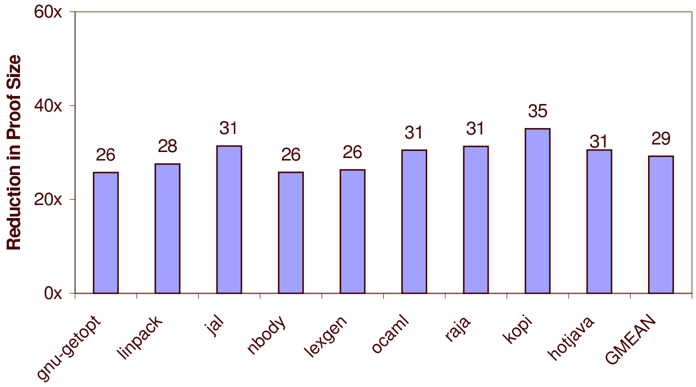

Fig. 8. Ratio between LF_i proof size and oracle size.

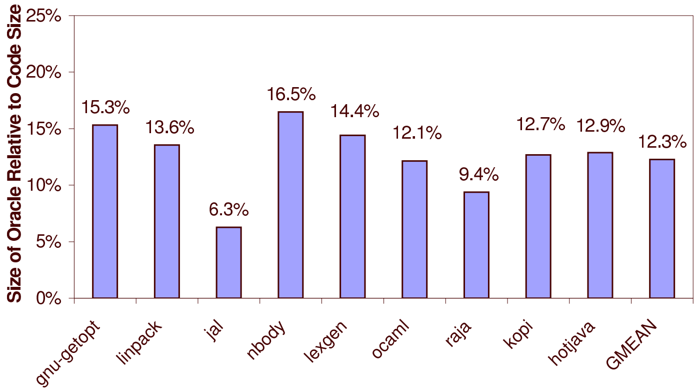

Fig. 9. Ratio between oracle size and machine code.

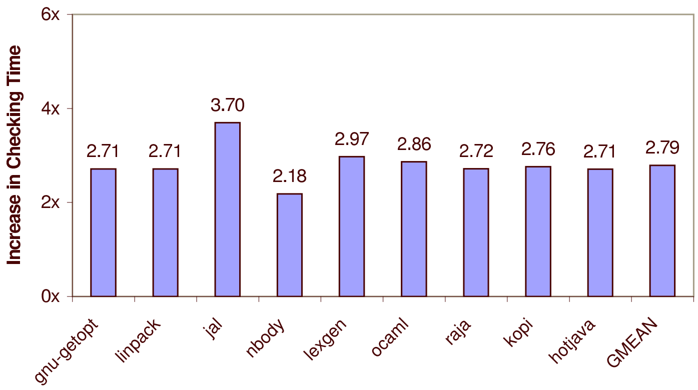

Fig. 10. Ratio between logic-interpretation time and LF_i type-checking time.

variable, the index into the signature for constants, or the number of elements in an application or abstraction.

We also compared the performance of our technique with that obtained by using a popular off-the-shelf compression tool, namely `gzip`. We do more than 3 times better than `gzip` with maximum compression enabled, without incurring the decompression time or the addition of about 8000 lines of code to the server side of the PCC system. That is not to say that oracles could not benefit from further compression. There could be opportunities for Lempel-Ziv compression in oracles, in those situations when sequences of deduction rules are repeated. Instead, we are looking at the possibility of compressing these sequences at a semantic level, by discovering lemmas whose proof can be factored out.

It is also interesting to note that logic interpretation is about 3 times slower than LF_i type checking. This is due to the overhead of constructing the decision trees used by ATI. There are some simple optimizations that one can do to reduce the checking time. For example, the results shown here are obtained by using an ATI truncated to depth 3. This saves time for the maintenance of the ATI but also loses precision thus leading to larger oracles. If we don't limit the size of the ATI we can save about 8% in the size of the oracles at the cost of increasing the checking time by 24%.

One interesting observation is that while LF_i checking is faster than oracle checking, it also uses a lot more memory. While oracles can be consumed a few bits at a time, the LF_i syntactic representation of a proof must be entirely brought in memory for checking. While we have not measured precisely the memory usage we encountered examples whose oracles can be checked using less than 1Mbyte of memory while the checking of the corresponding LF_i terms could not be performed even with 1Gbyte of virtual memory.

6 Conclusion

We presented in this paper an architecture for Proof-Carrying Code where proofs are replaced by an oracle guiding a non-deterministic checker for a given safety policy. The luxury of using non-determinism in the checker allows a simple checker to enforce even complex safety policies. Since many safety policies are relatively simple, the amount of non-determinism is low and this leads to small oracles that are required for checking compliance with such policies. In this sense the proposed PCC architecture is able to adapt the cost of verification to the complexity of the safety policy.

In designing this architecture we struggled to preserve a useful property of the previous implementation of PCC, namely that it can be easily configured to check different safety policies without changing the implementation. This has great software-engineering advantages and contributes to the trustworthiness of a PCC infrastructure since code that changes rarely is less likely to have bugs. To support this feature our choice for a non-deterministic checker is a non-deterministic logic interpreter that can be configured with the safety policy encoded as a logic program.

To achieve true scalability we had to solve several engineering problems, such as to design a low-cost interaction model between the various modules that compose the infrastructure. The code producer also must play an active role in ensuring that the verification process is quick. Through the combination of such techniques we have produced the first implementation of PCC that scales well to large programs at least in the context of a fairly simple type safety policy. What remains now to be seen is if Proof-Carrying Code can be practically applied to more complex safety policies.

Acknowledgments

We would like to thank Shree Rahul for the help with the collection of the experimental data presented in this paper and to Peter Lee, Mark Plesko, Chris Colby, John Gregorski, Guy Bialostocki and Andrew McCreight from Cedilla Systems Corporation who have implemented the certifying compiler for Java used in these experiments.

References

[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.

[DeB72] N. DeBruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34:381–392, 1972.

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

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[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.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementations*, pages 229–243. Usenix, October 1996.

[NL98] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Thirteenth Annual Symposium on Logic in Computer Science*, pages 93–104, Indianapolis, June 1998. IEEE Computer Society Press.

[NR01] George C. Necula and Shree P. Rahul. Oracle-based checking of untrusted programs. In *The 28th Annual ACM Symposium on Principles of Programming Languages*. ACM, January 2001. To appear.

[RRW95] R. Ramesh, I. V. Ramakrishnan, and David Scott Warren. Automata-driven indexing of Prolog clauses. *Journal of Logic Programming*, 23(2):151–202, May 1995.