Proof-Carrying Code

George C. Necula Peter Lee

September 1996
CMU-CS-96-165

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report describes *Proof-Carrying Code*, a software mechanism that allows a host system to determine with certainty that it is safe to execute a program supplied by an untrusted source. For this to be possible, the untrusted code supplier must provide with the code a *safety proof* that attests to the code's safety properties. The code consumer can easily and quickly validate the proof without using cryptography and without consulting any external agents.

In order to gain preliminary experience with proof-carrying code, we have performed a series of case studies. In one case study, we write safe assembly-language network packet filters. These filters can be executed with no run-time overhead, beyond a one-time cost of 1 to 3 milliseconds for validating the attached 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. In another case study we show how proof-carrying code can be used to develop safe assembly-language extensions of the a simplified version of the TIL run-time system for Standard ML.

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.

Keywords: Operating System Security and Protection, System Extensibility, Safety of Untrusted Code, Network Packet Filters, Assembly Language, Program Verification, Type Theory, LF, Proof-Carrying Code.

Contents

1 Introduction **2**

2 Proof-Carrying Code **5**
2.1 Defining a Safety Policy 6
2.1.1 An Abstract Machine for Memory-Safe DEC Alpha Machine Code 7
2.1.2 The Formalism for Expressing the Safety Policy 9
2.1.3 The Proof System 10
2.2 A Sample Application and its Safety Policy 11
2.2.1 An Example of an Untrusted Program 12
2.3 Certifying the Safety of Programs 13
2.3.1 Computing the Safety Predicate 13

3 Encoding Safety Proofs **17**
3.1 LF Representation 17
3.1.1 Representing Abstract Syntax: Expressions, Types and Predicates 18
3.1.2 Representing Semantics: Proofs 18
3.2 LF_0 Type Checking 21
3.3 A Binary Encoding of LF objects 24

4 Correctness of the Proof-Carrying Code Technique **26**
4.1 Soundness of the VC-based Certification 26
4.2 Adequacy of the LF Representation of Proofs 29

5 Case Study: Safe Packet Filters **34**
5.1 The Safety Policy 34
5.2 The Foreign Code 36
5.3 Performance Comparisons with Previous Approaches 40
5.4 Expressing the Safety Policy with Low-Level Constructs 44

6 Case Study: Safe Extensions of the TIL Run-Time System **46**
6.1 The Safety Policy 46
6.2 The Foreign Function 48

7 Case Study: Internet Checksum **51**
7.1 The Safety Policy 51
7.2 The Foreign Code 51
7.3 Performance Measurements 52
7.4 Expressing the Safety Policy with Low Level Constructs 52

8 Practical Difficulties **54**
8.1 Establishing the Safety Policy 54
8.2 Generating the Safety Proofs 55
8.3 Safety Proof Size 56

9 Conclusion **57**

10 Acknowledgements **59**

1 Introduction

In this report we address the problem of how a host (the *code consumer*) can determine with absolute certainty that it is safe to execute code supplied by an untrusted agent (the *code producer*). There are many manifestations of this problem.

For example, in the realm of operating systems, it is often profitable to allow application programs to install code fragments in the operating system kernel. This mechanism allows applications to customize the operation of the kernel without incurring the cost of frequent address space changes and the limitations of a fixed application–kernel interface. The problem is how can the kernel determine that the inherently untrusted application code respects its internal invariants.

The problem is exacerbated in the case of distributed and web computing, when mobile code is allowed. In this kind of situation the code producer on one part of the network produces a software component that is transmitted to the code consumer on another node for execution. How can the code consumer know that the untrusted code behaves correctly according to a set of predefined safety rules?

Finally, consider an example from high-level programming languages, which are designed and implemented with the assumption of a closed world. Taking ML [15] as an example, the programmer must normally assume that all components of the program are written in ML in order to establish that the program will have the properties conferred by type safety. In practice, however, programs often have some components written in ML and others in a different language (perhaps C or even assembly language). In such situations, we lose the guarantees provided by the design of ML, unless extremely expensive mechanisms (such as sockets and processes) are employed. In implementation terms, it is extremely difficult to determine whether the invariants of the ML heap will be respected by the foreign code, and so we must use some kind of expensive firewall or simply live dangerously.

In the situations described above, a *code consumer* must somehow become convinced that the code supplied by an untrusted *code producer* has some (previously agreed upon) set of properties. Sometimes this is referred to as establishing "trust" between the consumer and producer. Cryptography can be used to ensure that the code was produced by a trusted person or compiler [1, 19]. This scheme is weak because of its dependency on personal authority—even trusted persons, or compilers written by them, can occasionally make errors.

We propose a mechanism that allows the *code consumer* to define a safety policy and then verify that this policy is respected by native-code binaries supplied to it by an untrusted *code producer*. The mechanism stipulates 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.

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 not only standard requirements such as memory safety, but also more abstract and fine-grained guarantees about the integrity of data-abstraction boundaries. In this respect, PCC goes beyond the safety guarantees provided by other mechanisms such as Software Fault Isolation. 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).

There is an analogy between safety proofs and types. The analogy carries over to proof validation and type checking. With this analogy in mind we note that most attempts to tamper with either the code or the safety proof result in a validation error. In the few cases when the code and the proof are modified such that validation still succeeds, the new code is also safe. Another feature of the PCC method is that the proof checking algorithm is very simple, allowing fast and easy-to-trust implementations.

In our main experiment, we implemented several network packet filters [12, 16] in DEC Alpha assembly language [21] 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.

Another case study we performed considers the case of DEC Alpha assembly language extensions to the TIL compiler [23] for Standard ML [15]. In this case the safety proof certifies that the foreign code preserves the data-type representation and the heap invariants of the run-time system. Currently we are only able to certify very simple extensions that do not contain function calls and heap allocation, but we plan to expand on this in the future.

The results of these and other experiments are encouraging. For our collection of packet filters and run-time extensions, 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 for our current examples 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. 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 [24], and programming in the safe subset of Modula-3 [1, 9, 17].

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 before the approach can be considered practical. To mention just one here, we do not know at this moment what is the most practical way to generate the safety proofs. A more in-depth discussion of where we see the major difficulties in using PCC is given in Section 8.

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 report 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 and a simple resource access service. In the context of this example, we show a sample formal system for PCC and state the necessary theorems for soundness and adequacy of the methodology. 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. We continue in Section 6 with a description of how we use PCC to develop safe extensions to language run-time systems. 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.

2 Proof-Carrying Code

Figure 1 depicts the typical process of generating and using proof-carrying code. The whole process is centered around the *safety policy*, which is defined and made public by the code consumer. Through this policy, the code consumer specifies precisely under what conditions it considers the execution of a foreign program to be safe.

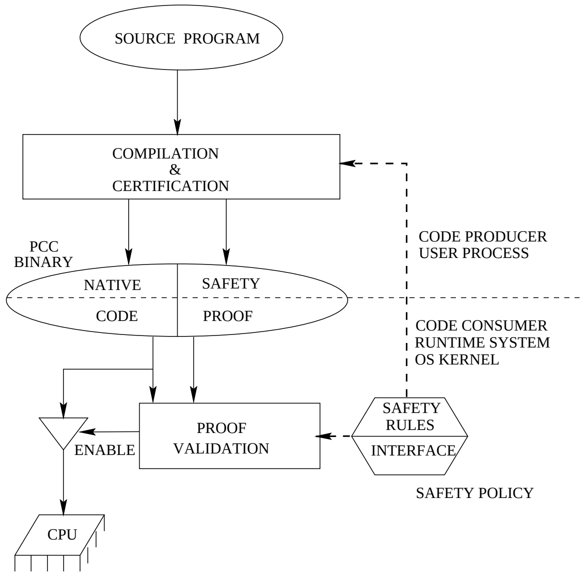

Figure 1: Overview of Proof-Carrying Code.

The safety policy consists of two main components: the *safety rules* and the *interface*. The safety rules describe all authorized operations and their associated safety preconditions. The interface describes the calling conventions between the code consumer and the foreign program, that is the invariants holding when the consumer invokes the foreign code and the invariants that the foreign code must establish before calling functions provided by the consumer. In the analogy with types, the safety rules are the typing rules and the interface is the signature that the foreign module must implement.

The life of a PCC binary spans three stages. In the first stage—called *certification*—the code producer compiles (or assembles) and generates a proof that a source program adheres to the safety policy. In the general case, certification is essentially a form of program verification with respect to the specification described by the safety policy. In addition, a proof of successful verification is produced and suitably encoded to yield the safety proof, which together with the native code component forms the PCC binary. The code producer can store the resulting PCC binary for future use, or can deliver it to code consumers for execution.

In the second stage—called *validation*—a code consumer validates the proof part of an PCC binary presented for execution and loads the native code component for execution. The validation is quick and driven by a straightforward algorithm. It is only the implementation of this simple algorithm that the consumer must trust in addition to the soundness of its safety policy.

This organization allows for the verification process to be performed off-line and only once for a given program, independently of the number of times it is executed. This has important engineering advantages, especially in cases when verification is hard and time consuming or requires user interaction. In such cases it would be undesirable to perform verification at the consumer site.

Finally, in the last stage of the process, the code consumer executes the machine-code program possibly many times. This stage can proceed without performing additional run-time checks because the previous validation stage ensures that the code obeys the safety policy.

This completes our overview of the general proof-carrying code technique. Before we can attempt a practical implementation of PCC we must decide on concrete representations for the safety policy, safety proofs and their validation procedure. We present next a summary of our current choices and continue in the next section with the details and formal soundness theorems.

In our current experiments we use extensions of first-order predicate logic as the basis for formalizing the safety policy. The extensions are predicates denoting application-specific safety requirements, together with their proof rules. In this setup, the interface part of the safety policy consists of a set of precondition and postcondition predicates for the foreign function and the functions exported by the code consumer. The safety rules are expressed as a Floyd-style verification condition generator, which given the program and a set of preconditions and postconditions produces a verification condition (also referred to as the safety predicate) in our logic. The safety predicate has the property that if it can be proved using the proof rules in our logic, then the program satisfies the safety requirements of the code consumer. In this case the safety proof is an appropriate encoding of a proof of the safety predicate, proof is reduced to theorem proving in our logic and validation to proof checking. For the particular safety policy of an example safety policy, we show that the above choices are adequate.

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 prove the safety predicate.

3. The *precondition*, which is essentially a "calling convention" that defines how the code consumer will invoke the PCC binaries, and the *postcondition* that specifies the properties of system state that the code must establish before returning to the consumer.

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 on the physical machine. 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 a subset of the DEC Alpha processor language, 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.

$$
\begin{array}{lcll}
\mathit{Instr} & ::= & \text{ADD} & \mathbf{r}_s, \mathit{Op}, \mathbf{r}_d \\
& | & \text{SUB} & \mathbf{r}_s, \mathit{Op}, \mathbf{r}_d \\
& | & \text{LD} & \mathbf{r}_d, n(\mathbf{r}_s) \\
& | & \text{ST} & \mathbf{r}_s, n(\mathbf{r}_d) \\
& | & \text{BEQ} & \mathbf{r}_s, n \\
& | & \text{BNE} & \mathbf{r}_s, n \\
& | & \text{RET} & \\
& | & \text{INV} & p \\
\\
\mathit{Op} & ::= & n \mid \mathbf{r}_i &
\end{array}
$$

Figure 2: The subset of DEC Alpha assembly language.

2.1.1 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 a subset of the Alpha architecture [21]. 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 the purpose of this report we are not considering function calls. With this arrangement we can easily ensure that the foreign code does not write the reserved or callee-saves registers by not making them available at all. We therefore 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 $\mathbf{r}_{10}$). Note that the invariant instruction INV is not an actual DEC Alpha instruction. It is just a code annotation whose purpose purpose will be explained shortly.

To define how programs are executed, we define an abstract machine as a state-transition function. The state of the abstract machine consists of the program counter pc, the values of machine registers and the state of the memory. This is sufficient for the experiments presented in this report. For applications that require more state (e.g., the state of a hardware device or the state of the code consumer's locks) the extra state can be added in a manner similar to the memory state presented below.

Following the model of the DEC Alpha processor, a machine register value is a positive integer in the range $0, \ldots, 2^{64}-1$, with negative values represented using the two's-complement representation. The state of the memory is modeled as a total mapping from the range of addresses to machine register values. In our case the range of addresses consists of those machine register values that are a multiple of 8.

It is convenient to introduce a pseudo-register of the abstract machine, called $\mathbf{r_m}$, that holds the state of the memory at any point in the computation. With this convention we denote the state of the abstract machine by the pair (ρ, pc), where ρ is the register state and pc is the program counter value. The register state is a mapping of register names to register values (or to memory values in the case of the register $\mathbf{r_m}$). The notation $\rho(\mathbf{r}_i)$ (often abbreviated as $\mathbf{r}_i$ when ρ can be recovered from the context) refers to the value of register $\mathbf{r}_i$ in state ρ. In particular the notation $\rho(\mathbf{r_m})(a)$ denotes the contents of the memory address a in state ρ. Also, we write $\rho[\mathbf{r}_i \leftarrow v]$ to denote the new register state obtained from ρ by setting the register $\mathbf{r}_i$ to v.

We introduce next a language of expressions e and of memory expressions m. The syntax of these expression languages is as follows:

$$
\begin{array}{lcll}
e & ::= & n \mid \mathbf{r}_i \mid e_1 \oplus e_2 \mid e_1 \ominus e_2 \mid \mathbf{sel}(m, e) & n \in [0, 2^{64} - 1] \\
m & ::= & \mathbf{r_m} \mid \mathbf{upd}(m, e_1, e_2) &
\end{array}
$$

The circled operations $\oplus$ and $\ominus$ denote addition and subtraction on 64-bits, as implemented by the arithmetic unit of a DEC Alpha processor. They are defined in terms of the usual arithmetic operations as follows:

$$
\begin{aligned}
e_1 \oplus e_2 &= (e_1 + e_2) \bmod 2^{64} \\
e_1 \ominus e_2 &= (e_1 - e_2) \bmod 2^{64}
\end{aligned}
$$

The expression $\mathbf{sel}(\mathbf{r_m}, e)$ in a state ρ denotes the contents of the memory location denoted by e. This is equivalent to $\rho(\mathbf{r_m})(\rho(e))$. The expression $\mathbf{upd}(\mathbf{r_m}, e_1, e_2)$ denotes the new memory state obtained by writing the register value denoted by the expression e_2 to the address denoted by e_1. These expressions are defined only in states where the address expressions denote values that are multiple of 8.

Finally, we extend the notation $\rho(e)$ to denote substitution in the expression e of register names with register values.

For a more succinct specification of the abstract machine we introduce a notation to express that in a given state an expression denotes an address that can be safely read or written. This will be useful for safety policies that include memory safety, as is the case for all examples presented in this report. For the purpose of specifying the abstract machine we do not care how it is established that a given memory location is readable or writable. The next section describes one method for the code consumer to communicate to the code producer information about memory accessibility. Yet other methods are discussed in Section 5.

Definition 2.1 *We write*

- $\rho \models e : \mathbf{ro\_addr}$ *iff the expression* $\rho(e)$ *denotes an aligned memory address* ($\rho(e) \bmod 8 = 0$) *that can be safely read in state* ρ.
- $\rho \models e : \mathbf{addr}$ *iff the expression* $\rho(e)$ *denotes an aligned memory address* ($\rho(e) \bmod 8 = 0$) *that can be safely read and written in state* ρ.

We are now prepared to give the specification of the abstract machine as presented in Figure 3. In this specification, a DEC Alpha program is a vector of instructions, Π, and the current instruction is referred to as Π_{pc}. The abstract machine is described as a state-transition function that maps a machine state (ρ, pc) into a new state (ρ', pc') by executing the current instruction Π_{pc}.

$$
(\rho, pc) \to \begin{cases}
(\rho[\mathbf{r}_d \leftarrow \rho(\mathbf{r}_s) \oplus \rho(op)], pc + 1), & \textit{if } \Pi_{pc} = \mathrm{ADD}\ \mathbf{r}_s, op, \mathbf{r}_d \\
(\rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\rho(\mathbf{r_m}), \rho(\mathbf{r}_s) \oplus n)], pc + 1), & \textit{if } \Pi_{pc} = \mathrm{LD}\ \mathbf{r}_d, n(\mathbf{r}_s) \textit{ and } \boxed{\rho \models \mathbf{r}_s \oplus n : \mathbf{ro\_addr}} \\
(\rho[\mathbf{r_m} \leftarrow \mathbf{upd}(\rho(\mathbf{r_m}), \rho(\mathbf{r}_d) \oplus n, \rho(\mathbf{r}_s))], pc + 1), & \textit{if } \Pi_{pc} = \mathrm{ST}\ \mathbf{r}_s, n(\mathbf{r}_d) \textit{ and } \boxed{\rho \models \mathbf{r}_d \oplus n : \mathbf{addr}} \\
(\rho, pc + n + 1), & \textit{if } \Pi_{pc} = \mathrm{BEQ}\ \mathbf{r}_s, n \quad \textit{and} \quad \mathbf{r}_s = 0 \\
(\rho, pc + 1), & \textit{if } \Pi_{pc} = \mathrm{BEQ}\ \mathbf{r}_s, n \quad \textit{and} \quad \mathbf{r}_s \neq 0 \\
(\rho, pc + 1), & \textit{if } \Pi_{pc} = \mathrm{INV}\ p
\end{cases}
$$

Figure 3: The Abstract Machine.

So, for example, the DEC Alpha instruction "ADD $\mathbf{r}_s, op, \mathbf{r}_d$" is defined in Figure 3 to have the following semantics:

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

where ρ is the current register and memory state. This specification states that the ADD instruction updates register $\mathbf{r}_d$ with the 64-bit sum of $\mathbf{r}_s$ and op, and also increments the program counter.

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 checks the assertions that are shown in boxes in Figure 3. For example, consider the definition of the "LD $\mathbf{r}_d, n(\mathbf{r}_s)$" instruction:

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

When executing a LD instruction the abstract machine first checks that it is safe to read from the corresponding address. The memory write operations are checked similarly. In conclusion our abstract machine only executes memory-safe programs. Of course, the presence of the safety checks means that the abstract machine is not a faithful abstraction of the DEC Alpha processor. Moreover, the careful reader might have noted that it is in fact impossible to have a practical realization of our abstract machine because the safety checks are defined in terms of the abstract notion of accessibility of memory locations. This is not a problem, however, because 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 safe behavior as on our abstract machine, even though the Alpha does not implement the safety checks.

Note that the pseudo-instruction INV behaves like a NOP for the abstract machine. This is appropriate as there is no such instruction in the DEC Alpha instruction set. In practice the invariants are kept separate from the code, allowing the code to be executed directly by the physical processor.

Mathematically, the abstract machine does not return errors when a safety check fails. Instead, the execution blocks because there are no transition rules covering the error cases. With this arrangement, a program is safe if and only if it runs without blocking on the abstract machine.

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

2.1.2 The Formalism for Expressing the Safety Policy

The abstract machine introduced above describes safety in terms of the abstract notions of readable and writable memory locations. It is the code consumer who, through the safety policy, specifies which locations are readable and which are writable. In particular the safety policy includes a *precondition* that specifies what such properties can be assumed to hold when the PCC code is invoked.

The preconditions are expressed in a language of predicates that includes the predicate **true**, conjunction, implication, universal quantification, expression equality and disequality and a typing predicate, as shown below:

$$\begin{array}{lcl} P & ::= & \mathbf{true} \mid P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall \mathbf{r}_i.P \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e : \tau \\ \tau & ::= & \mathbf{addr} \mid \mathbf{ro\_addr} \end{array}$$

Conjunction is used to specify sequences of safety requirements, while implication is used to introduce assumptions, usually due to program branching constructs or the precondition. We use universal quantification to specify that a given predicate should hold in any state.

The typing predicate is meant as a general way of expressing properties of expressions by defining appropriate types. For example, in the present case we are interested in memory safety and we define the types **addr** and **ro_addr** to express memory read/write rights. The type **addr** is used to denote valid memory addresses that can be read or written. This implies not only that the address is accessible but also that it is aligned on an 8-byte boundary. The type **ro_addr** is similar with the exception that no writes are permitted to the given memory address. We note here that the type **addr** is a subtype of **ro_addr** in the sense that an address that has type **addr** also has type **ro_addr**.

Together with the syntax of the logic, the safety policy designer must also give an interpretation of the syntactic logical constructs. This can be done in two ways. The way that we are taking in this section is to give a set of proof rules that allows the code producer to prove predicates in the logic. Another possibility for giving a meaning to predicates in our logic is to define rules for computing their truth values in a given machine state. This is effectively done by extending the Definition 2.1 to cover all the expressible predicates in our logic. The main purpose of such an interpretation of predicates is to serve as a model for proving correctness of the proof system. We defer this construction to Section 4 where we discuss the correctness of the PCC technique.

2.1.3 The Proof System

The proof system is an integral part of the safety policy and consists of axioms and inference rules that are declared valid by the code consumer for the purpose of proving predicates in the given logic. Because the logic introduced in the previous sections is composed of a fragment of first-order predicate logic and application-dependent predicates, the proof system is similarly composed of first-order predicate logic proof rules and application-dependent proof rules. We denote by $\mathcal{L}$ the entire set of proof rules in our logic and we write $\triangleright P$ when the predicate P can be proved using the proof rules in $\mathcal{L}$.

Figure 4 shows the inference rules of first-order predicate logic from $\mathcal{L}$. For our current experiments we only need conjunction, implication and universal quantification. If more complicated safety conditions need to be expressed, the technique can be extended to negation, disjunction and existential quantification by adding the appropriate proof rules.

The proof rules are shown in natural deduction style. The rules for implication and universal quantification introduction are hypothetical in u and parametrical in v. This means that the hypothesis u and the parameter v can only be used locally for the purpose of proving the implication or quantification.

In addition to the rules of first-order predicate logic, the code consumer must specify proof rules for dealing with application-specific predicates. In our case this means reasoning about the typing predicates (typing rules) and two's complement arithmetic (arithmetic rules). There is only one typing rule we need in our example, namely the subtyping relation between **addr** and **ro_addr**.

$$\frac{}{\rhd\ \mathbf{true}}\,\mathbf{true\_i} \qquad \frac{\rhd P_1 \qquad \rhd P_2}{\rhd P_1 \wedge P_2}\,\mathbf{and\_i} \qquad \frac{\rhd P_1 \wedge P_2}{\rhd P_1}\,\mathbf{and\_el} \qquad \frac{\rhd P_1 \wedge P_2}{\rhd P_2}\,\mathbf{and\_er}$$

$$\frac{\begin{array}{c}\overline{\rhd P_1}^{\,u}\\ \vdots u\\ \rhd P_2\end{array}}{\rhd P_1 \supset P_2}\,\mathbf{impl\_i}^u \qquad \frac{\rhd P_1 \supset P_2 \qquad \rhd P_1}{\rhd P_2}\,\mathbf{impl\_e} \qquad \frac{\begin{array}{c} v\\ \vdots v\\ \rhd [v/x]P\end{array}}{\rhd \forall x\ .\ P}\,\mathbf{all\_i}^v \qquad \frac{\rhd \forall x\ .\ P}{\rhd [e/x]P}\,\mathbf{all\_e}$$

Figure 4: Fragment of the first-order predicate logic proof rules.

From the arithmetic rules, we only need for our example the five rules shown in Figure 5. The choice of axioms and inference rules for reasoning about arithmetic is a delicate point. In our experiments, we have chosen the rules and axioms a bit haphazardly, extending the logic as the need arose. While this approach might be workable in some circumstances, widespread use of PCC for, say, safe applets would require that all proof validators adopt the same logic. How to choose the right system may be a difficult task, though in practice this amounts to establishing a kind of standard basis library.

$$\frac{\rhd e : \mathbf{addr}}{\rhd e : \mathbf{ro\_addr}}\,\mathbf{sub\_addr} \qquad \frac{\rhd e : \tau \qquad e = e'}{\rhd e' : \tau}\,\mathbf{tp\_congr}$$

$$\frac{e = e'}{\rhd \mathbf{sel}(m, e) = \mathbf{sel}(m, e')}\,\mathbf{sel\_congr} \qquad \frac{\rhd e \neq e'' \qquad \rhd e = e'}{\rhd e' \neq e''}\,\mathbf{neq\_congr}$$

$$\frac{}{(e \oplus e') \ominus e' = e}\,\mathbf{eq\_{+}{-}} \qquad \frac{\rhd e' = e}{\rhd e = e'}\,\mathbf{eq\_sym}$$

Figure 5: Application specific proof rules.

This completes the presentation of the basic mechanisms required for expressing the safety policy. For illustration purposes we introduce a specific application of PCC and show in that context an example safety policy.

2.2 A Sample Application and its Safety Policy

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 and data 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:

$$Pre_r = \mathbf{r}_0 : \mathbf{ro\_addr} \ \wedge \ \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr} \ \wedge \ \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0) \neq 0 \supset \mathbf{r}_0 \oplus 8 : \mathbf{addr}$$

What remains now is to prove for a particular client of the resource access service that all typing assertions 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 certification 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 postcondition) that the code releases the semaphore before returning. Again, for purposes of the current presentation, we stick to the simpler memory-safety requirements.

2.2.1 An Example of an Untrusted Program

We continue the presentation of the PCC technique in the context of the resource access service introduced in the previous section and a sample client for this service. Consider the DEC Alpha assembly language program shown in Figure 6. The overall effect of this program is to increment the data word if it is writable. We have intentionally written the program in Figure 6 in a slightly complicated way, to show that low-level code transformations 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[1], (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 complexity of the PCC technique.

It is a simple exercise for the reader familiar with assembly-language programming to verify that the code shown in Figure 6 is indeed correct with respect to the resource access safety policy. The problem, of course, is how to convince even the most suspicious kernel that this code is absolutely safe. We discuss next how PCC can be used for this purpose.

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

```
                           %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 == 0
     ST    r0, 0(r1)       %Write back data
L1   RET                   %Done
```

Figure 6: 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 $\mathbf{r}_0$.

2.3 Certifying the Safety of Programs

To create safety proofs for a program, we must prove that executing it does not violate any of the abstract machine safety checks (and also that the postcondition, if one is given, is satisfied if and when the program execution completes). 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 typing assertions.

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

These steps are described in the following subsections.

2.3.1 Computing the Safety Predicate

In order to illustrate the complete algorithm for computing the safety predicate we add to the untrusted program shown in Figure 6 two invariant instructions. The resulting program is shown in Figure 7.

The purpose of the invariant instruction in line 6 is to communicate to the certification process a hint about the state invariant at that point in the program. Specifically the invariant instruction says that if the branch in line 5 is not taken, it is invariably true that the address is register $\mathbf{r}_1$ can be safely read and written. This invariant is not trusted blindly: it is first verified and only then used, as will become clear in the next subsection. We want to stress that, for the example at hand, it is not required to have invariant instructions. We use them here only so that our example uses all of the current features of the PCC technique. Invariant instruction are crucial for dealing with programs with loops and procedure calls. Such non-trivial uses of invariants are shown in Sections 6 and 7 where examples with loops are considered.

Another invariant instruction was added at program point 0 with the invariant predicate being the precondition. We make the convention that before the safety predicate is computed for a program the precondition is prepended to it. We allow invariants to be associated with arbitrary points in the program. These points are marked by INV pseudo-instructions, and their set is denoted by Inv. For such a point i, we write Inv_i to denote the corresponding invariant. With the above convention regarding the precondition we have that $0 \in Inv$ and $Inv_0 = Pre$.

```
0    INV  Pre_r            %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 == 0
6    INV  r1 : addr
7    ST   r0, 0(r1)        %Write back data
L1   RET                   %Done
```

Figure 7: 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 $\mathbf{r}_0$.

$$VC_i = \begin{cases} [\mathbf{r}_s \oplus op/\mathbf{r}_d]VC_{i+1}, & \textit{if } \Pi_i = \text{ADD } \ \mathbf{r}_s, op, \mathbf{r}_d \\ \mathbf{r}_s \oplus n : \mathbf{ro\_addr} \wedge [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)/\mathbf{r}_d]VC_{i+1}, & \textit{if } \Pi_i = \text{LD } \ \mathbf{r}_d, n(\mathbf{r}_s) \\ \mathbf{r}_d \oplus n : \mathbf{addr} \wedge [\mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)/\mathbf{r_m}]VC_{i+1}, & \textit{if } \Pi_i = \text{ST } \ \mathbf{r}_s, n(\mathbf{r}_d) \\ (\mathbf{r}_s = 0 \supset VC_{i+n+1}) \wedge (\mathbf{r}_s \neq 0 \supset VC_{i+1}), & \textit{if } \Pi_i = \text{BEQ } \ \mathbf{r}_s, n \\ \textit{Post}, & \textit{if } \Pi_i = \text{RET} \\ \mathcal{P}, & \textit{if } \Pi_i = \text{INV } \ \mathcal{P} \end{cases}$$

Figure 8: The verification condition generator.

The safety predicate of a program is a function of the code itself, the precondition and the postcondition that the code consumer specifies, and of the invariants present in the code. The computation of the safety predicate requires a vector VC of predicates, one for each instruction, that is computed using the rules shown in Figure 8. The notation $[e/\mathbf{r}_i]P$ stands for the predicate obtained from P by substituting the expression e for all occurrences of $\mathbf{r}_i$.

Our current implementation requires that every backward-branch target be an invariant instruction. In these conditions the entire vector VC can be computed in one backward pass through the program. The only cases when the computation of an element of the VC vector would require a not-yet-computed value is for the backward branches. If backward-branch targets are invariant instructions, then VC_i can be found by a table lookup in Inv. We could relax the requirement on the placement of invariants but we must still require that every loop in the program contains at least one invariant instruction.

The rules in Figure 8 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. In an ideal world, the VC generator rules might even be taken from a standards document.

Based on the vector VC, we define the safety predicate for the entire program as follows:

$$SP(\Pi, Inv, Post) = \forall \mathbf{r}_k. \bigwedge_{i \in Inv} Inv_i \supset VC_{i+1}$$

The intuition behind a valid safety predicate is that for any initial state that satisfies the precondition Inv_0, the code Π starting at the first instruction executes without blocking and, if it terminates, the final state satisfies the postcondition $Post$.

For every invariant instruction a separate conjunct is added to the safety predicate. In essence the safety predicate contains a conjunct for every path through the program starting at the beginning or immediately after an invariant instruction and ending at a return or invariant instruction.

This effectively breaks the task of proving a large program into several simpler proving tasks for smaller segments of code. We have found that this use of invariant instructions greatly simplifies the generation of the safety proofs, especially when user interaction is involved.

Returning to our example program, the vector Π is as shown in Figure 7, the postcondition is the predicate **true** and the invariants are defined for program points 0 and 6. With this data, we apply the above formula and, after some minor simplifications, obtain the following safety predicate:

$$\begin{array}{lll} SP_r = \forall \mathbf{r}_0.\forall \mathbf{r}_1.\forall \mathbf{r_m}. \quad (Pre_r & \supset & ((\mathbf{r}_0 \oplus 8) \ominus 8 : \mathbf{ro\_addr} \wedge \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr}) \wedge \\ & & (\mathbf{sel}(\mathbf{r_m}, (\mathbf{r}_0 \oplus 8) \ominus 8) = 0 \supset \mathbf{true}) \wedge \\ & & (\mathbf{sel}(\mathbf{r_m}, (\mathbf{r}_0 \oplus 8) \ominus 8) \neq 0 \supset \mathbf{r}_0 \oplus 8 : \mathbf{addr})) \\ \qquad\qquad\qquad\qquad \wedge\, (\mathbf{r}_1 : \mathbf{addr} & \supset & \mathbf{r}_1 : \mathbf{addr}) \end{array}$$

The above safety predicate is composed of two conjuncts, one corresponding to the precondition and one to the invariant from line 6. Informally, the first conjunct says that for all values of registers $\mathbf{r}_0$, $\mathbf{r}_1$ 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. The second conjunct is not very informative, demonstrating that the use of the invariant instruction in line 6 is gratuitous.

Up to this point we have shown how to compute the safety predicate given the untrusted code, the invariants (including the precondition) and the postcondition. In order to prove the safety predicate, the code producer uses the safety rules which are part of the safety policy.

The proof $\mathcal{D}$ of the safety predicate for the example program is shown in a tree form in Figure 9. To simplify the figure we introduce some abbreviations that are shown at the bottom of the figure. Also, several proof subtrees, labelled $\mathcal{D}_1$ to $\mathcal{D}_8^{u_1,u_4}$ are shown separately. The superscripts on the proof subtrees denote proof dependency on logical parameters introduced by the implication introduction rules.

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. Each node in the proof tree is labelled with the proof rule that is used at that stage.

The proof shown in Figure 9 was generated automatically by our PCC system, which incorporates a simple theorem prover. We defer the discussion of proof generation to Section 8.

$$\mathcal{D} \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}\mathcal{D}_3^{u_1}\\ \rhd P_3\end{matrix} \quad \begin{matrix}\mathcal{D}_4^{u_1}\\ \rhd \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr}\end{matrix}}{\rhd P_3 \wedge \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr}}\,\mathbf{and\_i} \qquad \dfrac{\begin{matrix}\mathcal{D}_5^{u_1}\\ \rhd P_5\end{matrix} \quad \begin{matrix}\mathcal{D}_6^{u_1}\\ \rhd P_6\end{matrix}}{\rhd P_5 \wedge P_6}\,\mathbf{and\_i}}{\rhd (P_3 \wedge \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr}) \wedge (P_5 \wedge P_6)}\,\mathbf{and\_i}}{\rhd Pre_r \supset P_1}\,\mathbf{impl\_i}^{u_1} \qquad \begin{matrix}\mathcal{D}_2\\ \rhd P_2\end{matrix}}{\rhd (Pre_r \supset P_1) \wedge P_2}\,\mathbf{and\_i}}{\rhd \forall \mathbf{r_m}.(Pre_r \supset P_1) \wedge P_2}\,\mathbf{all\_i}^{\mathbf{r_m}}}{\rhd \forall \mathbf{r}_1.\forall \mathbf{r_m}.(Pre_r \supset P_1) \wedge P_2}\,\mathbf{all\_i}^{\mathbf{r}_1}}{\rhd \forall \mathbf{r}_0.\forall \mathbf{r}_1.\forall \mathbf{r_m}.(Pre_r \supset P_1) \wedge P_2}\,\mathbf{all\_i}^{\mathbf{r}_0}}{}$$

$$\mathcal{D}_2 \quad = \dfrac{\overline{\rhd \mathbf{r}_1 : \mathbf{addr}}^{\,u_2}}{\rhd \mathbf{r}_1 : \mathbf{addr} \supset \mathbf{r}_1 : \mathbf{addr}}\,\mathbf{impl\_i}^{u_2}$$

$$\mathcal{D}_3^{u_1} \quad = \dfrac{\begin{matrix}\mathcal{D}_7^{u_1}\\ \rhd \mathbf{r}_0 : \mathbf{ro\_addr}\end{matrix} \qquad \dfrac{\overline{\rhd (\mathbf{r}_0 \oplus 8) \ominus 8 = \mathbf{r}_0}\,\mathbf{eq\_{+}\text{-}}}{\rhd \mathbf{r}_0 = \mathbf{r}_0 \oplus 8 \ominus 8}\,\mathbf{eq\_sym}}{\rhd \mathbf{r}_0 \oplus 8 \ominus 8 : \mathbf{ro\_addr}}\,\mathbf{tp\_congr}$$

$$\mathcal{D}_4^{u_1} \quad = \dfrac{\dfrac{\overline{Pre_r}^{\,u_1}}{\rhd \mathbf{r}_0 : \mathbf{ro\_addr} \wedge \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr}}\,\mathbf{and\_el}}{\rhd \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr}}\,\mathbf{and\_er}$$

$$\mathcal{D}_5^{u_1} \quad = \dfrac{\overline{\rhd \mathbf{true}}\,\mathbf{true\_i}}{\rhd \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) = 0 \supset \mathbf{true}}\,\mathbf{impl\_i}^{u_3}$$

$$\mathcal{D}_6^{u_1} \quad = \dfrac{\dfrac{\dfrac{\overline{Pre_r}^{\,u_1}}{\rhd \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0) \neq 0 \supset \mathbf{r}_0 \oplus 8 : \mathbf{addr}}\,\mathbf{and\_er} \qquad \begin{matrix}\mathcal{D}_8^{u_1,u_4}\\ \rhd \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0) \neq 0\end{matrix}}{\rhd \mathbf{r}_0 \oplus 8 : \mathbf{addr}}\,\mathbf{impl\_e}}{\rhd \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0 \supset \mathbf{r}_0 \oplus 8 : \mathbf{addr}}\,\mathbf{impl\_i}^{u_4}$$

$$\mathcal{D}_7^{u_1} \quad = \dfrac{\dfrac{\overline{Pre_r}^{\,u_1}}{\rhd \mathbf{r}_0 : \mathbf{ro\_addr} \wedge \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr}}\,\mathbf{and\_el}}{\rhd \mathbf{r}_0 : \mathbf{ro\_addr}}\,\mathbf{and\_el}$$

$$\mathcal{D}_8^{u_1,u_4} = \dfrac{\overline{\rhd \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0}^{\,u_4} \qquad \dfrac{\overline{\rhd \mathbf{r}_0 \oplus 8 \ominus 8 = \mathbf{r}_0}\,\mathbf{eq\_{+}\text{-}}}{\rhd \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) = \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0)}\,\mathbf{sel\_congr}}{\rhd \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0) \neq 0}\,\mathbf{neq\_congr}$$

Abbreviations:

$$\begin{array}{lcl}
P_6 & = & \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0 \supset \mathbf{r}_0 \oplus 8 : \mathbf{addr} \\
P_5 & = & \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) = 0 \supset \mathbf{true} \\
P_3 & = & \mathbf{r}_0 \oplus 8 \ominus 8 : \mathbf{ro_addr} \\
P_2 & = & \mathbf{r}_1 : \mathbf{addr} \supset \mathbf{r}_1 : \mathbf{addr} \\
P_1 & = & (P_3 \wedge \mathbf{r}_0 \oplus 8 : \mathbf{ro_addr}) \wedge (P_5 \wedge P_6) \\
SP_r & = & \forall \mathbf{r}_0.\forall \mathbf{r}_1.\forall \mathbf{r_m}.(Pre_r \supset P_1) \wedge P_2 \\
Pre_r & = & (\mathbf{r}_0 : \mathbf{ro_addr} \wedge \mathbf{r}_0 \oplus 8 : \mathbf{ro_addr}) \wedge (\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0) \neq 0 \supset \mathbf{r}_0 \oplus 8 : \mathbf{addr})
\end{array}$$

Figure 9: The formal proof of the predicate SP_r in tree form.

3 Encoding Safety Proofs

We argued in the previous section and we prove in Section 4.1 that a proof of validity of the VC predicate is sufficient to ensure compliance with the safety policy. The safety proof must therefore be a suitable encoding of a derivation $\rhd SP(\Pi, \mathit{Inv}, \mathit{Post})$.

We use a two-stage encoding of derivations. In the first stage we represent predicates and proofs as expressions in the Edinburgh Logical Framework (also referred to as LF) [7]. The major benefit of this encoding is that checking the validity of a proof is equivalent to typechecking the LF representation of the proof. In the second stage, we encode LF objects in a compact binary format, suitable for storage or transmission to code consumers.

3.1 LF Representation

LF has been introduced by Harper, Honsell and Plotkin [7] as a metalanguage for high-level specification of logics. LF provides natural support for the management of binding operators, hypothetical and schematic judgments. For example it captures the convention that expressions that differ only in the names of bound variables are considered identical. Similarly, it allows direct expression of contexts and variable lookup as they arise in a hypothetical and parametrical judgement. The fact that these techniques are supported by the logical framework is a crucial factor for the succinct formalization of proofs.

It is convenient to use the LF representation both for encoding safety proof and the safety policy itself. In our implementations it is the LF representation of the safety policy that is exported to code producers and it is the LF representation of the safety predicate that is expected to accompany the untrusted code. We proceed therefore to present the representation of our logic in LF.

The formalization task ahead consists of two stages. The first stage is the representation of the abstract syntax of the logic under investigation. For example, we will show how to represent expressions, types and predicates in the logical framework. The second stage is the representation of the semantics. We do this by representing in LF the proof rules set $\mathcal{L}$. Then we show how actual proofs can be constructed from instances of proof rules.

The LF type theory is a language with entities of three levels: objects, types and kinds. Types are used to qualify expressions and similarly, kinds are used to qualify types. The abstract syntax of these entities is shown below:

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

Here `Type` is the base kind, a is a type constant and c is an object constant.

We represent our logic in LF by means of a signature Σ that assigns types to a set of constants describing the syntax of expressions and predicates, and the proof rules of our logic. Then we define a representation function that will map expressions, types, predicates and their proofs in our logic to LF objects constructed with constants declared in the signature Σ. It is the code consumer who specifies the LF signature Σ and the representation function. In fact it is the signature Σ, together with the VC generator, which constitutes the concrete representation of the safety policy exported by the code consumer.

The main representation strategy in LF is that judgements (e.g., statements about the validity of predicates) are represented as LF types and judgement derivations (e.g. a proof of a predicate) are represented as objects whose type is the representation of the judgments they prove. Type checking in the LF type discipline can then be used to check the validity of logic proofs.

We start now to present the signature Σ corresponding to our example resource access service introduced in Section 2.2. Most of the signature can then be reused for other applications.

3.1.1 Representing Abstract Syntax: Expressions, Types and Predicates

First, we define in Figure 10 the LF types `exp` of expressions, `pred` of predicates and `tp` of types. All of these are atomic LF types of kind `Type`.

$$\begin{array}{lcl} \texttt{exp} & : & \texttt{Type} \\ \texttt{pred} & : & \texttt{Type} \\ \texttt{tp} & : & \texttt{Type} \end{array}$$

Figure 10: LF signature (part 1). Base type constants.

Then for each expression, type and predicate constructor we define an LF constant as shown in Figure 11. One of the most interesting cases is the universal quantification. Care must be taken when dealing with universal quantification because of the presence of bound variables. For example, we must ensure that the representation captures the fact that the bound variable is local to the body of the quantification and that two expressions differing only in the name of the bound variables are equal. Moreover, when an expression is substituted for the bound variable we must ensure that no free variable of the substituted expression is captured.

One of the main reasons we chose LF as a proof representation language is that it provides mechanisms for dealing with bound variables. Note how the universal quantification is represented as a higher-order construct by representing the bound logical variable by a bound LF variable. This effectively delegates all the tedious manipulations of bound variables to LF.

In the predicate and proof examples we have seen so far, the machine register names are used as logical variables. In particular, once the safety predicate is computed and completely quantified over all registers, the register names lose all their special significance for a physical machine. Moreover, the problem of proving the safety predicate as well as checking its proof is independent of any physical machine or safety policy. This separation of phases allows us to use the same theorem prover independently of the specific application or machine considered and similarly to use the same proof validator independently of the theorem prover used.

The LF representation function $\ulcorner\cdot\urcorner$ is inductively defined on the structure of expressions, types and predicates as shown in Figures 12 and 13.

3.1.2 Representing Semantics: Proofs

Up to this point we have defined the representation of expressions, types and predicates in LF. Our ultimate goal is to be able to represent proofs of predicates or equivalently derivations of the validity of predicates. We follow the same pattern as for syntactic constructs and we introduce a type of proofs `pf` and then define each proof rule as an LF constant of this type. Things are actually more involved due to the fact that we want the type of a proof to determine the predicate that is being proved. In this way we verify by type checking not only that a proof is valid but also that it proves the desired predicate. This is possible to express in the LF type discipline by using type families indexed by terms.

Thus `pf` is actually a type family indexed by LF representation of predicates:

$$\texttt{pf} \quad : \quad \texttt{pred} \rightarrow \texttt{Type}$$

$$
\begin{array}{lcl}
0 & : & \mathtt{exp} \\
\mathtt{oplus} & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\
\mathtt{ominus} & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\
= & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\
<> & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\
\mathtt{sel} & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\
\mathtt{upd} & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\
= & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{pred} \\
\mathtt{hastype} & : & \mathtt{exp} \to \mathtt{tp} \to \mathtt{pred} \\
\mathtt{addr} & : & \mathtt{tp} \\
\mathtt{ro\_addr} & : & \mathtt{tp} \\
\mathtt{true} & : & \mathtt{pred} \\
\mathtt{and} & : & \mathtt{pred} \to \mathtt{pred} \to \mathtt{pred} \\
\mathtt{impl} & : & \mathtt{pred} \to \mathtt{pred} \to \mathtt{pred} \\
\mathtt{all} & : & (\mathtt{exp} \to \mathtt{pred}) \to \mathtt{pred}
\end{array}
$$

Figure 11: LF signature (part 2). Expression, types and predicate constructors.

$$
\begin{array}{rcl}
\ulcorner x \urcorner & = & x \\
\ulcorner e_1 \oplus e_2 \urcorner & = & \mathtt{oplus}\,\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \\
\ulcorner e_1 \ominus e_2 \urcorner & = & \mathtt{ominus}\,\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \\
\ulcorner e_1 = e_2 \urcorner & = & =\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \\
\ulcorner e_1 \neq e_2 \urcorner & = & <>\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \\
\ulcorner e : \tau \urcorner & = & \mathtt{hastype}\,\ulcorner e \urcorner \ulcorner \tau \urcorner \\
\ulcorner \mathbf{sel}(m, e) \urcorner & = & \mathtt{sel}\,\ulcorner m \urcorner \ulcorner e \urcorner \\
\ulcorner \mathbf{upd}(m, e_1, e_2) \urcorner & = & \mathtt{upd}\,\ulcorner m \urcorner \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner
\end{array}
$$

Figure 12: LF representation (part 1). Expressions.

Following the model of expressions and predicates we add to the signature Σ a constant for each proof rule in $\mathcal{L}$. The constants corresponding to the proof rules used by our example are shown in Figure 14 (first-order logic proof rules) and Figure 15 (application-specific proof rules).

We then extend the representation function $\ulcorner\cdot\urcorner$ to derivations. When doing so care must be taken with hypothetical and schematic judgments, such as the implication introduction and the universal quantification introduction rules. We show in Figure 17 the representation of the introduction rules for conjunction, implication and universal quantification. The representation of the conjunction introduction is typical for all other rules not shown here, including the typing and arithmetic proof rules.

The implication introduction rule introduces the hypothesis labelled u for the purpose of deriving P_2. Checking an instance of this rule schema involves verifying that it discharges properly the hypothesis u. Equivalently, the derivation $\mathcal{D}_u$ must be hypothetical in u. This is expressed naturally in LF by representing the hypothesis as a variable bound in the derivation $\mathcal{D}_u$. Finally, the LF representation of our logic contains also the representation of the application-specific proof rules. Their representation is straightforward because they do not involve hypothetical judgments. As an example we show below the LF representation of the subtyping rule **sub_addr**:

As an example of a proof representation in LF we show in Figure 18 the representation of the

$$
\begin{array}{rcl}
\ulcorner \mathbf{addr} \urcorner & = & \texttt{addr} \\
\ulcorner \mathbf{ro\_addr} \urcorner & = & \texttt{ro\_addr} \\
\\
\ulcorner \mathbf{true} \urcorner & = & \texttt{true} \\
\ulcorner P \wedge R \urcorner & = & \texttt{and}\, \ulcorner P \urcorner \ulcorner R \urcorner \\
\ulcorner P \supset R \urcorner & = & \texttt{impl}\, \ulcorner P \urcorner \ulcorner R \urcorner \\
\ulcorner \forall x. P \urcorner & = & \texttt{all}\ (\lambda x : \texttt{exp}. \ulcorner P \urcorner)
\end{array}
$$

Figure 13: LF representation (part 2). Types and predicates.

$$
\begin{array}{lcl}
\texttt{true\_i} & : & \texttt{pf true} \\
\texttt{and\_i} & : & \Pi p : \texttt{pred}. \Pi r : \texttt{pred}. \texttt{pf}\ p \to \texttt{pf}\ r \to \texttt{pf}\ (\texttt{and}\ p\ \ r) \\
\texttt{and\_el} & : & \Pi p : \texttt{pred}. \Pi r : \texttt{pred}. \texttt{pf}\ (\texttt{and}\ p\ \ r) \to \texttt{pf}\ p \\
\texttt{and\_er} & : & \Pi p : \texttt{pred}. \Pi r : \texttt{pred}. \texttt{pf}\ (\texttt{and}\ p\ \ r) \to \texttt{pf}\ r \\
\texttt{impl\_i} & : & \Pi p : \texttt{pred}. \Pi r : \texttt{pred}. (\texttt{pf}\ p \to \texttt{pf}\ r) \to \texttt{pf}\ (\texttt{impl}\ p\ \ r) \\
\texttt{impl\_e} & : & \Pi p : \texttt{pred}. \Pi r : \texttt{pred}. \texttt{pf}\ (\texttt{impl}\ p\ \ r) \to \texttt{pf}\ p \to \texttt{pf}\ r \\
\texttt{all\_i} & : & \Pi p : \texttt{exp} \to \texttt{pred}. (\Pi v : \texttt{exp}. \texttt{pf}\ (p\ v)) \to \texttt{pf}\ (\texttt{all}\ \ p) \\
\texttt{all\_e} & : & \Pi p : \texttt{exp} \to \texttt{pred}. \Pi e : \texttt{exp}. \texttt{pf}\ (\texttt{all}\ \ p) \to \texttt{pf}\ (p\ e)
\end{array}
$$

Figure 14: LF signature (part 3). First-order logic proof constants (see Figure 4).

proof of the safety predicate for our example SP_r. Notice the one-to-one mapping between the proof in the tree-form and in the LF form. Actually what is shown in Figure 18 is only the skeleton of the proof representation. For example for the `and_i` constant we only show the last two arguments and we omit the predicates. Also we do not show the argument types in LF abstractions. The complete representation is about an order of magnitude larger and somewhat difficult to read.

We have discussed in this section how to represent the abstract syntax (expressions, types and predicates) and semantics (logic proofs). This is enough for our purposes but we could go even further using the same logical framework. For example we could represent meta-theory of the logic in LF. In the case of PCC this could mean introducing uniform derivations of validity and prove that this constitutes an equivalent proof system. This gives the code producer the flexibility of choosing the proof system it wants to use, provided it convinces the code consumer of its soundness with respect to the "official" proof system. For example, it is often the case that for given logics, uniform derivations are significantly easier to produce.

$$
\begin{array}{lcl}
\texttt{sub\_addr} & : & \Pi e : \texttt{exp}. \texttt{pf}\ (\texttt{hastype}\ e\ \texttt{addr}) \to \texttt{pf}\ (\texttt{hastype}\ e\ \texttt{ro\_addr}) \\
\texttt{tp\_congr} & : & \Pi e_1 : \texttt{exp}. \Pi e_2 : \texttt{exp}. \Pi t : \texttt{tp}. \\
 & : & \qquad \texttt{pf}\ (\texttt{hastype}\ e_1\ t) \to \texttt{pf}\ (=e_1\ e_2) \to \texttt{pf}\ (\texttt{hastype}\ e_2\ t) \\
\texttt{sel\_congr} & : & \Pi m : \texttt{exp}. \Pi e_1 : \texttt{exp}. \Pi e_2 : \texttt{exp}. \texttt{pf}\ (=e_1\ e_2) \to \texttt{pf}\ (=(\texttt{sel}\ m\ e_1)\ (\texttt{sel}\ m\ e_2)) \\
\texttt{neq\_congr} & : & \Pi e_1 : \texttt{exp}. \Pi e_2 : \texttt{exp}. \Pi e_3 : \texttt{exp}. \texttt{pf}\ (\texttt{neq}\ e_1\ e_2) \to \texttt{pf}\ (=e_2\ e_3) \to \texttt{pf}\ (\texttt{neq}\ e_1\ e_3) \\
\texttt{eq\_+-} & : & \Pi e_1 : \texttt{exp}. \Pi e_2 : \texttt{exp}. \texttt{pf}\ (=(\texttt{ominus}\ (\texttt{oplus}\ e_1\ e_2)\ e_2)\ e_1) \\
\texttt{eq\_sym} & : & \Pi e_1 : \texttt{exp}. \Pi e_2 : \texttt{exp}. \texttt{pf}\ (=e_1\ e_2) \to \texttt{pf}\ (=e_2\ e_1)
\end{array}
$$

Figure 15: LF signature (part 4). Application-specific proof constants (see Figure 5).

$$\ulcorner \frac{\triangleright\, e : \mathbf{addr}}{\triangleright\, e : \mathbf{ro\_addr}} \urcorner = \mathtt{sub\_addr}\ \ulcorner e \urcorner$$

Figure 16: LF representation (part 4). Fragment of the application-specific rule representation (see Figure 5).

$$\ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \triangleright\, P_1 & \triangleright\, P_2\end{array}}{\triangleright\, P_1 \wedge P_2} \urcorner = \mathtt{and\_i}\ \ulcorner P_1 \urcorner\, \ulcorner P_2 \urcorner\, \ulcorner \mathcal{D}_1 \urcorner\, \ulcorner \mathcal{D}_2 \urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}\overline{\triangleright\, P_1}^{\,u} \\ \vdots\ \mathcal{D}_u \\ \triangleright\, P_2\end{array}}{\triangleright\, P_1 \supset P_2}\,u \urcorner = \mathtt{impl\_i}\ \ulcorner P_1 \urcorner\, \ulcorner P_2 \urcorner\ (\lambda u\!:\!\mathtt{pf}\ \ulcorner P_1 \urcorner.\ulcorner \mathcal{D}_u \urcorner)$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}_v \\ \triangleright\, [v/x]P_x\end{array}}{\triangleright\, \forall x.P_x} \urcorner = \mathtt{all\_i}\ (\lambda x\!:\!\mathtt{exp}.\ulcorner P_x \urcorner)\ (\lambda v\!:\!\mathtt{exp}.\ulcorner \mathcal{D}_v \urcorner)$$

Figure 17: LF representation (part 3). Fragment of the first-order logic rule representation (see Figure 4).

Our purpose in using the LF representation of proofs is to use the LF type-checking algorithm for checking the validity of proofs. This has the advantage that the code consumer need only trust one implementation of proof checking. Other logics can be encoded in the same framework and their derivations checked by the same type checker just by changing the signature. Furthermore, the LF typing rules are so simple that a naive implementation takes only about 4 of pages of C code. This is important because it minimizes the concern that the type checker must be trusted.

We do not show here the typing rules for full LF. Instead we define next a fragment of LF that is expressive enough to encode first-order and higher-order logics but is strictly simpler and less expressive than full LF. For this fragment, called LF_0, we show the typing rules and the adequacy of the encoding of predicates and derivations.

3.2 $\mathbf{LF}_0$ Type Checking

In this section we introduce LF_0, a fragment of full LF defined in [7]. The benefits of using LF_0 instead of full LF for proof representation and validation is that LF_0 admits a simpler type-checking algorithm that is still complete for our purposes. The results that we are deriving in this section for LF_0 mirror similar results for full LF [7].

The first observation is that LF_0 has the same syntactical elements as full LF. As a consequence the signature and LF representation function presented in the previous subsection are inherited to LF_0. It is only the typechecking algorithm that changes from full LF to LF_0.

We note also that the set of kinds in our logic (and in many possible extensions of it) is very simple and is also introduced by the trusted code consumer. We only make use of the kinds `Type`

```
all_i (λ r₀ : exp .
   all_i (λ r₁ : exp .
        all_i (λ rₘ : exp .
                 and_i
                    (impl_i (λu₁ : _ .
                             and_i (and_i (tp_congr (and_el (and_el u₁))
                                                    (eq_sym (eq_+- r₀)))
                                          (and_er (and_el u₁)))

                             (and_i (impl_i (λu₃ : _ . true_i))
                                    (impl_i (λu₄ : _ .
                                             impl_e (and_er u₁)
                                             (neq_congr u₄
                                                      (sel_congr (eq_+- r₀))))))))

                    (impl_i (λu₂ : _ . u₂)))))
```

Figure 18: LF representation of the proof shown in Figure 9.

and pred → Type. This suggests that the process of checking proofs does not have to get involved in checking well-formedness of kinds.

Similar simplifications can be made with types. All of the types involved in checking the validity of proofs are specified in the signature. These types can be assumed to be well-formed because the signature is defined by the code consumer. The first simplification of LF_0 over full LF is that signatures are trusted to be well-formed.

Another distinguishing feature of LF_0 is that it only allows second-order constants and first-order abstractions. This means that the type of all constants involved are at most second-order and in all abstractions the type of the argument is first-order. This is enough for representing a wide array of first-order and higher-order logics [7]. The benefit gained is that the normalization judgment is syntax directed and admits simple and efficient implementations. Intuitively, functions can only be applied to atomic arguments, which when substituted in the body of the function cannot generate new redices.

Finally, by examination of the LF encoding functions we notice that only LF objects in canonical form are produced. This is in fact a crucial technical detail in the proofs of adequacy in [7]. In LF_0 we define typing judgments only for objects in canonical form, thus simplifying the typing rules and the adequacy proofs. An object is in canonical form if it is in $\beta\eta$–long-normal-form.

We start by defining canonical forms for objects and types. An object (type) is in *canonical* form when it is either an abstraction or an atomic object (type) of type (kind) non-functional. An object is *atomic* if it is a constant or variable applied to "enough" canonical arguments. Formally this is defined by the judgments:

Well-typed canonical objects	$\Gamma \vdash_\Sigma M :_c A$
Well-typed atomic objects	$\Gamma \vdash_\Sigma M :_a A$

We write $\Gamma \vdash_\Sigma M :_c A$ if the object M is in canonical form of type A with respect to the type assignment Γ and the LF_0 signature Σ. This judgment is defined in Figure 19 in terms of the atomic typing judgment $\Gamma \models_\Sigma M :_a A$. An object is atomic if it is a constant or a variable applied to zero or more arguments. Enough arguments must be present such that the application has a non-functional (atomic) type.

Canonical Objects

$$\frac{\Gamma, x :_a A \models_\Sigma M :_c B \qquad \Gamma \models_\Sigma A :_a \texttt{Type}}{\Gamma \models_\Sigma \lambda x : A.M :_c \Pi x : A.B}\texttt{can\_pi}$$

$$\frac{\Gamma \models_\Sigma M :_a A \qquad \Gamma \models_\Sigma A :_a \texttt{Type}}{\Gamma \models_\Sigma M :_c A}\texttt{can\_at}$$

Atomic Objects

$$\frac{\Gamma(x) = A}{\Gamma \models_\Sigma x :_a A}\texttt{at\_var} \qquad \frac{\Sigma(c) = A}{\Gamma \models_\Sigma c :_a A}\texttt{at\_ct} \qquad \frac{\Gamma \models_\Sigma M :_a \Pi x : A.B \quad \Gamma \models_\Sigma N :_c A \quad [N/x]B \Downarrow B'}{\Gamma \models_\Sigma M\ N :_a B'}\texttt{at\_app}$$

Atomic Types

$$\frac{\Sigma(a) = K}{\Gamma \models_\Sigma a :_a K}\texttt{t\_a} \qquad \frac{\Gamma \models_\Sigma A :_a B \to K \qquad \Gamma \models_\Sigma M :_c B}{\Gamma \models_\Sigma A\ M :_a K}\texttt{t\_pi}$$

Normalization

$$\frac{}{a \Downarrow a}\texttt{na\_a} \qquad \frac{A \Downarrow A' \qquad M \Downarrow M'}{A\ M \Downarrow A'\ M'}\texttt{na\_app} \qquad \frac{A \Downarrow A' \qquad B \Downarrow B'}{\Pi x : A.B \Downarrow \Pi x : A'.B'}\texttt{na\_pi}$$

$$\frac{}{x \Downarrow x}\texttt{nm\_var} \qquad \frac{}{c \Downarrow c}\texttt{nm\_c} \qquad \frac{M \Downarrow M' \qquad N \Downarrow N'}{M\ N \Downarrow M'\ N'}\texttt{nm\_app} \qquad \frac{M \Downarrow \lambda x : A.M' \qquad N \Downarrow N'}{M\ N \Downarrow [N'/x]M'}\texttt{nm\_beta}$$

Figure 19: Typing rules for LF_0

One variation from typical presentations of LF is that instead of a definitional equivalence judgment we use a normalization judgment. Furthermore, use of normalization is localized to the `at_app` rule. This makes both the canonical and atomic typing judgments syntax directed, which simplifies the adequacy proofs.

Abstractions are restricted to be first order by the `can_pi` rule, because an atomic type cannot be a function type. This in turn, justifies the syntax-directed form of the normalization judgment. In particular, in the `nm_beta` rule, the term $[N'/x]M'$ is known to be in canonical form if M' is canonical and N' has an atomic type.

We first state a theorem that justifies the claim the LF_0 is a fragment of full LF. We use the judgement $\Gamma \models_\Sigma M :_{LF} A$ to denote typing in the full LF type discipline. The following theorem says that typing in LF_0 is a subrelation of typing in full LF. The converse holds only for canonical objects and types. Furthermore the theorem states that reduction to canonical form as defined in LF_0 preserves definitional equality of full LF.

Theorem 3.1 (Soundness of $\mathbf{LF}_0$)

1. *If* $\Gamma \models_\Sigma M :_c A$ *then* $\Gamma \models_\Sigma M :_{LF} A$.

2. *If* $\Gamma \models_\Sigma M :_a A$ *then* $\Gamma \models_\Sigma M :_{LF} A$.

3. *If* $\Gamma \vdash_\Sigma A :_a K$ *then* $\Gamma \vdash_\Sigma A :_{LF} K$.

4. *If* $\Gamma \vdash_\Sigma A :_{LF} \texttt{Type}$ *and* $A \Downarrow A'$ *then* $A \equiv_{LF} A'$ *and* $\Gamma \vdash_\Sigma A' :_{LF} \texttt{Type}$.

5. *If* $\Gamma \vdash_\Sigma M :_{LF} A$ *and* $M \Downarrow M'$ *then* $M \equiv_{LF} M'$ *and* $\Gamma \vdash_\Sigma M' :_{LF} A$.

Proof: The proof is by simultaneous induction on the structure of LF_0 derivations.

□

To recapitulate, we have given a representation function which can be used effectively to translate expressions, types, predicates and their proofs to LF. The LF objects produced are constructed using type constants and object constants drawn from the signature Σ. Both the signature Σ and the representation function are defined by the code consumer as part of specifying the safety policy. In fact the signature Σ is the concrete representation of the logic $\mathcal{L}$ that we have established in previous sections for the purpose of expressing the safety policy.

The code producer has to compute the safety predicate SP of the code that it intends to submit for execution to the code consumer. Then the producer must generate a proof $\mathcal{D}$ of the safety predicate according to the proof rules established by the consumer and described in the signature Σ. Finally, the last task that the producer must perform is to represent the proof as an LF object $M = \ulcorner \mathcal{D} \urcorner$. Note that the code producer can choose to use the LF representation of proofs as its own internal representation, saving thus the expense of converting its internal representation of proofs to LF.

In the next stage of the process the code consumer receives the machine code program Π together with the LF object M that represents the proof of safety predicate of Π. The code consumer then computes the safety predicate SP on its own. The code consumer is most likely using internally the LF representation of predicates, and thus the concrete safety predicate that is computed is $\ulcorner SP \urcorner$. The final validation step is to perform the typechecking $\cdot \vdash_\Sigma M :_c \mathtt{pf}\ \ulcorner SP \urcorner$. If this check is successful, the code consumer knows that the safety predicate is valid or, equivalently that there exists a proof of it using only the proof rules established as part of the safety policy. The formal proof that this procedure does indeed establish safety with respect to the safety policy see Section 4.

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 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. This kind of tampering with a PCC binary can be viewed as a valid code transformation.

3.3 A Binary Encoding of LF objects

At the end of the previous section we have given an overview of the PCC technique when the safety is enforced through Floyd-style verification conditions and the proofs are encoded in LF_0. We have skipped over an important engineering aspect that must be addressed by any practical implementation of PCC: the binary representation of LF objects and types. We feel that this is an important issue because in our experiments we achieved improvements in the size of proof representations and the time to typecheck them of an order of magnitude just by optimizing the representation.

A typical PCC binary in our system 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, the binary encoding of the LF representations of invariants, and the binary encoding of the LF representation of the safety proof. We briefly describe here our current binary format for LF objects.

There are four kinds of LF objects and types: constants, variables, abstractions and applications. We represent constants as indices in the current signature. This is appropriate because each constant must be declared in the signature. However, for flexibility it is sometimes useful to allow the code consumer and the code producer to use different signatures, as long as they agree on the set of constants used in the proof representation. (Such a situation occurs when the code producer uses a signature containing declarations for multiple applications and logics.) To achieve this flexibility we include with each PCC binary a symbol table, which contains a list of the external names of LF constants that occur in the proof or invariant representations. Constants in the invariant or proof are then represented externally as an index in the symbol table. When the PCC binary is loaded in memory each symbol in the symbol table is resolved using the resident signature and each constant is changed to point in the resident signature.

The next issue to be addressed is the issue of LF variables. We chose to represent variables as deBrujin indices. That means that each variable occurrence in a closed LF object is represented as an integer whose value is the number of nested abstractions between the binding abstraction and the actual occurrence. This representation has the advantage that two objects which are α-equivalent have identical representations. The disadvantage is that substitution has to take care of avoiding variable capture.

Finally abstraction and application are represented by a tag followed by a variable number of subterms. The tag specifies whether it is an application or an abstraction and the number of subterms that follow. In the case of an application the first subterm is the application head followed by all the arguments. For an abstraction the last subterm is the body and all the preceding subterms are the types of the parameters.

We stop here with the presentation of the binary representation of LF terms, noting that although the representation that we are currently using seems to be quite practical, we have only explored a tiny part of a huge design space. Figure 20 shows the sizes of these sections for the PCC binary corresponding to the resource access example. In our experiments we have observed that the size of the symbol table increases only very slightly with the size of the proof.

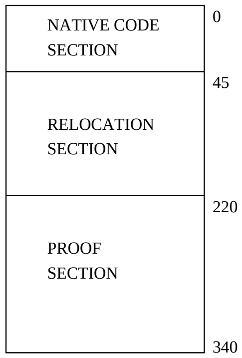

Figure 20: The layout of the PCC binary for the resource access example. The offsets are in bytes.

4 Correctness of the Proof-Carrying Code Technique

In the previous sections we have described a particular realization of the Proof-Carrying Code concept. We have used Floyd-style verification conditions to capture properties about programs and we represented proofs of first-order predicate logic in the Logical Framework (LF) for the purpose of easing their verification. In this section we argue formally that these choices lead to a sound technique for establishing the safety of untrusted code. Even though the proofs in this section are somewhat customized to the particular choices we made and sometimes even to a particular safety policy, this section is intended to be a model for proving the safety of any particular realization of PCC.

There are several issues to be dealt with when considering correctness of PCC. The first issue is the soundness of the safety policy. We show in the next subsection how this can be done for the case when the safety policy is expressed as a VC-generator (Figure 8) together with some application specific proof rules (Figure 5). Then we consider in Section 4.2 the issue of adequacy of using LF_0 typechecking as a means of validating proofs of safety predicates.

The main result of this section is Theorem 4.1 stated below. The statement of the safety theorem refers to the abstract machine defined in Figure 3. The purpose of the abstract machine is to provide a model for specifying the properties of safety predicates. If the safety policy designer does not feel the need for a formal proof of soundness then the abstract machine is not strictly required.

Theorem 4.1 (Safety) *Let Π be the native code part of a PCC binary and M be the LF object representing the safety proof contained in the PCC binary. If M such that $\cdot \vdash_\Sigma M :_c \ulcorner SP(\Pi, Inv, Post) \urcorner$ then for any initial state ρ_0 that satisfies the precondition ($\rho_0 \models Inv_0$) 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 $(\rho, pc) \to (\rho', pc')$.*

Proof: By Corollary 4.14 (see Section 4.2) we know that there exists a derivation $\mathcal{D} :: \triangleright\, SP(\Pi, Inv, Post)$, that is the safety predicate is valid. Now by using Theorem 4.7 (see Section 4.1) we immediately get the desired result.

□

Since the abstract machine gets stuck when there is any violation of a typing assertion, 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. For the particular case of our safety policy this means that the program only reads from memory locations that are defined as readable by the safety policy and only writes to writable memory locations.

The rest of this section presents the technical details of the main lemmas and theorems used in the proof of safety above. This is necessarily somewhat technical and thus the reader might want to skip any or all of the subsections and go directly to the next section where we present the first of our case studies.

4.1 Soundness of the VC-based Certification

In this subsection we prove that the safety predicate as defined in the body of the paper is indeed sufficient to ensure compliance with the safety policy. In the context of our example, this means that every memory read operation references a readable address and every write operation references a writable memory location. Also we check that upon termination, the postcondition holds. This soundness proof can be extended easily to other examples.

The first order of business is to prove define a meaning of the predicates in our logic, following the model of Definition 2.1, and then prove the soundness of the proof system $\mathcal{L}$. This is required because the safety predicate is proved using the rules in $\mathcal{L}$ while the abstract machine safety checks are defined in the terms of the $\models$ relation.

We extend the $\models$ relation introduced in Definition 2.1 to the entire set of predicates. We write $\rho \models P$ when the predicate P is true in the state ρ. This relation is defined below.

Definition 4.2 *The relation $\rho \models P$ is defined on the structure of P as follows:*

- $\rho \models \mathbf{true}$.
- $\rho \models P_1 \wedge P_2$ *iff* $\rho \models P_1$ *and* $\rho \models P_2$.
- $\rho \models P_1 \supset P_2$ *iff whenever* $\rho \models P_1$ *then* $\rho \models P_2$.
- $\rho \models \forall \mathbf{r}_i.P$ *iff* $\rho[\mathbf{r}_i \leftarrow n] \models P$ *for any* $n \in [0, 2^{64} - 1]$.
- $\rho \models e_1 = e_2$ *iff* $\rho(e_1) = \rho(e_2)$.
- $\rho \models e_1 \neq e_2$ *iff* $\rho(e_1) \neq \rho(e_2)$.
- $\rho \models e : \mathbf{ro\_addr}$ *iff the expression* $\rho(e)$ *denotes an aligned memory address* ($\rho \models e \bmod 8 = 0$) *memory address that can be safely read in state* ρ.
- $\rho \models e : \mathbf{addr}$ *iff the expression* $\rho(e)$ *denotes an aligned memory address* ($\rho \models e \bmod 8 = 0$) *memory address that can be safely read and written in state* ρ.

We say that the predicate P (possibly containing free register names) can be derived in state ρ if and only if $\triangleright\ \rho(P)$. The following soundness theorem establishes that the proof rules in Σ preserve the intuitive meaning of predicates as defined by the $\models$ relation.

Theorem 4.3 (Soundness of $\mathcal{L}$) *If* $\triangleright\ \rho(P)$ *then* $\rho \models P$.

Proof: The proof is by induction on the structure of the derivation $\mathcal{D} :: \ \triangleright\ \rho(P)$. The cases when the last rule in $\mathcal{D}$ is one of the first-order predicate logic rules is very similar to the traditional soundness proofs for first-order predicate logic. Of the application-specific rules we consider here only the **sub_addr** rule.

Case:

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\triangleright\ \rho(e : \mathbf{addr})}}{\triangleright\ \rho(e : \mathbf{ro\_addr})}\mathbf{sub\_addr}$$

By induction hypothesis on the derivation $\mathcal{D}'$ we get that $\rho \models e : \mathbf{addr}$. By definition of $\models$ we deduce that $\rho \models e \bmod 8 = 0$ and that in state ρ the memory address denoted by $\rho(e)$ can be safely read and written. Because of the readability property we can use again the definition of $\models$ and deduce the desired conclusion: $\rho \models e : \mathbf{ro\_addr}$.

$\square$

We also need to check the substitutivity property of the $\models$ relation, as defined by the following lemma.

Lemma 4.4 (Substitutivity) *If* $\rho \models [e/\mathbf{r}_i]P$ *then* $\rho[\mathbf{r}_i \leftarrow \rho(e)] \models P$.

The central result in this subsection is the progress lemma for the abstract machine. Informally, this lemma says that if the current state satisfies the VC predicate for the current instruction then either the execution terminates immediately in a state the satisfies the postcondition, or else there is a subsequent state (the execution does not halt there.)

Lemma 4.5 (Progress) *For any program* Π, *precondition Pre and postcondition Post such that* $\Pi_0 = \texttt{INV}\ Pre$, *if* $\triangleright SP(\Pi, Inv, Post)$ *and* $\rho \models VC_{pc}$ *then either:*

- $\Pi_{pc} = RET$, *and* $\rho \models Post$, *or*
- *Exists a new state* ρ' *such that* $(\rho, pc) \rightarrow (\rho', pc')$ *and* $\rho' \models VC_{pc'}$.

Proof: The proof is by case analysis of the current instruction.
Case: $\Pi_{pc} = \texttt{RET}$. Because $\rho \models VC_{pc}$ and $VC_{pc} = Post$, we conclude that $\rho \models Post$.
Case: $\Pi_{pc} = \texttt{ADD}\ \mathbf{r}_s, op, \mathbf{r}_d$. From the hypothesis $\rho \models [\mathbf{r}_s \oplus n/\mathbf{r}_d]VC_{pc+1}$. Using the substitutivity lemma we get that $\rho[\mathbf{r}_d \leftarrow \rho(\mathbf{r}_s) \oplus n] \models VC_{pc+1}$. The conclusion follows immediately if we pick $pc' = pc + 1$ and $\rho' = \rho[\mathbf{r}_d \leftarrow \rho(\mathbf{r}_s) \oplus n]$.
Case: $\Pi_{pc} = \texttt{LD}\ \mathbf{r}_d, n(\mathbf{r}_s)$. From the hypothesis

$$\rho \models \mathbf{r}_s \oplus n : \mathbf{ro\_addr} \wedge [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)/\mathbf{r}_d]VC_{pc+1}$$

By definition of $\models$ in the conjunctive case, it follows that $\rho \models \mathbf{r}_s \oplus n : \mathbf{ro\_addr}$ and $\rho \models [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)/\mathbf{r}_d]VC_{pc+1}$, which means that the side condition in the memory read rule of the abstract machine is satisfied. If we pick $pc' = pc + 1$ and $\rho' = \rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\rho(\mathbf{r_m}), \rho(\mathbf{r}_s) \oplus n)]$ we deduce (using the substitutivity lemma) that $\rho' \models VC_{pc'}$.
Case: $\Pi_{pc} = \texttt{BEQ}\ \mathbf{r}_s, n$. We distinguish two cases depending on the value of $\rho(\mathbf{r}_s)$. We only show here the case when $\rho(\mathbf{r}_s) \neq 0$. The other case is similar. From the hypothesis we get $\rho \models \mathbf{r}_s = 0 \supset VC_{pc+1} \wedge \mathbf{r}_s \neq 0 \supset VC_{pc+n+1}$. Using the definition of $\models$ for conjunction and implication we get that $\rho \models VC_{pc+n+1}$, which is exactly what we have to prove.
Case: $\Pi_{pc} = \texttt{INV}\ \mathcal{P}$. From the hypothesis we have that $\rho \models \mathcal{P}$. Now we use the validity of the safety predicate. By universal quantification elimination (with the instantiation ρ) and conjunction elimination on the proof of the safety predicate we get that $\triangleright \rho(\mathcal{P} \supset VC_{pc+1})$. By the soundness of $\mathcal{L}$ we deduce that $\rho \models \mathcal{P} \supset VC_{pc+1}$ and from here using the definition of $\models$ that $\rho \models VC_{pc+1}$, which is the desired conclusion.

□

Lemma 4.6 *For any program* Π, *set of invariants Inv and postcondition Post such that* $\Pi_0 = \texttt{INV}\ Pre$, *if* $\triangleright SP(\Pi, Inv, Post)$ *and the initial state* ρ_0 *satisfies the precondition Pre (* $\rho \models Pre$ *), then for any subsequent state* ρ *of the abstract machine such that* $(\rho_0, 0) \rightarrow^* (\rho, pc)$, *we have that* $\rho \models VC_{pc}$.

Proof: By induction on the length of the derivation $(\rho_0, 0) \rightarrow^* (\rho, pc)$. The base case follows immediately from the hypothesis observing that $VC_0 = Pre$. The inductive case is Lemma 4.5.

□

Lemmas 4.6 and 4.5 can be easily used to show that at any point during the execution of a program with a valid safety predicate, the safety check in the memory load rule is satisfied, and furthermore whenever the program terminates, it does so in a state that satisfies the postcondition. This is stated formally as Theorem 4.7, which is the main correctness result related to using verification conditions for ensuring compliance with the safety policy.

Theorem 4.7 *For any program Π such that $\Pi_0 = \texttt{INV}\ Pre$, if $\triangleright SP(\Pi, Inv, Post)$ and the initial state satisfies the precondition Pre, then 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 ($\rho \models Post$), or*

2. *The execution is not stuck, i.e., there exists a new state (ρ', pc') such that $(\rho, pc) \rightarrow (\rho', pc')$.*

Proof: From the Lemma 4.6 we know that for any state (ρ, pc) following from the initial state $\rho \models VC_{pc}$. Then by using the Lemma 4.5 we get the desired conclusion.

□

The proof of soundness of VC-based certification shown here is much simpler than other correctness arguments for Floyd's VC generators, mainly because of the more precise definition of programs, invariants and program points for assembly language programs than for flowcharts. This concludes the issue of soundness of using Floyd-style verification conditions to capture program safety. We explore next the issue of using LF_0 typing as a way to check the validity of first-order predicate logic proofs.

4.2 Adequacy of the LF Representation of Proofs

We have argued in previous sections that there are many advantages of using LF for proof representation. The major advantage is that verifying the validity of a proof is reduced to type-checking its LF representation. Therefore, we can base our proof validation on well-understood principles from type theory. Furthermore, the same type-checking algorithm is utilized for many logics and applications, just by varying the signature.

We prove in this section that with the LF_0 proof representation given in Section 3.1 and the typing rules for LF_0 shown in Figure 19, LF_0 typechecking is sufficient to ensure the validity of a proof. At the same time, by typechecking the representation of a proof we can check the identity of the proved predicate. This property is called in the literature [7] the adequacy of the proof representation.

We state below the adequacy theorems for expression, predicate and proof representation as defined by the signature Σ. The proofs for the adequacy theorems follow closely the model of similar adequacy theorems for full LF in [7]. Technically, the proofs are somewhat simpler for LF_0 because of the syntax-directed form of the typing judgments and canonical forms. If we extend the signature of first-order predicate logic with first-order proof constants, the adequacy still holds because typically problems only arise for hypothetical and parametric judgements. The theorems below are intended to be a model for proving representation adequacy of any such extension of first-order predicate logic.

In order to prove the main result we first need to prove representation adequacy for expressions and predicates. We state these theorems without proof here. Their proof is very similar but simpler than that for derivation representation.

Theorem 4.8 (Adequacy of Expression Representation.) *There is a compositional bijection $\ulcorner \cdot \urcorner$ between expressions e with free variables among $x_1, \ldots, x_n$ and atomic LF objects $\ulcorner e \urcorner$ such that $x_1 :_a \texttt{exp}, \ldots, x_n :_a \texttt{exp} \vdash_\Sigma \ulcorner e \urcorner :_c \texttt{exp}$. The bijection is compositional in the sense that $\ulcorner [e_1/x]e_2 \urcorner = [\ulcorner e_1 \urcorner / x] \ulcorner e_2 \urcorner$.*

Theorem 4.9 (Adequacy of Predicate Representation.) *There is a compositional bijection* $\ulcorner\cdot\urcorner$ *between predicates* P *with free variables among* $x_1, \ldots, x_n$ *and canonical LF objects* $\ulcorner P \urcorner$ *such that* $x_1 :_a \mathtt{exp}, \ldots, x_n :_a \mathtt{exp} \vDash_\Sigma \ulcorner P \urcorner :_c \mathtt{pred}$. *The bijection is compositional in the sense that* $\ulcorner [e/x]P \urcorner = [\ulcorner e \urcorner / x]\ulcorner P \urcorner$.

Theorem 4.10 (Adequacy of Derivation Representation.) *There is a bijection* $\ulcorner\cdot\urcorner$ *between derivations* $\mathcal{D} :: \ \triangleright P$ *with parameters* v_i $(i = 1, \ldots, n)$ *and from hypotheses* $u_j :: \ \triangleright P_j$ $(j = 1, \ldots, m)$ *and canonical LF objects* $\ulcorner \mathcal{D} \urcorner$ *such that* $v_i :_a \mathtt{exp}, u_j :_a \mathtt{pf}\ \ulcorner P_j \urcorner \vDash_\Sigma \ulcorner \mathcal{D} \urcorner :_c \mathtt{pf}\ \ulcorner P \urcorner$.

An important observation is that a canonical object of type $\mathtt{pf}\ \ulcorner P \urcorner$ for some predicate P must be also an atomic object (by inversion on the rules of canonical typing). Then we observe that an atomic object must be a variable or a constant applied to "enough arguments of the right type." This notion will be stated formally starting with the next paragraph. The next observation is that if the head of the application is a variable it can only be one of the assumptions u_j. The case of constant application head is done by case analysis on all the constants in the signature.

We introduce two new judgements for dealing with applications represented as head and argument list. Let Δ range over argument lists:

$$\Delta \quad ::= \quad \cdot \quad | \quad M, \Delta$$

We define the completion judgement $M + \Delta \Rightarrow M'$ to say that the term M applied in order to all the arguments in Δ yields term M'. This is actually needed because application is left-associative. Formally the completion judgement is defined by the two rules below:

$$\frac{}{M + \cdot \Rightarrow M}\mathbf{compl\_0} \qquad \frac{M N + \Delta \Rightarrow N'}{M + N, \Delta \Rightarrow N'}\mathbf{compl\_1}$$

Then we introduce the canonical typing judgement for argument lists $\Gamma \vDash_\Sigma \Delta :_c A \Rightarrow A'$. Informally this judgement is saying that the arguments in Δ can be applied in order to any term of type A. Furthermore the type of the resulting term would be A'. This judgement is defined by the rules below:

$$\frac{}{\Gamma \vDash_\Sigma \cdot :_c A \Rightarrow A}\mathbf{arg\_0} \qquad \frac{\Gamma \vDash_\Sigma M :_c A_1 \qquad [M/x]A_2 \Downarrow A_2' \qquad \Gamma \vDash_\Sigma \Delta :_c A_2' \Rightarrow A_2''}{\Gamma \vDash_\Sigma M, \Delta :_c \Pi x : A_1 . A_2 \Rightarrow A_2''}\mathbf{arg\_1}$$

Now we can state formally the fact that an atomic object must be a variable or a constant applied to "enough arguments of the right type."

Lemma 4.11 (Atomic Forms.) *If* $\Gamma \vDash_\Sigma M :_a A$ *and* $\Gamma \vDash_\Sigma \Delta :_c A \Rightarrow A'$ *and* $M + \Delta \Rightarrow M'$, *then exactly one of the following is true:*

1. *There is a variable* x *and an argument list* Δ' *such that* $x + \Delta' \Rightarrow M'$ *and* $\Gamma \vDash_\Sigma \Delta' :_c \Gamma(x) \Rightarrow A'$
2. *There is a constant* c *and an argument list* Δ' *such that* $c + \Delta' \Rightarrow M'$ *and* $\Gamma \vDash_\Sigma \Delta' :_c \Sigma(c) \Rightarrow A'$

Proof: The proof is by induction on the structure of the derivation $\mathcal{D} :: \ \Gamma \vDash_\Sigma M :_a A$.
Case:

$$\mathcal{D} = \frac{\Gamma(x) = A}{\Gamma \vDash_\Sigma x :_a A}\mathbf{at\_var}$$

Then $M = x$ and $A = \Gamma(x)$. We pick $\Delta' = \Delta$ and then verify using the rule **arg_0** that case 1 is satisfied.
Case:

$$\mathcal{D} = \frac{\Sigma(c) = A}{\Gamma \models_\Sigma c :_a A}\textbf{at\_ct}$$

Similar with the previous case.
Case:

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \models_\Sigma M_1 :_a \Pi x\!:\!A_1.A_2\end{array} \quad \begin{array}{c}\mathcal{D}_2 \\ \Gamma \models_\Sigma M_2 :_c A_1\end{array} \quad \begin{array}{c}\mathcal{D}_3 \\ [M_2/x]A_2 \Downarrow A\end{array}}{\Gamma \models_\Sigma M_1\ M_2 :_a A}\textbf{at\_app}$$

We try now to setup the required facts for applying the induction hypothesis on $\mathcal{D}_1$. By assumption $M_1 M_2 + \Delta \Rightarrow M'$. By using the rule **compl_1** we get that $M_1 + M_2, \Delta \Rightarrow M'$. Let Δ' be M_2, Δ. Using the rule **arg_1** with hypotheses $\mathcal{D}_2$, $\mathcal{D}_3$ and the assumption $\Gamma \models_\Sigma \Delta :_c A \Rightarrow A'$, we get that $\Gamma \models_\Sigma \Delta' :_c \Pi x\!:\!A_1.A_2 \Rightarrow A'$. Now we finish be applying the induction hypothesis on $\mathcal{D}_1$ and get the desired conclusion.

$\square$

The above lemma is stated in a more general form than we will use it. That was required for the induction proof to go through. The corollary that we actually use is:

Corollary 4.12 (Canonical Forms.) *If $\Gamma \models_\Sigma M :_c \texttt{pf}\ \ulcorner P \urcorner$ then there exists M' which is either a variable or a constant of type A', and a list of arguments Δ such that $M' + \Delta \Rightarrow M$ and $\Gamma \models_\Sigma \Delta :_c A' \Rightarrow \texttt{pf}\ \ulcorner P \urcorner$.*

Proof: The proof of the corollary is immediate by observing that M must be also atomic and then using Lemma 4.11 with an empty initial argument list.

$\square$

The next lemma states the difficult part of the adequacy for derivations, the completeness of LF representation of derivations. This is actually the implication direction that is most relevant to PCC because it states that the existence of a well-typed LF object of type $\texttt{pf}\ \ulcorner P \urcorner$ guarantees that there exists a derivation of the proof of P.

Lemma 4.13 (Completeness of LF representation for derivations.) *If $\Gamma = v_i :_a \texttt{exp}, u_j :_a \texttt{pf}\ \ulcorner P_j \urcorner$ and if $\Gamma \models_\Sigma M :_c \texttt{pf}\ \ulcorner P \urcorner$ then there exists a derivation $\mathcal{D} :: \ \triangleright P$ with parameters v_i and hypotheses u_j such that $M = \ulcorner \mathcal{D} \urcorner$.*

Proof: The proof is by induction on the structure of the derivation $\mathcal{E} = \Gamma \models_\Sigma M :_c \texttt{pf}\ \ulcorner P \urcorner$. By inversion on the structure of $\mathcal{E}$ we deduce that M must actually be an atomic object. Then we use Corollary 4.12 to deduce that there exist M' (a variable or a constant) of type A' and the list of arguments Δ such that $M' + \Delta \Rightarrow M$ and $\mathcal{F} ::\ \Gamma \models_\Sigma \Delta :_c A' \Rightarrow \texttt{pf}\ \ulcorner P \urcorner$.
We continue by case analysis on whether M' is a variable or a constant from the signature Σ.
Case: $M' = x$ and $A' = \Gamma(x)$. Because all the types in Γ are atomic it must be the case that $\Delta = \cdot$ (only rule **arg_0** could be the last rule in $\mathcal{F}$). Then $M = x$ and the derivation $\mathcal{E}$ is

$$\mathcal{E} = \frac{\dfrac{\Gamma(x) = \texttt{pf}\ \ulcorner P \urcorner}{\Gamma \models_\Sigma x :_a \texttt{pf}\ \ulcorner P \urcorner}\textbf{at\_var} \qquad \Gamma \models_\Sigma \texttt{pf}\ \ulcorner P \urcorner :_a \texttt{Type}}{\Gamma \models_\Sigma x :_c \texttt{pf}\ \ulcorner P \urcorner}\textbf{can\_pi}$$

It must be the case that $M = u_j$ for some j and therefore the derivation that we are looking for is the hypothesis u_j.
Case: $M = c$ and $A' = \Sigma(c)$. Of all the possible constants in the signature Σ we only consider here the cases of `true` (as an example of a case that actually does not apply here) and `impl_i`.
Case: $M = \texttt{true}$ and $A' = \texttt{pred}$. Following a similar line of reasoning as in the variable case it must be that the argument list is empty and **arg_0** is the last rule in $\mathcal{F}$. It follows that $\texttt{pred} = \texttt{pf}\ \ulcorner P \urcorner$ which is false. The significance of the contradiction is that the constant `true` is not an actual possibility in our case. Similar contradictions occur for all constants in Σ whose result type is not a proof.
Case: $M = \texttt{impl\_i}$ and $A = \Pi p : \texttt{pred}.\Pi r : \texttt{pred}.(\texttt{pf}\ p \to \texttt{pf}\ r) \to \texttt{pf}\ (\texttt{impl}\ p\ r)$. We follow a sequence of deductions as follows:

1. Only the rule **arg_1** could be the last in $\mathcal{F}$, thus
2. $\Delta = M_1, \Delta_1$, and
3. $\Gamma \models_{\Sigma} M_1 :_c \texttt{pred}$, and
4. $[M_1/p](\Pi r : \texttt{pred}.(\texttt{pf}\ p \to \texttt{pf}\ r) \to \texttt{pf}\ (\texttt{impl}\ p\ r)) \Downarrow A_1$. It follows that $A_1 = \Pi r : \texttt{pred}.(\texttt{pf}\ M_1 \to \texttt{pf}\ r) \to \texttt{pf}\ (\texttt{impl}\ M_1\ r)$, and
5. $\Gamma \models_{\Sigma} \Delta_1 :_c A_1 \Rightarrow \texttt{pf}\ \ulcorner P \urcorner$.
6. Only the rule **arg_1** could be the last one in derivation 5, thus
7. $\Delta_1 = M_2, \Delta_2$, and
8. $\Gamma \models_{\Sigma} M_2 :_c \texttt{pred}$,
9. $[M_2/r]((\texttt{pf}\ M_1 \to \texttt{pf}\ r) \to \texttt{pf}\ (\texttt{impl}\ M_1\ r) \Downarrow A_2$. It follows that $A_2 = (\texttt{pf}\ M_1 \to \texttt{pf}\ M_2) \to \texttt{pf}\ (\texttt{impl}\ M_1\ M_2)$, and
10. $\Gamma \models_{\Sigma} \Delta_2 :_c A_2 \Rightarrow \texttt{pf}\ \ulcorner P \urcorner$.
11. Only the rule **arg_1** could be the last one in derivation 10, thus
12. $\Delta_2 = M_3, \Delta_3$, and
13. $\Gamma \models_{\Sigma} M_3 :_c \texttt{pf}\ M_1 \to \texttt{pf}\ M_2$,
14. $[M_3/x]\texttt{pf}\ (\texttt{impl}\ M_1\ M_2) \Downarrow A_3$. It follows that $A_3 = \texttt{pf}\ (\texttt{impl}\ M_1\ M_2)$, and
15. $\Gamma \models_{\Sigma} \Delta_3 :_c A_3 \Rightarrow \texttt{pf}\ \ulcorner P \urcorner$.
16. Only the rule **arg_0** could be the last in derivation 15. By inversion we get that $\texttt{impl}\ M_1\ M_2 = \ulcorner P \urcorner$. Also Δ_3 must be empty.
17. Because of 3 and the adequacy of predicate representation there exists P_1 with appropriate free variables such that $\ulcorner P_1 \urcorner = M_1$.
18. Because of 8 there exists P_2 such that $\ulcorner P_2 \urcorner = M_2$. Therefore P must be $P_1 \supset P_2$.
19. By inversion on the derivation 13 we get that $M_3 = \lambda u : \texttt{pf}\ \ulcorner P_1 \urcorner.M_3'$ and $\Gamma, u :_a \texttt{pf}\ \ulcorner P_1 \urcorner \models_{\Sigma} M_3' :_c \texttt{pf}\ \ulcorner P_2 \urcorner$.

20. From 2, 7, 12 and 16 we get that $\Delta = M_1, M_2, M_3, \cdot$. Remember that $\texttt{impl\_i} + \Delta \Rightarrow M$. A simple calculation shows that $M = \texttt{impl\_i}\ M_1\ M_2\ M_3$. We can recover now the structure of $\mathcal{D} :: \ \Gamma \models_\Sigma M :_c \texttt{pf}\ \ulcorner P \urcorner$ (which we do not show here). Note that the derivations 3, 8, 13 and therefore 19 are included in $\mathcal{D}$. This allows us to apply the induction hypothesis on the derivation 19 and deduce that there exists a derivation $\mathcal{D}' :: \ \triangleright P_2$ from hypothesis $u :: \ \triangleright P_1$ (in addition to all of $u_j :: \ \triangleright P_j$).

21. Now we construct the derivation

$$\mathcal{D} = \dfrac{\begin{array}{c} \overline{\triangleright P_1}^{\,u} \\ \vdots \mathcal{D}' \\ \triangleright P_2 \end{array}}{\triangleright P_1 \supset P_2}\,\textbf{impl\_i}$$

It is easy to verify that $\ulcorner \mathcal{D} \urcorner = M$.

□

Now we return to the proof of adequacy of derivations (Theorem 4.10).
Proof: In one direction it is easy to verify that $\ulcorner \mathcal{D} :: \ \triangleright P \urcorner$ is a canonical LF object of type $\texttt{pf}\ \ulcorner P \urcorner$. In the other direction we use Lemma 4.11 to decompose the atomic object of type $\texttt{pf}\ \ulcorner P \urcorner$ in a variable or constant head followed by arguments of the right type. Then we use Lemma 4.13 to deduce the existence of the derivation $\mathcal{D} :: \ \triangleright P$.

□

The adequacy of derivation representation is the central result that justifies the use of LF_0 type checking as a sufficient procedure for checking validity of proofs. This is stated formally in the following corollary.

Corollary 4.14 *If P is a closed predicate and M is a canonical LF object such that $\cdot \models_\Sigma M :_c$ $\texttt{pf}\ \ulcorner P \urcorner$, then there exists a derivation $\mathcal{D} :: \ \triangleright P$, that is P is valid. Furthermore, $M = \ulcorner \mathcal{D} \urcorner$.*

Proof: The proof is immediate from Theorem 4.10 by using an empty typing environment Γ.

□

5 Case Study: Safe 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 5.3, we compare it with other approaches including interpreted packet filters (as exemplified by the BSD Packet Filter system), 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 [16], packet filters have been used successfully in network monitoring and diagnosis.

5.1 The Safety Policy

In our approach a 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).

There is a wide range of techniques that the safety policy designer can use to express the precondition and the application-specific safety rules. For uniformity we choose to use the framework developed in Section 2 for the resource access service and extend it so that it covers the case at hand. We extend the logic with another kind of a typing predicate written $m \vdash e : \tau$. The difference between this typing predicate and the one already existent in the logic is the presence of the memory state. We extend the system of types as follows:

$$\tau \;::=\; \mathbf{addr} \;\mid\; \mathbf{ro\_addr} \;\mid\; \mathbf{unit} \;\mid\; \mathbf{array}(\tau_1, \tau_2, e_0, e_l)$$

The type **unit** describes an expression of which no significant properties can be asserted. It is used for convenience in places where a type is required but there is nothing interesting to say about the qualified expression. We also introduce the type `array` to characterize a sequence of memory locations. Informally, $m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l)$ means that in the memory state m, the expression e denotes a memory address of a subarray of length e_l. The subarray is part of the array starting at e_0 and each of its elements have type τ_2 (possibly `unit`). The type τ_1 can be either **ro_addr** or **addr** or and is used to distinguish between a read-only array and a writable one.

The definition of an array implies more that the memory addresses covered by the array are accessible. For example, a subarray always starts on an aligned address. Also all arrays are allocated far-enough from the end of the virtual memory range (2^{64}) such that overflow cannot occur when doing address arithmetic. Finally, arrays are allocated such that that there is no aliasing between

$$\frac{\triangleright m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l)}{\triangleright e_0 \le e}\mathbf{arr\_lbound} \qquad \frac{\triangleright m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l)}{\triangleright e \bmod 8 = 0}\mathbf{arr\_align}$$

$$\frac{\triangleright m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l)}{\triangleright e : \tau_1}\mathbf{arr\_idx} \qquad \frac{\triangleright m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l)}{\triangleright m \vdash \mathbf{sel}(m, e) : \tau_2}\mathbf{arr\_sel}$$

$$\frac{\triangleright m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l) \qquad \triangleright e' \bmod 8 = 0 \qquad \triangleright e'_l + e' \le e_l}{\triangleright m \vdash e + e' : \mathbf{array}(\tau_1, \tau_2, e_0, e'_l)}\mathbf{arr\_sub}$$

$$\frac{\triangleright m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l) \qquad \triangleright e'_l \le e_l}{\triangleright m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e'_l)}\mathbf{arr\_sub0} \qquad \frac{\triangleright m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l)}{\triangleright e \oplus e_l = e + e_l}\mathbf{arr\_overflow}$$

$$\frac{\triangleright m \vdash e' : \mathbf{array}(\tau'_1, \tau'_2, e'_0, e'_l) \qquad \triangleright m \vdash e'' : \mathbf{array}(\tau''_1, \tau''_2, e''_0, e''_l) \qquad \triangleright e'_0 \ne e''_0}{\triangleright e' \ne e''}\mathbf{arr\_alias}$$

Figure 21: Application-dependent typing rules.

two arrays starting at different addresses e_0. This final property is crucial when reasoning about programs with side-effects.

We now extend the definition of the $\models$ relation to cover the new types.

Definition 5.1 $\rho \models m \vdash e : \mathbf{array}(\tau_1, \tau_2, e_0, e_l)$ *iff*

- τ_1 *is either* **addr** *or* **ro_addr**, *and*
- $\rho \models e_0 \bmod 8 = 0$, *and*
- $\rho \models e \ge e_0$ *and* $\rho \models e \bmod 8 = 0$, *and*
- $\rho \models e \oplus e_l = e + e_l$, *and*
- *For any* i *such that* $i >= \rho(e_0)$ *and* $i < \rho(e + e_l)$ *and* $i \bmod 8 = 0$ *we have that* $\rho \models i : \tau_1$ *and* $\rho \models m \vdash \mathbf{sel}(m, i) : \tau_2$, *and*
- *For any* e' *such that* $\rho \models m \vdash e' : \mathbf{array}(\tau'_1, \tau'_2, e'_0, e'_l)$ *such that* $\rho(e_0) \ne \rho(e'_0)$ *we have that* $\rho \models e \ne e'$.

We note that it is in fact possible to express all the above properties of arrays within the logic we have used for the resource access service without introducing the type **array**. In fact we briefly explore this alternative in Section 5.4. The major advantage of using high-level types, such as **array**, is that it is easier to reason about them than about the equivalent universally quantified formulas and consequently the proofs involving them are shorter and easier to produce.

The properties of the `array` type are expressed formally in Figure 21. Keep in mind that all of the arithmetic in our logic involves only positive values. For convenience, we have used in the proof rules for arrays the usual arithmetic operators $+$ and $-$ (in addition to the machine operations $\oplus$ and $\ominus$). These operations could have results outside the range $0, \ldots, 2^{64}$ but they are only used in assertions that rule out these cases.

As expected we want to convince ourselves that the set of proof rules from Figure 21 are sound with respect to the agreed upon definition of arrays. This is formalized in the following theorem:

Theorem 5.2 (Soundness of array proof rules) *If* $\rhd \rho(P)$ *then* $\rho \models P$.

Proof: We only consider here the rules dealing with arrays. The proof is as usual by induction on the structure of the derivation $\mathcal{D} :: \ \rhd \rho(P)$. Note that the soundness of the rules **arr_lbound**, **arr_align** and **arr_alias** follow directly from the induction hypothesis and the definition of $\models$ for arrays. The rest of the rules can be proven easily if we remember that all expressions e are positive. $\square$

Returning to the precondition of packet filters in the BPF semantics we express it naturally using arrays as follows:

$$\mathit{Pre} = \mathbf{r_m} \vdash \mathbf{r}_1 : \mathbf{array}(\mathbf{ro\_addr}, \mathtt{unit}, \mathbf{r}_1, \mathbf{r}_2) \ \wedge \ \mathbf{r_m} \vdash \mathbf{r}_3 : \mathbf{array}(\mathbf{addr}, \mathtt{unit}, \mathbf{r}_3, 16) \ \wedge \ \mathbf{r}_1 \neq \mathbf{r}_3$$

The above precondition is read as follows: register $\mathbf{r}_1$ points to a read-only array of length denoted by $\mathbf{r}_2$ (the packet); register $\mathbf{r}_3$ points to a writable array of length 16 (the scratch memory); the packet and the scratch memory are distinct arrays (there is no aliasing).

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

5.2 The Foreign Code

We have implemented four typical packet filters in assembly language and certified their safety with respect to the packet filter safety policy. In addition to the DEC Alpha instructions introduced in Section 2 we also use the following instructions in developing packet filters:

Instr	::=	...		Previously defined instructions
	|	`SHL`	$\mathbf{r}_s, op, \mathbf{r}_d$	Shift left $\mathbf{r}_s$ by *op* bits
	|	`SHR`	$\mathbf{r}_s, op, \mathbf{r}_d$	Shift right $\mathbf{r}_s$ by *op* bits
	|	`EXTW`	$\mathbf{r}_s, op, \mathbf{r}_d$	Extract a word (2 bytes) from position *op* in the register $\mathbf{r}_s$
	|	`EXTB`	$\mathbf{r}_s, op, \mathbf{r}_d$	Extract a byte from position *op* in the register $\mathbf{r}_s$
	|	`LDAH`	$\mathbf{r}_d, n[\mathbf{r}_s]$	Add $n * 2^{16}$ to $\mathbf{r}_s$

The definition of `EXTW` $\mathbf{r}_s, op, \mathbf{r}_d$ is $(\mathbf{r}_s{>>}(op*8)) \bmod 2^{16}$. The abstract machine definitions for the other instructions are obvious. Also the *VC* rules for the new instructions are very similar to those for previously defined instructions.

Although in the previous sections of the paper we have used register names from 0 through 10 we remind the reader that in fact when we write actual assembly language we use the names of the 11 caller-saves and temporary registers. In the code listings below we use the actual DEC Alpha registers. In particular the address of the packet is in register $\mathbf{r}_{16}$, the length of the packet in $\mathbf{r}_{17}$ and the address of the scratch memory is in $\mathbf{r}_{18}$. We now proceed with the description of the 4 filter that we implemented.

Filter 1 accepts all IP packets. This is done by comparing a 16-bit word at offset 12 in the packet to the constant `ETHER_IP`. The DEC Alpha assembly language code for this packet filter is given in Figure 22.

Filter 2 accepts IP packets originating from a given network (with number 128.2.206). This involves checking a 24-bit value (at offset 27 in the packet) in addition to the work done by Filter 1. The DEC Alpha assembly language code for this packet filter is given in Figure 23.

Filter 3 accepts IP or ARP packets exchanged between two given networks (128.2.206 and 128.2.209). This is the filter with the most complex control-flow among those considered in our experiments. This filter first branches depending on whether the packet is an IP packet or an ARP

```
        LD    r0, 8[r16]     % Load bytes at offset 8-15
        EXTW  r0, 4, r1      % Extract 2 bytes at offset 12
        SUB   r1, 8, r2      % Compare to ETHER_IP
        BNE   r2, Lfail
        MOV   r0, 1          % Accept
        RET
Lfail : MOV   r0, 0          % Fail
        RET
```

Figure 22: DEC Alpha assembly language for Filter 1. The address of the packet is in $\mathbf{r}_{16}$ and its length in $\mathbf{r}_{17}$. This filter accepts exactly the IP packets.

```
        LD    r0, 8[r16]     % Load bytes at offset 8-15
        MOV   r2, 640        % Prepare address 128.2 in big-endian
        EXTW  r0, 4, r1      % Extract 2 bytes at offset 12
        LDAH  r2, 206[r2]    % Prepare address 128.2.206 in big-endian
        SUB   r1, 8, r0      % Compare to ETHER_IP
        BNE   r0, Lfail
        LD    r0, 24[r16]    % Load bytes at 24-31
        SHL   r0, 24, r1
        SHR   r1, 40, r1     % 3 bytes at offset 27
        SUB   r1, r2, r0     % Compare with address
        BNE   r0, Lfail
        MOV   r0, 1          % Accept
        RET
Lfail : MOV   r0, 0          % Fail
        RET
```

Figure 23: DEC Alpha assembly language for Filter 2. On input the start address of the packet is in $\mathbf{r}_{16}$ and its length in $\mathbf{r}_{17}$. This filter accepts exactly the IP packets originating from the network 128.2.206.

packet (the packet is rejected if neither is true). Then in each case the originating network address (3 bytes at offset 27 for IP packets and at offset 28 for ARP packets) is compared to 128.2.206 and then to 128.2.209. If the originating address test passes, the same test is done for the destination address (3 bytes at offset 30 for IP packets and at offset 38 for ARP packets). The filter code is further complicated by the fact that the DEC Alpha only loads aligned 8-byte words and some of the quantities we compare spread across alignment boundaries. The DEC Alpha assembly language code for this filter is given in Figure 24. We have made an attempt to schedule the code for this filter to accommodate the DEC Alpha pipeline latencies.

Finally, 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, then that this is the first packet in a sequence and lastly that the destination port matches a given value. The interesting feature of this filter over the others is that the offset TCP destination port is computed based on a byte in the IP header (the length of the IP header). Because it is not guaranteed that this computation yields an offset less than the minimum packet size (64 bytes), filter 4 contains a run-time bounds check. Furthermore the possible lack of alignment of the TCP port byte requires a few extra instructions. The DEC Alpha assembly language code for this filter is shown in Figure 25.

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

	LD	$\mathbf{r}_6, 8[\mathbf{r}_{16}]$	% Load bytes at offset 8-15
	MOV	$\mathbf{r}_3, 640$	% Prepare address 128.2 in big-endian
	EXTW	$\mathbf{r}_6, 4, \mathbf{r}_1$	% Extract 2 bytes at offset 12
	LDAH	$\mathbf{r}_4, 206[\mathbf{r}_3]$	% Prepare address 128.2.206 in big-endian
	SUB	$\mathbf{r}_1, 8, \mathbf{r}_5$	% Compare to ETHER_IP
	LDAH	$\mathbf{r}_3, 209[\mathbf{r}_3]$	% Prepare address 128.2.209 in big-endian
	LD	$\mathbf{r}_6, 24[\mathbf{r}_{16}]$	% Load bytes at offset 24-31
	BEQ	$\mathbf{r}_5, L_{\mathrm{IP}}$	
	SUB	$\mathbf{r}_1, 1544, \mathbf{r}_2$	% Compare to ETHER_ARP
	BNE	$\mathbf{r}_2, L_{\mathrm{fail}}$	
	JMP	L_{ARP}	
L_{IP} :	LD	$\mathbf{r}_7, 32[\mathbf{r}_{16}]$	% Load bytes at offset 32-39
	SHL	$\mathbf{r}_6, 24, \mathbf{r}_0$	
	SHR	$\mathbf{r}_0, 40, \mathbf{r}_1$	% 3 bytes at offset 27-29
	SUB	$\mathbf{r}_1, \mathbf{r}_3, \mathbf{r}_0$	% Compare to 128.2.209
	BEQ	$\mathbf{r}_0, L_{\mathrm{IPdest}}$	
	SUB	$\mathbf{r}_1, \mathbf{r}_4, \mathbf{r}_0$	% Compare to 128.2.206
	BNE	$\mathbf{r}_0, L_{\mathrm{fail}}$	
L_{IPdest} :	EXTW	$\mathbf{r}_6, 6, \mathbf{r}_0$	% Extract 2 bytes at offset 30-31
	SUB	$\mathbf{r}_0, 640, \mathbf{r}_0$	% Compare to 128.2
	BNE	$\mathbf{r}_0, L_{\mathrm{fail}}$	
	AND	$\mathbf{r}_7, 255, \mathbf{r}_1$	% Byte at offset 32
	SUB	$\mathbf{r}_1, 206, \mathbf{r}_0$	% Compare to 128.2.206
	BEQ	$\mathbf{r}_0, L_{\mathrm{acc}}$	
	SUB	$\mathbf{r}_1, 209, \mathbf{r}_0$	% Compare to 128.2.209
	BNE	$\mathbf{r}_0, L_{\mathrm{fail}}$	
	JMP	L_{acc}	
L_{ARP} :	SHL	$\mathbf{r}_6, 8, \mathbf{r}_0$	
	LD	$\mathbf{r}_6, 32[\mathbf{r}_{16}]$	% Load bytes at offset 32-39
	SHR	$\mathbf{r}_0, 40, \mathbf{r}_1$	% 3 bytes at offset 28-30
	SUB	$\mathbf{r}_1, \mathbf{r}_3, \mathbf{r}_0$	% Compare to 128.2.209
	LD	$\mathbf{r}_7, 40[\mathbf{r}_{16}]$	% Load bytes at offset 40-47
	BEQ	$\mathbf{r}_0, L_{\mathrm{ARPdest}}$	
	SUB	$\mathbf{r}_1, \mathbf{r}_4, \mathbf{r}_0$	% Compare to 128.2.206
	BNE	$\mathbf{r}_0, L_{\mathrm{fail}}$	
L_{ARPdest} :	EXTW	$\mathbf{r}_6, 6, \mathbf{r}_0$	% Extract 2 bytes at offset 38-39
	SUB	$\mathbf{r}_0, 640, \mathbf{r}_1$	% Compare to 128.2
	BNE	$\mathbf{r}_1, L_{\mathrm{fail}}$	
	AND	$\mathbf{r}_7, 255, \mathbf{r}_1$	% Byte at offset 40
	SUB	$\mathbf{r}_1, 206, \mathbf{r}_0$	% Compare to 128.2.206
	BEQ	$\mathbf{r}_0, L_{\mathrm{acc}}$	
	SUB	$\mathbf{r}_1, 209, \mathbf{r}_0$	% Compare to 128.2.209
	BNE	$\mathbf{r}_0, L_{\mathrm{fail}}$	
L_{acc} :	MOV	$\mathbf{r}_0, 1$	% Accept
	RET		
L_{fail} :	MOV	$\mathbf{r}_0, 0$	% Fail
	RET		

Figure 24: DEC Alpha assembly language for Filter 3. On input the start address of the packet is in $\mathbf{r}_{16}$ and its length in $\mathbf{r}_{17}$. This filter accepts IP or ARP packets with originating network 128.2.206 or 128.2.209 and destination network 128.2.206 or 128.2.209.

```
        LD    r0, 8[r16]      % Load bytes at offset 8-15
        EXTW  r0, 4, r1       % Extract 2 bytes at offset 12
        SUB   r1, 8, r2       % Compare to ETHER_IP
        BNE   r2, L_fail
        LD    r1, 16[r16]     % Load bytes at offset 16-23
        MOV   r5, 23          % Prepare Telnet port
        EXTB  r1, 7, r2       % Extract byte at offset 23
        SUB   r2, 6, r3       % Compare to IP_TCP
        SHL   r5, 8, r5       % Prepare Telnet port in big-endian
        BNE   r3, L_fail
        EXTW  r1, 4, r2       % Extract 2 bytes at offset 20
        BNE   r2, L_fail      % Reject if not first in sequence
        SHR   r0, 46, r2
        AND   r2, 60, r3
        ADD   r3, 16, r2      % Offset of the TCP port
        AND   r2, 7, r4       % Check alignment
        SUB   r2, r4, r3      % Align
        SUB   r3, r17, r0     % Bounds check
        BGE   r0, L_fail
        ADD   r16, r3, r1     % Address of TCP port
        LD    r0, 0[r1]       % Load TCP port
        EXTW  r0, r4, r3      % Extract TCP port
        SUB   r3, r5, r0      % Compare with Telnet port
        BNE   r0, L_fail
        MOV   r0, 1           % Accept
        RET
L_fail: MOV   r0, 0           % Fail
        RET
```

Figure 25: DEC Alpha assembly language for Filter 4. On input the start address of the filter is in $\mathbf{r}_{16}$ and its length $\mathbf{r}_{17}$. This filter accepts only TCP packets destined for the Telnet application (TCP port = 23).

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 and the precondition show before. 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 $\mathbf{r}_1 > 0 \supset \mathbf{r}_1 \geq 0$). 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.

5.3 Performance Comparisons with Previous Approaches

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.

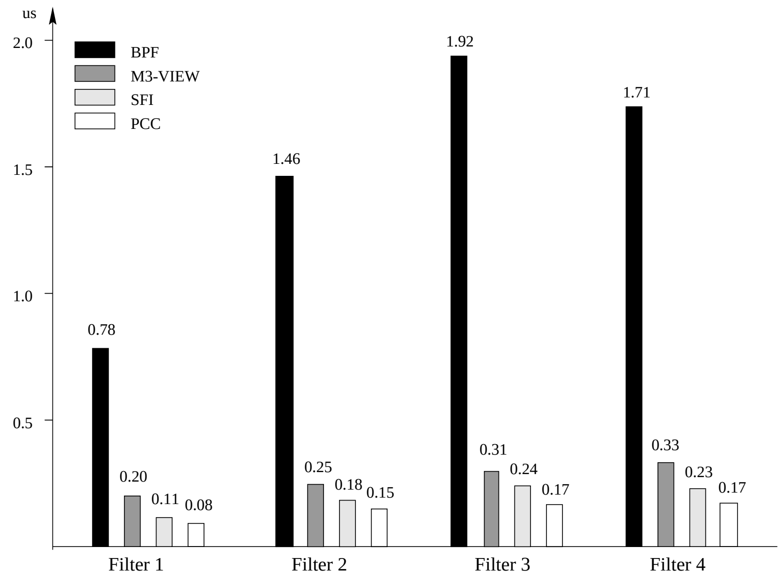

Figure 26: Comparison of average per-packet run time.

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 26. 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, 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.

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 27 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 860 packets when compared to the BPF version of the filter, after 8300 packets when compared to the Modula-3 version and after 22,000 packets when compared to the SFI packet filter. Note that at

Packet Filter		1	2	3	4
Instructions		8	15	47	28
Binary Size (bytes)		315	404	835	757
Validation Cost	Time (μs)	358	546	1327	850
	Space (KB)	3.3	5.3	15.4	8.6

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

the time we collected the packet trace used for the experiments we counted about 1000 Ethernet packets per second on the average.

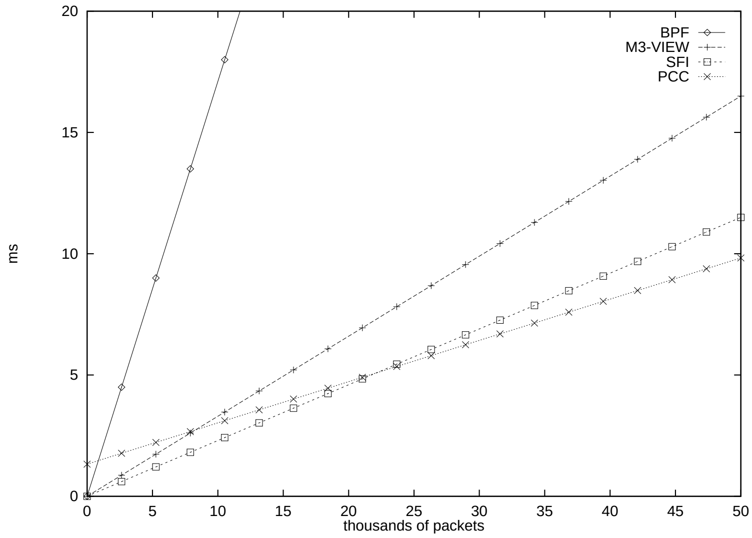

Figure 27: Startup cost amortization for Filter 4.

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], 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, 25]. 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) [24]. 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% [24]. 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 [24]. 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 [25]. 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 [20] 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 [22] 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.

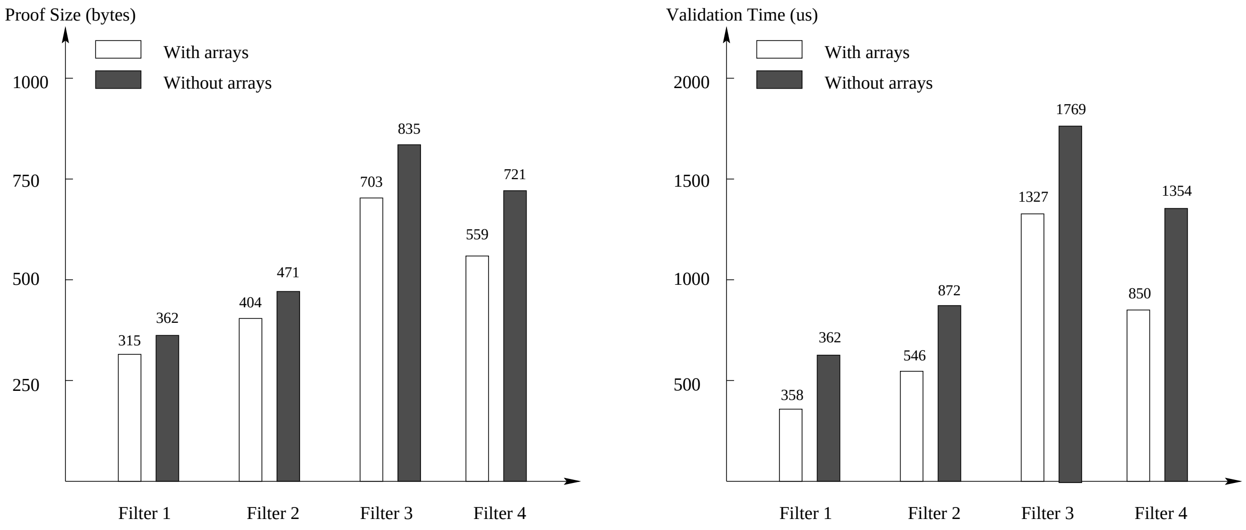

Figure 28: Comparison of proof sizes and validation times between the safety policy with and without arrays.

5.4 Expressing the Safety Policy with Low-Level Constructs

In this section we briefly explore a variation of the PCC technique for safe packet filters. The variation is that we express the same BPF packet filter safety but without using the type **array**. There are various reasons for doing this. First, this is the most obvious thing to do, especially for a safety policy designer who is not familiar with type systems. Secondly, a safety policy designer might feel more comfortable avoiding high-level types such as arrays with the purpose of avoiding semantic gaps between the low-level machine (or abstract machine) and the high-level type system. In reality the series of soundness Theorems 4.3 and 5.2 completely close the semantic gap, but nevertheless some people may want to avoid high-level type systems. We performed such an experiment to measure the benefits of using a safety policy based on high-level types in the case of packet filters.

We can define the precondition for packet filters in low-level terms as follows:

$$
\begin{array}{rl}
Pre = & \mathbf{r}_2 \geq 64 \wedge \\
& \forall i.(i \geq 0 \wedge i < \mathbf{r}_2 \wedge i \bmod 8 = 0) \\
& \qquad \supset \mathbf{r}_1 \oplus i : \mathbf{ro\_addr} \qquad \wedge \\
& \forall j.(j \geq 0 \wedge j < 16 \wedge (j \bmod 8) = 0) \\
& \qquad \supset \mathbf{r}_3 \oplus j : \mathbf{addr} \qquad \wedge \\
& \forall i.\forall j.(i \geq 0 \wedge i < \mathbf{r}_2 \wedge j \geq 0 \wedge j < 16) \\
& \qquad \supset (\mathbf{r}_1 \oplus i \neq \mathbf{r}_3 \oplus j)
\end{array}
$$

By careful comparison of the above precondition and the one using arrays (from page 36) we note that they are equivalent in all respects but size. Up to this point the major drawback of a bulkier safety precondition seems to be the danger of making a mistake and the increased difficulty in understanding it. Our experience is that in addition to these, the size of the safety proofs and the time to certify them is increased considerably. The main culprit is the presence of the universally quantified formulas in the precondition. It is much easier to reason about the typing predicates involving arrays than about the universally quantified formulas that appear in the precondition. The only drawback of using arrays is the need to extend the safety policy with proof rules involving arrays and to prove the soundness of such rules. But once this is done, all future proofs of client will be easier to generate, smaller and faster to validate. The results of our experiments are shown in Figure 28.

One conclusion that we draw from these experiments is that there is much hope that with careful choice of the safety policy and application-specific proof rules the theorem proving effort as well as the PCC binary size and the validation time can be further improved.

6 Case Study: Safe Extensions of the TIL Run-Time System

The practice of software development in languages such as ML and Haskell often involves extending the run-time system, usually by writing C code, to implement new primitive types and operations or functionality that is not easily programmed in the high-level language. This raises the question of how to ensure that the foreign code respects the basic assumptions of the run-time system. Even without considering user-extensions, the run-time systems of high-level languages usually include a sizeable part written in unsafe languages such as C or even assembly language. The mechanism that allows an untrusted user to safely extend the run-time can also be used by a small kernel of the run-time system to bootstrap the rest, increasing the level of confidence in the system.

We propose the use of proof-carrying code to allow arbitrary untrusted users to link foreign functions to a safe programming language run-time system. For this to be safe the compiler designer defines the safety policy, which is basically a formal description of the data-representation invariants to be preserved and calling conventions to be obeyed by foreign functions. Then, the user produces and attaches to the foreign code a safety proof attesting to the preservation of the invariants.

To make the presentation more concrete we show in detail how we use PCC to develop safe DEC Alpha assembly-language [21] extensions to the run-time system of the TIL [23] compiler for Standard ML [15]. We consider here only a very small example with several simplifying assumptions. For example, we only consider leaf procedures that do not allocate memory. Scaling the technique to the entire Standard ML language is subject of current research.

Consider the Standard ML program fragment shown in Figure 29. This program defines a union type `T` and a function `sum` that adds all the integers in a `T list`. The plan for the rest of this section is to define a safety policy for extensions to the TIL run-time system and then prove the type safety of a hand-optimized assembly language version of the `sum` function.

```
datatype T = Int of int | Pair of int * int

fun sum (l : T list) =
 let
    fun foldr f [] acc = acc
      | foldr f (h :: t) acc = foldr f t (f(acc, h))
 in
    foldr (fn (acc, Int i) => acc + i
            | (acc, Pair (i, j)) => acc + i + j)
          l 0
 end
```

Figure 29: The Standard ML source program.

6.1 The Safety Policy

The first order of business is to define the safety policy for the TIL run-time system in the presence of foreign functions. This is the job of the compiler designer, or a trusted person who is intimately familiar with the data-representation conventions and basic invariants maintained by the TIL compiler and run-time system.

The safety policy in our case requires that foreign code maintains the data-representation invariants chosen by the TIL compiler. Data representation in TIL is type directed and the types involved in our example are the following:

$$\tau ::= \texttt{int} \quad | \quad \tau_1 * \tau_2 \quad | \quad \tau_1 + \tau_2 \quad | \quad \tau\ \texttt{list}$$

For convenience we use `T` as an abbreviation for the type $\texttt{int} + (\texttt{int} * \texttt{int})$. For this subset of ML types the TIL data-representation rules are as follows: an integer value is represented as an untagged 32-bit machine word; a pair is represented as a pointer to a sequence of two memory locations containing values of appropriate types; a value of type $\tau_1 + \tau_2$ is represented as a pointer to a pair of locations containing respectively the constructor value (0 for $\texttt{inj}_l$ and 1 for $\texttt{inj}_r$) and the value carried by the constructor; the empty list is represented as the value 0 and the non-empty list as a pointer to a list cell. See Figure 30 for examples of TIL representations of several SML values.

```
val r0 : int = 5
val r1 : int * int = (2, 3)
val r2 : T = Pair r1
val r3 : T = Int 6
val r4 : T list = [r3, r2]
```

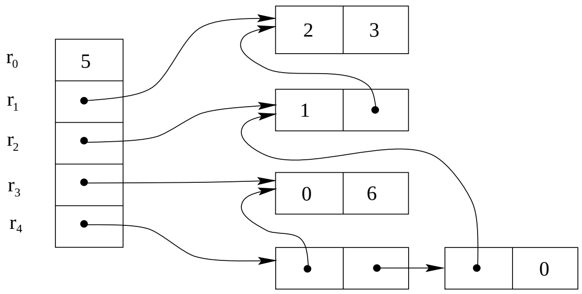

Figure 30: Data Representation in TIL. Each box represents a machine word.

The compiler designer describes formally the data-representation strategy by means of the typing predicate $m \vdash e : \tau$ introduced in Section 5. The meaning of this typing judgement in the case of the newly introduced types is:

Definition 6.1 *We say that* $\rho \models m \vdash e : \tau_1 * \tau 2$ *iff*

- $\rho \models e \bmod 8 = 0$, *and*
- $\rho \models e \oplus 4 = e + 4$, *and*
- $\rho \models e : \textbf{ro\_addr}$, *and*
- $\rho \models e \oplus 4 : \textbf{ro\_addr}$, *and*
- $\rho \models m \vdash \textbf{sel}(m, e) : \tau_1$, *and*
- $\rho \models m \vdash \textbf{sel}(m, e \oplus 4) : \tau_2$.

Definition 6.2 *We say that* $\rho \models m \vdash e : \tau_1 + \tau 2$ *iff*

- $\rho \models e \bmod 8 = 0$, *and*
- $\rho \models e \oplus 4 = e + 4$, *and*
- $\rho \models e : \textbf{ro\_addr}$, *and*
- $\rho \models e \oplus 4 : \textbf{ro\_addr}$, *and*

- $\rho \models m \vdash \mathbf{sel}(m, e) : \mathbf{int}$, *and*
- *if* $\rho \models \mathbf{sel}(m, e) = 0$ *then* $\rho \models m \vdash \mathbf{sel}(m, e \oplus 4) : \tau_1$, *and*
- *if* $\rho \models \mathbf{sel}(m, e) \neq 0$ *then* $\rho \models m \vdash \mathbf{sel}(m, e \oplus 4) : \tau_2$.

Definition 6.3 *We say that* $\rho \models m \vdash e : \tau\ \texttt{list}$ *iff whenever* $\rho(e) \neq 0$ *we have:*

- $\rho \models e \bmod 8 = 0$, *and*
- $\rho \models e \oplus 4 = e + 4$, *and*
- $\rho \models e : \mathbf{ro\_addr}$, *and*
- $\rho \models e \oplus 4 : \mathbf{ro\_addr}$, *and*
- $\rho \models m \vdash \mathbf{sel}(m, e) : \tau$, *and*
- $\rho \models m \vdash \mathbf{sel}(m, e \oplus 4) : \tau\ \texttt{list}$.

A major difference between previous examples and this case is that the precondition and post-condition are different for each foreign function. They are computed according to the declared type of the foreign function as shown in the next subsection. Another difference is that for the purpose of this section we use 32-bit memory operations and modify the alignment requirement accordingly.

6.2 The Foreign Function

In our experiment, the code producer writes a DEC Alpha assembly language implementation of the `sum` function, as shown in Figure 31. This code assumes that register $\mathbf{r}_0$ contains the argument of type `T list` on entry and the integer result on exit. The registers $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathbf{r}_3$ are used as temporaries. This code is written to respect the TIL data-representation strategy, and it is this fact that must be proved for the consumer.

Note that the above assembly-language program is optimized by hand. One of our goals is to show that proof-carrying code does not pose restrictions on using register allocation, scheduling or other low level optimization techniques.

The precondition and the postcondition for this experiment vary depending on the type of the foreign function. In the case of the function `sum` of type `T list` $\rightarrow$ `T list` the precondition and the postcondition are as follows:

$$\begin{array}{rcl} \mathit{Pre} & \equiv & \mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{T list} \\ \mathit{Post} & \equiv & \mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{int} \end{array}$$

As we promised before, the foreign code explored here exhibits a required use of the invariant instruction. The invariant in line 2 is the loop invariant and corresponds to the backward branch from line 13. By examining the code we note that at the loop entry point (line 2) there are two live registers: $\mathbf{r}_0$ of type `T list`, corresponding to the rest of the list to be scanned and, $\mathbf{r}_1$ of type **int** corresponding the the accumulated sum so far. The invariant derived from this information is

$$\mathit{Inv}_2 = \mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{T list} \wedge \mathbf{r_m} \vdash \mathbf{r}_1 : \texttt{int}$$

In general, the loop invariant for type-safety policies is the conjunction of all typing predicates for the registers that are live at the invariant point.

```
                          %r0 is l
sum : INV  rm ⊢ r0 : T list
                          %r1 is acc
      MOV  r1, 0          %Initialize acc
L2    INV  rm ⊢ r0 : T list ∧ rm ⊢ r1 : int
                          %Loop invariant
      BEQ  r0, L14        %Is list empty?
      LD   r2, 0(r0)      %Load head
      LD   r0, 4(r0)      %Load tail
      LD   r3, 0(r2)      %Load constructor
      LD   r2, 4(r2)      %Load data
      BEQ  r3, L12        %Is an integer?
      LD   r3, 0(r2)      %Load i
      LD   r2, 4(r2)      %Load j
      ADD  r2, r3, r2     %Add i and j
L12   ADD  r1, r2, r1     %Do the addition
      BR   L2             %Loop
L14   MOV  r0, r1         %Copy result in r0
      RET                 %Result is in r0
```

Figure 31: DEC Alpha assembly language implementation of the `sum` function.

For our example program, the VC predicate has two conjuncts, one for the precondition and another for the invariant associated with L_2. The first conjunct corresponds to the control path from the function entry point to the start of the loop. This conjunct says that the loop invariant is established when the loop conditional is first executed:

$$\mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{Foo list} \supset (\mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{Foo list} \wedge \mathbf{r_m} \vdash 0 : \texttt{int})$$

The second conjunct corresponds to the rest of the program and says both that the the loop invariant is preserved around the loop and that it entails the postcondition when the loop finishes. This part of the VC predicate is more complicated and we do not show it here.

In order to prove the safety predicate the code producer needs proof rules specific to this application. These proof rules are shown in Figure 32. By careful examination of the rules we note that there is a close relationship between these rules and the definition of the $\models$ relations for the newly introduced types. Thus it is not difficult to verify that the soundness theorem for the proof system still holds:

Theorem 6.4 *If* $\triangleright\, \rho(P)$ *then* $\rho \models P$.

The type **int** has a different purpose in this example. It is not used to express representation invariants[2] but to enforce abstraction boundaries. The intention is for **int** to be an abstract type whose values can only be constructed using certain operations. In the case at hand we only consider two valid constructors: the constant `0` and the addition of two integers. This is enforced in the safety policy by ensuring that the only proof rules that introduce integer typing predicates correspond to the valid constructors.

[2]In TIL any machine word can be legitimately viewed as the representation of an integer.

$$\frac{\rhd\, m \vdash e : \tau_1 * \tau_2}{\rhd\, e \oplus 4 = e + 4 \ \wedge\ e : \mathtt{addr} \ \wedge\ e \oplus 4 : \mathtt{addr} \ \wedge\ m \vdash \mathbf{sel}(m, e) : \tau_1 \quad \wedge \quad m \vdash \mathbf{sel}(m, e \oplus 4) : \tau_2}\,\mathbf{tp\_*}$$

$$\frac{\rhd\, m \vdash e : \tau_1 + \tau_2}{\rhd\, e \oplus 4 = e + 4 \ \wedge\ e : \mathtt{addr} \ \wedge\ e \oplus 4 : \mathtt{addr}}\,\mathbf{tp\_{+}1}$$

$$\frac{\rhd\, m \vdash e : \tau_1 + \tau_2}{\rhd\, \mathbf{sel}(m, e) = 0 \supset m \vdash \mathbf{sel}(m, e \oplus 4) : \tau_1 \ \wedge\ \mathbf{sel}(m, e) \neq 0 \supset m \vdash \mathbf{sel}(m, e \oplus 4) : \tau_2}\,\mathbf{tp\_{+}2}$$

$$\frac{\rhd\, m \vdash e : \tau\ \mathtt{list} \qquad \rhd\, e \neq 0}{\rhd\, m \vdash e : \tau * \tau\ \mathtt{list}}\,\mathbf{tp\_list}$$

$$\frac{\rhd\, e_1 : \mathtt{int} \qquad \rhd\, e_2 : \mathtt{int}}{\rhd\, e_1 + e_2 : \mathtt{int}}\,\mathbf{tp\_int{+}} \qquad \frac{}{\rhd\, 0 : \mathtt{int}}\,\mathbf{tp\_0}$$

Figure 32: The typing rules.

This example of abstract types is admittedly contrived but it still illustrates the basic methodology for enforcing abstraction boundaries. More involved examples could be constructed if the framework supported function calls. Then the framework would check that values declared to be of an abstract type are only manipulated with functions intended for that purpose. More concretely, static checking has the ability to distinguish between two machine words with the same value but of different types. For example, the machine word 0 can be the representation of the integer 0 and the empty list. In the former case it can be validly incremented but in the latter not.

Extending the framework to deal with the entire Standard ML is subject of current research. As we mentioned, we want to deal with function calls. We also want to integrate well with the garbage collection, exceptions and higher-order functions.

7 Case Study: Internet Checksum

The experiment we present in this section has been inspired by the network packet filter experiment and is intended to test the viability of the PCC technique for dealing with loops. Another inspiration source was the FoxNet, a TCP/IP protocol stack developed by the Fox Project at Carnegie Mellon University. It is well-known to the networking community that the most expensive operations in a TCP/IP protocol stack are those that touch the data, namely the checksumming and copying. This is also the case in the FoxNet protocol stack. What differentiates FoxNet from other implementation of TCP/IP is the use of advanced programming language constructs for achieving performance while maintaining a high degree of flexibility, safety and robustness. In particular, the FoxNet is written almost entirely in Standard ML.

In the context of the FoxNet it would not be acceptable to implement performance-critical operations in machine language because it would destroy all the safety guarantees that the programming language offers. It seemed natural to think that if the PCC technique were able to guarantee the safety of assembly language extensions to the runtime of an ML system then we could achieve the best of two worlds: strong safety guarantees and high-performance. Incidentally, the current state of the PCC is able to certify the safety of a IP checksum routine because there is no interaction with garbage collection and such a routine can be implemented easily as a leaf-procedure.

7.1 The Safety Policy

The safety policy in this case specifies that the register $\mathbf{r}_{16}$ (the standard argument register) holds on input the start of the packet whose length is in register $\mathbf{r}_{17}$. There is no postcondition. This is expressed as follows:

$$\begin{array}{rcl} \mathit{Pre} & = & \mathbf{r_m} \vdash \mathbf{r}_{16} : \mathbf{array}(\mathbf{ro\_addr}, \mathbf{unit}, \mathbf{r}_{16}, \mathbf{r}_{17}) \\ \mathit{Post} & = & \mathbf{true} \end{array}$$

As for the application specific proof rules we use those for packet filters.

7.2 The Foreign Code

We wrote the IP checksum routine in DEC Alpha assembly language assuming that the register $\mathbf{r}_{16}$ holds on input the address of the data to be checksummed, and register $\mathbf{r}_{17}$ holds the length of this data. We also assume that the packet is aligned on a 8-byte boundary. We make no assumptions about its length being a multiple of 8. The code is shown in Figure 33. The program is composed of three main sections. The first section is the main loop stating at L_{loop}, which iterates through the data, loading 8 bytes at a time and accumulating a 32-bit checksum in the register $\mathbf{r}_0$. The register $\mathbf{r}_3$ holds the current pointer in the data array and register $\mathbf{r}_4$ contains 8 less than the number of remaining bytes to be summed. Thus the loop invariant is expressed as:

$$\mathit{Inv}_{\mathrm{loop}} = \mathbf{r_m} \vdash \mathbf{r}_3 : \mathbf{array}(\mathbf{ro\_addr}, \mathbf{unit}, \mathbf{r}_{16}, \mathbf{r}_4 + 8)$$

Note that we preserve the information that $\mathbf{r}_3$ points somewhere in the array starting at address $\mathbf{r}_{16}$. We make no use of this fact here because we need not reason about pointer aliasing. Also note that the loop is only started if the register $\mathbf{r}_4$ has a value that is smaller than 2^{63} (i.e., is the two's complement representation of a positive integer).

The second section of the checksumming program starts at L_{end} and its purpose is to accumulate in the checksum the leftover bytes after the loop terminates. The number of such bytes is $\mathbf{r}_4 \oplus 8$, which must be from 0 to 7. Note that it is important here to use the modulo 2^{64} addition as the value of $\mathbf{r}_4$ represents a negative integer. We have introduced an invariant instruction at the beginning of this section, although it was not strictly necessary. As we have mentioned before the invariant instructions break a large safety predicate in smaller conjuncts, making the theorem proving task simpler.

The third and last section of the checksum program computes the final 16-bit checksum from the accumulator $\mathbf{r}_0$. There are no memory operations in this section, thus the safety predicate corresponding to it is **true**.

7.3 Performance Measurements

The entire PCC binary for the IP checksum example is 859 bytes. On our DEC Alpha running at 175MHz it takes 1.3ms to validate the safety proof. As far as runtime performance is concerned we have compared it with the standard C version that is part of the OSF/1 kernel. We noticed that our code runs two times faster than the OSF/1 version over a wide range of packet sizes (140 MBytes/s compared to 65 MBytes/s for packets of 1500 bytes and 100 MBytes/s compared to 47 MBytes/s for packets of 100 bytes). The C version has the main loop unrolled 16 times but it does all memory operations on 32 bits. Upon closer inspection we noticed that for each 8 bytes processed the C version uses 16 instructions, of which 2 are loads. The assembly language version uses 8 instruction (of which one is a load) for the same number of bytes. We believe that this is the main contributing factor in the performance discrepancy.

The PCC version has an initial latency of 1.3ms due the proof validation cost. However, this cost is only incurred once and it is completely amortized after processing approximately 150 KBytes of data.

We did not perform this experiment to surpass the performance of the OSF/1 implementation of the IP checksum. Our purpose was to show again that even hand-optimized code is considered the PCC technique can still be used to certify the safety of programs. Performance does not have to be sacrificed to achieve safety with PCC.

7.4 Expressing the Safety Policy with Low Level Constructs

We have used arrays to express the safety policy and the invariants for the IP checksum program presented before. We consider here a small variation of the experiment in which we do not use arrays to express the safety policy. The safety precondition becomes:

$$
\begin{array}{rcl}
\mathit{Pre} & = & \forall i. i \geq \mathbf{r}_{16} \wedge i < \mathbf{r}_{16} + \mathbf{r}_{17} \wedge i \bmod 8 = 0 \supset i : \mathbf{ro\_addr} \\
 & & \quad \wedge\, \mathbf{r}_{16} \oplus \mathbf{r}_{17} = \mathbf{r}_{16} + \mathbf{r}_{17} \wedge \mathbf{r}_{16} \bmod 8 = 0 \\
\mathit{Post} & = & \mathbf{true} \\
\mathit{Inv}_{\mathrm{loop}} & = & \forall i. i \geq 0 \wedge i < \mathbf{r}_4 \oplus 8 \supset \mathbf{r}_3 \oplus i : \mathbf{ro\_addr} \\
 & & \quad \wedge\, \mathbf{r}_4 \oplus 8 < 2^{63} \wedge \mathbf{r}_3 \bmod 8 = 0
\end{array}
$$

Not only the invariants become more clumsy but also the theorem proving process becomes more difficult to supervise. And ultimately the proof size that we obtained was 1620 bytes (88% larger than before) and the proof validation time was 3.6ms (176% larger than before). These results are in line with the similar experiments done with packet filters. Our conclusion is that it is beneficial for the safety policy to be expressed in terms of higher-level constructs that subsume universal quantification.

	MOV	$\mathbf{r}_3, \mathbf{r}_{16}$	% Copy the packet start address in $\mathbf{r}_3$
	SUB	$\mathbf{r}_{17}, 8, \mathbf{r}_4$	% Packet length - 8 in $\mathbf{r}_4$
	MOV	$\mathbf{r}_6, 1$	
	SHL	$\mathbf{r}_6, 32, \mathbf{r}_6$	
	MOV	$\mathbf{r}_0, 0$	% Init the checksum in $\mathbf{r}_0$
	SUB	$\mathbf{r}_6, 1, \mathbf{r}_6$	% Put in $\mathbf{r}_6$ the mask 0xFFFFFFFF
	BLT	$\mathbf{r}_4, L_{\mathrm{end}}$	% Jump if less than 8 bytes left
L_{loop} :	INV	$\mathbf{r_m} \vdash \mathbf{r}_3 : \mathbf{array}(\mathbf{ro\_addr}, \mathbf{unit}, \mathbf{r}_{16}, \mathbf{r}_4 + 8)$	
	LD	$\mathbf{r}_5, 0[\mathbf{r}_3]$	% Load next 8 bytes
	ADD	$\mathbf{r}_3, 8, \mathbf{r}_3$	% Advance pointer
	AND	$\mathbf{r}_5, \mathbf{r}_6, \mathbf{r}_7$	% Get LO 4 bytes
	SHR	$\mathbf{r}_5, 32, \mathbf{r}_8$	% Get HI 4 bytes
	ADD	$\mathbf{r}_0, \mathbf{r}_7, \mathbf{r}_7$	% Accumulate LO
	SUB	$\mathbf{r}_4, 8, \mathbf{r}_4$	% Decrease remaining length
	ADD	$\mathbf{r}_7, \mathbf{r}_8, \mathbf{r}_0$	% Accumulate HI
	BGE	$\mathbf{r}_4, L_{\mathrm{loop}}$	% Loop if not done
L_{end}	INV	$\mathbf{r}_4 \oplus 8 < 2^{63} \supset \mathbf{r}_3 : \mathbf{ro\_addr}$	
	ADD	$\mathbf{r}_4, 8, \mathbf{r}_4$	% This is the number of bytes left
	BLE	$\mathbf{r}_4, L_{\mathrm{done}}$	% Nothing left
	LD	$\mathbf{r}_5, 0[\mathbf{r}_3]$	% Load the leftover
	SHL	$\mathbf{r}_4, 3, \mathbf{r}_4$	% Number of bits left
	MOV	$\mathbf{r}_7, 1$	
	SHL	$\mathbf{r}_7, \mathbf{r}_4, \mathbf{r}_7$	
	SUB	$\mathbf{r}_7, 1, \mathbf{r}_7$	% The mask for extracting the leftover
	AND	$\mathbf{r}_5, \mathbf{r}_7, \mathbf{r}_5$	% Extact the leftover
	SHR	$\mathbf{r}_5, 32, \mathbf{r}_7$	% This is HI
	AND	$\mathbf{r}_5, \mathbf{r}_6, \mathbf{r}_5$	% This is LO
	ADD	$\mathbf{r}_0, \mathbf{r}_7, \mathbf{r}_7$	% Accumulate HI
	ADD	$\mathbf{r}_7, \mathbf{r}_8, \mathbf{r}_0$	% Accumulate LO
L_{done}	AND	$\mathbf{r}_0, \mathbf{r}_6, \mathbf{r}_7$	% checksum LO
	SHR	$\mathbf{r}_0, 32, \mathbf{r}_0$	% checksum HI
	ADD	$\mathbf{r}_0, \mathbf{r}_7, \mathbf{r}_0$	% Now at most 33 bits
	SHR	$\mathbf{r}_6, 16, \mathbf{r}_6$	% Mask is 0xFFFF
	SHR	$\mathbf{r}_0, 16, \mathbf{r}_7$	% HI 17 bits
	AND	$\mathbf{r}_0, \mathbf{r}_6, \mathbf{r}_0$	% LO 16 bits
	ADD	$\mathbf{r}_0, \mathbf{r}_7, \mathbf{r}_0$	% At most 17 bits
	SHR	$\mathbf{r}_0, 16, \mathbf{r}_7$	
	AND	$\mathbf{r}_0, \mathbf{r}_6, \mathbf{r}_0$	
	AND	$\mathbf{r}_0, \mathbf{r}_7, \mathbf{r}_0$	
	RET		

Figure 33: The DEC Alpha code for the Internet Checksum. On input the packet start address is in $\mathbf{r}_{16}$ and its length in $\mathbf{r}_{17}$.

8 Practical Difficulties

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 this section we discuss some of the problematic issues that were brought to light by our experiment. Fortunately we think that we have found good practical solutions to the part of the PCC pertaining to the code consumer: safety proof representation and validation. Moreover, in this we are faced with such a vast and unexplored range of engineering choices that we think major performance improvements can still be achieved on the code consumer side. We consider next the three main issues that concern us: establishing the safety policy, generating the safety proof and the size of the safety proofs.

8.1 Establishing the Safety Policy

Establishing a sound safety policy is a critical step in the use of PCC. This is known to be a problem even without considering automatic verification of compliance with the safety policy. The first difficult problem that arises in the process of establishing a formal safety policy is to find the most appropriate logic for expressing the required properties. For example, the first-order predicate logic as described in this report is not expressive enough to encode temporal properties easily. Also, concurrent programs typically require a different formalism, and the same might be true for programs that use higher-order functions. We recognize this problem as important and the PCC technique does not alleviate it in any way. In our own experiments we have relied on expertise developed in the program specification and verification field for answers to such questions.

As we have seen in the experiments with packet filters and the IP checksum it is often the case that a given formalism can be extended to suit the needs of the safety policy. We have observed in these practical cases that extending the notion of types with arrays made the safety policy simpler and the theorem proving task easier.

Once the issue of the formalism is settled, the safety policy designer has to specify the safety precondition and the postcondition, as well as the proof rules. The major danger at this step is that the safety policy contains "loopholes" that the foreign code can exploit to circumvent the protection boundaries. In the particular case of PCC such "loopholes" can be either a precondition that is too strong (i.e., allows the code to assume fact that are false), a postcondition that is too weak (i.e., fail to request the code to establish some properties upon termination) or an unsound proof system.

The problem of specifying a safety policy is a difficult one independently of the technology used to enforce it. In the case of PCC, the safety policy designed is forced to use formal specification techniques that could increase the effort required for specification. However, it is generally believed that using formal specification forces a better and safer design. In addition, a good choice of the underlying logic and proof system for the safety policy can make the interface simpler and less error prone (see for example the packet fitters with and without arrays).

As for the soundness of the proof system, there are well-known techniques for verifying it. All the application-specific proof systems presented in this report are accompanied by soundness proofs, that relate the derivability relation $\triangleright$ with the intuitive definition of truth (the $\models$ relations). We think that it is important that in all application of PCC the soundness of the proof system be proven. A flaw in the proof system will allow a malicious code producer to produce a "certified" program that does anything the physical machine can do.

8.2 Generating the Safety Proofs

One of the most difficult problems that must be solved for any practical implementation of PCC is the methodology for producing the safety proofs.

In the experiments reported here, 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 [24]), which would have the effect of simplifying the proving process at the cost of reducing the performance of the foreign code. 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.

The process of generating a proof is closely related with the process of "guessing" the loop invariants. 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 often 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.

We currently produce the proofs using a very simple theorem prover that outputs a witness for every successful proof. For our experiments we use the programming language Elf [18] to prove VC predicates and produce LF representation of their proofs. Elf is a logic programming language based on LF. A program in Elf is an LF signature and execution in Elf is search for canonical LF objects inhabiting an LF type in the context of a signature. In our case the program is the signature Σ and we are interested in finding a closed object M of type $\texttt{pf}\ \ulcorner SP \urcorner$ for some safety predicate SP. If such an object is found, according to Theorem 4.1, it constitutes the canonical LF representation of a proof $\triangleright\ SP$. Incidentally, this is exactly the required safety proof.

Proof search in Elf is performed in depth-first fashion, as in Prolog. With this operational view, the natural deduction style presentation of our logics is not appropriate for proof search, because any of the elimination rules would lead to non-termination. Our solution to this problem is based on the observation that all of the VC predicates in our current experiments are either first-order Horn clauses, or first order hereditary Harrop formulas. These fragments of first-order logic admit a complete sequent-style proof system where the declarative meaning of logical connectors coincides with their search-related reading [14]. The resulting proofs are called uniform. The LF representation of an uniform proof system for our logic can then be used as a logic program to perform proof search.

We represent in LF the uniform derivation rules for our logic in a manner similar to the natural deduction representation. We use this representation in Elf to perform a goal-directed search for a uniform derivation of the validity of the VC predicate. We also represent in LF the proof of soundness of uniform derivations with respect to the natural deduction formulation of our logic. We exploit the operational reading of this soundness proof in Elf to convert the uniform derivation of the VC predicate to a natural deduction proof of it.

Using this method, 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. The same is true for other experiments with the difference that currently we have to insert all the loop invariants by hand.

Of all the experiments presented in this paper the one involving safe runtime extensions to the TIL runtime system stands out as the one that is more promising. Remember that this example involved loop invariants, but there was a simple algorithm that produced them. Similarly the precondition and the postcondition are generated by the same type-directed algorithm.

8.3 Safety Proof Size

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.

9 Conclusion

We have presented *proof-carrying code,* a mechanism that allows a code consumer to interact safely with native code supplied by untrusted code producers. PCC does not incur the run-time overhead of previous solutions to this problem. Instead, the code producer is required to generate a proof that attests to the code's safety properties. 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 proof in an PCC binary is either detected or harmless.

The main contribution of the work presented in this paper is the principle of staging program verification into certification and proof validation, with the proof acting as a witness that the certification was performed correctly. This staging has great engineering advantages, all based on the intuition that proof checking is, in general, much easier than proof generation.

For example, the application-specific proving strategies—goal directed search, interactive theorem proving or just brute-force search guided by heuristics—and their associated complexity and computational costs are moved off-line to the certification stage. In the validation stage, we only need a simple and reliable proof checker which in many cases is inexpensive enough to be used in performance critical paths. Moreover, the same proof checker covers many practical applications, which increases the reliability of the methodology. Lastly, the certification must be done only once independently how many times the code is used.

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.

We have also shown a way to use standard verification techniques to check type safety at the assembly-language level. This is important for certifying extensions to safe programming languages and as a main building block in constructing certifying compilers. Similar techniques have been applied to assembly language before [2, 3] but neither as a basis for creating safety proofs nor for checking type safety.

We show an encoding of safety proofs as first-order logic derivations in LF. Our contribution in this area is to identify a fragment of LF which is both sufficient for many applications of PCC and also admits a simple and fast type-checking algorithm.

Proof-carrying code is an application of ideas from program verification, logic and type theory, in this case to extend to low-level languages safety properties that are normally enjoyed only by high-level languages. We have shown that this technique is useful both for safe interoperability of programming languages and operating system components. 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.

While we hope to have brought convincing arguments in favor of the "Proof-Carrying Code" technique, we recognize that there are difficult practical obstacles to wide-spread application of this technology. For example, we do not know at this moment what is the most practical way to generate safety proofs. While this problem is undecidable in general it is possible in restricted settings to achieve automatic generation of proofs. However, more exploration is needed to validate the PCC technique and to discover new application domains where it can be applied effectively.

10 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 for the OSDI'96 version of this report, 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.

[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] Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic 51* (1991), 125–157.

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

[16] 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.

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

[18] Pfenning, F. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science* (Pacific Grove, California, June 1989), IEEE Computer Society Press, pp. 313–322.

[19] Rouaix, F. A Web navigator with applets in Caml. *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking 28*, 7–11 (May 1996), 1365–1371.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.