Safe, Untrusted Agents using Proof-Carrying Code

George C. Necula and Peter Lee

Carnegie Mellon University
School of Computer Science
Pittsburgh PA 15217, USA

Abstract. Proof-Carrying Code (PCC) enables a computer system to determine, automatically and with certainty, that program code provided by another system is safe to install and execute without requiring interpretation or run-time checking. PCC has applications in any computing system in which the safe, efficient, and dynamic installation of code is needed. The key idea of Proof-Carrying is to attach to the code an easily-checkable proof that its execution does not violate the safety policy of the receiving system. This paper describes the design and a typical implementation of Proof-Carrying Code, where the language used for specifying the safety properties is first-order predicate logic. Examples of safety properties that are covered in this paper are memory safety and compliance with data access policies, resource usage bounds, and data abstraction boundaries.

1 Introduction

Proof-Carrying Code (PCC) enables a computer system to determine, automatically and with certainty, that program code, also referred to as an *agent*, provided by another system is safe to install and execute. The key idea behind PCC is that the external system, which we shall henceforth refer to as the *code producer*, provides an encoding of a proof that the code adheres to a safety policy defined by the recipient of the code, which we shall call the *code consumer*. The proof is encoded in a form that can be transmitted digitally to the consumer and then quickly validated using a simple, automatic, and reliable proof-checking process.

PCC is useful in many applications. It enhances the ability of a collection of software systems to interact flexibly and efficiently by providing the capability to share executable code safely. Typical examples of code consumers include

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.
Submitted to the Lecture Notes in Computer Science Special Issue on Mobile Agents.

operating system kernels and World-Wide Web browsers, which must allow untrusted applications and Internet hosts to install and execute code. Indeed, PCC is useful in any situation where the safety in the presence of newly installed code is paramount.

PCC has several key characteristics that, in combination, give it an advantage over previous approaches to safe execution of foreign code:

1. *PCC is general.* The code consumer defines the safety policy, and this policy is not limited to a particular notion of "safety." We have experimented both with simple safety properties, such as memory and type safety, and with properties that are normally difficult to verify, such as time limits on execution and resource usage bounds.
2. *PCC is low-risk and automatic.* The proof-checking process used by the code consumer to determine code safety is completely automatic, and can be implemented by a program that is relatively simple and easy to trust. Thus, the *safety-critical infrastructure* that the code consumer must rely upon is reduced to a minimum.
3. *PCC is efficient.* In practice, the proof-checking process runs quickly. Furthermore, in contrast to previous approaches, the code consumer does not modify the code in order to insert costly run-time safety checks, nor does the consumer perform any other checking or interpretation once the proof itself has been validated and the code installed.
4. *PCC does not require trust relationships.* The consumer does not need to trust the producer. In other words, the consumer does not have to know the identity of the producer, nor does it have to know anything about the process by which the code was produced. All of the information needed for determining the safety of the code is included in the code and its proof.
5. *PCC is flexible.* The proof-checker does not require that a particular programming language be used. PCC can be used for a wide range of languages, even machine languages.

This paper describes Proof-Carrying Code and how it can be used to enforce safety in the presence of untrusted agents. We begin with a general overview of the basic elements of PCC. Then, each of the major components of PCC are described in the subsequent sections. For a more concrete presentation of the implementation details of PCC, we introduce in Sect. 4 a stripped-down example of a safety policy and an associated agent that can be used for agent-based shopping. After we complete the discussion of PCC we review the agent example in the context a more complex safety policy (Sect. 7). We conclude with a comparison with related work and a presentation of the exeprimental result showing that agents using PCC can be much faster than agents whose safety is enforced through run-time checking.

2 Basic Elements of Proof-Carrying Code

Proof-Carrying Code has many applications, and each such application may entail some variations on the precise details of the approach. We will have more to say about some of these variations in the next section. In this section we describe at a high-level a canonical implementation of PCC, which is general enough that any of the variations can be seen as optimizations or special cases.

In its most general form PCC involves, in addition to a *code consumer* and *code producer*, a *proof producer*. In practice, it often turns out that the code producer and proof producer are the same system, though in general they may be separate entities.

A central component of any PCC implementation is the *safety policy*, which is specified by the code consumer. In this paper, we use the term "safety policy" in two distinct ways. First, we use it to denote the abstract set of safety rules that the code consumer desires to enforce for the untrusted code. We also use the term "safety policy" to denote the concrete realization of these safety rules as two components of the code consumer: a program called *VCGen* and a *proof-checker configuration file* that defines the *logic*, i.e., the valid axioms and inference rules that may be used in the proofs. Details about the implementation of the safety policy are given in Sect. 4.

Once the safety policy is defined, PCC involves a two-stage interaction process. In the first stage, the code consumer receives the untrusted code and extracts from it a *safety predicate* that can be proved only if the execution of the code does not violate the safety policy. This predicate is then sent to the proof producer who proves it and returns its proof back to the consumer. In the second stage, the code consumer checks the validity of the proof using a simple and fast proof checker. If the proof is found to be a valid proof of the safety predicate, then the untrusted code is installed and executed.

This two-stage verification process is a key design element contributing to the advantages listed in the previous section. In particular, this is the reason PCC can be used to certify code properties that would be very difficult, or even impossible to infer from the code directly. Also, by staging the verification into a difficult phase (proof generation) and a simple phase (proof checking) we are able to minimize the complexity of the safety-critical infrastructure, that is, the implementation of the proof-checking process, thereby greatly reducing the risk that a bug in the system leads to the failure to detect unsafe programs. In fact, we have made it a design goal of PCC that any task whose result can be more easily checked than generated, should be performed by an untrusted entity (the code or the proof producer) and then checked by the code consumer.

We now give a step-by-step description of a typical PCC session, glossing over many implementation details. These details are deferred to later parts of this paper. Fig. 1 shows a session based on the canonical PCC implementation, where the sequence of steps is determined by the arrows.

Step 1. A PCC session starts with the code producer preparing the untrusted code to be sent the code consumer. As part of this preparation, the producer adds *annotations* to the code. The annotations, whose exact nature is discussed later, contain information that helps the code consumer understand the safety relevant properties of the code. The code producer then sends the annotated code to the code consumer, requesting its execution.

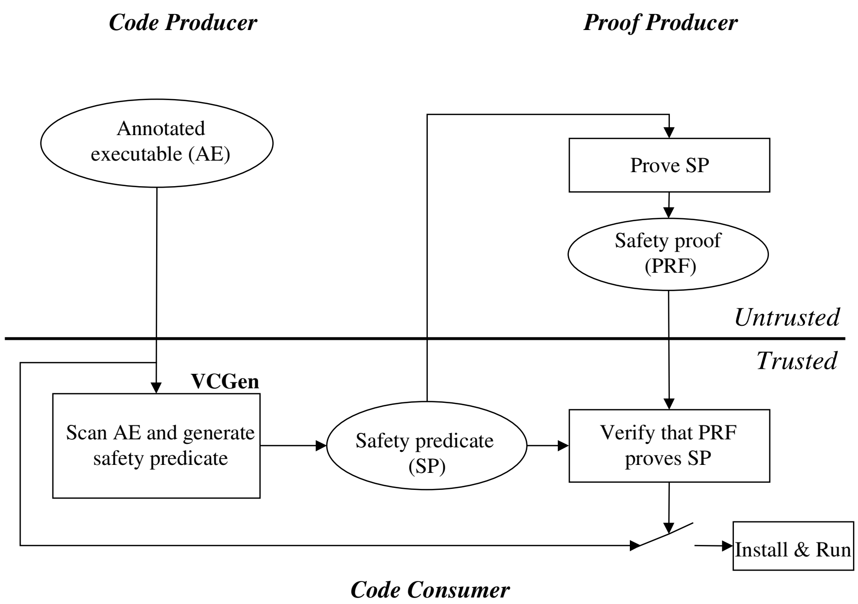

Fig. 1. Overview of Proof-Carrying Code.

Step 2. Upon receiving the annotated code, the code consumer performs a fast but detailed inspection of the annotated code. This is accomplished using a program, called *VCGen*, which is one component of the consumer-defined safety policy. VCGen performs two tasks. First, it checks simple safety properties of the code. For example, it verifies that all immediate jumps are within the code-segment boundaries. Second, VCGen watches for instructions whose execution might violate the safety policy. When such an instruction is encountered, VCGen emits a predicate that expresses the conditions under which the execution of the instruction is safe. Following the standard terminology from the field of automatic program verification, we refer to such predicates as *verification conditions* (and hence VCGen can be seen as a classical verification condition generator). The collection of the verification conditions, together with some control flow information, make up the *safety predicate*, a copy of which is sent to the proof producer.

Step 3. Upon receiving the safety predicate, the proof producer attempts to prove it, and in the event of a success it sends an encoding of a formal proof back to the code consumer. Because the code consumer does not have to trust the proof producer, any system can act as a proof producer. In particular the code producer can also act as the proof producer.

Step 4. The next step in a PCC session is the proof validation step performed by the code consumer. This phase is performed using a program which we refer to as the *proof checker*. The proof checker verifies that each inference step in the proof is a valid instance of one of the axioms and inference rules specified as part of the safety policy. In addition, the proof checker verifies that the proof proves the same safety predicate generated in Step 2 and not another predicate.

Step 5. Finally, after the executable code has passed both the VCGen checks and the proof check, it is trusted not to violate the safety policy and it is installed for execution, without any further need of run-time checking.

3 Variants of Proof-Carrying Code

Fig. 1 and the five-step process described above present a canonical view of Proof-Carrying Code. However, this approach to PCC is not the only one. By redistributing the tasks between the entities involved we can adapt PCC to special practical circumstances while maintaining the same safety guarantees. In this section we briefly discuss several such variations. Then, in the subsequent sections, we present details of the implementation of each component of PCC.

- In one variant of PCC the code producer runs VCGen itself and then submits the resulting predicate to the proof producer. Then the code and the proof are sent together to the code consumer that runs VCGen again and verifies that the incoming proof proves the resulting safety predicate. This arrangement is possible because there is nothing secret about VCGen and it can therefore be given to untrusted code producers to use. To retain the safety guarantees of original PCC, it is necessary that the code consumer repeats the VCGen step in order to produce a trustworthy safety predicate. Because this version saves a communication step in generating the safety predicate, it is preferred over the interactive version when the latency of the verification must be minimized.
- In another variant of PCC the code consumer does the proof generation. For this to be possible it must be the case that the safety predicate be relatively easy to prove automatically without extra knowledge about the program. This variant of PCC is useful in situations when the generated proof would be too large to send over the communication channel between the proof producer and the code consumer.
  Even though the code consumer does more work in this variant of PCC, the safety-critical infrastructure, consisting of VCGen and the proof checker, remains the same. One could be tempted to save the cost of generating, storing and verifying the proof altogether by trusting the theorem prover on the consumer side. But this saving is at the expense of greatly increasing the size and complexity of the safety-critical infrastructure, and our experience suggests that relying on the correctness of a complex theorem prover is a dangerous game.
- Yet another scheme for employing PCC is to use one of the variants above to establish the safety of the code on given code consumer system C, and then to forward this code for execution to any other system that trust C. This trust can be established by any convenient means such as digital signatures. This

scheme might be useful in enclaves where there are some trusted machines with the computational power to perform the VCGen and proof-checking phases, and other machines that do not have this power but still want to execute the untrusted code. For example, a firewall might certify external code using PCC and then forward the code to other machines inside the firewall.

No matter which of these or other variants are chosen, they all share the same characteristic of depending on a small and well-defined safety-critical infrastructure, given by a simple proof checker and VCGen.

4 Design Details of VCGen

Starting with this section, we reconsider in more detail the main building blocks of a PCC system. We start with VCGen and continue in the next section with a description of the proof checker and the proof generator.

The purpose of VCGen is twofold: to perform simple checks on the untrusted code, and to emit verification conditions for all checks that are mandated by the safety policy but difficult to perform at this time. In order to simplify the adaptation of VCGen to different safety policies, it is useful to restrict the checks performed by VCGen to those code properties that are likely to be encountered in many safety policies (e.g., that branch targets are within the code boundaries, or that the function invocations follow a stack discipline). All other checks should be emitted as general verification conditions, whose interpretation is left to the logic used in the proofs of the safety predicates.

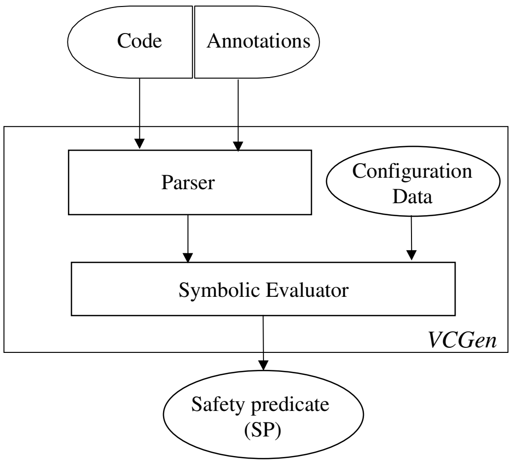

Fig. 2. The structure of VCGen

For example, we have found it to be useful to assume that some form of memory safety is always going to be part of the safety policy. However, we do not want to hard-wire in VCGen a particular form of memory safety. Hence, we designed VCGen to emit a generic verification condition `saferd` $(mem, addr)$ for each attempted read access from address *addr* in memory state *mem*. Then, it would be left to the proof logic to determine the meaning of the `saferd` predicate. A given logic might even say that `saferd` is never true, thus effectively disallowing memory reads.

Several techniques for implementing VCGen have been described in the literature [3, 8]. As shown in Fig. 2, our approach to implementing VCGen involves two main components. One is the language-dependent *parser*, whose purpose is to translate the instructions of the untrusted annotated code to a stream of instructions in a generic *intermediate language* (IL) that is understood by the second component, the *symbolic evaluator*. VCGen can be customized to a particular safety policy by using a configuration file that is provided as part of the safety policy by the code consumer.

Example: An Agent-Based Travel Agency. For a more concrete presentation of the implementation details that follow, we introduce here a simple example of a safety policy and an associated agent that can be used for agent-based shopping.

Assume that a travel agency host **H** records a database of pricing information for airline tickets between various destinations. A distinguishing feature of this particular travel agency is that mobile untrusted agents are allowed to scan the database and then communicate back to their parent host. To control the access to the information in the database, the travel agency implements a multi-level access protection scheme for the database. For this purpose, it assigns access levels to the agent and to each record in the database, and requires that an agent can only access those records whose access level is less or equal to its own access level. And to make things even more interesting the travel agency decides not to mediate at all the interaction between the agent and the database, but instead to use Proof-Carrying Code to select the agents that obey the safety policy from those whose behavior is uncertain.

For the purpose of this paper we are using a simple agent that computes the best available price for a trip. The actual agent is expressed in DEC Alpha assembly language, but for clarity we only show here the C source code (Fig. 3). We make the simplifying assumption that the database is an array of pricing entries, each entry containing four 32-bit fields encoding the entry access level, the source and destination airport and the associated price, in that order. We focus here only on the operation of reading the price fields. This agent will be used in future sections to exemplify the operation of the building blocks of PCC. For the purpose of providing a simple example, we initially ignore the communication between the agent and its parent host. Then in Sect. 7, we will expand the safety policy to allow communication and to restrict the agent's use of CPU cycles and network bandwidth.

```
int main(ENTRY tab[], int len, ACCESS acc) {
  for(j=0;j<len;j++) {
    if(tab[j].access <= acc) {int p = tab[j].price; ... }
  }
}
```

Fig. 3. A fragment of an agent that computes the best available price for a trip.

4.1 The Code Annotations

Some of the code properties of interest are, in general, difficult to infer from the code directly. In such cases VCGen relies on the code annotations provided by the producer as an extra source of information about the code behavior. But in doing so it must be careful not to allow incorrect annotations to hide unsafe program behavior.

There are several kinds of annotations that we currently use, some of them mandatory and others used only for optimization purposes. The most important mandatory annotations are the *loop invariants*. Their main purpose is to associate with each loop a set of properties that are preserved by the execution of the loop body, and that are sufficient for proving the safety of the code. The presence of a loop invariant for each loop in the code makes it possible for VCGen to extract the safety predicate in just one pass thorough the code. The requirement that every loop has an associated invariant can be easily satisfied by associating an invariant with every backward-branch target.

For each loop invariant two verification conditions are emitted as part of the safety predicate: one verifies that the invariant holds on loop entry and the other verifies that it is preserved through one loop iteration. If both these are proved, then, by induction, the loop invariant can be assumed valid for the purpose of inspecting the loop body and the code following the loop.

Another example of code annotations are the *call-target annotations*, which are required to disambiguate the target of an indirect call instruction. This kind of annotations is needed, for example, when the untrusted code is written in the DEC Alpha assembly language where only indirect function calls exist. VCGen reads a call-target annotation and continues its code inspection assuming that the annotation is correct. To prevent safety breaches, VCGen also generates a safety condition for the call site requiring that the value of the expression used in the indirect call is equal to the address of the destination declared through the annotation.

There are other, less important annotations, that we do not describe here. However, they all share the property that are untrusted and, following the model of the loop invariants and call-target annotations, they are checked using verification conditions.

4.2 The VCGen Configuration File

In order to reduce the need for dynamic checking of parameters, the code consumer usually declares a *precondition*, which is essentially a description of the calling convention the consumer will use when invoking the untrusted code. For example, if the untrusted code needs to access Ethernet network packets, the code consumer might declare that the first argument passed to the code is an array of length at least 64 bytes. With this assumption array accesses to most packet header fields can be proved safe without the need for run-time array-bounds checking. The safety policy can also declare *postconditions* for the untrusted code. These are constraints on the final execution state of the untrusted code.

Both the precondition and postcondition are parameters of VCGen and are part of the safety policy. The preconditions and the postconditions for all the functions declared by the untrusted code, as well as for the functions exported by the code consumer, are expressed as first-order logic predicates in a VCGen configuration file. The code consumer guarantees that the code precondition holds when the untrusted code is invoked. In turn, the untrusted code must ensure that the postcondition holds on return. For functions exported by the consumer the situation is reversed and the precondition is a predicate that the untrusted code must establish before calling the function while the postcondition is a predicate that the untrusted code may assume to hold upon return from the function.

Example: The Safety Policy for the Agent-Based Travel Agency. We return now to the example introduced before with the purpose of defining its precondition and postcondition. For this purpose we define the predicate $\mathtt{entry}(m, e, a)$ to denote that, in memory state m and for an agent whose access level is a, the address e points to an entry in the pricing database. This is described formally as follows:

$$\begin{aligned}\mathtt{entry}(m,e,a) = {} & \mathtt{saferd}(m, e+0) \wedge \mathtt{saferd}(m, e+4) \wedge \mathtt{saferd}(m, e+8) \wedge \\ & \mathtt{sel}(m, e+0) \leq a \;\supset\; \mathtt{saferd}(m, e+12)\end{aligned}$$

Informally, the above definition says that the 32-bit words situated at offsets 0, 4 and 8 from the start of the entry are always readable, while the word situated at offset 12 (the price) is only readable if the value of the first word (the access level) is less or equal to the agent's own access level.

The safety policy requires that each agent contains a function $\mathtt{main}(tab, len, acc)$ that expects as arguments the starting address of the table, the number of entries in the table and the access level that was assigned to this agent. No postcondition is assigned to the agent at this time, meaning that **H** imposes no requirements on the agent's result. This calling convention is expressed as a pair of a precondition and a postcondition, as shown below:

$$\begin{aligned}Pre_{\mathtt{main}} &= \forall i. 0 \leq i \wedge i < len \supset \mathtt{entry}(mem, tab + i \times 16, acc) \\ Post_{\mathtt{main}} &= \mathtt{true}\end{aligned}$$

Variables	*Vars*	x
Variable sets	$\mathcal{P}(\mathit{Vars})$	s
Labels	*Label*	l
Expressions	*Expr*	$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid$ $\texttt{sel}\ (e_1, e_2) \mid \texttt{upd}\ (e_1, e_2, e_3)$
Predicates	*Pred*	$P ::= \texttt{true} \mid \texttt{false} \mid P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x.P_x \mid$ $e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \mid$ $\texttt{saferd}\ (e_1, e_2) \mid \texttt{safewr}\ (e_1, e_2, e_3)$
Instructions	*Instr*	$c ::= \texttt{SET}\ x, e \mid \texttt{ASSERT}\ P \mid \texttt{CALL}\ l \mid \texttt{RET} \mid$ $\texttt{BRANCH}\ P_1 \rightarrow l_1 \square\ P_2 \rightarrow l_2 \mid$ $\texttt{INV}\ P, s \mid \texttt{MEMRD}\ e \mid \texttt{MEMWR}\ e_1, e_2$

Table 1. The syntax of the intermediate language (IL).

This safety policy could be enforced at runtime only if each access to the table is mediated by the host. In our example, the untrusted agent performs the access checks itself without invoking the host, thus greatly reducing the run-time penalty for enforcing safety.

4.3 The Code Parser

The purpose of the code parser is to provide a machine and language-independent interface to the symbolic evaluator. In fact, the parser is the only language-dependent component of a PCC system. While translating instructions from the incoming language to the intermediate language it abstracts over language and code details that are not relevant to the safety policy.

The intermediate language (IL) syntax is shown in Tab. 1. To simplify the parsing of machine code we use a generic assembly language as the IL. The examples presented in this paper require one distinguished variable *mem* for denoting the state of the memory during symbolic evaluation. If the safety policy makes explicit use of other state components besides memory, then they should be modeled in a similar fashion. Beyond this requirement the sets of variables and labels are left abstract at this time because they depend on the particular source language being parsed.

For expository purposes, the intermediate language presented here is restricted to the few constructors that we shall use in our examples. In practice, expressions for most arithmetic and logical operation would also need to be included. Also, depending on the needs of the safety policy an extension of first-order logic might be used, such as temporal, linear or higher-order logic. Two special expression constructors merit some discussion. The expression $\texttt{sel}\ (e_1, e_2)$ denotes the contents of memory address e_2 in the memory state denoted by e_1. The expression $\texttt{upd}\ (e_1, e_2, e_3)$ denotes the new memory state obtained from the old state e_1 by updating the location e_2 with the value denoted by e_3.

At the level of predicates we introduce two special predicates for dealing with memory safety. The predicate `saferd` (e_1, e_2) is valid if in the memory state denoted by e_1 it is safe to read from the address denoted by e_2. The predicate `safewr` is used similarly for memory writes, with the extra argument denoting the value being written.

The invariant instruction `INV` P, s requires that predicate P be valid at the corresponding program point and it also declares the maximal set of variables that might be modified on all loop paths that contain this instruction.[1] The instructions `MEMRD` and `MEMWR` are used by the parser to signal to the symbolic evaluator that a memory access instruction was decoded. They can be safely ignored if the safety policy is not concerned with memory safety because their state changing semantics is redundantly expressed as `SET` instructions (see Tab. 2 for examples).

For example, Fig. 4 shows a possible IL representation of the agent of Fig. 3. The loop invariant for the main loop is $j \geq 0$ and the only changed variable in the loop is j.

$$
\begin{array}{lll}
 & \texttt{SET} & j, 0 \\
\texttt{Loop}: & \texttt{INV} & j \geq 0,\ \{j\} \\
 & \texttt{BRANCH} & j < len \rightarrow \texttt{L}_1 \ \square\ j \geq len \rightarrow \texttt{Done} \\
\texttt{L}_1: & \texttt{BRANCH} & \texttt{sel}(mem, tab + 16 \times j) \leq acc \rightarrow \texttt{L}_2 \ \square\ \ldots < acc \rightarrow \texttt{Next} \\
\texttt{L}_2: & \texttt{MEMRD} & tab + 16 \times j + 12 \\
 & \texttt{SET} & p, \texttt{sel}(mem, tab + 16 \times j + 12) \\
 & \ldots & \\
\texttt{Next}: & \texttt{SET} & j, j + 1 \\
 & \texttt{BRANCH} & \texttt{true} \rightarrow \texttt{Loop}
\end{array}
$$

Fig. 4. The intermediate language representation for the agent shown in Fig. 3.

Example: Parsing DEC Alpha Machine Code. As an example we describe here a parser from a subset of DEC Alpha machine code to the intermediate language IL. The set of variables in this case are the 32 machine registers of the DEC Alpha plus the special memory variable *mem*. In order to simplify the resulting safety predicate we might want to let the parser interpret the spill area of the stack frame as an extension of the register file.[2] For this purpose we extend the set of IL variables with $f_0, \ldots, f_{F-1}$ where F is a limit we impose on the number of spill slots. In order to keep the parser simple, we require the

[1] The set of modified variables is an optimization allowing the symbolic evaluator to construct smaller safety predicates. If it is missing then it is conservatively approximated with the set of all variables in scope.

[2] To avoid the danger of aliasing in the spill area the safety policy must ensure that this area is not declared as "safe-to-access" by arbitrary memory operations.

DEC Alpha	*IL*	*Observations*
start of function	`SET` sp_0, sp `SET` ra_0, ra	At the start of function save the values of the stack pointer and return address.
`addl` r_1, r_2, r_d	`SET` $r_d, r_1 + r_2$	Most arithmetic and logical instructions are done similarly.
`ANN_CALL` (f) `jsr` $ra, (pv)$	`ASSERT` $sp = sp_0 - fsz$ `ASSERT` $pv = f$ `CALL` f `SET` $ra, pc + 4$	Require a `CALL` annotation. Emit checks for the stack pointer and the correctness of the annotation. The return address register is changed by the call.
`jsr` $zero, (ra)$	`ASSERT` $sp = sp_0$ `ASSERT` $ra = ra_0$ `RET`	On return, verify the stack pointer and the return address.
`ANN_INV` (P, s)	`INV` P, s	Invariant annotations are propagated unchanged.
`ldl` $r_d, n(sp)$	`ASSERT` $sp = sp_0 - fsz$ `SET` r_d, f_j	Check that $0 <= n < fsz$ and $n \bmod 4 = 0$. Let $j = n/4$. Emit check for the stack pointer.
`ldl` $r_d, n(r_b)$	`MEMRD` $r_b + n$ `SET` r_d, `sel` $(mem, r_b + n)$	For other load instructions signal the read and its effect of the state.
`stl` $r_s, n(sp)$	`ASSERT` $sp = sp_0 - fsz$ `SET` f_j, r_s	Check that $0 <= n < fsz$ and $n \bmod 4 = 0$. Let $j = n/4$. Emit check for the stack pointer.
`stl` $r_s, n(r_b)$	`MEMWR` $r_b + n, r_s$ `SET` mem, `upd` $(mem, r_b + n, r_s)$	For other store instructions signal the write and its effect on the memory state.
`beq` r_s, n	`BRANCH` $r_s = 0 \rightarrow L(pc + n + 4)$ $\Box\ r_s \neq 0 \rightarrow L(pc + 4)$	L is a mapping from DEC Alpha machine code addresses to indices within the stream of IL instructions.

Table 2. Parser for DEC Alpha machine code. The current function being parsed is declared to use a frame of size *fsz*. At each line, *pc* is the machine code index of the DEC Alpha instructions.

untrusted code to declare its frame size *fsz*. In the same spirit of simplicity, only memory accesses through the register *sp* will be interpreted as accesses to the stack. All other accesses are treated as ordinary memory accesses.

For the purpose of checking procedure calls and returns we define two other special variables, sp_0 and ra_0, that are used to keep the initial values of the stack pointer and return address registers. The DEC Alpha has only indirect procedure calls that are difficult to translate to the IL call syntax, which requires an immediate label. This information gap is bridged by requiring, in the position immediately preceding the call instruction, a call-target annotation `ANN_CALL` that declares the actual call target.

Table 2 shows the definition of the parser for the DEC Alpha machine code as a mapping from sequences of DEC Alpha machine instructions and annotations to sequences of IL instructions. Each line in the table is assumed to occur at index pc in the machine code. For convenience we assume that annotations are in-lined in the machine code. In practice, the actual implementation stores the annotations off-line in the data segment.

4.4 The Symbolic Evaluator

The symbolic evaluator executes the intermediate code produced by the parser. As opposed to a concrete evaluator, it does not actually perform the basic operations of the IL, but instead it computes the result as a symbolic expression. The symbolic evaluator can be implemented as a linear pass through the code because all loops are required to have an invariant.

The output of the symbolic evaluator is the safety predicate, which consists mainly of verification conditions. A verification condition is emitted, for example, whenever the symbolic evaluator encounters memory operations. Besides verification conditions, the symbolic evaluator also emits predicates corresponding to taken branches and invariants reflecting the control structure of the code. In many respects, the safety predicate is an expression of that part of the operational semantics of the untrusted code that is relevant to the safety policy.

In order to define the symbolic evaluator, we introduce some notation. The mapping Π associates function labels to triplets containing a precondition, a postcondition and a set of modified global variables. We write $\Pi_f = (Pre, Post, s)$ when function f is declared with the precondition Pre, postcondition $Post$ and the set of modified global variables s.

The state of the symbolic evaluator consists of the current index i in the IL instruction stream and a partial mapping from variables to symbolic expressions $\rho \in \mathit{VarState} = \mathit{Vars} \rightarrow \mathit{Expr}$. We write $\rho[x \leftarrow e]$ to denote the state obtained from ρ by setting the variable x to e and we write $\rho(e)$ to denote the expression resulting from the substitution of variables in e with their values in ρ. We extend this substitution notation to predicates.

For the evaluation of the invariant instructions, the symbolic evaluator keeps track of the invariants seen so far on the path from the start to the current instruction. For each such invariant, the symbolic evaluator also remembers the execution state at the time the invariant was encountered. This is accomplished with a mapping $\mathcal{L}$ from instruction indices (labels) to states $\mathcal{L} \in \mathit{Loops} = \mathit{Label} \rightarrow \mathit{VarState}$, such that at any moment during symbolic execution $\mathit{Dom}(\mathcal{L})$ is the set of invariants on the current path from the start. The symbolic evaluator is also parameterized by the current function being evaluated and the mapping Π, although we shall often omit these subscripts:

$$\mathit{SE}_{f,\Pi} \in (\mathit{Label} \times \mathit{VarState} \times \mathit{Loops}) \rightarrow \mathit{Pred}$$

To simplify the presentation of the evaluator we assume that prior to the evaluation we prepend for the IL representation of each function f the instruction `INV` Pre, s, where Pre and s are the precondition and the set of modified registers of f.

$SE(i+1, \rho[x \leftarrow \rho(e)], \mathcal{L})$	$\mathtt{if}\ IL_i = \mathtt{SET}\ x, e$
$\rho(P) \wedge SE(i+1, \rho, \mathcal{L})$	$\mathtt{if}\ IL_i = \mathtt{ASSERT}\ P$
$(\rho(P_1) \supset SE(i_1, \rho, \mathcal{L})) \wedge (\rho(P_2) \supset SE(i_2, \rho, \mathcal{L}))$	$\mathtt{if}\ IL_i = \mathtt{BRANCH}\ \ P_1 \rightarrow i_1 \Box P_2 \rightarrow i_2$ $i_1 < i \supset IL_{i_1} = \mathtt{INV}$ $i_2 < i \supset IL_{i_2} = \mathtt{INV}$
$\mathtt{saferd}\ (\rho(m), \rho(e)) \wedge SE(i+1, \rho, \mathcal{L})$	$\mathtt{if}\ IL_i = \mathtt{MEMRD}\ e$
$\mathtt{safewr}\ (\rho(m), \rho(e_1), \rho(e_2)) \wedge SE(i+1, \rho, \mathcal{L})$	$\mathtt{if}\ IL_i = \mathtt{MEMWR}\ e_1, e_2$
$\rho(P) \wedge \forall y_1 \ldots y_k . \rho'(P) \supset SE(i+1, \rho', \mathcal{L}[i \leftarrow \rho'])$	$\mathtt{if}\ \ IL_i = \mathtt{INV}\ P, \{x_1, \ldots, x_k\}$ and $i \notin Dom(\mathcal{L})$ $\{y_1, \ldots, y_k\}$ are new variables $\rho' = \rho[x_1 \leftarrow y_1, \ldots, x_k \leftarrow y_k]$
$\rho(P) \wedge \mathtt{checkEq}\ (\rho, \mathcal{L}_i, s)$	$\mathtt{if}\ IL_i = \mathtt{INV}\ P, s$ and $i \in Dom(\mathcal{L})$
$\rho(Pre) \wedge \forall y_1 \ldots y_k . \rho'(Post) \supset SE(i+1, \rho', \mathcal{L})$	$\mathtt{if}\ \ IL_i = \mathtt{CALL}\ l$ $\Pi_l = (Pre, Post, \{x_1, \ldots, x_k\})$ $\{y_1, \ldots, y_k\}$ are new variables $\rho' = \rho[x_1 \leftarrow y_1, \ldots, x_k \leftarrow y_k]$
$\rho(Post) \wedge \mathtt{checkEq}\ (\rho, \mathcal{L}_f, s)$	$\mathtt{if}\ IL_i = \mathtt{RET}\ \ \mathtt{and}\ \Pi_f = (Pre, Post, s)$

Table 3. The definition of $SE_{f,\Pi}(i, \rho, \mathcal{L})$, a symbolic evaluator for generic memory safety.

Table 3 presents the definition of the symbolic evaluator function $SE_{f,\Pi}(i, \rho, \mathcal{L})$ by cases depending on the instruction being evaluated (IL_i). For each kind of instruction there is one case, except for the invariant instructions, which are treated differently the first time when they are encountered.

The evaluation of a `SET` instruction consists of updating the state and continuing with the next instruction. In the case of an assertion the symbolic evaluator emits the asserted predicate with variables substituted according to the current state. For a conditional branch the symbolic evaluator considers both branches recursively and then builds the safety predicate as a conjunction of implications. The left side of each implication is the guard predicate of the branch. This way control flow information is made available for the purpose of proving the verification conditions arising from the evaluation of the branches. For `MEMRD` and `MEMWR` instructions the evaluator emits appropriate verification conditions for the safety of the memory access.

The evaluation of a loop invariant instruction that is encountered for the first time (its index is not in $Dom(\mathcal{L})$) consists of asserting the invariant, then altering the values of the variables that might be changed by the loop and finally processing the loop body in the new state. The invariant predicate is also assumed to hold before considering the loop body and the new state is recorded in $\mathcal{L}$. When the same invariant instruction is encountered again, the evaluator asserts the invariant predicate and checks that variables not declared as modified have not been changed. The verification conditions corresponding to these equality checks are generated by the auxiliary function `checkEq` :

$$\texttt{checkEq}\,(\rho, \rho', s) = \bigwedge_{x \in (Dom(\rho) \cap Dom(\rho')) - s} \rho(x) = \rho'(x)$$

Not surprisingly, the function call and return instructions are processed in a manner similar to the loop invariants, with just a few minor differences. In processing the `RET` instruction the reference state for the equality check is recovered from $\mathcal{L}$. This is possible because the first instruction in each function (at index f) is the invariant instruction `INV` Pre, s.

As an optimization, the evaluator might verify itself some of the simple verification conditions. This has the effect of reducing the safety predicate size and implicitly the proof size. The cost of this optimization is increased code complexity in the symbolic evaluator, and so it must only be employed when the verification conditions are trivial to check. One such case are those variable-equality checks emitted by `checkEq` that are syntactic identities.

Finally, the safety predicate of a function is obtained by evaluating it symbolically starting in a state that maps the global variables[3] and function formal parameters to new variables $y_1, \ldots, y_k$. Then the resulting predicate is quantified over $y_1, \ldots, y_k$:

$$SP_f = \forall y_1 \ldots y_k . SE_{f,\Pi}(f, [g_1 \leftarrow y_1, \ldots, g_k \leftarrow y_k], \{\})$$

As an example we show in Fig. 5 the safety predicate obtained from the symbolic evaluation of the agent whose IL code is shown in Fig. 4. The first three lines of the safety predicate represent, in order: the final quantification, the precondition and the initial invariant. The rest of the predicate originates from the evaluation of the loop body.

$$\begin{array}{l}
\forall mem.\ \forall tab.\ \forall len.\ \forall acc. \\
\quad (\forall i.\ 0 \le i \wedge i < len \supset \texttt{entry}(mem, tab + 16 \times i, acc)) \supset \\
\qquad (0 \ge 0 \wedge \\
\qquad \forall j.\ (j \ge 0 \supset \\
\qquad\qquad (j < len \supset ((\texttt{sel}(mem, tab + 16 \times j) \le acc \supset \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{saferd}(mem, tab + 16 \times j + 12)) \wedge \\
\qquad\qquad\qquad\qquad (\texttt{sel}(mem, tab + 16 \times j) > acc \supset j + 1 \ge 0)) \\
\qquad\qquad (j >= len \supset \texttt{true}))
\end{array}$$

Fig. 5. The safety predicate of the function `main` of Fig. 3.

[3] The special memory variable *mem* is also a global variable.

5 Encoding and Checking Proofs

The next building block of PCC that we describe is the proof checker. Its purpose is to verify that the proof supplied by the untrusted proof producer uses only allowed axioms and inference rules and that it is a proof of the required safety predicate and not another one. In order to isolate the dependencies on the safety policy we have built a generic proof checker parameterized by a configuration file. The proof checker itself does not contain *any* details of the safety policy, not even of the logic being used. All such details are segregated to the configuration file.

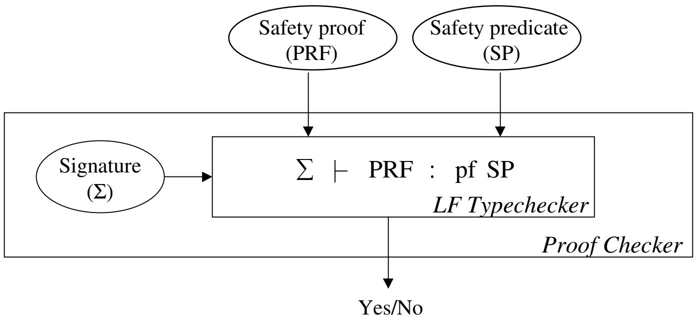

Fig. 6. The structure of the proof checker

To achieve the independence on the safety policy and logic we are currently encoding both the safety predicates and their proofs in the *Edinburgh Logical Framework* [10] (LF), which was specifically designed as a metalanguage for high-level specification of logics. In the rest of this section we only give a brief overview of LF and its use for PCC. The reader interested in a more comprehensive discussion of the subject, including numerous implementation details, should consult [12].

The Logical Framework is a very simple typed language (dependent-typed λ-calculus) with four expression constructors (variables, constants, functions and function application), and a similar set of type constructors:

$$\begin{array}{llcl} \text{Types} & A & ::= & a \mid A\ M \mid \Pi x\!:\!A_1.A_2 \\ \text{Objects} & M & ::= & c \mid x \mid M_1 M_2 \mid \lambda x\!:\!A.M \end{array}$$

To encode the syntax of a particular logic (the predicates) and its semantics (the axioms and inference rules) the safety policy declares a set of LF expression constants together with their types. We refer to this set of constant declarations as the *LF signature* Σ that defines the logic. Fig. 7 shows a small fragment of the signature that defines the first-order predicate logic with integers. The first two lines declare the constant `0` to be an expression (the representation of the numeral 0), and `plus` to be a binary constructor. If M_1 and M_2 are the LF representations of respectively e_1 and e_2, then the LF object "plus M_1 M_2" is the LF representation of $e_1 + e_2$. The middle section of the figure shows some predicate constructors.

The particular feature that makes LF an excellent choice for proof checking is the richness of the type system, and in particular the power to encode predicates as LF types. If P is a predicate then pf P is the type of proofs of P in our logic. In the bottom third of Fig. 7 we show the declaration of two proof constructors. true_i is a nullary proof constructor (a proof constant) that represents the axiom that the predicate true is always valid. The proof constructor mp is used to represent the "modus ponens" proof of a predicate. For example, if M_1 and M_2 are the LF representations of the proofs of $P_1 \supset P_2$ and respectively P_1 then the LF object "mp P_1 P_2 M_1 M_2" is the LF representation of a proof of the predicate P_2.

$$
\begin{array}{ll}
0 & : \mathtt{exp} \\
\mathtt{plus} & : \mathtt{exp} \rightarrow \mathtt{exp} \rightarrow \mathtt{exp} \\
\ldots & \\
\mathtt{true} & : \mathtt{pred} \\
\mathtt{impl} & : \mathtt{pred} \rightarrow \mathtt{pred} \rightarrow \mathtt{pred} \\
= & : \mathtt{exp} \rightarrow \mathtt{exp} \rightarrow \mathtt{pred} \\
\ldots & \\
\mathtt{true\_i} & : \mathtt{pf}\ \mathtt{true} \\
\mathtt{mp} & : \Pi p{:}\mathtt{pred}.\Pi r{:}\mathtt{pred}.\mathtt{pf}\ (\mathtt{impl}\ p\ r) \rightarrow \mathtt{pf}\ p \rightarrow \mathtt{pf}\ r
\end{array}
$$

Fig. 7. A fragment of the LF signature describing the syntax and semantics of first-order logic.

The actual proof-checking operation is done by verifying that the proof object *PRF* has type pf *SP*, where *SP* is the safety predicate of interest. This LF typechecking operation, written as $\vdash_{\Sigma} PRF : \mathtt{pf}\ SP$, verifies both the fact that the proof only contains the proof constants declared in Σ, and that it proves the right predicate.

An important feature of LF type-checking is that it is very simple and can be completely described by fifteen inference rules and implemented is less than five pages of C code [12]. Furthermore, because the LF type checker is completely independent of the particular logic being used by the safety policy, we can reuse it for checking proofs in other logics. A standing proof of reusability is that in all our past and present experiments with PCC we have used the same unmodified implementation of LF type-checking with safety policies ranging from memory safety and type safety to termination and resource usage bounds. Furthermore, because LF type-checking is simple and abstract, it is possible to prove formally its adequacy for proof checking [10, 12].

In our PCC implementation the LF signatures are expressed in proof-checker configuration files using a format that is virtually identical to the one used in

Fig. 7. To further increase our confidence in the proof-checking infrastructure, the configuration file must itself pass the LF type-checker before being used for type-checking proofs.

6 Generating Proofs for PCC

The safety predicates are expressed using first-order logic, with all the language and machine details either abstracted or modeled in the logic, so the proof generator must be a theorem prover for a fragment of first-order logic. For first-order logic, many theorem-proving systems have been implemented [2, 4, 5, 7, 9, 14]. To our knowledge, all of these are able to prove typical safety predicates, sometimes with the help of additional tactics that might be provided by the code consumer. However, for some safety properties, automatic decision procedures do not exist or are not effective. In such cases it is more practical to use a semi-interactive theorem prover guided by a person with a deep understanding of the reasons underlying the safety of the untrusted code.

To be usable as a PCC proof producer, a theorem prover must not only be able to prove safety predicates but must be also capable of generating detailed proof of them. Furthermore these proofs must be expressed in the particular logic (i.e., using the axioms and inference rules specified as part of the safety policy) used by the code consumer. The major difficulty here is to make the theorem prover output the proof, because once we have all the proof details, it is generally easy to transform them in the format expected by the consumer.

In our implementations of PCC we have used two different theorem provers so far. The first, and most primitive, theorem prover that we used was developed using the Elf [15] implementation of LF. The Elf system is able to read an LF signature describing a logic and then answer queries of the form: *Is there any LF object M having type* `pf` *P?* If P is the representation of the predicate that we want to prove and the answer to the query is yes, then M is a representation of the proof, and by construction it is a valid proof. So basically, the theorem prover consists of the Elf system together with the LF signature part of the safety policy.

The major problem with the Elf approach to theorem proving is that Elf uses a very simple search algorithm that is inappropriate for many logics. In some cases, mostly having to do with integer arithmetic, we had to add redundant inference rules to the safety policy so that Elf can search for a proof.

Lately we have switched to our own implementation of a theorem prover based on the Nelson-Oppen architecture for cooperating decision procedures [13], also implemented in the Stanford Pascal Verifier [6] and the Extended Static Checking [7] systems. The distinguishing feature of our implementation is that it outputs an LF representation of the proof of successfully proved predicates.

The theorem prover uses several decision procedures, the most notable ones being Simplex, for deciding linear inequalities, and the congruence closure, for deciding equalities. In addition it also incorporates a decision procedure for modular arithmetic and a simple matcher. The theorem prover is a complicated system implementing complex algorithms, but we do not have to rely on its soundness. We just have to check every proof it outputs. In fact, by doing so we were able to discover subtle bugs in the theorem prover. With this theorem prover we are currently able to prove completely automatically most safety predicates arising from our PCC experiments.

7 Controlling Resource Usage with PCC

Previous sections describe Proof-Carrying Code and an example of its use for certifying the memory safety and the conformance with a simple access protection scheme for a simple shopping agent. In this section we expand the safety policy to allow communication from the agent to its parent host and also to restrict the agent's use of CPU cycles and network bandwidth. Then we consider an agent that exploits this extended policy to gather and sends to its parent host all entries satisfying an arbitrary predicate.

7.1 Extending the Safety Policy with Resource Usage Bounds

In order to control the total execution time of an agent we extend VCGen to perform a simple timing analysis of the agent's code. To keep things simple we use instruction counts as estimates for execution time and we define a pseudo-variable *icount* that is incremented by one for every instruction that is being parsed.[4] Then we change the parser to prepend a "SET $icount, icount + 1$" instruction to the translation of every instruction from the agent's code. For example, the translation of an addition instruction from the DEC Alpha machine language to IL is now described as follows:

$$\begin{array}{lll} \texttt{addl}\ r_1, r_2, r_d & \longrightarrow & \texttt{SET}\ icount, icount + 1 \\ & & \texttt{SET}\ r_d, r_1 + r_2 \end{array}$$

The instruction count variable can be referred to in preconditions and postconditions in order to specify timing constraints. For example, to limit the execution of the agent to maximum MAXRUN instructions we add the predicate $icount \leq icount^0 + \texttt{MAXRUN}$ to the agent's postcondition. (The superscript 0 on a variable refers to its value at function entry.) But this is not enough because the symbolic evaluator of Sect. 4 enforces only partial correctness, and thus the postcondition is enforced only in the event the function terminates. Total correctness is important not only for termination but for a large class of program properties, usually referred to as *liveness* properties, that require the occurrence of certain events during the execution (as opposed to safety properties that prevent the occurrence of certain events).

Fortunately, with only a minor change to the symbolic evaluator we can also enforce total correctness. If the symbolic evaluator automatically adds the

[4] A more precise estimate of the execution time can be obtained by increasing the increment amounts for instructions that are likely to take more time to complete.

predicate $icount \leq icount^0 + \texttt{MAXRUN}$ to every loop invariant and function postcondition, then a valid safety predicate ensures the termination of every loop or function, and therefore the postconditions can be used to express total correctness properties.[5]

Next, we extend the safety policy to allow communication between an agent and its parent host. For this purpose the host **H** supplies the function `sendBack`, which the agent can invoke with a memory address and the number of words to be sent. The precondition of `sendBack` states that the entire memory range referred to by the actual arguments is agent readable. To address the resource usage issues arising during the communication, the safety policy limits both the memory and network bandwidth usage. This is achieved in a conservative manner by limiting the size of a single message and the timing between successive invocations. Concretely, the safety policy requires that the length argument passed to `sendBack` is less that `MAXSEND` and that at least $\texttt{MINWAIT} \times len$ instructions have to have passed since the previous communication before a message of length len can be sent. This conservatively limits the memory buffer usage to `MAXSEND` and the network bandwidth used to $\texttt{Freq}/(\texttt{MINWAIT} \times \texttt{CPI})$, where `Freq` is the processor's clock frequency and `CPI` is the average number of cycles-per-instruction for the agent execution.

The implementation of the `sendBack` function has to be customized for every individual agent, for example with the address of the parent host. A simple way to do this is for the code consumer to prepare a closure data structure containing the customization information, and to give pointer to it to the `main` function of the untrusted code. Then the agent is required to use this closure every time it invokes `sendBack`. The safety issue that arises is how can the safety policy ensure that the untrusted agent does not invoke `sendBack` with a phony or tampered closure argument? The answer is inspired by abstract types. We define an abstract type `closureSB` with no constructors and we require that one argument to the `sendBack` function have this type. The untrusted code must then prove that the type of the actual argument to the `sendBack` function has the abstract closure type, and the only way it can do that is by passing along the closure that was supplied by the host.

Finally, we extend the safety policy with a host-provided function `sleep` that the agent can use to delay its execution for the equivalent of a number of instructions passed as an argument.

Having described the safety policy informally, we proceed now with its formalization in first-order logic. We start with the simpler task of formalizing the abstract closure type for `sendBack` and continue with the formalization of the bandwidth limitation.

The logic counterpart of an abstract type with no constructors is an uninterpreted unary predicate symbol with no introduction axioms. Thus we introduce

[5] Programs that are intended not to terminate but which must have certain liveness properties can be structured as a function that is invoked repeatedly by the host. The liveness properties are then verified on this function.

the predicate $\texttt{closureSB}(x)$ to denote that the expression x is a valid closure for the `sendBack` function.

To formalize the bandwidth limitation we consider the more general problem of restricting the timing between various events during execution. For our particular example we use two event constructors: `start`, to denote the start of agent execution, and `send`, to denote a communication event. Then we define a pseudo-variable *log* that is used to keep a log of events and their occurrence times. This is a global variable and, just like the memory and instruction count pseudo-variables, it is implicitly passed to, and potentially modified by every function. Valid values of the *log* variable are expressions of the form $\texttt{new}(l, e, c)$, denoting that a new event e occurred when the instruction count was c in the state of the log denoted by l.

In order to manipulate the event occurrence times we define the expression $\texttt{timeOf}(l, e)$ to denote, in a log state l, the value of the instruction count at the last occurrence of event e. If the event e never occurred, the function is undefined. This meaning of `timeOf` is expressed using the following axioms:

$$\frac{}{\texttt{timeOf}(\texttt{new}(l,e,c),e) = c} \qquad \frac{e \neq e'}{\texttt{timeOf}(\texttt{new}(l,e,c),e') = \texttt{timeOf}(l,e')} \qquad \frac{}{\texttt{send} \neq \texttt{start}}$$

To conclude the safety policy description we show in Tab. 4 the preconditions and postconditions for the functions involved. Recall that only `main` is untrusted, and as such the precondition of `main` and the postconditions of `sendBack` and `sleep` are used as assumptions in the safety predicate; the postcondition of `main` and the preconditions of `sendBack` and `sleep` are actual verification conditions in the safety predicate, along with the verification conditions arising from loop invariants and memory operations.

Function	Precondition	Postcondition
`main`(cl,tb,ln,sl)	$\texttt{closureSB}(cl)$ $\forall i. 0 \leq i \wedge i < ln \supset$ $\texttt{entry}(mem, tb + 16 \times i, sl)$ $\texttt{timeOf}(log, \texttt{start}) = icount$ $\texttt{timeOf}(log, \texttt{send}) = icount$	$icount \leq icount^0 + \texttt{MAXRUN}$
`sendBack`(cl,dt,ln)	$\texttt{closureSB}(cl)$ $\forall i. 0 \leq i \wedge i < ln \supset$ $\texttt{safeRd}(mem, dt + 4 \times i)$ $ln \leq \texttt{MAXSEND}$ $icount - \texttt{timeOf}(log, \texttt{send}) \geq$ $ln \times \texttt{MINWAIT}$	$icount \leq icount^0 + \texttt{MAXSB}$ $icount \geq icount^0 + \texttt{MINSB}$ $log = \texttt{new}(log^0, \texttt{send}, icount)$
`sleep`(c)	$c \geq 0$	$icount = icount^0 + c$

Table 4. The extended travel agency safety policy. Each precondition and postcondition is shown as a list of conjuncts. The pseudo-variables *mem*, *log* and *icount* are considered implicit inputs and outputs of all functions.

```
1  void main(cl, tab, len, acc) {
2   timesend = 4 * MINWAIT; timeout = MAXRUN - 6; i = 0;
3   while(true) {
4     if(timeout < MAXLOOP) return;
5     if(i >= len) return;
6     i+=; timeout -= 8 + MAXFIL; timesend -= 8 + MINFIL;
7     if(tab[i-1].access > acc) continue;
8     if(!filter(&tab[i-1])) continue;
9     if(timesend >= 0) { sleep(timesend); timeout -= 2 + timesend;}
10    sendBack(cl, & tab[i-1], 16);
11    timeout -= (4 + MAXSB); timesend = 4 * MINWAIT;
12 }}
```

Fig. 8. The skeleton of a shopping agent that attempts to send back to its parent host all pricing table entries that match a certain predicate.

7.2 Extending the Shopping Agent

We now swap the travel-agency administrator's hat with that of the code producer's and we design a shopping agent that uses an arbitrary predicate to select database records to be sent to the parent host. Then we discuss a few key points in proving the safety predicate for the new agent and finally, we describe our experimental results gathered from the actual implementation of the agents.

For the purpose of this paper we stick with the agent design that leads to a simpler proof of safety. Our agent, whose main function is shown in Fig. 8, keeps track explicitly of the instruction counts by using the variables `timeout`, which conservatively estimates the number of instructions that are still available for execution, and `timesend`, which conservatively estimates the number of instructions that must pass before a new communication can be initiated. The constants `MINSB`, `MAXSB`, `MINFIL` and `MAXFIL` are the minimum and maximum number of instructions required for the execution of the functions `sendBack` and `filter` respectively. The decrement operations from line 6 account for the instructions in lines 3–8, while those from line 11 account for the program lines 9–11. Note that the variable `timeout` is always decremented by the maximum possible execution time, while `timesend` is decremented by the minimum. The purpose of the constant 6 in the initialization of `timeout` is to account for the loop preamble and exit (lines 2 and 4). The loop terminates when all the entries have been scanned (line 5), or when not enough instruction are left to perform one more iteration through the loop (line 4). The constant `MAXLOOP` is a conservative estimate of the number of instructions executed in a loop iteration ($\mathtt{MAXLOOP} = \mathtt{MAXFIL} + \mathtt{MAXSB} + 4 \times \mathtt{MINWAIT}$) assuming a maximum length wait has to be performed in line 10.

We conclude the presentation of the new agent with the loop invariant of the main loop:

$$\begin{aligned} Inv = {} & i \geq 0 \wedge \mathtt{timeout} \geq 0 \quad \wedge \\ & \mathtt{MAXRUN} - 3 - \mathtt{timeout} \geq icount - \mathtt{timeOf}(log, \mathtt{start}) \quad \wedge \\ & 0 \leq 4 \times \mathtt{MINWAIT} - \mathtt{timesend} \quad \wedge \\ & 4 \times \mathtt{MINWAIT} - \mathtt{timesend} \leq icount - \mathtt{timeOf}(log, \mathtt{send}) \end{aligned}$$

The first conjunct of the invariant is inherited from the simple agent presented before. The third conjunct specifies that $\mathtt{MAXRUN} - 3 - \mathtt{timeout}$ is a conservative estimate of the total number of instruction executed (three instructions are being subtracted to allow for the time-out return of line 4). The last two conjuncts of the invariant claim than $4 \times \mathtt{MINWAIT} - \mathtt{timesend}$ is always positive and a conservative estimate of the number of instructions executed since the last communication operation. The modified symbolic evaluator is adding to the above invariant the predicate $icount - icount^0 \leq \mathtt{MAXRUN}$. This additional conjunct is weaker than our loop invariant and we ignore it from now on.

7.3 Proving the Safety of the Extended Agent

The safety predicate and the safety proof corresponding to this agent are too large to show here. Instead we only discuss a key point in the proof and then we report the data obtained from the actual implementation.

When the symbolic evaluator encounters the call to `sendBack` (line 10) it emits the verification condition obtained by substituting the symbolic values of variables in the precondition of `sendBack`. From this verification condition we focus on the conjunct that specifies the timing of `sendBack`. This conjunct is shown below the horizontal line in Fig. 9. Line-number subscripts on variables denote the value of the variable right before the execution of the corresponding line. The proof of this conjunct is by cases, depending whether the `sleep` function is called or not. For the case when `sleep` is called, the proof follows by adding the assumptions shown above the horizontal line in Fig. 9. The first assumption is the loop invariant, the second is from the postcondition of `filter` and the third is from the postcondition of `sleep`.

$$\frac{\begin{aligned} icount_4 - \mathtt{timeOf}(log_4, \mathtt{send}) &\geq 4 \times \mathtt{MINWAIT} - \mathtt{timesend}_4 \\ icount_9 &\geq icount_4 + 7 + \mathtt{MINFIL} \\ icount_{10} &= icount_9 + (\mathtt{timesend}_4 - 8 - \mathtt{MINFIL}) + 3 \end{aligned}}{icount_{10} - \mathtt{timeOf}(log_4, \mathtt{send}) \geq 4 \times \mathtt{MINWAIT}}$$

Fig. 9. A fragment of the proof that the precondition of `sendBack` holds at line 10, in the case when the test at line 9 succeeds. Above the line we have assumptions that when added yield the desired conclusion, shown below the line.

The other case of the proof of the precondition of `sendBack`, as well as all the other verification conditions arising from the symbolic evaluation of `main` are proved in a similar manner.

8 Experimental Results

We have implemented the extended safety policy presented in this section in our Proof-Carrying Code system. We constructed a configuration file for VCGen describing the functions involved in the experiment and their preconditions and postconditions. This file is literally a transcription of Tab. 4. We have also created an LF signature describing the first-order logic and the axioms that define the predicates `entry`, and `timeOf`. This is again a literal transcription in the LF syntax of the axioms presented in this paper.

We wrote the agents first in C and then we compiled them to DEC Alpha assembly language. The program for the extended agent had to be then manually edited to adjust for the discrepancy between the number of instructions counted at the C source level and the number of assembly language instructions. We also had to manually add the loop invariant annotations. The resulting programs were submitted to VCGen, which produced a safety predicate for each agent.

We finally proved the safety predicates using a theorem prover developed by us for other applications of PCC. The proof of the safety predicate for the simple agent was done completely automatically. For proving the timing verification conditions for the extended agent we had to customize the theorem prover by providing it with a list of axioms for `timeOf`.

The purpose of the experiments presented here is two-fold. First, we want to show the costs that are specific to PCC, that is the safety predicate generation time, the proof generation time, the proof size and the cost of proof checking. Second, we want to quantify the run-time penalty imposed on agent execution by other techniques that are typically used to enforce the same level of safety. Finally we integrate the data obtained in these two experiments to compute, for the simple agent, the minimum number of table entries such that the cost of safety predicate generation and proof checking is amortized.

The first set of experiments was done to ascertain the costs of proof generation and proof checking. It is important to note that these costs are incurred only once per agent, independently how many times the agent is executed by the consumer. For this experiment we have measured the proof size and the times required for VCGen, for proof generation and for proof checking. To put the proof size into context we also report the machine code size for the agents. As in other experiments with PCC we obtain proofs that are between three and ten times larger than the code, with the larger factors observed for more complex safety policies. These experiments confirm the usual intuition that proof generation is more expensive than proof checking.

The second set of experiments compared agents using PCC to agents implemented in Java [16] and agents isolated using either hardware-memory protection or Software Fault Isolation (SFI) [18]. The comparison is not entirely fair because Java and SFI cannot enforce the database access policy, and thus offer weaker safety guarantees than PCC and hardware-memory protection. All the measurements are done for the implementation of the simple agent on a 175 MHz DEC Alpha running DEC OSF 1.3.

Experiment	Code size (bytes)	Proof size (bytes)	VCGen (ms)	Proof Generation (ms)	Proof Checking (ms)
Simple agent	112	370	0.4	40	1.2
Extended agent	250	2012	1.5	800	12.3

Table 5. The cost of PCC for the two example agents presented in this paper.

For the Java experiment we have embedded the code of Fig. 3 in a simple Java applet containing timing code. Unfortunately, we were not able to find a JIT compiler for the DEC Alpha and thus the Java measurements were done using the bytecode interpreter of Netscape 4.0. [*We are trying to remedy this situation for the final version of the paper.*]

Software Fault Isolation is a technique by which the code consumer inspects the untrusted agent code and inserts instructions for memory-bounds checking before each memory operation. This code inspection process is similar to VCGen. To simulate the effects of SFI, we have instructed the compiler to insert bounds-checking operations for all memory accesses.

To simulate an agent that runs in a hardware-protected memory space, we have modified the agent to invoke a consumer-supplied function that checks the access level and then copies the entry to the agent's memory space. To simulate more accurately the cost of a function call across different protection domains we have inserted an idling loop lasting 50 instructions in the checking function.

Experiment	Running time (us)	Slowdown	Cross-over (# table entries)
Proof-Carrying Code	0.030	1.0x	-
Software Fault Isolation	0.036	1.2x	200,000
Indirect Access to Data	0.280	9.3x	8,400
Java Interpretation	1.230	41.0x	1,200
Java JIT	–	–	–

Table 6. A comparison of the per-table-entry running time of the simple agent of Fig. 3 when the safety policy is enforced using PCC, SFI, Java and consumer-intermediated access to data. The last column, computes the number of table entry to amortize the cost of VCGen and proof checking (1.6ms). See the text for caveats regarding these experiments.

Table 6 shows the running time divided by the number of table entries of the simple agent of Fig. 3 when the safety policy is enforced using PCC, SFI, Java and hardware-memory protection. These times are the average of 100 runs. These results are in-line with those measured in other similar experiments.

In addition to the running time of agents, PCC differs from the other techniques considered here in the one-time cost incurred for VCGen and proof checking. In some cases, especially for short-running agents or for agents that are only executed once, it is more efficient to use one of the other techniques. In the case of our example, this might happen for a small database. The last column in Tab. 6 shows the minimum size of a table for which that the checking time (1.6ms for the simple agent) added to the running time of the PCC agent is smaller than the running time of the agent using a competing technique. In the case of SFI we have considered that the cost of scanning the code and instrumenting it is the same as the cost of VCGen. However, note that only PCC and the Indirect Access method can enforce the desired safety policy.

9 Discussion and Future Work

The most important consideration in the design of the extended agent of Sect. 7.2 is the ultimate requirement that we must prove that it satisfies the safety policy. This imposes a delicate balance between code optimizations and the difficulty of the proof, because the safety predicate for an optimized agent is usually more difficult to prove than for an unoptimized version. The safety policy does not unduly restrict the optimizations that can be applied to an agent, and should we decide to spend more effort for proof generation we can develop more optimized agents.

We were forced to write the extended agent to maintain explicit instruction counts because the safety policy specifies very strict timing requirements. If the safety policy is relaxed to impose an instruction count limit that varies with the size of the database, the timing constraints have to be proved only for the loop body, making it unnecessary to maintain explicit instruction counts at run-time.

The most unpleasant aspect of writing the extended agent is for the programmer to keep track of assembly language instruction counts. We think that it is feasible to write an automatic tool to do this and to insert the appropriate decrement instructions and loop termination tests. For this to be possible we need to find a simpler way to specify timing constraints, maybe as code or typing annotations.

In the direction of automation of PCC we have obtained promising results by using a prototype implementation of an optimizing compiler from a type-safe subset of the C language to DEC Alpha assembly language. Not only is the performance of the resulting code comparable to that of `cc` and `gcc` with all optimizations enabled, but it produces the required annotations and also a proof of type safety and memory safety. Automation was possible in this case because the compiler need only preserve these properties from the source language. Similar technology might be used to certify more complex safety policies if we start with a restricted or a domain specific source language.

The most unpleasant aspect of the experimental results are the proof sizes, which can be an order of magnitude larger than the code, and in certain cases can grow exponentially with the size of the code. This worst case occurs when the program has long sequences of conditionals without intervening loop invariants. For many safety properties however, the size of the proof is linear in the size of the program.

At the moment, we have only scratched the surface of proof representation optimizations that can be applied to reduce the size of the proofs. For example, it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once as lemmas. Also, common subproofs can be identified among proofs from different experiments for the same safety policy. These common parts can be proved once and then assumed as theorems of a given safety policy. Finally, one can apply compression algorithms to the binary representation of proofs. In our superficial experiments with compression we observed a reduction by a factor of 2 in the proof size. We believe that by serious proof optimization, the size of the proofs for memory and type safety will approach the size of the code.

Another direction of future work is in identifying more examples of code properties that can be verified using PCC. The most challenging properties to verify seem to be the liveness properties and those involving dynamic safety requirements. In this direction we have obtained promising results in dealing with locks and memory allocation by extending the model of a log of events presented here. The greatest challenge in this area is the serious difficulty of proving the resulting safety predicates.

Proof-Carrying Code compares favorably with other techniques used to prevent untrusted code to step outside a safety policy. When compared with run-time techniques such as hardware or software memory protection [18] and interpretation [11, 16] the advantage is the run-time performance and the simplicity of the safety-critical infrastructure. Another advantage over run-time checking is that PCC avoids the possibility that the untrusted code must be terminated abruptly because of a run-time error before it has a chance of cleaning-up the modified state. Furthermore, certain safety properties (e.g. compliance with data abstraction boundaries) cannot be checked at run-time without significant penalties.

When compared with approaches based on type-safety [1, 17] the advantage of PCC is the increased expressiveness of first-order logic over traditional type systems, which, for example, cannot express resource usage bounds or the arithmetic properties that enable the elimination of array-bounds checking.

Finally, using Proof-Carrying Code is qualitatively better than using digital signatures for the purpose of certifying the safety of agents, because it does not rely on the assumption that the owner of a particular encryption key writes only well-behaved code.

10 Conclusion

This paper presents the details of Proof-Carrying Code and its use to certify the safety of untrusted code. The safety properties that are explored here are memory safety, compliance with simple data access policies and resource usage bounds, and data abstraction. PCC, however, can be used for any safety and liveness properties that can be expressed in first-order logic.

Proof-Carrying Code has the potential to free the host-system designer from relying on run-time checking as the sole means of ensuring safety. Traditionally, system designers have always viewed safety simply in terms of memory protection, achieved through the use of rather expensive run-time mechanisms such as hardware-enforced memory protection and extensive run-time checking of data. By being limited to memory protection and run-time checking, the designer must impose substantial restrictions on the structure and implementation of the entire system, for example by requiring the use of a very restricted agent-host interaction model (to intermediate the access to critical data and resources, for example).

Proof-Carrying Code, on the other hand, provides greater flexibility for designers of both the host system and then agents, and also allows safety policies to be used that are more abstract and fine-grained than memory protection. We believe that this has the potential to lead to great improvements in the robustness and end-to-end performance of systems.

References

1. Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating System Principles* (Dec. 1995), pp. 267–284.
2. Boyer, R., and Moore, J. S. *A Computational Logic*. Academic Press, 1979.
3. Burstall, R., and Landin, P. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4 (1969).
4. Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
5. Coquand, T., and Huet, G. Constructions: A higher order proof system for mechanizing mathematics. In *Proc. European Conf. on Computer Algebra (EUROCAL'85), LNCS 203* (1985), Springer-Verlag, pp. 151–184.
6. D.C. Luckham, e. Stanford pascal verifier user manual. Tech. Rep. STAN-CS-79-731, Dept. of Computer Science, Stanford Univ., Mar. 1979.
7. Detlefs, D. An overview of the Extended Static Checking system. In *Proceedings of the First Formal Methods in Software Practice Workshop* (1996).
8. Dijkstra, E. W. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM 18* (1975), 453–457.
9. Gordon, M. HOL: A machine oriented formulation of higher-order logic. Tech. Rep. 85, University of Cambridge, Computer Laboratory, July 1985.
10. Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. *Journal of the Association for Computing Machinery 40*, 1 (Jan. 1993), 143–184.
11. McCanne, S. The Berkeley Packet Filter man page. BPF distribution available at `ftp://ftp.ee.lbl.gov`, May 1991.

12. Necula, G. C., and Lee, P. Efficient representation and validation of logical proofs. Technical Report CMU-CS-97-172, Computer Science Department, Carnegie Mellon University, Oct. 1997.
13. Nelson, G., and Oppen, D. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1*, 2 (Oct. 1979), 245–257.
14. Owre, S., Rushby, J. M., and Shankar, N. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)* (Saratoga, NY, June 1992), D. Kapur, Ed., vol. 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 748–752.
15. Pfenning, F. Elf: A meta-language for deductive systems (system description). In *12th International Conference on Automated Deduction* (Nancy, France, June 26–July 1, 1994), A. Bundy, Ed., LNAI 814, Springer-Verlag, pp. 811–815.
16. Sun Microsystems. The Java language specification. Available as `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`, 1995.
17. Sun Microsystems. The Java Virtual Machine specification. Available as `ftp://ftp.javasoft.com/docs/vmspec.ps.zip`, 1995.
18. Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles* (Dec. 1993), ACM, pp. 203–216.

This article was processed using the LaTeX macro package with LLNCS style