Proofs of Safety for Untrusted Code

Grigore Roşu and Nathan Segerlind
Department of Computer Science & Engineering
University of California at San Diego

May 18, 1999

Abstract

Proof-carrying code is a technique that can be used to execute untrusted code safely. A code consumer specifies requirements and safety rules which define the safe behavior of a system, and a code producer packages each program with a formal proof that the program satisfies the requirements. The consumer uses a fast proof validator to check that the proof is correct, and hence the program is safe. In this tutorial, we discuss applications for which proof-carrying code is appropriate, explain the mechanics of proof-carrying code, compare it with other techniques and suggest two research directions for the method.

1 Introduction

We cannot always trust computer programs to be safe, let alone correct. Given that many programs are of questionable or unknown origin, for example, mobile agents and games downloaded from shady websites, there are often reasons to believe that a piece of code may attempt to gain access to private data or cause harm to the system.

This problem has been identified for a long time, and there have been numerous solutions implemented. Not every solution will always be appropriate for each use of untrusted code. For a security mechanism to be useful in a given situation, it should have the following properties:

Efficacy The system should not have loopholes which can be used by a clever hacker to circumvent the system.

Efficiency The system should not incur a cost which makes running the untrusted code undesirable.

Expressiveness The system should allow a user to guarantee the untrusted code obeys whatever properties are desired.

Minimal Trust The user should not have to trust many third parties in order to feel safe when using the system.

Ease of Use The system should be easy to use, both for the producer and consumer of the untrusted code.

These criteria can seem daunting. Many protection methods have been designed which monitor the code and prevent disallowed accesses, but this can be expensive and inflexible. Some systems use cryptographic protocols to authenticate the code producer, as if large software companies did not produce buggy software or have nefarious motives. Indeed, it can seem that there may be no better solution than to inspect the code, and convince yourself that it does nothing malicious.

In fact, this solution is not bad at all, rather, it can be very efficient and it can allow great freedom in the properties that it can ensure. *Proof-carrying code* requires the code producer to provide the consumer with a formal proof that the given code meets a specification. The consumer is then able to quickly verify that the proof is correct and therefore the given code is safe. Because the verification of safety is done once, the amortized cost of using the system is very close to running native code. Because the the only limit on the system is in what the code producer can prove about the code, it can be very expressive.

We believe that proof-carrying code is an exciting technique, and that it shows great promise for ensuring safety in performance critical code and systems with complicated requirements. In section 2, we describe some applications which need efficient and flexible mechanisms for handling untrusted code. Other methods developed over the years are briefly described in section 3. Section 4 is a tutorial on the mechanics of proof-carrying code. Comparisons between PCC and the other methods are made in section 5, and section 6 sketches some promising research directions.

2 Applications

Modern operating systems and network architectures present us with numerous situations in which the safety of untrusted code is a major concern. We describe some of these applications here, and the reader is encouraged to keep them in mind throughout the rest of the paper.

Mobile Code There is a very fine line delimiting a mobile agent, a useful self-replicating program which travels the internet performing a task for its master, and a computer virus, a dangererous self-replicating program which spreads across the internet wreaking havok on those unfortunates in its path. For a mobile agent sytem to be viable, it would be necessary for a host to guard against viruses while welcoming mobile agents. Telling the difference is not always an easy task. Indeed, a mobile agent with an error in its replication routines can become a virus. Mobile agent applications may require safety requirements beyond those presently enforced, such as bounds on resource consumption, and guarantees of termination.

Extensible Operating Systems An *extensible* operating system [2], is one which allows the user to easily replace the kernel code to increase performance or customize functionality.

There are numerous situations when it is desirable to modify kernel provided functions. Video stream applications can achieve a 50% increase in throughput by modifying the kernel to transfer data between the network and the video driver, bypassing a user process [8]. Similarly, it can be more efficient to demultiplex network packets in the kernel rather than a user process. However, kernel level techniques can be crude, and require customization. This trade-off led to packet filters being run as interpreted code within the kernel [16, 15]. For memory intensive applications such as databases, it can beneficial to involve the application with cache management [14].

In an extensible operating system, the safety of the code in question is a critical issue: a bug or Trojan horse inserted into the kernel could have grave consequences. However, the safety mechanisms should not incur too much cost, lest they negate the desired performance increases.

Active Networks Active networks are a currently hot technology [25], in which users distribute customized programs about the network to be executed at various nodes. A simple example of this would be giving a packet filter to distribution centers in the network so that unwanted packets addressed to a given user can be filtered early, thereby reducing network congestion. A more exotic possibility would be to replace network packets with "code capsules" which are executed by each node upon receipt. Certainly, for such technologies to come into widespread use, it should not be possible for a malevolent user to create code capsules capable of bringing down the network. Depending on what the code capsules are allowed to do, a wide-array of safety requirements may have to be ensured, with great efficiency.

3 Methods

The problem of taming untrusted code has been around since the design of the early time-sharing mainframes. The advent of portable disks and the internet only made it easier for shady programs to enter a system. Different methods for ensuring that a piece of untrusted code will not cause any harm have been developed, each with its advantages and disadvantages.

3.1 Interpretation

A common way to ensure the safe execution of untrusted code is to interpret each command and perform run-time checks. Interpreters implement a virtual machine defining a set of virtual commands which provide a portable interface between the program and the processor. User applications can be written either directly in the virtual machine's code or in a higher level language, like Java, which is then compiled to the virtual code. Some kernel extensions, such as the BSD packet filter [15], are currently imlemented with interpreted languages.

3.2 Hardware Protection

Many processors provide hardware protection, preventing one process from accessing the memory of another. This is often done by using dedicated *base* and *limit* registers, which only the kernel can set, to delimit the range of memory a process is allowed to access. The permission entries in the TLB can also be used to restrict memory accesses, provided that only the kernel is allowed to manipulate TLB entries. Calls between protected domains are handled by the kernel through a remote procedure call mechanism, [3, 4].

3.3 Software Fault Isolation

Software fault isolation, or "sandboxing" [26], was introduced to avoid the high costs incurred by placing untrusted code in its own address space and accessing it via remote procedure calls. The suspicious code is modified to include address checks before each memory access or branch. Then, to avoid context switching, it is placed in a reserved region of memory belonging to the process calling it. If the untrusted code ever attempts to read memory or jump to a location outside of this region, the process is terminated and an error is raised.

At present, the performance of sandboxed code is generally at least an order of magnitude better than that seen with code that uses hardware protections. Indeed, depending on the applications, sandboxing incurs a slowdown between 20% and 50% over unmodified code [26, 5, 22].

3.4 Author Authentication

Cryptographic message-authentication protocols, [24], make it possible for a code producer to sign a piece of code, ensuring the consumer that it came from a reputable source.

This approach has been used in commerce, because it is efficient, and relatively easy to implement with existing technologies. Furthermore, "guarantees" any property which the consumer trusts the producer to implement. However, the method is only as strong as the consumer's faith in the producer. The consumer must believe that the code producer is both competent and benevolent, a questionable assumption at best.

4 Proof-Carrying Code

The proof carrying code mechanism [20, 17, 21], or simply PCC, ensures the safety of a mobile agent by requiring that the code producer provide a proof of its correctness, which the consumer compares against the object code. If the proof checks and is correct, the consumer safely runs the code. Because the code is not interpreted and no tests have been added to the code, the only cost incurred is the one-time off-line cost of verifying the proof.

Given a safety-specification and a piece of code[1] it is possible to construct a formula which is valid if and only if the code satisfies the specification. This formula is called the *safety predicate* and is the key mechanism underlying proof-carrying code. The code consumer can quickly construct this formula given the code, and rather

[1] It is usually necessary to annotate the code with invariants in order for the process to work. This is explained later in the section.

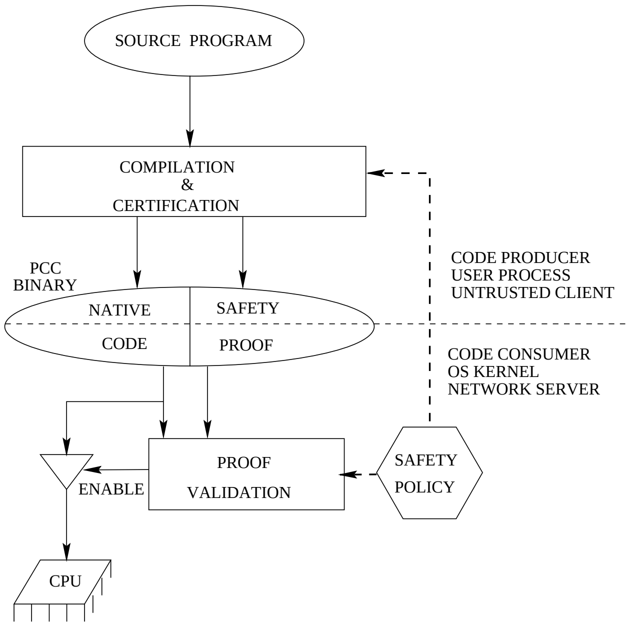

Figure 1: Overview of Proof-Carrying Code.

than proving it himself, he asks the code producer to provide a correct proof. Whether or not a proof is correct can be checked quickly, so the consumer has a rapid mechanism for recognizing a safe program.

A typical proof-carrying code system would proceed as follows (see Figure 1):

1. Consumer specifies requirements for the system and provides them to the producer;
2. Producer generates code, keeping in mind the safety specification and an eventual proof;
3. Producer annotates the code with invariants and automatically generates the safety predicate;
4. Producer produces a formal proof of the safety predicate and encodes it together with the code in the PCC binary;
5. Consumer receives the PCC binary, extracts the code and the proof, generates the safety predicate, and then uses a proof-checker to ensure that the proof of safety is indeed valid.

The remainder of this section illustrates these steps in more detail on an example.

One should keep in mind that proof-carrying code is a general technique, and it can be parameterized by the logical framework used to represent and verify the proofs, and the machine model and instruction set for the code. In this section, we use familiar first-order logic and an instruction set similar to the DEC Alpha ISA. For simplicity, we do not make specific the formal representation of the proofs. In existing implementations, Necula and Lee [17, 21] encode both the proofs and the safety predicates in the Edinburgh Logical Framework [13], and the Appendix presents a formalization based on equational logic using the specification language OBJ [11, 12].

Step 1: Safety Policy

The following example is adapted from [20]. Suppose that a 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 provides a resource access service through which user processes are given permission to access their table entry by installing native code into the kernel. The kernel invokes the user-installed code with the address of the table entry corresponding to the parent process in register $\mathtt{r}_0$. This address is guaranteed by the kernel to be valid and aligned on an 8-byte boundary.

The safety policy requires the code to follow these three rules:

1. the user code cannot access other table entries than the one poined to by $\mathtt{r}_0$,
2. the tag is read only,
3. if the tag is zero than the data item is also read only,

We now express this safety policy as a formal safety specification, in first-order logic, the language of our PCC example. This formula is all the consumer needs to give the producer.

$$\begin{array}{l}\mathsf{ReadAccess}(\mathtt{r}_0)\ \wedge \mathsf{ReadAccess}(\mathtt{r}_0+8)\ \wedge \\ \wedge\ (\mathsf{Memory}[\mathtt{r}_0] \neq 0 \Rightarrow \mathsf{WriteAccess}(\mathtt{r}_0+8))\end{array}$$

Step 2: Untrusted Code

It can be easily seen that the DEC Alpha assembly language program shown in Figure 2 satisfies the above safety policy. Initially, r_0 holds the address of the tag and the data is at the offset 8 from r_0.

```
                        % Address of tag in r0
1    ADD   r0, 8, r1    % Address of data in r1
2    LD    r0, 8(r0)    % Data in r0
3    LD    r2, -8(r1)   % Tag in r2
4    ADD   r0, 1, r0    % Increment data in r0
5    BEQ   r2, L1       % Skip if tag is zero
6    ST    r0, 0(r1)    % Write back data
L1   RET
```

Figure 2: DEC Alpha code for resource access.

The problem now is how to convince the consumer that the code does not violate the safety rules.

Step 3: Safety Predicate

The safety predicate is a formula which depends on both the safety specification and the code. It will be a valid formula if and only if the code satisfies the specification. To generate the safety predicate, the code must be annotated with invariants, similar to the method of preconditions, postconditions and loop invariants taught in undergraduate programming courses. From the annotated code, the safety predicate is constructed recursively.

Code Annotations The PCC method is different from other methods in that it requires the producer include more specific information in the code, to make the generation of safety predicates automatic.

One way to enrich code is to include *invariants*, assertions about the state of execution at certain steps, which provide a skeleton for forming the safety predicate. The sample code is annotated with invariants in Figure 3.

The invariant instructions give the certification process a hint about the state of execution at that point in the program They are not executable instructions, they are discarded after the safety check.

```
0    INV   Prec                % Precondition
1    ADD   r0, 8, r1           % Address of data in r1
2    LD    r0, 8(r0)           % Data in r0
3    LD    r2, -8(r1)          % Tag in r2
4    ADD   r0, 1, r0           % Increment data in r0
5    BEQ   r2, L1              % Skip if tag is zero
6    INV   WriteAccess(r1)
7    ST    r0, 0(r1)           % Write back data
L1   RET
```

Figure 3: Enriched code for resource access.

The precondition contains the definitions and assumptions of the consumer's safety policy. In our example, this includes the definition of the non-logical predicates ReadAccess and WriteAccess. The precondition is simply the specification formula:

$$Pre_c = \mathsf{ReadAccess}(\mathtt{r}_0) \ \wedge \mathsf{ReadAccess}(\mathtt{r}_0 + 8) \ \wedge$$
$$\wedge \ (\mathsf{Memory}[\mathtt{r}_0] \neq 0 \Rightarrow \mathsf{WriteAccess}(\mathtt{r}_0 + 8))$$

The second invariant (the one at line 6) hints to the certification process that if the branch in line 5 is not taken, then the address in register $\mathtt{r}_1$ can be safely written.

Sometimes the safety specification may include a *postcondition* which must be satisfied when the procedure returns, ensuring the machine is left in a desirable state. The example specification has no postcondition, or, its postcondition is simply "true", a condition which is always satisfied.

It should be emphasized that the invariants are not trusted blindly. If the code does not respect the given invariants, then the safety predicate generated will not be valid, and no proof provided will convince the user of safety. How the safety predicate depends upon the annotations will be explained next.

The Safety Predicate The safety predicate $\mathsf{P}_{\mathsf{Safety}}$ is a function of the code, the invariants provided, and the safety specification that the consumer specifies.

The construction of the safety predicate works

$Code_i$		VC_i
ADD	$\mathtt{r_a}, \mathtt{op}, \mathtt{r_d}$	$[\mathtt{r_d} \leftarrow \mathtt{r_s} + \mathtt{op}]\mathsf{VC}_{i+1}$
LD	$\mathtt{r_d}, \mathtt{n}(\mathtt{r_s})$	$\mathsf{ReadAccess}(\mathtt{r_s} + \mathtt{n}) \ \wedge \ [\mathtt{r_d} \leftarrow \mathsf{Memory}[\mathtt{r_s} + \mathtt{n}]]\mathsf{VC}_{i+1}$
ST	$\mathtt{r_s}, \mathtt{n}(\mathtt{r_d})$	$\mathsf{WriteAccess}(\mathtt{r_d} + \mathtt{n}) \ \wedge \ [\mathsf{Memory} \leftarrow \mathsf{Update}[\mathsf{Memory}, \mathtt{r_d} + \mathtt{n}, \mathtt{r_s}]]\mathsf{VC}_{i+1}$
BEQ	$\mathtt{r_s}, \mathtt{n}$	$((\mathtt{r_s} = 0) \Rightarrow \mathsf{VC}_{i+n+1}) \ \wedge \ ((\mathtt{r_s} \neq 0) \Rightarrow \mathsf{VC}_{i+1})$
RET		$Post$
INV	$\mathcal{I}$	$\mathcal{I}$

Table 1: The verification condition generator.

by recursively calculating a formula VC_i, called a verification condition, for each line of code, $Code_i$. The recursive definition is given in Table 1. The formulas treat the contents of registers and memory as free variables, the notation $[\text{lhs} \leftarrow \text{rhs}]\,\theta$, denotes simple substitution performed on the formula θ, in which the text on the right-hand side is substituted for the text on the left-hand side throughout the formula.

Informally, the formula VC_{i+1} is the precondition needed to safely execute the code starting with the $i+1$th line of code. That is, it is true whenever the code executes safely through the ith line of code.

The safety predicate is calculated as follows:

$$\mathsf{P}_{\mathsf{Safety}} = (\forall r_k) \bigwedge_{i \in Inv} (Inv_i \Rightarrow \mathsf{VC}_{i+1})$$

where Inv is the set of all invariants introduced in the code, while Inv_i is the ith invariant; notice that Inv_0 is the precondition. The safety predicate is quantified over all registers appearing in its conjuncts, that is, over all starting environments. An involved induction argument can be used to show that this formula is valid if and only if the code satisfies the specification.

Putting all this together, Table 2 shows the safety predicate $\mathsf{P}_{\mathsf{Safety}}$ of our code from Figure 3.

Given the specification, generating the safety predicate efficiently can be done by linearly scanning the object code from the end to the beginning, in a method similar to one-dimensional dynamic programming.

Step 4: Proof of Correctness

Provided that the code actually satisfies the specification, the code producer must now generate a formal proof of the validity of the safety predicate. This step is can be partially or fully automated[2]. The development of such tools is currently a broad area of research in theoretical computer science. A large number of tools intended to generate and check proofs have been devised recently. Such a tool, called KUMO [9, 10], is being developed at UCSD.

In this section, we deliberately avoid using a specific formalism, as doing so would be beyond the scope of this paper and complicate matters greatly with enormous amounts of notation. The reader should believe that the following proof of the sample safety predicate, expressed in English, can be formalized. This formalization is carried out in the appendix using the OBJ system.

> The last conjunct of $\mathsf{P}_{\mathsf{Safety}}$ has the form $P \Rightarrow P$ which is always true, so it can be eliminated. Similarly, the second conjunct in the first implication has the form $P \Rightarrow\ true$ which is also always true. Since $(\mathtt{r_0} + 8) - 8$ is equal to $\mathtt{r_0}$, one gets that $\mathsf{P}_{\mathsf{Safety}}$ can be reduced to "true".

[2] It should be said that the process cannot be fully automated in *all* cases. The undecidability of the halting problem prevents the development of an automated theorem prover capable of proving or disproving whether an arbitrary piece of code meets a specification. However, in most cases the programs and specifications under consideration are simple enough that the process can be automated. This has been the case in practice.

$$
\begin{aligned}
\mathsf{P}_{\mathsf{Safety}} = (\forall\, \mathtt{r}_0, \mathtt{r}_1, \mathsf{Memory})\{Pre_c \quad \Rightarrow& \quad ((\mathsf{ReadAccess}((\mathtt{r}_0 + 8) - 8) \ \wedge\ \mathsf{ReadAccess}(\mathtt{r}_0 + 8)) \ \wedge \\
& \quad \wedge\ ((\mathsf{Memory}[(\mathtt{r}_0 + 8) - 8] = 0) \Rightarrow \mathsf{true}) \ \wedge \\
& \quad \wedge\ ((\mathsf{Memory}[(\mathtt{r}_0 + 8) - 8] \neq 0) \Rightarrow \mathsf{WriteAccess}(\mathtt{r}_0 + 8)))\} \ \wedge \\
\wedge\ \{\mathsf{WriteAccess}(\mathtt{r}_1) \quad \Rightarrow& \quad \mathsf{WriteAccess}(\mathtt{r}_1)\}
\end{aligned}
$$

Table 2: Safety Predicate for the code in Figure 3

The producer's job is now finished. All that is left to do is to package the code with the proof and provide it to the consumer.

Steps 5: Consumer End Verification

Upon receipt of the code, the consumer separates the code and the proof, generates a safety predicate, and then uses a proof-checker to ensure that the proof of safety is a correct proof of the safety predicate.

Separating the code and the proof is simply a matter of conventions. A suggested convention is to store the proof of correctness in the data segment of the executable, [21, 17] , although other methods could certainly be envisioned.

The consumer generates the safety predicate S from the code using the same algorithm as the producer used in Step 3.

The consumer now checks that the proof provided by the producer is a correct proof of S. This can be done automatically and efficiently, by checking that the lines of the producer's proof follow from the rules of inference of the formal system used. In our example, this would amount to scanning the lines of the formal proof, and checking that each inference made is allowed by the rules of first-order logic. The easiest way to do this may perhaps be to have a proof assistant, for example KUMO, as part of the PCC system that can be used both by producer to generate the proof and by consumer to check it.

Discussion

This brief description of the PCC mechanism leaves two crucial issues unclarified.

The first issue is why the validity of the safety predicate ensures that the code satisfies the specification. We discuss this issue only informally here, the interested reader is referred to [18, 19] for the details of the formal proofs. The safety predicate is true iff each of its conjuncts $Inv_i \Rightarrow \mathsf{VC}_{i+1}$ is true. The hasty reader can imagine that the invariants separate basic blocks of code ended with either a branch or a return command. It can be shown that the first-order formula $Inv_i \Rightarrow \mathsf{VC}_{i+1}$ is true iff

> assuming that the safety rules specified by Inv_i are satisfied by a certain state of the system, then the code that follows up to the next invariant in the execution flow does not violate the safety rules and moreover, state arrived at upon execution satisfies the subsequent invariant

From this, a simple induction argument can show that a safety predicate is true if and only if the safety rules specified by the consumer are not violated during the execution of producer's code.

The second issue is whether or not the consumer can be fooled into accepting a piece of unsafe code as safe. Consider the case when the code is unsafe, the safety predicate S is then invalid: it has no correct proof. The proof provided by the producer will either be incorrect or prove a different predicate altogether. The consumer will not be fooled.

Therefore, as long as the safety specification was correctly formulated, the received code can be safely executed.

5 Comparisons with Other Methods

In order to highlight when its use would be most appropriate, we compare proof-carrying code with the other methods using the criteria defined in the introduction.

5.1 Efficacy

All of the techniques we have described, with the arguable exception of author authentication, are effective when properly implemented. Of course, author authentication does not guarantee safety *per se*, but rather guarantees the source of the code, and in this message authentication protocols are effective.

However, there are problems with particular implementations of these methods. The most well-known are in the Netscape Java interpreter, [6]. Complicated software opens the door for possible bugs and loopholes, and the correctness of the implementation of the safety system is a matter the consumer must take on faith. For this reason, we consider the correctness of the safety software a matter of trust, and defer its discussion until the subsection on minimal trust.

5.2 Efficiency

We have results from two performance tests comparing proof-carrying code with other methods [19, 22]. Because code using a PCC system is run as native code, it is not surprising that it is more efficient than other techniques. However, one should be careful to amortize the cost of verifying and transmitting a possibly large proof into the performance of the system.

In the first test [19], four packet filters were implemented and executed many times. The packet filters varied in complexity. Filter 1 accepted all IP packets, filter 2 accepted all IP packets originating from a given network, filter 3 accepted all IP and ARP packets exchanged between two networks, and filter 4 accepted all TCP packets with a given destination port. The safety policy enforced was that (1) memory reads are restricted to packet and scratch memory (2) memory writes are limited to scratch memory (3) all branches are forward and (4) reserved and callee-saved registers are not modified. Proof-carrying code was compared against running the packet filter in the BPF interpreter, compiling it from a safe fragment of Modula-3, and inserting software fault isolation checks into the code. The results are summarized in Figure 4.

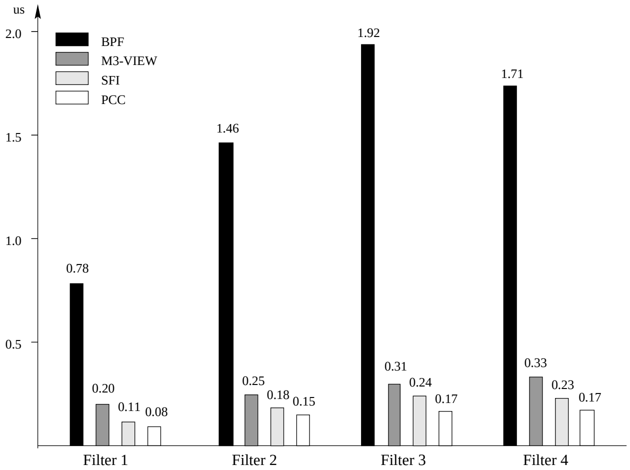

Figure 4: Average per-packet run time [19]

The second test simulated the performance of an untrusted mobile agent [22]. The mobile agent browses through a database of airline ticket and calculates the best price for a trip. The safety policy enforced was a simple one: First, memory accesses had to be properly aligned. Second, an agent was assigned an access level by the server, and it could access only records of equal or lesser access level. The running time was averaged over the number of records in the database. The results are summarized in Table 3.

System	Time (μs)	Slowdown
Proof-Carrying Code	0.030	1.0
Software Fault Isolation	0.036	1.2
Hardware Protections	0.280	9.3
Interpreted Java	1.230	41.0

Table 3: Run Time per Record and Slowdown Factors for Safety Mechanisms on a Mobile Agent [22]

A discussion of the efficiency of proof-carrying code would not be complete or honest without analyzing the cost of verifying the proof of correctness. From our perspective, this cost should be amortized.

For the mobile agent discussed in the second experiment, the proof of correctness required 370 bytes, for 112 bytes of object code. The generation of the safety-predicate required $400\mu s$, the verification of the proof required $1200\mu s$, for a one-time verification cost of $1600\mu s$. Simple algebra shows that proof-carrying code will outperform the hardware protection scheme provided that the size of the database contains a modest 6400 entries, and it will outperform interpreted Java when the database has 1334 entries. Comparison against software fault isolation cannot be made in this way because the one-time cost of inserting the checks was not provided.

In the first experiment, it was seen that the amortized cost of proof-carrying code overtook that of the BPF interpreter after about 1000 packets, Modula-3 after about 7500 packets and software fault isolation after about 22,000 packets. Given the high use of a routine such as a packet filter, it seems fair to conclude that proof-carrying is a more efficient system.

A factor which has not been measured, and is particularly relevant to mobile agents, is the increased expense of transmitting the proof along with the code. The proofs tend to be three to six times larger than the code itself, so this cost cannot be ignored. Further studies to establish the added cost of transmission for proof carrying code would be advised.

5.3 Expressiveness

The systems which predate proof-carrying code ensure that untrusted code causes no harm by monitoring and restricting its behavior. For this reason, they are limited in the properties which they can ensure.

The use of hardware protected memory domains is the least flexible safety mechanism we have mentioned: it cannot enforce any policy other than than restricting addresses memory accesses and branch targets.

Interpreters and software fault isolation systems can be more flexible. Theoretically, any property which can be detected by monitoring the execution of the program can be enforced. The difficulty is then making the interpreter or the sandbox able to monitor user-specified safety properties. We do not know if this has been done for interpreters, but there have been attempts at extending sandbox techniques, [23]. This has been done by formulating safety properties as finite automata and implementing them with the dedicated registers. This has shown limited promise because of the incurred cost, and the awkwardness of specifying the finite automaton for a complicated safety policy.

Proof-carrying code is limited only in what the formal language used to encode proofs can specify and what the code producer can prove about the code. For this reason, it is much more flexible in the properties it can guard against. Proof-carrying code has been used to ensure that mobile agents do not consume too many of the host's CPU cycles or too much of the host's network bandwidth, [22].

It may be possible to extend the proof-carrying code system to ensure termination and liveness properties, which are impossible to enforce with monitor methods. This possibility is discussed in the section on research extensions.

5.4 Minimal Trust

Author authentication is a system in which trust is the most critical resource. Is it always acceptable to assume that because you have heard of a programmer, you can trust their work? Moreover, the method cannot be used by code producers who wish to remain anonymous or who do not yet have a reputation. A possible solution to this would be to form trusted third-parties capable of certifying a piece of code as "officially safe", but this pushes the trust problem onto different shoulders, and it is unclear how such third-parties could be both effective and efficient.

The more mechanical solutions are as good as their implementations. Interpreted languages and virtual machines can be rather complicated, so it is no surprise that errors have been found in commercial implementations of safe interpreted languages, [6]. The technique for inserting safety-checks into code can be very complicated, depending on whether on how expressive system is, and whether or not the safety-checks require the registers to be reallocated.

A proof-carrying code system requires only two pieces of trusted code: the safety-predicate generator and the proof-checker. Both of these make use of simple algorithms and can be implemented in a few pages of code, thereby minimizing the possibility of loopholes being introduced by an implementation bug.

5.5 Ease of Use

Proof-carrying code achieves efficiency and flexibility at the cost of placing the responsibility of producing a formal proof of safety on the shoulders of the code producer. It is for this reason that proof-carrying code will most likely remain out of use for some time.

Interpretation and placing untrusted code in hardware protected memory segments are well-understood and implemented in contemporary systems, [16, 15, 3] Software-fault isolation requires a special program to be run by the user, to insert the bounds checking, but it imposes no restrictions on the code producer other than leaving a few unused registers, which can be easily handled by the compiler.

In contrast, proof-carrying code requires substantial effort on the part of the code producer.

The first difficulty is getting a proof of correctness out of the code producer. Requiring the progranner to prove the correctness of a piece of code is not unreasonable. In a professional situation, any programmer who cannot provide a high-level argument as to why their program works should be rightly canned. However, requiring the programmer to eke out the details of a formal proof would be cruelly inefficient and unreasonable. Thankfully this is usually unnecessary. The proofs of correctness from [20] and [22] were generated automatically, given the safety-predicates, which were in turn generated automatically from specifications and annotated code.

A more substantial difficulty is annotating the code. Providing loop invariants for a high-level language such as C is a mundane task taught is most undergraduate computer science curriculums. However, translating high-level invariants of C code into annotations for the compiled assembly code would be quite a chore. For this reason, for proof-carrying code to become practical, compilers capable of transforming high-level invariants into low-level invariants seem necessary.

Finally, there is a difficulty in getting the consumer's specification of safety to the code producer. Without the prior knowledge of the specification, the programmer cannot create the invariants necessary to generate a proof. We propose two possible solutions for this. One, is to have a trusted organization capable of setting standards publish the safety specifications. For example, Netscape could publish specifications which must be met by any applet to be downloaded and run by their browser, and Sun Microsystems could publish specifications which must be met by any module to be inserted into one of their kernels. The second is for the code producer to provide a proof of a tight, all-purpose safety specification for their code. Then, it is up to the code consumer to verify that this proof is correct and that their own desired safety specification is implied by the producer's specification. This possibility is discussed in the next section.

6 Extensions

In this section we disscuss possible extensions of the PCC mechanism. First, we present, from a software engineering point of view, a general framework for combining specifications and implementations, and then we discuss a promising example of this when the specification language is that of temporal logic.

$$
\begin{array}{ll}
\mathsf{Spec}_{Start} & \\
 & \textsc{Commands}_{Start} \\
\mathsf{Spec}_1 & \\
 & \textsc{Commands}_1 \\
\mathsf{Spec}_2 & \\
 & \textsc{Commands}_2 \\
 & \vdots \\
\mathsf{Spec}_n & \\
 & \textsc{Commands}_n \\
\mathsf{Spec}_{End} &
\end{array}
$$

Figure 5: A program layout.

6.1 Specification Extension

Combining specification and implementation languages is a continuous challenge for software engineers. Unfortunately, the two tend to be kept separate, and often the specification language is just English or another natural language. This is acceptable in most situations, in which security and reliability are not critical. However, specification languages have the advantage that they allow high level representation, design, and testing and verifying properties of systems *a priori*; additionally, specification languages allow rigorous proofs of correctness, which is a precious advantage in developing highly secure and reliable systems.

Inspired by the proof-carrying code mechanism, we propose a technique to combine specifications and implementations and then we test it on the code in Section 4 using a specification language whose underlying logic is equational logic.

We regard programs as collections of threads of execution, each thread starting with a specification, as in Figure 5. A specification expresses machine's state before the execution of a thread and they should be inserted in such a way to easily allow tracking the state changes between two specifications.

The control flow of a program can be seen as a graph whose nodes are specifications and whose edges are sequences of commands. The meaning is that of a state transition system, where a machine is in a certain node iff its state satisfies the spec and it changes the state by executing a series of commands.

Now we define the formal notion of correctness. A program is *correct* if and only if for each edge in the control flow graph, say $\textsc{Commands}_j : \mathsf{Spec}_j \to \mathsf{Spec}_k$, and for each state s_j satisfying Spec_j, if t_j is the state obtained after executing the series of commands $\textsc{Commands}_j$, then t_j satisfies Spec_k.

Spec_{Start} and Spec_{End} specify an abstract environment in which producer's program executes. So that the producer does not have to produce separate proofs of correctness for each possible consumer with a potentially different safety specification, and to increase the consumer's privacy, we do not ask the safety specification to be provided by the consumer. Rather, the producer, having in mind the intended uses of his code, specifies the safety properties of the code as tightly as possible. An eventual consumer of producer's code can either prove that his safety specification is implied by the one provided by the producer.

Despite its simplicity, this technique can be very powerful depending on the expressiveness of the specification language. We tested it on the ressource access program introduced in Section 4, using OBJ [12, 11] as a specification language for states. The correctness in this case is the program's safety. The program in Figure 3 translates to the one in Figure 6. The interested reader can find the complete proof score of correctness of this program in the Appendix.

6.2 Liveness Specifications

Suppose you don't like your system's thread scheduler, and you are considering downloading one off the internet for experimentation. Proof-carrying code, and the other systems we have discussed, can ensure that this unfamiliar code does not touch memory or devices which are off limits. But, there are still other properties one would like in a scheduler. For example, that no thread is starved (that each unblocked thread will be scheduled in the future). This is very basic property to be expected of a scheduler. Unfortunately, it is impossible to ensure by methods

```
Spec_Start {
  eq Safe(state) = true.
  eq okRead(state, state[r0]) = true.
  eq okRead(state, 8 + state[r0]) = true.
  cq okWrite(state, 8 + state[r0]) = true
    if not state{state[r0]} is 0.
}
            ADD  r0, 8, r1
            LD   r0, 8(r0)
            LD   r2, -8(r1)
            ADD  r0, 1, r0
            BEQ  r2, 2

Spec_1 {
  eq Safe(state) = true.
  cq okWrite(state, state[r1]) = true
    if not state[r2] is 0.
}
            ST   r0, 0(r1)
            RET

Spec_End {
  eq Safe(state) = true.
}
```

Figure 6: Ressource access program.

which simply monitor the execution of the code and prevent it from taking dangerous actions. However, it should be possible to ensure with an appropriate proof-carrying code system.

Properties such as liveness can be expressed in *temporal logic*, [7], a variation of first-order logic extended by the operators $\Box$ (always) and $\diamond$ (eventually). A simple liveness property could formulated "at all times, each thread will eventually be scheduled". Temporal logic has simple axiom schemas and inference rules, so there is little added difficulty for efficiently recognizing correct proofs. People have long used temporal logic to prove the formal correctness of concurrent and real time systems, [1], so the system is usable. All that would be necessary to extend proof-carrying code to prove such a property would be to expand the specification predicate generator and the proof-checker to accept the added syntax, axioms and inference rules for temporal logic.

This extension is a conceptually simple one, the formal language for specifications and proofs is enriched but the basic technique is unchanged. However, it would substantially widen the class of properties we would be able to guarantee of a piece of untrusted code.

7 Conclusions

Proof-carrying code can be thought of as a consumer rights system for computer users: I want safe, efficient code, and you have to prove it to me.

Of course, this is not always needed or even appropriate. For applications which are not performance critical, the simplicity of an interpreted language is quite convenient. For applications which are not executed very often, and the cost of transmitting and checking the proofs does not have time to be amortized, hardware-protections or sandboxing may provide better efficiency.

However, proof-carrying code does ensure very high amortized efficiency, and it is capable of guaranteeing more requirements than the "wait till it tries to do something bad and then kill it" methods. For high-use applications, such as kernel extensions, or applications with complicated requirements, such as mobile agents, proof-carrying code is promising.

We identify three areas of future research. The first is making proof-carrying code usable, by improving automated and semi-automated theorem provers for proving safety and by developing compilers which correctly transform high-level loop invariants into assembly level annotations. The second is extending proof-carrying code to ensure properties which cannot possibly be ensured by other systems, by developing a proof-carrying code system which makes use of temporal logic. Finally, performance measurements and comparisons with other methods should be broadened to allow engineers to make informed decisions about the use of proof-carrying code, in particular, measurements which take into account the transmission of the lengthy proofs should be made.

Acknowledgement: We thank George Ciprian Necula (a.k.a. Count Necula) for providing us with two of the figures used in this project.

References

[1] B. Banieqbal, H. Barringer, and A. Pnueli, editors. *Temporal Logic in Specification.* Lecture Notes in Computer Science. Springer Verlag, 1987.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] Peter M. Chen, Wee Teck Ng, Subachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: Surviving operating systems crashes. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[6] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to Netscape and beyond. In IEEE, editor, *1996 IEEE Symposium on Security and Privacy: May 6–8, 1996, Oakland, California*, pages 190–200, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[7] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[8] Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In USENIX Association, editor, *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 327–333, Berkeley, CA, USA, Winter 1993. USENIX.

[9] Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Distributed cooperative formal methods tools. In Michael Lowry, editor, *Proceedings, Automated Software Engineering*, pages 55–62. IEEE, 1997.

[10] Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Tools for distributed cooperative design and validation. In *Proceedings, CafeOBJ Symposium.* Japan Advanced Institute for Science and Technology, 1998. Nomuzu, Japan, April 1998.

[11] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison Wesley, 1985, pages 391–420.

[12] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies.* Academic, to appear. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.

[13] Robert W. Harper, F. Housell, and G. Plotkin. A framework for defining logic. *JACM*, 40(1):143–184, 1993.

[14] K. Harty and D. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1993.

[15] S. McCanne and V. Jacobson. The bsd packet filter: A new arhitecture for user-level packet capture. In *The Winter 1993 USENIX Conference*, 1993.

[16] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles*, 1987.

[17] George C. Necula. Proof-carrying code. In *POPL'97*, 1997.

[18] George C. Necula. *Compiling with Proofs.* PhD thesis, Carnegie Melon University, 1998. School of Computer Science.

[19] George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., November 1996.

[20] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation, Seattle, WA*, 1996.

[21] George C. Necula and Peter Lee. Research on proof-carrying code for untrusted-code security. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy, Oakland, 1997*, 1997.

[22] George C. Necula and Peter Lee. Safe, Untrusted Agents Using Proof-Carrying Code. In Giovanni Vigna, editor, *Mobile Agent Security*, Lecture Notes in Computer Science No. 1419, pages 61–91. Springer-Verlag, 1998.

[23] Fred B. Schneider. Enforceable security policies. Technical report, Cornell University, 1998.

[24] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley and Sons, Inc., New York, NY, USA, second edition, 1996.

[25] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), April 1996.

[26] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.

A A Formal PCC Sytem

An automated, formal correctness proof for the code in Figure 6 is presented. It is done using the specification language OBJ [12, 11] whose operational engine is based on term rewriting.

```
open SAFETY-CHECK .
*** ************** ***
***> Initial state ***
*** ************** ***
  op init : -> State .
***> Safety Specification for init
  eq Safe( init ) = true .
  eq okRead (init,     init[r0]) = true .
  eq okRead (init, 8 + init[r0]) = true .
  cq okWrite(init, 8 + init[r0]) = true
       if not init{init[r0]} is 0 .
***> First segment of code is safe
  let temp = init ;
           ADD r0, 8, r1  ;
           LD  r0, 8(r0)  ;
           LD  r2, -8(r1) ;
           ADD r0, 1, r0  ;
           BEQ r2, 2      .
  red Safe( temp ) .
***> Some properties of it
*** 1) r2 = value found at address r0
  red temp[r2] == init{init[r0]} .
*** 2) if the branch is not taken (r2 =/= 0)
***    then the address r1 is writable
  eq init{init[r0]} is 0 = false .
  red okWrite(temp, r1) .

*** ************* ***
***>  Next State  ***
*** ************* ***
  op state1 : -> State .
***> Safety Specification for State1
  eq Safe(state1) = true .
  eq okWrite(state1, state1[r1]) = true .
***> Second segment of code is safe
  red Safe(state1 ;
           ST  r0, 0(r1)  ;
           RET            ) .
*** ********************************** ***
*** End.  Therefore, the code is safe. ***
*** ********************************** ***
close
```

Notice that a template of the proof above can be generated automatically from the code, but there might still be intermediate proofs that require human intervention. Our experience with OBJ suggests that most proofs can be performed automatically using OBJ.

The proof score above can be now given to OBJ for certification, obtaining the output:

```
        \|||||||||||||||||/
      --- Welcome to OBJ3 ---
        /|||||||||||||||||\
OBJ3 version 2.04oxford built: 1994 Feb 28
Copyright 1988,1989,1991 SRI International
      1999 May 13 Thu 4:20:36
OBJ> =====================================
obj DEC-COMMANDS
==========================================
obj DEC-EXP
==========================================
obj ABSTRACT-MACHINE
==========================================
obj SAFETY-CHECK
==========================================
open
==========================================
***> Initial state ***
==========================================
op init : -> State .
==========================================
***> Safety Specification for init
==========================================
eq Safe ( init ) = true .
==========================================
eq okRead (init, init [ r0 ] ) = true .
==========================================
eq okRead (init, 8 + init[r0]) = true .
==========================================
cq okWrite (init,8 + init[r0]) = true
     if not init { init [ r0 ] } is 0 .
==========================================
***> First segment of code is safe
==========================================
let temp = init ;
         ADD r0, 8, r1  ;
         LD  r0, 8(r0)  ;
         LD  r2, -8(r1) ;
         ADD r0, 1, r0  ;
         BEQ r2, 2      .
==========================================
reduce in SAFETY-CHECK : Safe(temp1)
rewrites: 48
result Bool: true
```

```
==========================================
***> Some properties of it
==========================================
reduce in SAFETY-CHECK :
  temp[r2] == init{init[r0]}
rewrites: 63
result Bool: true
==========================================
eq init { init [ r0 ] } is 0 = false .
==========================================
reduce in SAFETY-CHECK : okWrite(temp,r1)
rewrites: 213
result Bool: true
==========================================
***>  Next State  ***
==========================================
op state1 : -> State .
==========================================
***> Safety Specification for State1
==========================================
eq Safe ( state1 ) = true .
==========================================
eq okWrite ( state1, state1[r1]) = true .
==========================================
***> Second segment of code is safe
==========================================
reduce in SAFETY-CHECK :
  Safe((state1 ; ST r0,0(r1)) ; RET)
rewrites: 3
result Bool: true
==========================================
close
OBJ> Bye.
```

Now the code can be safely executed on the consumer's machine.

The DEC Alpha's specification, the abstract machine associated with it, and the safety checker are independent from programs and hopfully are modules in specialized libraries. In the sequel we list these modules.

```
obj DEC-COMMANDS is pr INT .
  sorts Reg Operand Cmd .
  subsorts Reg Int < Operand .
  ops r0 r1 r2 : -> Reg .

  op ADD _,_,_  : Reg Operand Reg -> Cmd .
  op LD  _,_(_) : Reg Int Reg -> Cmd .
  op ST  _,_(_) : Reg Int Reg -> Cmd .
  op BEQ _,_    : Reg Int -> Cmd .
  op RET        : -> Cmd .
endo

obj DEC-EXP is pr DEC-COMMANDS .
  sort Exp .
  subsorts Operand < Exp .
  vars E E' : Exp .
  op _+_ : Exp Exp -> Exp .

  op _ is _ : Exp Exp -> Bool .
  cq E is E' = true if E == E' .
endo

obj ABSTRACT-MACHINE is pr DEC-EXP .
  sort State .
  op _;_  : State Cmd -> State .

*** value in a memory location
  op _{_} : State Exp -> Int [prec 5] .
*** evaluation of an expression
  op _[_] : State Exp -> Int [prec 5] .

  vars R R' R'' : Reg .  var OP : Operand .
  vars E E' E'' E1 E2 : Exp .
  var S : State . var N : Int .

  eq S[N] = N .
  eq S[E1 + E2] = S[E1] + S[E2] .

  op subst : Reg Exp Exp -> Exp .
  eq subst(R, E, OP) =
       if OP == R then E else OP fi .
  eq subst(R, E, E1 + E2) =
       subst(R, E, E1) + subst(R, E, E2) .
  eq subst(R, E, S{E'}) = S{subst(R, E, E')} .
  eq subst(R, E, S[E']) = S[subst(R, E, E')] .

  op update : State Exp Exp -> State .
  cq update(S, E, E'){E''} = S[E']
       if S[E''] is S[E] .

  eq (S ; ADD R, OP, R') {E} =
       S{subst(R', R + OP, E)} .
  eq (S ; ADD R, OP, R') [E] =
       S[subst(R', R + OP, E)] .

  eq (S ; LD R, N(R')) {E} =
       S{subst(R, S{N + S[R']}, E)} .
  eq (S ; LD R, N(R')) [E] =
       S[subst(R, S{N + S[R']}, E)] .

  eq (S ; ST R, N(R')) {E} =
       update(S, N + S[R'], R){E} .
  eq (S ; ST R, N(R')) [E] =
       update(S, N + S[R'], R)[E] .

  eq (S ; BEQ R, N) {E} = S{E} .
  eq (S ; BEQ R, N) [E] = S[E] .

  eq (S ; RET) {E} = S{E} .
  eq (S ; RET) [E] = S[E] .
endo

obj SAFETY-CHECK is pr ABSTRACT-MACHINE .
  op Safe     : State -> Bool .
  op okRead   : State Exp -> Bool .
  op okWrite  : State Exp -> Bool .

  vars R R' R'' : Reg .  var OP : Operand .
  vars E E' E'' E1 E2 : Exp .
  var S : State .  var N : Int .

*** ****************************
  eq Safe(S ; ADD R, OP, R') = Safe(S) .
  eq Safe(S ; LD  R, N(R'))  =
       okRead (S, N + S[R']) .
  eq Safe(S ; ST  R, N(R'))  =
       okWrite(S, N + S[R']) .
  eq Safe(S ; BEQ R, N)      = Safe(S) .
  eq Safe(S ; RET)           = Safe(S) .
*** ****************************
```

```
*** *****************************
  eq okRead((S ; ADD R, OP, R'), E) =
       okRead(S, subst(R',S[R] + S[OP],E)) .
  eq okRead((S ; LD  R, N(R')), E)  =
       okRead(S, N + S[R']) and
       okRead(S, subst(R, N + S[R'], E)) .
  eq okRead((S ; ST  R, N(R')), E)  =
       okWrite(S, N + S[R']) and
       okRead(update(S, N + S[R'], S[R]), E) .
  eq okRead((S ; BEQ R, N), E)      =
       okRead(S, E) .
  eq okRead((S ; RET), E)           =
       okRead(S, E) .
*** *****************************

*** *****************************
  eq okWrite((S ; ADD R, OP, R'), E) =
       okWrite(S,subst(R',S[R] + S[OP],E)) .
  eq okWrite((S ; LD  R, N(R')), E) =
       okRead(S, N + S[R']) and
       okWrite(S, subst(R, N + S[R'], E)) .
  eq okWrite((S ; ST  R, N(R')), E) =
       okWrite(S, N + S[R']) and
       okWrite(update(S, N + S[R'], S[R]), E) .
  eq okWrite((S ; BEQ R, N), E) = okWrite(S, E) .
  eq okWrite((S ; RET), E) = okWrite(S, E) .
*** *****************************
endo
```