Safe Kernel Extensions Without Run-Time Checking

George C. Necula and Peter Lee
Carnegie Mellon University

Safe Kernel Extensions Without Run-Time Checking

George C. Necula Peter Lee

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891

{necula,petel}@cs.cmu.edu

Abstract

This paper describes a mechanism by which an operating system kernel can determine with certainty that it is safe to execute a binary supplied by an untrusted source. The kernel first defines a safety policy and makes it public. Then, using this policy, an application can provide binaries in a special form called *proof-carrying code*, or simply PCC. Each PCC binary contains, in addition to the native code, a formal proof that the code obeys the safety policy. The kernel can easily validate the proof without using cryptography and without consulting any external trusted entities. If the validation succeeds, the code is guaranteed to respect the safety policy without relying on run-time checks.

The main practical difficulty of PCC is in generating the safety proofs. In order to gain some preliminary experience with this, we have written several network packet filters in hand-tuned DEC Alpha assembly language, and then generated PCC binaries for them using a special prototype assembler. The PCC binaries can be executed with no run-time overhead, beyond a one-time cost of 1 to 3 milliseconds for validating the enclosed proofs. The net result is that our packet filters are formally guaranteed to be safe and are faster than packet filters created using Berkeley Packet Filters, Software Fault Isolation, or safe languages such as Modula-3.

1 Introduction

In this paper we address the problem of how an operating-system kernel or a server can determine with absolute certainty that it is safe to execute code supplied by an application or other untrusted source. We propose a mechanism that allows a kernel or server—from now on referred to as the *code consumer*—to define a safety policy and then verify that the policy is respected by native-code binaries supplied to it by an untrusted *code producer*.

In contrast to some previous approaches, we do not rely on the usual authentication or code-editing mechanisms. Instead, we require that the code producer creates its binaries in a special form, which we call *proof-carrying code*, or simply PCC. A PCC binary contains an encoding of a formal proof that the enclosed native code respects the safety policy. The proof is structured in such a way that makes it easy and foolproof for any agent (and in particular, the code consumer) to verify its validity *without* using cryptographic techniques or consulting with external trusted entities; there is also no need for any program analysis, code editing, compilation, or interpretation. Besides being safe, PCC binaries are also extremely fast because the safety check needs to be conducted only once, after which the consumer knows it can safely execute the binary without any further run-time checking.

In a PCC binary, the proof is linked with the native code so that its validity guarantees the code's safety. Furthermore, proof-carrying code is tamperproof; the consumer can easily detect most attempts by any malicious agent to forge a proof or modify the code. Tampering can go undetected only if the adulterated code is *still* guaranteed to respect the consumer-defined safety policy. Another feature of the PCC method is that the proof checking algorithm is very simple, allowing fast and easy-to-trust implementations.

The safety policy is defined and published by the code consumer and comprises a set of proof-formation rules, along with a set of preconditions. Safety policies can be defined to stipulate standard

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.

requirements such as memory safety, as well as more abstract and fine-grained guarantees about the integrity of data-abstraction boundaries. To take a simple example, consider the abstract type of file descriptors. In this case, a client is said to preserve the abstraction boundaries if it does not exploit the fact that file descriptors are represented as integers (by incrementing a file descriptor, for example).

Although we have worked out many of the theoretical underpinnings for PCC (and indeed, most of the theory is based on old and well-known principles from logic, type theory [4, 11], and formal verification [5, 6, 8]), there are many difficult problems that remain to be solved. In particular we do not know at this point the most practical way to generate the proofs. We have thus set out to gain some preliminary experience, both to measure the benefits and to identify the practical problems.

In the experiments reported in this paper, we have in fact achieved fully automatic proof generation. In general, however, this problem is similar to program verification and is not completely automatable. Actually, the problem is somewhat easier than verification because we have the option of inserting extra run-time checks (as is done in Software Fault Isolation), which would have the effect of simplifying the proving process at the cost of reducing performance. By "extra", we mean run-time checks that are not intrinsically a part of the algorithm of the extension code. (For example, SFI will actually edit the code and insert "extra" checks; PCC does not normally do this.) Fortunately, we have not yet had any need or desire to insert extra run-time checks in any of our PCC examples. Still, automation of proof generation remains as one of the most serious obstacles to widespread practical application of PCC.

In our main experiment, we implemented several network packet filters [12, 15] in DEC Alpha assembly language [19] and then used a special prototype assembler to create PCC binaries for them. We were motivated to use an unsafe assembly language in order to place equal emphasis on both performance and safety, as well as to demonstrate the generality of the PCC approach. In addition to the assembler, we implemented a proof validator that accepts a PCC binary, checks its safety proof, and if it is found to be valid, loads the enclosed native code and sets it up for execution.

The results of this and other experiments are encouraging. For our collection of packet filters, we are able to automate completely the generation of the PCC binaries. The one-time cost of loading and checking the validity of the safety proofs is between 1 and 3 milliseconds. Because a safety proof guarantees safety, our hand-tuned packet filters can be executed safely in the kernel address space without adding any run-time checks. Predictably, they are much faster than safe packet filters produced by any other means with which we are familiar.

We believe that our early results show that proof-carrying code is a new point in the design space that is worthy of further attention and study. This paper presents an overview of the approach. We begin with a brief overview of the process of generating and validating the safety proofs. Then, we make this more concrete by showing how a safety policy can be defined and proofs created for a generic assembly language. This is followed by a description of our main experiment involving safe network packet filters. The benchmark results provide some preliminary indication that the PCC methodology has the potential to surpass traditional approaches from a safety point of view while maintaining or improving performance. In particular, we show that PCC leads to faster and safer packet filters than previous approaches to code safety in systems software, including Berkeley Packet Filters [12], Software Fault Isolation [23], and programming in the safe subset of Modula-3 [1, 9, 17]. Finally, we conclude with a discussion of the remaining difficulties and speculate on what might be necessary to make the approach work on a practical scale.

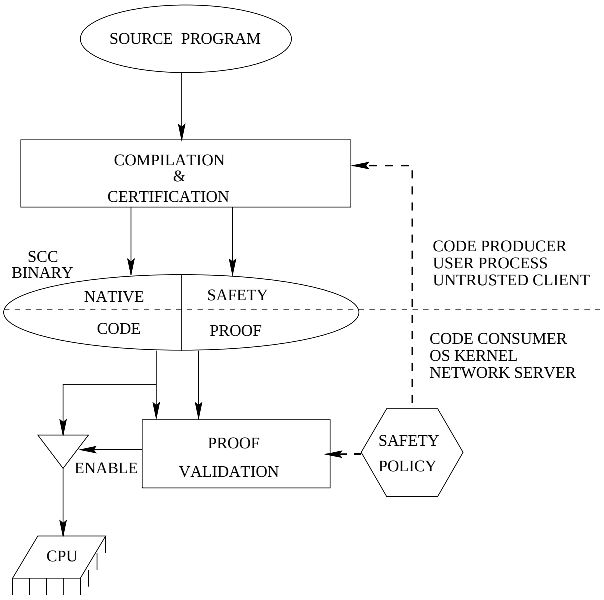

Figure 1: Overview of Proof-Carrying Code.

2 Proof-Carrying Code

Figure 1 depicts the process of generating and using a PCC binary. The process begins with the code consumer defining and publicizing a safety policy. This policy defines formally what is meant by "safety" and also specifies the interface between the consumer and any binary provided by the producer. Taking the policy into account, the code producer compiles (or assembles) and proves the safety of a source program, through a process which we call certification. This results in a PCC binary that can be delivered to the code consumer. Upon receipt, the consumer validates the safety proof enclosed in the PCC binary. Finally, if the proof is found to be valid, the code consumer can safely execute the native-code part of the PCC binary.

The following subsections describe each of these phases in more detail. The whole process is based on concepts from logic, semantics, and type theory, and so the rest of this section is necessarily somewhat technical, with most details beyond the scope of this paper. We will thus attempt to explain only the basic technicalities and key intuitions here. Those readers who would like more details on the underlying theory can find them in a separate technical report [16]. The impatient reader may want to skip ahead to Section 3 where we show, for the case of network packet filters, that proof-carrying code surpasses previous approaches in both safety and performance.

2.1 Defining a Safety Policy

The first order of business is to define precisely what constitutes safe code behavior. We do this by specifying a *safety policy* in three parts:

1. A Floyd-style *verification-condition generator* (also referred to as the VC generator) [6], which is a procedure that computes a predicate in first-order logic based on the code to be certified. We will refer to this predicate as the *safety predicate.*

2. A set of axioms that can be used to validate the safety predicate.

3. The *precondition*, which is essentially a "calling convention" that defines how the code consumer will invoke the PCC binaries.

It is the job of the designer of the code consumer (e.g., the operating system designer) to define the safety policy. In practice, several different safety policies might be used, each one tailored to the needs of specific tasks or services.

We obtain the VC generator by first specifying an *abstract machine* (also called the *operational semantics*), that simulates the execution of safe programs. The abstract machine is not strictly required but it simplifies the design of the safety policy and provides a basis for proving the soundness of the whole approach.

In order to make all of this more concrete, we will now present an example of an abstract machine that specifies a general form of memory safety for the DEC Alpha processor, and then show how the safety policy of a simple resource access service can be defined by a precondition. The VC generator and axioms will then be given in the next subsection.

An abstract machine for memory-safe DEC Alpha machine code

Because the experiments in this paper use the DEC Alpha assembly language, our abstract machine is essentially a high-level formal description of the Alpha architecture [19]. To see how this is done, consider the subset of the Alpha instruction set shown in Figure 2. (Actually, we use a larger subset of the DEC Alpha assembly language in our experiments, but this smaller subset will suffice for presentation purposes.) In this table, n denotes an integer constant and $\mathbf{r}_i$ refers to machine register i. All instructions operate on 64-bit values. For simplicity we allow the use of only 11 temporary and caller-save machine registers (which, for the purpose of this presentation, we rename $\mathbf{r}_0$ through $\mathbf{r}_{10}$). The consequence of this is that programs cannot write into reserved and callee-save registers (according to the standard C calling convention for the DEC Alpha architecture), and are thus trivially safe with respect to these registers.

To define how programs are executed, we define an abstract machine as a state-transition function, the essential core of which is shown in Figure 3. In this specification, the DEC Alpha program is a vector of instructions, Π, and the current instruction is Π_{pc}, where pc is the program counter. The variable ρ denotes the state of the machine registers and memory. The state-transition function maps a machine state (ρ, pc) into a new state (ρ', pc') by executing the current instruction Π_{pc}.

The notation $\rho[\mathbf{r}_i]$ (often abbreviated as $\mathbf{r}_i$) refers to the value of register $\mathbf{r}_i$ in state ρ.[1] The expres-

[1] Valid register values are positive integers in the range 0 to $2^{64} - 1$. This constraint is expressed formally by the equation "$\mathbf{r}_i \bmod 2^{64} = \mathbf{r}_i$", which is applied to all register val-

$$
\begin{array}{llll}
op & ::= & n \mid \mathbf{r}_i & i \in 0 \ldots 10 \\
al & ::= & \text{ADDQ} \mid \text{SUBQ} \mid \text{AND} \mid \text{OR} \mid \text{SLL} \mid \text{SRL} & \\
br & ::= & \text{BEQ} \mid \text{BNE} \mid \text{BGE} \mid \text{BLT} & \\
instr & ::= & \text{LDQ } \mathbf{r}_d, n(\mathbf{r}_s) \mid \text{STQ } \mathbf{r}_s, n(\mathbf{r}_d) \mid al\ \mathbf{r}_s, op, \mathbf{r}_d \mid br\ \mathbf{r}_s, n \mid \text{RET} &
\end{array}
$$

Figure 2: The subset of DEC Alpha assembly language.

sion $\rho[\mathbf{r}_d \leftarrow \mathbf{r}_d \oplus 1]$ denotes the new state obtained from state ρ by incrementing the value of register $\mathbf{r}_d$. So, for example, the Alpha "ADDQ $\mathbf{r}_s, op, \mathbf{r}_d$" instruction is defined by Figure 3 to have the following semantics:

$$(\rho[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], pc + 1)$$

where ρ is the current register and memory state. This specification states that the ADDQ instruction updates register $\mathbf{r}_d$ with the sum of $\mathbf{r}_s$ and op, and also increments the program counter. We use the "circled" operation $\oplus$ to denote two's-complement addition on 64 bits. This operation is defined in terms of the usual integer arithmetic operations as

$$e_1 \oplus e_2 = (e_1 + e_2) \bmod 2^{64}$$

To model the state of memory, we use a pseudo register, called $\mathbf{r_m}$, that gives the content of each memory location. We write $\mathbf{sel}(\mathbf{r_m}, a)$ for the contents of memory address a, and $\mathbf{upd}(\mathbf{r_m}, a, \mathbf{r}_s)$ for the new memory state resulted from writing register $\mathbf{r}_s$ to address a. Memory operations work on 64-bits and the addresses involved must be aligned on an 8-byte boundary.

In the definition of the load and store instructions, there is a crucial difference between the DEC Alpha processor and our abstract machine. The difference is that our abstract machine performs the safety checks that are shown in boxes in Figure 3. For example, consider the definition of the "LDQ $\mathbf{r}_d, n(\mathbf{r}_s)$" instruction:

$$(\rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], pc + 1), \ \textit{if} \ \boxed{\mathbf{rd}(\mathbf{r}_s \oplus n)}$$

The predicate $\mathbf{rd}(a)$ is true when it is safe to read the word at memory address a, which for the DEC Alpha implies that a is aligned on an 8-byte boundary. Similarly, the predicate $\mathbf{wr}(a)$ is true when the address a denotes an aligned location that can be safely read or written. In essence, these checks define what is meant by safety, and more specifically for this example, memory safety. For the purpose of this paper, the predicates $\mathbf{rd}(a)$ and $\mathbf{wr}(a)$ are defined by the safety policy through the precondition, as shown in the next subsection.

ues. Negative values are represented using two's-complement representation.

Mathematically, the abstract machine does not return errors when a $\mathbf{rd}(a)$ or $\mathbf{wr}(a)$ check fails. Instead, the execution blocks because there are no transition rules covering the error cases. In this setting, a program is safe if and only if it runs without blocking on the abstract machine. Of course, the presence of these safety checks means that the abstract machine is not a faithful abstraction of the DEC Alpha processor. However, the purpose of certification is to prove that all safety checks always succeed. If we have a valid safety proof for a program, we know that we can safely execute it on a real DEC Alpha and get the same behavior as on our abstract machine, even though the Alpha does not implement the safety checks.

There are other notable differences between our abstract machine and a real DEC Alpha. For example, to simplify the presentation in this paper, we have restricted all branches to be only forward. Allowing backward branches and loops introduces a number of complications, but is handled in a conceptually straightforward manner through the addition of explicit loop invariants. As it turns out, the packet filter examples we use in our experiments do not have any loops, and so it is not inconvenient to eliminate them here. In a later section we will briefly describe our experiments with looping programs, including a safe IP-header checksum routine.

Another interesting aspect of the abstract machine is the level of abstraction of our specification. We might try to be ambitious and make a complete specification of the DEC Alpha processor. However, this would be extremely complex and probably difficult to trust. And, as a practical matter, for specific tasks such as the ones we are considering, many details and features of the Alpha are irrelevant. This justifies working at a higher level of abstraction above the details of the pipeline, cache, timing, and interrupt behavior.

We can also consider encoding other kinds of safety checks into our abstract machine. For the sake of simplicity, we have specified only a notion of fine-grained memory safety. With some ingenuity, an abstract machine designer can define safety policies involving other kinds of safety, like control over resource usage or preservation of data-abstraction

$$(\rho, pc) \rightarrow \begin{cases} (\rho[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], pc+1), & \mathit{if} \quad \Pi_{pc} = \mathrm{ADDQ} \;\; \mathbf{r}_s, op, \mathbf{r}_d \\ (\rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], pc+1), & \mathit{if} \quad \Pi_{pc} = \mathrm{LDQ} \;\; \mathbf{r}_d, n(\mathbf{r}_s) \quad \mathit{and} \quad \boxed{\mathbf{rd}(\mathbf{r}_s \oplus n)} \\ (\rho[\mathbf{r_m} \leftarrow \mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)], pc+1), & \mathit{if} \quad \Pi_{pc} = \mathrm{STQ} \;\; \mathbf{r}_s, n(\mathbf{r}_d) \quad \mathit{and} \quad \boxed{\mathbf{wr}(\mathbf{r}_d \oplus n)} \\ (\rho, pc+n+1), & \mathit{if} \quad \Pi_{pc} = \mathrm{BEQ} \;\; \mathbf{r}_s, n \quad \mathit{and} \quad \mathbf{r}_s = 0 \\ (\rho, pc+1), & \mathit{if} \quad \Pi_{pc} = \mathrm{BEQ} \;\; \mathbf{r}_s, n \quad \mathit{and} \quad \mathbf{r}_s \neq 0 \end{cases}$$

Figure 3: The Abstract Machine.

$$VC_{pc} = \begin{cases} VC_{pc+1}[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], & \mathit{if} \; \Pi_{pc} = \mathrm{ADDQ} \;\; \mathbf{r}_s, op, \mathbf{r}_d \\ \mathbf{rd}(\mathbf{r}_s \oplus n) \wedge VC_{pc+1}[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], & \mathit{if} \; \Pi_{pc} = \mathrm{LDQ} \;\; \mathbf{r}_d, n(\mathbf{r}_s) \\ \mathbf{wr}(\mathbf{r}_d \oplus n) \wedge VC_{pc+1}[\mathbf{r_m} \leftarrow \mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)], & \mathit{if} \; \Pi_{pc} = \mathrm{STQ} \;\; \mathbf{r}_s, n(\mathbf{r}_d) \\ (\mathbf{r}_s = 0 \Rightarrow VC_{pc+n+1}) \wedge (\mathbf{r}_s \neq 0 \Rightarrow VC_{pc+1}), & \mathit{if} \; \Pi_{pc} = \mathrm{BEQ} \;\; \mathbf{r}_s, n \\ Post, & \mathit{if} \; \Pi_{pc} = \mathrm{RET} \end{cases}$$

Figure 4: The Verification-Condition Generator.

boundaries. Once a safety policy is defined, application writers are free to use it to create PCC binaries that guarantee safety.

A sample application and its precondition

The abstract machine as given above describes safety in terms of the abstract notions of readable and writable memory locations. For this to be useful, the code consumer must specify an interface to PCC binaries that identifies the readable and writable memory locations. We do this by specifying a *precondition*, which is a predicate in first-order logic that the code consumer guarantees to be valid when the PCC binary is invoked.

Consider the following simple example. Suppose an operating-system kernel maintains an internal table with data pertaining to various user processes. Each table entry consists of two consecutive memory words—a tag and a data word. The tag describes whether the data word is user writable or not. The kernel also provides a *resource access service* through which user processes are given permission to access their table entry by installing native code in the kernel. To make this possible the kernel invokes the user-installed code with the address of the table entry corresponding to the parent process in machine register $\mathbf{r}_0$. This address is guaranteed by the kernel to be valid and aligned on an 8-byte boundary.

Although this example is somewhat contrived, we can imagine that entries in the table represent capabilities (perhaps file descriptors), and so we would like to provide user-installed code with full access to the correct table entries, while maintaining the integrity of the rest of the table and other parts of the kernel state.

Informally, the safety policy for the resource access service requires that: (1) the user code cannot access other table entries besides the one pointed to by $\mathbf{r}_0$, (2) the tag is read only, (3) the data word is also read only unless the tag value is non zero, and, (4) the code does not modify reserved and callee-saves registers. The last condition ensures that the kernel can safely invoke the user code using a normal C function call.

More formally, the kernel specifies a precondition Pre_r, which states that it is safe to read the tag pointed to by $\mathbf{r}_0$, and that it is also safe to write the data at offset 8 from $\mathbf{r}_0$ if the contents of the tag is not 0. In formal notation, this is written as follows:

$$\begin{aligned} Pre_r = \quad & \mathbf{r}_0 \bmod 2^{64} = \mathbf{r}_0 \;\wedge\; \mathbf{rd}(\mathbf{r}_0) \wedge \mathbf{rd}(\mathbf{r}_0 \oplus 8) \\ & \wedge\; \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r}_0 \oplus 8) \end{aligned}$$

What remains now is to prove for a particular client of the resource access service that all $\mathbf{rd}(a)$ and $\mathbf{wr}(a)$ checks will always succeed, given this precondition and abstract machine. In general, we can also specify a postcondition as part of the safety policy, which would require particular invariants to be valid when the user code terminates. Conceptually, in our example the postcondition is the predicate **true**, meaning that no additional conditions are imposed on the final machine state.

Before moving on to a discussion of the proof generation process, we note that the safety policy we have described here can be thought of as enforcing fine-grained memory protection. In general, one could imagine having much more involved safety requirements. For example, we could change the tag word in the table entry to be a semaphore that the user code must acquire (e.g., atomically test-and-set to zero) before trying to write the data word; furthermore, we could also require (via a simple post-condition) that the code releases the semaphore before returning. Again, for purposes of the current presentation, we stick to the simpler memory-safety requirements.

2.2 Certifying the Safety of Programs

To create safety proofs for a program, we must prove that executing it does not violate any of the safety checks (and the postcondition, if one is given, is also satisfied). Standard techniques exist for building such proofs. Our technique is based on Floyd's verification conditions [6], because they are powerful enough to deal with unstructured assembly-language programs and a broad range of safety invariants. Similar techniques have been used before to verify assembly-language programs [2, 3].

Certification of programs involves two steps:

1. Compute the *safety predicate* for the program. This essentially encodes the semantic meaning of the program in logical form and constitutes a formal statement that the program, when executed, will not violate any safety checks.

2. Generate a *proof* of the safety predicate, written out in a checkable form.

Both these steps are described in the following subsections.

Computing the safety predicate

To compute the safety predicate, we first generate a vector VC of predicates, one for each instruction as specified by the rules in Figure 4. The notation VC_{pc} denotes the predicate for the current instruction. Since the rules specify VC_{pc} in terms of VC_{pc+1}, the verification-condition VC_0 for the beginning of the program can be computed by starting at the end of the program and working back towards the beginning.[2]

[2] This simple approach works because all branches are restricted to be forward-only. We discuss later what happens in the presence of loops.

The rules in Figure 4 are derived in a straightforward manner from the abstract machine specification of Figure 3; in fact, we imagine that experienced kernel and safety policy designers would skip the abstract machine specification and give only the VC generator rules. The notation $P[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op]$ stands for the predicate obtained from P by substituting $\mathbf{r}_s \oplus op$ for $\mathbf{r}_d$.

After computing the vector VC, the safety predicate is computed simply by plugging the program Π, precondition Pre, and postcondition $Post$ into the following formula:

$$SP(\Pi, Pre, Post) = \forall \mathbf{r}_0 \ldots \forall \mathbf{r}_{10} \forall \mathbf{r_m} . Pre \Rightarrow VC_0$$

The intuition behind a valid safety predicate is that for any initial state that satisfies the precondition Pre, the code Π starting at the first instruction executes without failure and, if it terminates, the final state satisfies the postcondition $Post$.

```
                           %Address of tag in r0
      ADDQ   r0, 8, r1     %Address of data in r1
      LDQ    r0, 8(r0)     %Data in r0
      LDQ    r2, -8(r1)    %Tag in r2
      ADDQ   r0, 1, r0     %Increment Data in r0
      BEQ    r2, L1        %Skip if tag == 0
      STQ    r0, 0(r1)     %Write back data
L1    RET                  %Done
```

Figure 5: DEC Alpha assembly code for resource access. Initially register $\mathbf{r}_0$ holds the address of the tag. The data is at the offset 8 from $\mathbf{r}_0$.

For a concrete example of client code for the resource access service, consider the small program in Figure 5. The overall effect of this program is to increment the data word if it is writable. We first compute VC_0 for this program using the rules in Figure 4; then we compute the safety predicate SP_r using the above formula with the precondition Pre_r and the postcondition **true**. After a few trivial simplifications, the resulting safety predicate is the following:

$$\begin{aligned} SP_r = \forall \mathbf{r}_0 . \forall \mathbf{r_m} . Pre_r \Rightarrow & \ \mathbf{rd}(\mathbf{r}_0 \oplus 8) \wedge \mathbf{rd}(\mathbf{r}_0 \oplus 8 \ominus 8) \\ & \wedge \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) = 0 \Rightarrow \mathbf{true} \\ & \wedge \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r}_0 \oplus 8) \end{aligned}$$

Informally, the SP_r predicate says that for all values of register $\mathbf{r}_0$ and states of memory $\mathbf{r_m}$ satisfying the precondition Pre_r, the memory locations $\mathbf{r}_0 \oplus 8$ and $\mathbf{r}_0 \oplus 8 \ominus 8$ must be readable and if the tag (at address $\mathbf{r}_0 \oplus 8 \ominus 8$) is non zero, the data (at address $\mathbf{r}_0 \oplus 8$) must be writable. All these conditions must be true for the code to be safe with respect to the resource access safety policy.

Proving the safety predicate

We have intentionally written the program in Figure 5 in a slightly complicated way, to show that low-level optimizations do not pose significant problems in generating and validating safety proofs. Three of the interesting properties of this program are (1) the instructions are somewhat scheduled, including speculative execution of the load in line 2 and of the addition in line 4, to accommodate the DEC Alpha pipeline latency[3], (2) register $\mathbf{r}_0$ is reused in line 2 to hold the data word instead of the tag address, and (3) even though the precondition is expressed as a function of the value in register $\mathbf{r}_0$, some of the actual memory accesses are done through register $\mathbf{r}_1$. In general, we expect scheduling and register allocation to have no effect on the safety predicate and its proof.

It is a simple exercise for the reader familiar with assembly-language programming to verify that this code is indeed correct with respect to the safety policy. The problem, of course, is how to convince even the most suspicious kernel that this code is absolutely safe. To do this, we must prove the safety predicate according to the rules of first-order predicate calculus extended with two's-complement integer arithmetic. We refer to this set of proof rules as Σ and we write $\models_\Sigma SP$ when the safety predicate SP can be proved according to the rules in the set Σ. Most of the rules in Σ are simple. Below we show two of the rules we use, the first being a classical implication-elimination rule from the predicate calculus, and the second a rule about arithmetic:

$$\begin{array}{ll} \models_\Sigma Q, & \textit{if} \;\; \models_\Sigma P \Rightarrow Q \quad \textit{and} \quad \models_\Sigma P \\ \models_\Sigma e_1 \oplus e_2 \ominus e_2 = e_1, & \textit{if} \;\; \models_\Sigma e_1 \bmod 2^{64} = e_1 \end{array}$$

The second rule is perhaps a bit surprising because $e_1 + e_2 - e_2 = e_1$ is unconditionally true in integer arithmetic. However, for the machine implementation of arithmetic, this statement is true only if the original value of e_1 is a valid register value.

A large fragment of the proof of the safety predicate for our example program is shown in a proof-tree form in Figure 6. This proof was generated automatically by our PCC system, which incorporates a simple theorem prover. We use vertical dots to stand for extractions of a conjunct from the precondition. You can read the proof tree from top to bottom, interpreting every node as a valid inference of the predicate below the line using the assumptions above the line. For example, in the upper-right corner of the figure the predicate $\mathbf{r}_0 = \mathbf{r}_0 \oplus 8 \ominus 8$ is proved using the arithmetic rule we discussed with the assumption $\mathbf{r}_0 \bmod 2^{64} = \mathbf{r}_0$ extracted from the precondition. Then $\mathbf{wr}(\mathbf{r}_0 \oplus 8)$ is proved using the implication-elimination rule and the hypothesis u of the predicate $\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0$. This hypothesis is introduced at a lower level in the proof tree, at the node labeled u, for the purpose of proving the predicate $\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r}_0 \oplus 8)$.

[3]These operations are speculative because they are not required if the branch in line 5 is taken.

The guarantee of safety

We use the proof of the safety predicate, written out in an appropriate language (to be described in the next section), as the proof that the code obeys the safety policy. This is justified formally by the *safety theorem*, stated below:

Theorem 2.1 (Safety) *For any program Π, precondition Pre and postcondition $Post$, if $\models_\Sigma SP(\Pi, Pre, Post)$ then for any initial state ρ_0 that satisfies the precondition and for any abstract machine state (ρ, pc) originating from the initial state $(\rho_0, 0)$, one of the following is true:*

1. *The state (ρ, pc) is a final state (i.e. $\Pi_{pc} = \mathrm{RET}$) satisfying the postcondition Post, or*
2. *The execution is not stuck, i.e., there exists a new state (ρ', pc') such that $(\rho, pc) \rightarrow (\rho', pc')$.*

Since the abstract machine gets stuck when there is any violation of an $\mathbf{rd}(a)$ or $\mathbf{wr}(a)$ safety check, this theorem provides an absolute guarantee that a certified program will not have such violations, as long as its execution is started in a state that satisfies the precondition.

The proof of the Safety Theorem is beyond the scope of this paper, but can be found in a separate technical report [16].

2.3 Validating the Safety Proofs

A PCC binary consists of the assembled native code together with an encoding of the proof of its safety predicate. To validate the binary, the code consumer first extracts the native code and then computes its safety predicate using the VC rules. Then, it checks that the safety proof is a valid proof of the safety predicate.

This method ensures safety even if the native code or the proof in the PCC binary is tampered with. If the code is modified, then in all likelihood its safety predicate changes, so the given proof will not correspond to it. If the proof is modified, then either it will be invalid, or else not correspond to the safety predicate. If the code is modified in such

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{Pre_r}{\vdots}}{\mathbf{rd}(\mathbf{r_0})} \quad \dfrac{\overset{Pre_r}{\vdots}\atop \mathbf{r_0} \bmod 2^{64} = \mathbf{r_0}}{\mathbf{r_0} = \mathbf{r_0} \oplus 8 \ominus 8}}{\mathbf{rd}(\mathbf{r_0} \oplus 8 \ominus 8)} \quad \dfrac{\dfrac{\dfrac{\overset{Pre_r}{\vdots}}{\mathbf{sel}(\mathbf{r_m}, \mathbf{r_0}) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r_0} \oplus 8)} \quad \dfrac{\dfrac{u}{\mathbf{sel}(\mathbf{r_m}, \mathbf{r_0} \oplus 8 \ominus 8) \neq 0} \quad \dfrac{\overset{Pre_r}{\vdots}\atop \mathbf{r_0} \bmod 2^{64} = \mathbf{r_0}}{\mathbf{r_0} = \mathbf{r_0} \oplus 8 \ominus 8}}{\mathbf{sel}(\mathbf{r_m}, \mathbf{r_0}) \neq 0}}{\mathbf{wr}(\mathbf{r_0} \oplus 8)}\,u}{\mathbf{sel}(\mathbf{r_m}, \mathbf{r_0} \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r_0} \oplus 8)} \quad \ldots}{\mathbf{rd}(\mathbf{r_0} \oplus 8 \ominus 8) \wedge (\mathbf{sel}(\mathbf{r_m}, \mathbf{r_0} \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r_0} \oplus 8)) \wedge \ldots}\,Pre_r}{Pre_r \Rightarrow \mathbf{rd}(\mathbf{r_0} \oplus 8 \ominus 8) \wedge (\mathbf{sel}(\mathbf{r_m}, \mathbf{r_0} \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r_0} \oplus 8)) \wedge \ldots}}{\forall \mathbf{r_0}.\forall \mathbf{r_m}.Pre_r \Rightarrow \mathbf{rd}(\mathbf{r_0} \oplus 8 \ominus 8) \wedge (\mathbf{sel}(\mathbf{r_m}, \mathbf{r_0} \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r_0} \oplus 8)) \wedge \ldots}
$$

Figure 6: A Fragment of the formal safety proof of SP_r.

a way that the safety predicate is unchanged (for example, instruction scheduling and register allocation might do this in typical circumstances), or if both the code and the proof are modified so that we still have a valid proof of the new safety predicate, the validation succeeds and we continue to retain a guarantee of safety.

To automate the validation process, we must first choose a concrete representation language for predicates and their proofs. From the many available choices, we have selected the Edinburgh Logical Framework [7] (also called LF) as the representation framework for predicates and proofs. LF is an extension of the simply typed lambda calculus and was designed as a meta language for high-level specification of languages in logic and computer science. The most attractive property of LF is that it has a powerful yet simple typechecking algorithm, which we use to check the validity of proofs.

We represent the predicates and the proofs in LF in such a way that the validity of a proof is implied by the well typedness of the proof representation. Thus, proof validation amounts to typechecking. Also, LF allows us to represent in an elegant way a few key issues in logical proof correctness, such as the manipulation of logical parameters and assumptions. It is well beyond the scope of this paper to discuss in detail LF and the typechecking algorithm, however it is worth mentioning that typechecking is decidable and is described by a few simple rules. Indeed, typechecking is so simple that any programmers who do not trust the publicly available implementation can implement it easily themselves. Our implementation has about five pages of C code, even though it incorporates a few optimizations to the basic algorithm. With this implementation, it takes 1.4 milliseconds to validate the proof of the SP_r predicate.

For flexibility and to allow easy exchange of proofs between system components, we have designed a binary encoding of LF representations. Thus, a typical PCC binary contains a section with the native code ready to be mapped into memory and executed, followed by a symbol table used to reconstruct the LF representation at the code consumer site, and the binary encoding of the LF representation of the safety proof. The latter component is the safety proof. Figure 7 shows the sizes of these sections for the PCC binary corresponding to the resource access example.

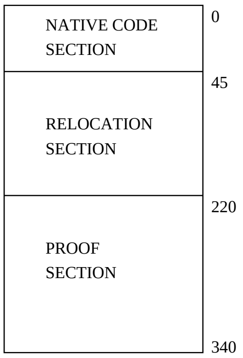

Figure 7: The layout of the PCC binary for the resource access example. The offsets are in bytes.

Currently, PCC binaries for standard packet filters, including the native code, safety proof, and relocation section, are about 400 to 1200 bytes in size, with the proof about 3 times larger than the code. The size of the relocation section increases linearly with the number of distinct proof rules used in the proof. In the case of packet filter safety proofs, the relocation section is a third of the binary but we expect this ratio be much smaller for larger proofs.

There is a considerable amount of design latitude in the encodings of the proofs, and we have barely scratched the surface on what can be done to reduce the size of the binaries as well as the time required for validation. But already, with relatively little effort, we have achieved acceptably small binaries and low validation times.

3 Application: Network Packet Filters

In order to gain more experience with PCC and to compare it with other approaches to code safety, we have performed a series of experiments with safe network packet filters. We describe in this section the particulars of the PCC approach to network packet filters. Then in Section 3.1, we compare it with other approaches including interpreted packet filters (as exemplified by the BSD Packet Filter), code editing (through Software Fault Isolation), and using a safe programming language (the approach taken in the SPIN kernel).

A packet filter is an application-provided subroutine that scans each incoming network packet and decides whether the user application is interested in receiving it or not. Packet filters are supported by most of today's workstation operating systems. Since their first introduction in [15], packet filters have been used successfully in network monitoring and diagnosis.

In the PCC approach the packet filter is a PCC binary whose native code component is invoked by the kernel on each incoming network packet. Kernel safety is ensured by validating the safety proof.

Following the procedure described in Section 2 we first establish a safety policy. To allow for a fair comparison we follow the BSD Packet Filter model of safety. The packet filter code can examine the packet at will and can also write to a statically allocated scratch memory. Informally, the safety policy requires that: (1) memory reads are restricted to the packet and the scratch memory; (2) memory writes are limited to the scratch memory; (3) all branches are forward; and (4) reserved and callee-saves registers are not modified. These rules establish memory safety and termination assuming that the kernel calls the packet filter with valid packet and scratch memory addresses.

We write the packet filter code assuming that the return value must be in $\mathbf{r}_0$, the aligned address and the length of the packet filter are given in $\mathbf{r}_1$ and $\mathbf{r}_2$ respectively, and the address of a 16-byte aligned scratch memory is given in $\mathbf{r}_3$. Moreover the packet's length is positive and at least 64-bytes (the minimum length of an Ethernet packet). Formally this is expressed as the precondition:

$$
\begin{aligned}
Pre = \;& \mathbf{r}_1 \bmod 2^{64} = \mathbf{r}_1 \wedge \\
& \mathbf{r}_2 \bmod 2^{64} = \mathbf{r}_2 \wedge \mathbf{r}_2 < 2^{32} \wedge \mathbf{r}_2 \geq 64 \wedge \\
& \mathbf{r}_3 \bmod 2^{64} = \mathbf{r}_3 \wedge \\
& \forall i.(i \geq 0 \wedge i < \mathbf{r}_2 \wedge (i \,\&\, 7) = 0) \\
& \qquad \Rightarrow \mathbf{rd}(\mathbf{r}_1 \oplus i) \qquad \wedge \\
& \forall j.(j \geq 0 \wedge j < 16 \wedge (j \,\&\, 7) = 0) \\
& \qquad \Rightarrow \mathbf{wr}(\mathbf{r}_3 \oplus j) \qquad \wedge \\
& \forall i.\forall j.(i \geq 0 \wedge i < \mathbf{r}_2 \wedge j \geq 0 \wedge j < 16) \\
& \qquad \Rightarrow (\mathbf{r}_1 \oplus i \neq \mathbf{r}_3 \oplus j)
\end{aligned}
$$

The first few conjuncts of the precondition restrict the values of input registers to valid machine word values. The last term of the precondition rules out the possibility of memory aliasing between packets and the scratch memory. This is useful when reasoning about filters that write to the scratch memory.

The postcondition in our packet filter experiment is the predicate **true**, meaning that no additional conditions are placed on the final state.

We have implemented four typical packet filters in assembly language and certified their safety with respect to the packet filter safety policy. Filter 1 accepts all IP packets. This is done by comparing a 16-bit word in the packet to a given value. Filter 2 accepts IP packets originating from a given network. This involves checking a 24-bit value in addition to the work done by Filter 1. Filter 3 accepts IP or ARP packets exchanged between two given networks. This includes all the work done by Filter 2 with the addition of checking the destination network address. Extra complexity is required because of different header layout of IP and ARP packets. Filter 4 accepts all TCP packets with a given destination port. This filter has to check that the Ethernet packet is an IP packet, then that it is a TCP packet, and lastly that the destination port matches a given value. The offset of the TCP destination port is computed based on a byte in the IP header (the length of the IP header).

The effort involved in hand-coding packet filters in assembly language is repaid in increased performance, because packet filters are usually small and very frequently executed. Hand-coding provides the opportunity to perform optimizations that are difficult to obtain from an optimizing compiler. The important point is that these optimizations are not an impediment to generation and validation of safety proofs. Here are a few optimizations that we incorporated in our packet filters:

- The number of memory operations is minimized by using the DEC Alpha 64-bit load followed by byte extraction.

- The TCP port number can be found at packet offset $([14]_8 \ \&\ 15) * 4 + 16$, where $[14]_8$ denotes the byte at offset 14. If loading 64 bits at a time on a little-endian machine, the formula becomes $((([8]_{64} \gg 48) \ \&\ 255) \ \&\ 15) * 4 + 16$. With further simplification we reduce this to $(([8]_{64} \gg 46) \ \&\ 60) + 16$, which is exactly how we coded Filter 4.

After we write a packet filter, our prototype assembler produces its safety predicate using the verification-condition method presented in Section 2. The safety predicate is then proved using a theorem prover. We currently use our own theorem prover, which is admittedly a toy. When it gets stuck, it requires intervention from the programmer, mainly to learn new axioms about arithmetic (for example, to know that $\mathbf{r}_1 > 0 \Rightarrow \mathbf{r}_1 \geq 0$). The process is easy, and because user-provided axioms are remembered for future sessions, by now our system works automatically for most practical packet filters. With state-of-the-art theorem proving technology we expect to be able to prove completely automatically most arithmetic facts involved in certifying packet filters.

With our primitive theorem-prover we can generate safety proofs for packet filters in about 5 to 10 seconds, in the cases when no user-intervention is required.

3.1 Performance Comparisons

All performance measurements were done on a DEC Alpha 3000/600 with a 175-MHz processor, a 2-MByte secondary cache and 64-MByte main memory, running OSF/1. All measurements were performed off-line using a 200,000-packet trace from a busy Ethernet network at Carnegie Mellon University.

We measured the average per-packet run time of the four PCC packet filters and of functionally equivalent filters implemented using alternative approaches: the BSD Packet Filter architecture, Software Fault Isolation and programming in the safe subset of Modula-3. In our experiments with Modula-3 packet filters we use the VIEW extension [9] for pointer-safe casting. The result of the measurements are shown in Figure 8. From a per-packet latency point of view, the PCC packet filters outperform filters developed using any other considered approach. However, the PCC method has a startup cost significantly larger than the other approaches. This cost is the proof validation time,

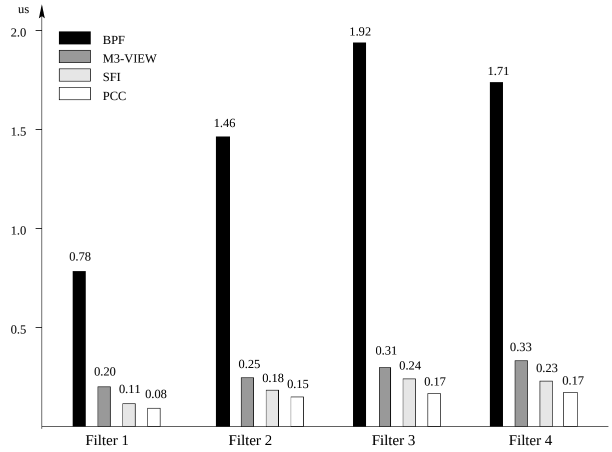

Figure 8: Comparison of average per-packet run time.

which is presented in Table 1 together with the PCC binary size for all four filters and maximum heap space used for validation. The maximum depth of the stack during validation was under 4 Kbytes.

Packet Filter		1	2	3	4
Instructions		8	15	47	28
Binary Size (bytes)		385	516	1024	814
Validation Cost	Time (μs)	780	1070	2350	1710
	Space (KB)	5.5	8.7	24.6	15.1

Table 1: Proof size and validation cost for PCC packet filters.

Despite the relatively high validation cost, the run-time benefits of PCC packet filters are large enough to amortize the startup cost after processing a reasonable number of packets. Figure 9 shows the overall running time, including startup cost, as a function of the number of packets processed, for Filter 4. In this particular case, the cost of proof validation is amortized after 1200 packets when compared to the BPF version of the filter, after 10500 packets when compared to the Modula-3 version and after 28,000 packets when compared to the SFI packet filter. Note that at the time we collected the packet trace used for the experiments we counted about 1000 Ethernet packets per second on the average.

We proceed now to describe in more detail each considered approach focusing on how it relates to PCC from the safety point of view, and how we set up the performance measurements.

The standard way to ensure safe execution of packet filters is to interpret the filter and perform extensive run-time checks. This approach is best exemplified by the BSD Packet Filter architecture [13],

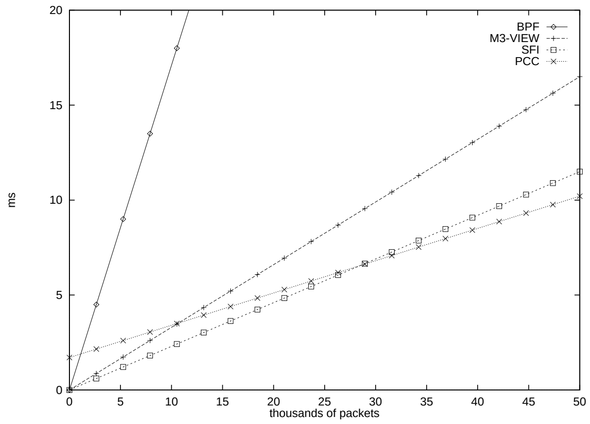

Figure 9: Startup cost amortization for Filter 4.

commonly referred to as BPF. In the BPF approach the filter is encoded in a restricted accumulator-based language. According to the BPF semantics, a filter that attempts to read outside the packet or the scratch memory, or to write outside the scratch memory, is terminated and the packet rejected.

The BPF interpreter makes a simple static check of the packet filter code to verify that all instruction codes are valid and all branches are forward and within code limits. We measured this one-time overhead to be a few microseconds, which is negligible. BPF packet filters, however, are about 10 times slower than our PCC filters. In the PCC approach all checks are moved to the validation stage, allowing for much faster execution.

In order to collect data for the BPF packet filters, we extracted the BPF interpreter as implemented by the OSF/1 kernel and compiled it as a user library.

It is possible, of course, to eliminate the need for interpretation. For example, we could replace the packet-filter interpreter with a compiler. This approach is taken by several researchers [10, 24]. The problem here is the startup cost and complexity of compilation, especially if serious optimizations are performed.

Another approach to safe code execution is Software Fault Isolation (SFI) [23]. SFI is an inexpensive method for parsing binaries and inserting run-time checks on memory operations. There are many flavors of SFI depending on the desired level of memory safety. If the entire code runs in a single protection domain whose size is a power of 2, and if only memory writes are checked, then the run-time cost of SFI is relatively small. If, on the other hand, the untrusted code interacts frequently with the code consumer or other untrusted components residing in different protection domains and the read operations must be checked also, the overhead of the run-time checks can amount to 20% [23]. A more serious disadvantage of SFI is that it can only ensure memory safety. We believe that this level of safety is not enough in general, and that it is important to be able to check abstraction boundaries and representation invariants, as shown by the resource access example in Section 2.

In order to accommodate SFI for packet filters, we allowed some concessions to the packet filter semantics. For example, we assumed that the kernel allocates the packets on a 2048-byte boundary. Furthermore, we assume that the filter can safely access the entire segment of 2048 bytes, independently of the packet size. Note that the BPF packet filter semantics, which we followed for all other experiments, specifies that a filter should be terminated if it tries to access beyond the packet boundary. This means that some working packet filters in the BPF semantics will not behave as expected in the SFI semantics for packet filters, and vice-versa.

One common way of performing SFI is at the code producer site, usually as part of the code-generation phase in a compiler. In this case, the code consumer performs a load-time checking that SFI was done correctly. The load-time SFI validator is reportedly simple if it must deal only with binaries for which run-time checks have been inserted on every potentially dangerous memory operation [23]. On the other hand, in the case where the validator must accept binaries for which the number of run-time checks has been optimized through program analysis, the validator itself has to redo the analysis that led to the optimization. This means a more complex and slower validation, and in fact such an SFI validator does not presently exist.

We inserted run-time checks for the memory operations in the assembly language packet filters implemented for the PCC experiment. This process can be done by a simple and easy-to-trust implementation of SFI. In our experiments, PCC packet filters run about 25% faster than SFI filters.

As part of our SFI experiment, we produced safety proofs attesting that the resulting SFI packet filter binaries are safe with respect to the packet filter safety policy. We achieve the same effect as an SFI load-time validator but using the universal typechecking algorithm and a few application-dependent proof rules. The precondition for this experiment says that it is safe to read from any aligned address that is in the same 2048-byte segment with the packet start address. Proof sizes and validation times are very similar to those for plain PCC packets.

Another approach to safe code is to use a type-safe programming language. This approach is taken by the SPIN extensible operating system [1], and the language used is Modula-3 [17] extended with pointer-safe casting (VIEW). SPIN allows applications to install extensions in the kernel but only if they are written in the safe subset of Modula-3. The extensions are compiled by a trusted compiler and the resulting executable code is then believed to be safe (at least according to the Modula-3 model of safety). Note that such extensions written in Modula-3 are intrinsically safe, as anyone who believes in the safety of Modula-3 can check their compliance with Modula-3 syntactic and typing rules.

We believe that encoding kernel extensions as PCC binaries instead of Modula-3 source code can provide important benefits. One such benefit is the increased flexibility for extension writers because any native code extension can be accepted, independent of the original source language or even the compiler used, as long as a valid safety proof accompanies it. Another potential benefit is overcoming the limitations of the Modula-3 safety model: the PCC safety proof should be able to express properties such as disciplined use of locks or array bounds compliance with no need for run-time checks.

We wrote the four packet filters in the safe subset of Modula-3 and compiled them with the version 3.5 of the DEC SRC compiler extended with the VIEW operation [24]. VIEW is used to safely cast the packet filter to an array of aligned 64-bit words allowing fewer memory operation for accessing packet fields. In contrast, in plain Modula-3 the packet fields must be loaded a byte at a time, and a safety bounds check is performed for each such operation. The compiler tries to eliminate some of these checks statically but it is not very successful for packet filters. The main reason is that a critical piece of information, the fact that packets are at least 64 bytes long, cannot be communicated to the compiler through the Modula-3 type system.

We measured a 20% improvement in the Modula-3 packet filter performance when using VIEW. Similar performance improvements over the DEC SRC Modula-3 compiler have been reported [18] for the more recent Vortex compiler. However, since we have not conducted any experiments with the Vortex compiler on our packet filters, it is not clear what kind of improvements we would realize in practice.

In an alternate implementation of untrusted code certification using Modula-3, the source code is compiled by a trusted and secure compiler that signs the executable for future use. Validation then means cryptographic signature checking and like in the PCC approach there is no run-time cost associated with it. We do not have a complete implementation of such a cryptographic validation, so we do not know exactly how large is the startup cost for the digital signature approach. It is likely however that a good implementation of digital signatures would achieve faster validation and significantly faster generation of certificates. The essential drawback of cryptographic techniques over PCC is that validation establishes only a trusted origin of the code and not its absolute safety relative to the safety policy. In particular, a digital signature can be ascribed to an unsafe program just as easily as to a safe one. Also, the cost of managing and transmitting encryption keys is not incurred by PCC.

We should mention here one more approach to safe code execution, although we do not have an actual quantitative comparison. The Java Virtual Machine [21] is a proposed solution to safe interaction of distributed, untrusted agents. Mobile code is encoded in the Java Virtual Machine Language (also referred to as Java Bytecode), which is basically a safe low-level imperative language. Safety is achieved through a combination of static typechecking and run-time checking.

However, the Java Bytecode safety model is relatively limited as a result of limitations of the type system. For example the Java Bytecode type information encoded in the instruction codes can only express a few basic abstract types (e.g., integers, objects) and has no provisions for expressing safety policies like the one for the resource access example in Section 2. Also, invariants involving array bounds compliance cannot be expressed in the Java Bytecode type system and must be checked at run time.

Although Java Bytecode is a low-level language, it still requires substantial processing before it can be executed on a general-purpose processor. In contrast, PCC segregates the safety proof from the program code, allowing for the code portion to be encoded in a variety of languages, including native code, without any safety loss.

4 Practical Problems and Future Work

In order to create a safety proof, the code producer must prove a predicate in first-order logic. In general, this problem is undecidable. However, as we mentioned in Section 1, the code producer can resort to "extra" run-time checks inserted in strategic locations, which have the tendency to simplify the certification.

Fortunately, in the packet-filter experiments, the certification process is nearly automatic, and we have not been forced to insert any extra run-time checks into the code. In fact, we find that safety predicates for packet filters are fairly easy handled by existing theorem-proving technology.

One of the simplifications in the packet filters is to restrict programs so that they do not contain loops. Although the general framework presented in this paper is easily extended to accommodate loops [5], this introduces a number of complications. One experiment we conducted involves an IP-header checksum routine, which is hand-coded in 39 DEC Alpha instructions. The core loop contains 8 instructions, and is optimized by computing the 16-bit IP checksum using 64-bit additions followed by a folding operation. The resulting PCC binary for this routine is, as expected, quite fast, beating the standard C version in the OSF/1 kernel by a factor of two. The PCC binary itself is 1610 bytes in size and proof validation takes 3.6 milliseconds.

This experiment brought to light several complications. For example, the standard approach of verifying loops using Floyd-style verification conditions involves introducing loop invariants explicitly, which is a challenge for any theorem-proving technology and ofter requires user intervention. In fact, for general assembly-language programs this represents the most important problem to be solved, as it is the main obstacle in automating the generation of proofs. Since this is beyond the capabilities of our system, we are forced to write the invariants out by hand. This also means that the native code must be accompanied by a loop invariant for every loop. Thus, the PCC binary contains a mapping between each loop and its invariant. Our convention is to have the PCC binary contain a table that maps each backward-branch target to a loop invariant.

Besides the problem of how to generate the proofs, there is also the matter of their size. In principle, the proofs can be exponentially large (in the size of the program). This has not been a problem for any of the examples we have tried thus far, however. The blowup would tend to occur in programs that contain long sequences of conditionals, with no intervening loops. Perhaps we have not yet seen the problem in a serious way because such programs tend to be hard for humans to understand, and we are writing the programs by hand. But as a general matter, the size of the PCC binaries is an issue that must be addressed carefully. We have implemented several optimizations in the representation of the proofs, and much more is possible here. But ultimately, we need more practical experience to know if this is a serious obstacle for PCC in practice.

For programs with loops, the loop invariants break a program with cycles into a set of acyclic code fragments. We treat each code fragment as a separate program, using the invariants as preconditions for each. This has the beneficial effect of partitioning the safety predicate and its proof into smaller pieces, and overall tends to reduce the size of the proof dramatically. For this reason, even for sections of programs that do not contain loops, it may be beneficial to introduce invariants, as a way of controlling the growth of the PCC binaries.

In addition to developing better certification technology, we see several other interesting directions for further research. One possibility that we intend to explore is the application of PCC to more dynamic properties, such as resource-usage guarantees. One example would be to certify that specific synchronization locks are always released prior to some action. The framework we have presented in this paper is already expressive enough to define such safety policies, and so what remains now is to try some experiments.

Another possibility is to allow the consumer and producer to "negotiate" a safety policy at run time. This would work by allowing the producer to send an encoding of a proposed safety policy (including the VC-generation rules, proof rules, and preconditions) to the consumer. If the consumer determines that the proposed policy implies some basic notion of safety, then it can allow the producer to produce PCC binaries using the new policy. This might be useful in distributed systems, in which one agent wants to define a language and then transmit to other agents code written in that language.

Finally, we believe there would be advantages to starting with a safe programming language and then implementing a *certifying compiler* that produces PCC binaries as target programs. For the safety properties that are implied by the source language, construction of the proofs is, in principle, a matter of having the compiler prove the correctness of the translation to target code. We have already experimented with a toy compiler of this sort for a small type-safe programming language, and hope to expand on this in the near future.

5 Conclusions

We have described *proof-carrying code,* a mechanism that allows a kernel or server to interact safely with binaries supplied by an untrusted source. PCC does not incur any run-time overhead for the kernel. Instead, the code producer is required to generate a formal proof that the code obeys the safety policy. The kernel can easily check the proofs for validity, after which it is absolutely certain that the code respects the safety policy. Furthermore, PCC binaries are completely tamper-proof; any attempt to alter either the native code or safety proof in a PCC binary is either detected or harmless. Our experiments with network packet filters show that PCC can lead to significant performance advantages over existing approaches to safe code, including code-editing techniques such as Software Fault Isolation.

Proof-carrying code has the potential to free the 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 restricted application-kernel interaction model (such as a fixed system call or application-program interface.)

Proof-carrying code, on the other hand, allows the safety policy to be defined by the kernel designer and then certified by each application. Not only does this provide greater flexibility for designers of both the system and applications, but 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.

6 Final Thoughts

The inspiration for proof-carrying code comes from the realm of static type systems, especially as embodied by the language Standard ML (SML). In the formal definition of SML [14], a formal theorem guarantees the safety of any type-correct SML program, for a rigorously defined notion of safety. There are, of course, many other type-safe programming languages, for example Modula-3 [17] and Java [20], but the use of mathematical formalism sets SML apart from the these languages, and as a practical matter this rigor provides the basic conceptual and technical foundations that we need to create checkable proofs.

With type-safe languages like SML in mind, we can get an intuitive idea about how proof-carrying code works. Consider a compiler for SML. Agent A writes an SML program and compiles it to a native-code target program. If we then throw away the source program, how can we later convince an agent B that the target program is safe? (We are assuming that agent B does not trust agent A.) One way to do this is to have the compiler *prove* that the target code correctly corresponds to the source code.[4] Now, as it turns out, in the type theory of SML, not only can such a proof be written out formally, but in fact it can be written in a typed language with the property that any well-typed proof is guaranteed to be valid.

Proof-carrying code is thus an application of ideas from programming-language theory, in this case used for defining notions of safety that are useful for operating systems, and flexible enough to accommodate both high-level and low-level languages. With the growth of interest in highly distributed computing, web computing, and extensible kernels, it seems clear to us that ideas from programming languages are destined to become increasingly critical for robust and good-performing systems.

7 Acknowledgements

We thank Robert Harper, Brian Noble, Daniel Jackson, Edo Biagioni, Greg Morrisett, Scott Draves, Chris Colby, Martin Abadi and Dave Detlefs for reading previous versions of this paper and for suggesting many improvements. We also thank Charles Garrett, Brian Bershad, Wilson Hsieh for suggesting many improvements to the methodology for the Modula-3 performance measurements. Finally, we thank the anonymous reviewers for their many suggestions for improving this paper. In particular we thank our shepherd, Jay Lepreau, who also suggested the PCC name.

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.

[4] This is essentially the same as having a compiler translate the types as well as the code, so that the target program will have types that can be checked. In fact, this approach to compiling is taken by the SML/TIL compiler [22].

[2] Boyer, R. S., and Yu, Y. Automated proofs of object code for a widely used microprocessor. *J. ACM 43*, 1 (Jan. 1996), 166–192.

[3] Clutterbuck, D., and Carré, B. The verification of low-level code. *IEEE Software Engineering Journal 3*, 3 (May 1988), 97–111.

[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] Dijkstra, E. W. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM 18* (1975), 453–457.

[6] Floyd, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. American Mathematical Society, 1967, pp. 19–32.

[7] 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.

[8] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12* (1969), 567–580.

[9] Hsieh, W. C., Fiuczynski, M. E., Garrett, C., Savage, S., Becker, D., and Bershad, B. N. Language support for extensible operating systems. In *The Inaugural Workshop on Compiler Support for Systems Software* (Feb. 1996), pp. 127–133.

[10] Lee, P., and Leone, M. Optimizing ML with run-time code generation. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996), pp. 137–148.

[11] Martin-Löf, P. A theory of types. Technical Report 71–3, Department of Mathematics, University of Stockholm, 1971.

[12] McCanne, S. The Berkeley Packet Filter man page. BPF distribution available at `ftp://ftp.ee.lbl.gov`, May 1991.

[13] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference* (Jan. 1993), USENIX Association, pp. 259–269.

[14] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[15] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles* (Nov. 1987), ACM Press, pp. 39–51. An updated version is available as DEC WRL Research Report 87/2.

[16] Necula, G. C., and Lee, P. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, Sept. 1996. Also appeared as FOX memorandum CMU-CS-FOX-96-03.

[17] Nelson, G. *Systems Programming with MODULA-3*. Prentice-Hall, 1991.

[18] Sirer, E. G., Savage, S., Pardyak, P., DeFouw, G. P., and Bershad, B. N. Writing an operating system with Modula-3. In *The Inaugural Workshop on Compiler Support for Systems Software* (Feb. 1996), pp. 134–140.

[19] Sites, R. L. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[20] Sun Microsystems. The Java language specification. Available as `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`, 1995.

[21] Sun Microsystems. The Java Virtual Machine specification. Available as `ftp://ftp.javasoft.com/docs/vmspec.ps.zip`, 1995.

[22] Tarditi, D., Morrisett, J. G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996), pp. 181–192.

[23] 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.

[24] Wallach, D. A., Engler, D., and Kaashoek, M. F. ASHs : Application-specific handlers for high-performance messaging. In *ACM SIGCOMM'96* (Oct. 1996), vol. 26, ACM.