A Formalization of the Proof-Carrying Code Architecture in a Linear Logical Framework

Mark Plesko and Frank Pfenning
Department of Computer Science
Carnegie Mellon University

`mp5f@andrew.cmu.edu` and `fp@cs.cmu.edu`

Draft of April 16, 1999

1 Introduction

One of the major challenges in the design of modular and extensible operating systems is to guarantee safety in the presence of untrusted code. A similar problem arises in the domain of mobile code. One solution, adopted, for example, in the Java Virtual Machine [LY97], is to perform extensive safety checks at run time. In the alternative paradigm of *proof-carrying code* (PCC) proposed by Necula and Lee [NL96, Nec97], the code producer attaches a safety proof to mobile code which can be independently verified by the code consumer before execution. This eliminates the need for run-time checks and leads to a small trusted computing base.

A difficulty with PCC is the complexity of proving the correctness of the architecture itself. In particular, we would like to ensure that a program which passes the safety check before execution will indeed run safely. While this can be quite difficult, it has to be done only once for each machine architecture and safety policy. For example, Necula [Nec98] has given a mathematical proof for the correctness of his safety policy. Unfortunately, minor changes or additions to a policy may require substantial changes in its correctness proof.

In this paper we formalize the PCC safety architecture in a logical framework, which constitutes an important first step towards an environment for experimentation and formal verification of properties of safety policies and their implementations in the PCC architecture. Our main tool is LLF [CP96], a logical framework based on linear logic [Gir87]. Linear logic provides natural means of describing programming languages and their semantics, especially those of an imperative nature. LLF permits us to give a high-level description of assembly code, safety policies, and safety proofs within the same language.

In future work we plan to formally verify safety policies based on their encoding in LLF. We also hope to introduce linearity to the PCC architecture itself in order to reduce the size of safety proofs.

We will first further describe the PCC infrastructure in Section 2 followed by a brief sketch of our meta-language, the linear logical framework in Section 3. In order to implement portions of the PCC system, we must choose a language for our simulated agent. We follow [Nec98] and use Safe Assembly Language (SAL), a generic RISC architecture. SAL is described in Section 4. Two execution models, one without and one with run-time safety checks, are described in Section 5. A formal connection between these two models is provided in Section 6, where we specify safety as a property of traces of unsafe execution. An implementation of the *Verification Condition Generator* (VCGen), an integral trusted component of the PCC infrastructure, is presented in Section 7. Some conclusions and ideas for further research are given in Section 8.

2 PCC Architecture

This section serves as an informal introduction to the components of proof-carrying code and the role that certifying compilation plays in PCC. For more detail the reader is encouraged to consult Necula [Nec97, Nec98] and Necula and Lee [NL96, NL98a, NL98b].

One important distinction to make among the different components of the PCC system is which components are trusted by the code consumer. This trusted infrastructure is kept simple in order to ensure correctness; it is also kept efficient since it must be executed by the code consumer. The complicated and difficult portions of PCC are saved for the code producer and proof producer, which may or may not be the same.

The first step must be taken by the code consumer; it must establish a *safety policy.* The safety policy has three parts which are part of the PCC infrastructure. First, a *logic* must be chosen to describe allowable agent actions, code annotations, verification conditions, and proofs (these are explained below). In this case we choose an extended first-order predicate logic, which is not to be confused with the linear logic that we are using in our method of formalization, though in future work we will explore the benefits of using linear logic in these components of the infrastructure. Second, *function specifications* must be formed. These are preconditions and postconditions that we will be able to assume on function invocation and return, respectively. Lastly, we must be able to determine what actions the agent code may take based on the state. This is done by the *verification-condition generator* (VCGen). The *verification condition* is a predicate in the logic that is provable only if the agent code is safe.

Once this preliminary step is completed, we can trace the actual operations of a PCC system. First the agent code must be annotated for use later in the system. These annotations must be sufficiently strong to ensure safety, yet weak enough so we can prove them. For the simple examples that were used as test cases in our implementation, these annotation were supplied by hand; however, in a production system, this task would be automated by a *certifying compiler* [NL98a]. The annotation process is the responsibility of the code producer and need not be trusted.

The code receiver then inspects the annotated code and produces a verification condition. A proof of this verification condition in the logic signifies that the agent code satisfies the safety policy. VCGen produces the condition by scanning through the code in one pass. VCGen requires annotations in the form of function specifications and loop invariants in order to simplify the process. These annotations are not trusted and result in additional components of the verification condition, but they make it possible to prove safety by showing which intermediate properties need to be shown. The VCGen is a trusted component.

The code consumer then sends the verification condition to the proof producer. A complex, proof-generating theorem-prover is used to produce the proof, which is then returned to the code consumer. It is possible that instead of producing proofs in real-time, the proof producer may have proofs on hand that correspond to several standard VCGen implementations, or even attaches them directly to the mobile code. This distinction is irrelevant for our purposes here as the proof producer is not trusted.

The final step prior to code execution is verification of the proof. This is a quick, simple process completed by the code consumer and obviously must be trusted.

3 Linear Logical Framework

A formalization of the essential components of the PCC architecture is a complex task. A logical framework appropriate to carry out this task must simultaneously provide means to specify logics and formal proofs, assembly language, machine computations, safety properties, and the verification condition generator. Ideally, we should also be able to express and prove meta-theoretic properties of these components in the framework.

A first candidate is the LF logical framework which is already used in Necula and Lee's implementation of PCC and the Touchstone certifying compiler [Nec98]. Unfortunately, encoding assembly language and the low-level machine is cumbersome due to the pervasively imperative nature of computation. This brings to mind linear logic [Gir87] which has been described as a "logic of state". Forum [Mil94] is a fragment of linear logic whose proof-theoretic properties make it particularly suitable as a specification language. Chirimar [Chi95] has demonstrated that a machine with a pipelined architecture can be effectively encoded in Forum. However, Forum lacks an internal notion of proof, which is essential for proof-carrying code. We therefore chose the *linear logical framework* (LLF) [CP96] which extends LF with connectives from linear logic. On the one hand, this immediately allows the encoding of logics and proofs as in the present PCC implementation. On the other hand, the linear connectives permit a high-level encoding of safe assembly language. Moreover, LLF is based on a type theory and has internal proof terms which can be used to encode and reason about computations themselves. We use this important feature, for example, to explicitly define safety conditions on computations.

The language of LLF is a linear type theory with types $\Pi x{:}A_1.\ A_2$ (dependent functions), $A \multimap B$ (linear functions), $A \,\&\, B$ (additive products) and $\top$ (additive unit). We also abbreviate $\Pi x{:}A_1.\ A_2$ as $A_1 \to A_2$ if x does not occur in A_2. Under the Curry-Howard isomorphism between propositions and types, Π can be seen as universal quantification, $\to$ as implication, $\multimap$ as linear implication, & as conjunction, and $\top$ as truth. We will freely switch between these different views, depending on the situation.

The type theory is defined by a judgment $\Gamma; \Delta \vdash M : A$ where $\Gamma = u_1 : A_1, \ldots, u_n : A_n$ declares types for unrestricted variables, $\Delta = x_1 : A_1, \ldots, x_m : B_m$ declares linear variables, M is a term, and A the type of M. The proof terms M are drawn from a λ-calculus—details are elided here for the sake of brevity. We adopt the customary view of Γ as consisting of logical assumptions (which may be used arbitrarily often in the derivation of A) and Δ consisting of resources (which must be used exactly once in the derivation of A). It is often helpful to think of A as a goal to achieve with resources in Δ according to the laws in Γ, rather than as a proposition to be proven.

The LLF type theory has the usual properties which make it suitable as the basis for a logical framework. In particular, canonical forms exist, and type-checking is decidable. The existence of canonical forms is tantamount to the completeness of uniform derivations [MNPS91]. This means that we can endow LF with an operational semantics in the style of Prolog which is sound and non-deterministically complete. Under this view, the logical assumptions Γ constitute a program, the resources Δ describe the current state, and the type A represents a goal. The proof term $M : A$ is generated by a successful search. This view of LLF as a logic programming language is important for our application, since it allows us to execute various specifications, including execution of the machine, verification of the safety property, and the actual generation of the verification condition from an assembly language program. We will return to these points below.

4 Safe Assembly Language

SAL [Nec98] is the generic assembly language that we use in our implementation of the PCC system. SAL assumes a machine with a RISC architecture that includes a set of registers including a special register `ra` for storing the return address of a function and another special register `sp` with dedicated instructions for the manipulation of the stack. SAL provides instructions for memory usage and function calls, and its syntax is presented in Figure 1. The operations "EOP" and "condCOP" are meant to be an arbitrary arithmetic operation and a conditional jump instruction, respectively. In our implementations, we have instantiated "EOP" as addition and "COP" as a comparison with zero. The addresses specified in this branching instruction and the `jump` instruction are both relative. The pair `call` and `ret` are simple, with the former not storing a return address and the latter relying on the `ra` register. By restricting `call` to only designated functions and not dynamically calculated addresses, we lose the possibilities of higher-order functions and dynamic method lookup. There are separate instructions for normal memory manipulation as well as stack access. `Annot` provides a way for annotations to be added to a SAL program but are ignored during execution. They communicate information that is useful to the PCC infrastructure, for example, to the Verification Condition Generator (see Section 7).

Registers:	r	$::=$	$\mathtt{r}_i \mid \mathtt{ra}$		$i = 1, \ldots, R$
Instructions:	I	$::=$	$r \leftarrow r'$	Move	
		$\mid$	$r \leftarrow n$	Initialize	
		$\mid$	$r \leftarrow r'$ `EOP` r''	Arithmetic/Logical operations	
		$\mid$	`jump` n	Jump	
		$\mid$	`condCOP`$(r), n$	Conditional branch	
		$\mid$	`ra` $\leftarrow$ `pc` $+$ n	Compute return address	
		$\mid$	`call` F	Function call	F a function
		$\mid$	`ret`	Function return	
		$\mid$	$r \leftarrow M[r']$	Memory read	
		$\mid$	$M[r'] \leftarrow r$	Memory write	
		$\mid$	`sp` $\leftarrow$ `sp` $+$ n	Advance the stack pointer	
		$\mid$	$r \leftarrow M[$`sp` $+$ $n]$	Stack read	
		$\mid$	$M[$`sp` $+$ $n] \leftarrow r$	Stack write	
		$\mid$	Annot	Annotations	discussed later
		$\mid$	End	Program termination	added for these implementations
Numerals:	n	$\in \mathbb{Z}$			

Figure 1: SAL Syntax — Taken from Necula [Nec98], Figure 3.1

We will be using the move ($r \leftarrow r'$) and memory read ($r \leftarrow M[r']$) instructions as examples throughout the paper. We show here some of the declarations which are used to represent SAL programs. In LLF, both type and term constants are declared in a signature. For now, we may think of a signature as simply of list of declarations of the form a : `type` for a type constant a and c : A for an object constant c of type A. We assume a type `exp` has already been declared which contains the values which can be held by a register and which may be the arguments to arithmetic operations. First, we declare the registers available in the machine as a type `rname`.

```
rname : type.
ra : rname.
r1 : rname.
r2 : rname.
```

We have the ability to create a machine with an arbitrary number of general purpose registers by simply adding more names for them in the form `r`n `: rname`. The different SAL instructions take varying numbers and types of parameters. The move and memory read instructions, our running examples, both require two registers for their operation. For of $r \leftarrow r'$ we write `mov` r r', while $r \leftarrow M[r']$ is represented as `memr` r r'.

```
instr   : type.
mov     : rname -> rname -> instr.
memr    : rname -> rname -> instr.
```

All the other instructions of the machine are encoded in a similar manner. The state of the machine consists of a program counter and the current values of the registers, stack pointer, and memory cells. They are represented by propositions of the form

`reg` r v	register r contains value v ,
`sp` a	stack pointer contains address a,
`mem` a v	memory at address a contains value v.

In LLF, propositions are coded as types, so `reg`, for example, is represented as a type family indexed by a register name r and a value v. This leads to the following declarations.

```
reg : rname -> exp -> type.
sp  : exp -> type.
mem : exp -> exp -> type.
```

Note that we allow not only types, but also type families to be declared in a signature.

5 SAL Execution

In this section we will concurrently develop two execution models of the SAL language. One is a standard execution model that lacks safety checks. This is the execution that would occur after code has passed through the PCC system. We will later be able to check properties of this type of execution. The other model includes the safety checks and corresponds to run-time verification.

The state of the SAL machine during execution is defined as a triple of values $\langle i, \rho, \mathcal{H}\rangle$, where i is the program counter, ρ is the state of the registers including `sp` and a pseudoregister `mem` that contains the contents of memory, and $\mathcal{H}$ is a sequence of register states that represents the call history of the program. The call history $\mathcal{H}$ is only required in the safe execution model as it is required to perform some of the safety checks. In the framework these values will be stored in the linear context, represented by Δ. The stack pointer is stored separately from the other registers simply to exert more control over it, i.e., so that general register operations do not apply to it.

The state of the machine also includes parameters that do not change such as the program, the amount of memory in the machine, and so on. The unrestricted context, represented by Γ, contains only these general properties. The construct `maxmem` *value* specifies the amount of memory in the machine. Each instruction of a program is stored as `prog` *n command.* The *n* is used as a line numbering scheme, though its real meaning would be the memory address at which the instruction is stored. We should note here that:

1. The code region is assumed to be completely separate from other regions of memory. In particular, it is a write-protected region.

2. Every instruction is assumed to consume one such address location.

It would not be difficult to relax both of these restrictions in the execution models; however, other components of this PCC system presently require these restrictions.

Two important definitions of type families for execution are shown below. The former, `run`, corresponds to the execution of a program from a certain program counter. The latter is used in implementing the operational semantics of each instruction.

```
run     : exp -> exp -> type.
exec    : instr -> exp -> exp -> type.
```

The main judgment we shall consider is

$$\Gamma; \Delta \vdash \texttt{exec}\, I\ X\ \Theta$$

which should be interpreted as in linear type theory, except that we have omitted the proof term M for the sake of brevity. The intended meaning is that with the assembly program and machine characteristics stored in Γ and with the register and memory state stored in current state Δ, the program can be run beginning with the program counter at X, which points to instruction I, to termination, with Θ being the final value of register $r1$. This register was chosen in an arbitrary manner; in fact, this is not necessary for the computation to succeed but allows the program to output a value.

We will now continue the specific examples that we started in the last section. Some of the notation that is used below is of the form

$i{+}{+}$	The address of the instruction after that of i.
	For our implementations this will be $i + 1$, but this is not the case in general.
$\rho(r)$	The value of register r under register state ρ.
$\rho[r \leftarrow v]$	The register state ρ with the value of register r updated to be v.
$\rho(\texttt{mem})$	The current memory state.
	Mathematically represented as a function from addresses to values

For the `mov` instruction, execution in state $\langle i, \rho, \mathcal{H}\rangle$ produces the state $\langle i{+}{+}, \rho[r \leftarrow \rho(r')], \mathcal{H}\rangle$. There are two stages that the framework must go through in order to simulate this. First is the lookup of the value of the register r' from the linear context. Because we usually do not want to permanently destroy this resource as the register should still exist with the same value, we define a lookup predicate, `rlook`, and connect it additively with the rest of the program. We would like it to satisfy

$$\frac{}{\Gamma; \Delta, \texttt{mem}\, M\ V \vdash \texttt{mlook}\, M\ V}$$

However, linear logic requires the resources in Δ to be used, so we use the additive unit $\top$ which consumes an arbitrary set of resources. The encoding then reads:

$$\frac{\Gamma;\Delta' \vdash \texttt{reg}\,R\,V \quad \Gamma;\Delta \vdash \top}{\Gamma;\Delta',\Delta \vdash \texttt{rlook}\,R\,V}\,rlook1$$

These judgments are meant to be read in a bottom-up fashion. Thus, in this case, the proof search for **rlook** $R\ V$ splits the linear context and attempts two sub-derivations. This also means that LLF code not only provides a specification of SAL execution but can also be used to simulate the running of SAL code. We can then easily see then that Δ' must be `reg` $R\ V$ for the left subgoal to succeed and that $\Gamma;\Delta,\texttt{reg}\,R\,V \vdash \texttt{rlook}\,R\,V$ is a derived rule of inference.

The code in LLF for this as as follows, where `<T>` denotes $\top$:

```
rlook   : rname -> exp -> type.
rlook1  : reg R V -o <T> -o rlook R V.
```

This is a direct transcription of the given rule with very little notational overhead. We think of the inference rule `rlook1` as a linear function from derivations of the premises to a derivation of the conclusion. This simple technique works for most of our implementation and attest to the appropriateness of the linear logical framework. The arithmetic operations are the only exception. This is because the framework does not have a built-in understanding of integers, which we therefore explicitly programmed in binary form.

The second phase of the `mov` command is to write this value to register r. Since all of the instructions that update state follow the model of doing preliminary calculations, updating one part of the state, incrementing the program counter, and continuing, it is useful to define an auxiliary judgment to combine the last three steps of the four. Therefore, we will let `upr1` $R\ V\ X\ \Theta$ be that judgment. The register being updated is R with the new value V. The code address and output variable are X and Θ, as usual. `runnext` does as its name suggests; it continues execution at the next address.

$$\frac{\Gamma;\Delta,\texttt{reg}\,R\,V \vdash \texttt{runnext}\,X\ \Theta}{\Gamma;\Delta,\texttt{reg}\,R\,V_2 \vdash \texttt{upr1}\,R\ V\ X\ \Theta}\,upr1a$$

The code is the following:

```
upr1a :     (reg R V -o runnext X Theta)
        -o (reg R V2 -o upr1 R V X Theta).
```

Note how the linear function type constructor in both premise and conclusion is used to model the consumption (conclusion) and introduction (premise) of a resource.

We continue similarly for memory. With these two assisting predicates in place, it is a simple matter to finish the `mov` command. The judgment and code are listed here, with & being the additive conjunction in the logic. This means that both premises obtain a complete copy of the resources Δ. There are no safety concerns with the `mov` command, so this is identicial in the unsafe and safe execution models.

$$\frac{\Gamma;\Delta \vdash \texttt{rlook}\,R_2\,V_2 \quad \Gamma;\Delta \vdash \texttt{upr1}\,R_1\,V_2\,X\ \Theta}{\Gamma;\Delta \vdash \texttt{exec}\ (\texttt{mov}\,R_1\ R_2)\ X\ \Theta}\,exmov$$

```
exmov :     (rlook R2 V2 & upr1 R1 V2 X Theta)
        -o exec (mov R1 R2) X Theta.
```

The `memr` command is quite similar. The operational semantics shows that from initial state $\langle i, \rho, \mathcal{H}\rangle$ the command `memr` $r\ r'$ produces state $\langle i{+}{+}, \rho[r \leftarrow \rho(\texttt{mem})(\rho(r'))], \mathcal{H}\rangle$. This requires use of `mlook`, `rlook`, and `upr1`, and that is all that we must do for the unsafe execution model. There is a safety check that we must do on memory reads, however. It is represented by the predicate `safeRd` a. `safeRd` represents an unspecified predicate defining memory read safety. It may check for memory alignment as well as ensuring that memory functions do not operate on the stack. `safeRd` uses the current state to determine whether it is safe to read from address a. The relevant judgment and definitions follow:

$$\frac{\Gamma;\Delta \vdash \texttt{rlook}\ R_2\ M \quad \Gamma;\Delta \vdash \texttt{safeRd}\ M \quad \Gamma;\Delta \vdash \texttt{mlook}\ M\ V \quad \Gamma;\Delta \vdash \texttt{upr1}\ R_1\ V\ X\ \Theta}{\Gamma;\Delta \vdash \texttt{exec}\ (\texttt{memr}\ R_1\ R_2)\ X\ \Theta}\ \textit{exmemr}$$

```
safeRd  : exp -> type.
exmemr  :    (rlook R2 M & safeRd M & mlook M V & upr1 R1 V X Theta)
          -o exec (memr R1 R2) X Theta.
```

We move to functions at this point as they require significant changes between the unsafe and safe execution models. In the unsafe model, there is no difficulty with functions. The `call` command must simply jump to the address that marks the function beginning. The `ret` command looks up the current value of register ra and jumps to that address. There are two important parts to the safety checks that must be done here. One half, the checking of the state against annotated function preconditions and postconditions, is very similar to the addition of the `safeRd` predicate above. The other half involves ensuring that the values in certain registers and stack locations are held constant across function calls. To do this we must manipulate the call history, which has been denoted by $\mathcal{H}$ previously. We have chosen to do this by annotating register and memory values also with their function depth. Then on function invocation we can copy the state to the next deeper level, and on function return we can compare the two adjacent depths before reducing the current depth. This requires the following addition and changes to our type families,

```
depth   : exp -> type.
reg     : exp -> rname -> exp -> type.
mem     : exp -> exp -> exp -> type.
```

where the first parameter to `reg` and `mem` now correspond to the function depth. We store `depth` d in the linear context Δ so that we can recall the current depth at any point without needing to change many of the definitions.

We would like to reemphasize that the encoding of all the operations of the machine are of a similarly direct nature as the examples shown in this section. In addition to providing a high-level specification, the two LLF signatures can also be viewed as a logic program to execute SAL code, in safe or unsafe mode, respectively. In the current LLF implementation which is closely related to the implementation of Twelf [PS99], this is efficient enough to write a simple iterative SAL programs and run them.

6 Properties of Computations

The safe execution model from the last section represents a run-time safety verification mechanism. In this section we will explore an alternative approach to specifying safety. The safety checks in the run-time checker apply to single instructions. Here we will consider safety as a property of a complete computation. The specification of safety will also provide a means for checking whether or not any given completed execution from the unsafe model was, indeed, safe, using the logic programming interpretation of LLF signatures.

A major advantage of this method of checking safety is an increase in modularity of the different components of safe execution. With the safety checks built in to the SAL interpreter, a change in safety policy requires the SAL interpreter itself to be modified. When we consider safety as a property of a computation, we can separate the interpreter and the safety checker into two distinct components. First, this separate interpreter is a more realistic execution. Second, a change in safety policy will leave the interpreter untouched while changes occur in the safety checker.

We formally specify safety of a computation by exploiting the *proof terms* that the framework provides. When we run a SAL execution in LLF, the rules as described in the previous sections are used to generate a proof that corresponds to our execution query. Because we have been careful in providing these rules, there is only one possible rule to use for each SAL command and in each auxiliary situation. Therefore there is exactly one ordering of rules that corresponds to a successful execution and it is not difficult for the framework to find it. If the execution should fail for some reason, for example by calling a function that does not exist, then there is no such ordering of rules and the proof fails. In the case that the proof succeeds, we would like more information than simply that the proof succeeded; we would like to see the proof itself. Hence, we label the rules that we provided above, and the proof term that is returned by LLF in a successful proof is constructed with these labels as well as constructs in the logic. Since proof terms are an integral part of the LLF type theory, we can write LLF programs to manipulate them and that is how we will check safety after-the-fact on executions in the unsafe model.

These proof terms contain complete records of SAL executions; they can be viewed as execution traces. As we traverse a proof term we are continuing along the execution and can check safety properties where necessary. Unfortunately, proofs terms do not explicitly carry enough information to make this completely trivial; the proof term does not directly contain any information about the state of the machine at specific points in the computation. It does provide enough information to construct this as we go, however, and this means that we need to, in essence, re-execute the program in order to keep the appropriate state data. We do not need to make decisions regarding flow control, as the proof term dictates the exact execution; we simply need to know the state (via watching of the update register clause for example) so that when the proof term signals, say, a function return, we can extract the appropriate information in order to do the safety check.

Since there are no safety checks on the `mov` instruction, it is an uninteresting case. Recall the unsafe version of the implementation of the `memr` command.

```
exmemr :      (rlook R2 M & mlook M V & upr1 R1 V X Theta)
           -o exec (memr R1 R2) X Theta.
```

Here, `exmemr` is a constructor for proof terms, which is applied to a triple representing the proofs of the three premises. In the notation of LLF:

```
exmemr ^ (RL , ML , U) : exec (memr R1 R2) X Theta
```

where `^` denotes application of a linear function and

```
RL : rlook R2 M
ML : mlook M V
U : upr1 R1 V X Theta
```

Next we define our predicates on proof terms. These are once again represented as type families, following the methodology of the the logical framework. The two that are part of the memr example are shown here

```
okexec : exec I X Theta -> type.
okupr1 : upr1 R V X Theta -> type.
```

For the actual rule, we note that all we need to do is check that the safeRd predicate is satisfied with the address that is used. The linear argument to exmemr, a subproof in the term, contains sub-derivations of the three clauses in the above memr execution rule. Braces, {}, are the LLF concrete syntax for universal quantification. Therefore, we can discover the address that is being read from by looking at the type of ML. We then continue traversing the proof term with the call to okupr1.

```
okexmemr : {ML:mlook M _}
           (safeRd M & okupr1 U) -o okexec (exmemr ^ (_, ML, U))
```

Several safety policies have been implemented in the checker. More can be added, and it is very simple to do so with checks that deal with a small number of commands (i.e. a small number of rules of the implementation). First, of course, are the standard SAL safety checks, the original motivation for the checker. We have also added, as options, several smaller safety properties. For example, we have implemented a check that disallows backwards jumps. This is a requirement in the Berkeley Packet Filter (BPF) [MJ93]. This is an interesting example to examine because a concrete check is done rather than calling an abstract safety predicate.

The code for the jump instruction is presented first. The reason for having a separate execjmp predicate is so that other instructions, for example condCOP, may make use of it.

```
exec    : instr -> exp -> exp -> type.
execjmp : exp -> exp -> exp -> type.
exjmp   : execjmp V X Theta -o exec (jmp V) X Theta.
execjmp1: run X2 Theta -o bplus V X X2 -> execjmp V X Theta.
```

New here is the use of unrestricted implication -> to indicate a premise which does not require access to the current machine state. In this case, it is bplus v x x_2 which will hold if and only if $v + x = x_2$.

Since all jumps will use the execjmp predicate, placing the safety check there will enforce the no backward jumping safety condition on all of these instructions instead of just the jmp instruction. The key then is to verify that the value of V in the above code is positive. Therefore, the code for checking the first part of the proof term is trivial.

```
okexjmp : okexecjmp E -o okexec (exjmp ^ E).
```

To extract the relative jump value, we examine the proof term for the second premise of the execjmp1 rule from above. The predicate ble v_1 v_2 is provable if and only if $v_1 \leq v_2$.

```
okexecjmp1 : {B:bplus V _ _}
             okrun R -o ble 1 V -> okexecjmp (execjmp1 ^ R B).
```

We also have a simple check that disallows function calls. The last policy actually addresses a minor deficiency in the implementation. It is difficult to express the idea of bounded numbers in the framework, so registers may take on unbounded values. This is not a faithful representation, and could lead to a loophole if an overflow value causes a safety problem. However, it is easy to add an inequality check to ensure that the register values never go beyond some bound. We should note that the bounds check could also be built into the actual register updating procedure so that overflow could be simulated, but that it could not easily be built into the actual expression type.

7 VCGen

Generation of the verification condition (VC) follows the same pattern as execution but with a few differences. First, we will also instantiate register and memory location values as variables or expressions involving variables. Second, we will only reach any given line of code (except annotations) at most once due to care in handling loops and function calls. Last, because of the first two differences, we will usually not have explicit values in register and therefore may not know at what address a memory read or write occurs. Thus, we must deal with memory as one entity instead of many different locations as done in the execution models. The judgments for VCGen are therefore similar to those for execution but reflect these differences. The VC that we generate does not have any linear components.

To handle these new requirements for the memory implementation, we define a new memory state expression `stexp` as well as two important functions for use on variables of type `stexp`,

```
stexp   : type.
sel     : stexp -> exp -> exp.
upd     : stexp -> exp -> exp -> stexp.
mem     : stexp -> type.
mlook   : stexp -> type.
```

with the following usage

`sel` $s\ a$	the value at address a in memory state s
`upd` $s\ a\ v$	the memory state equivalent to s but with the value at address a set to v.

and `mem` and `mlook` being a means of storing the entire memory state in the linear context.

Many of the predicates correspond exactly to their execution counterparts, with the output value (Θ in past judgments) replaced by the VC, an object of type `pred`. These are a few of the relevant definitions, where `/\` is defined to be an infix conjunction operator.

```
true    : pred.
/\      : pred -> pred -> pred.
saferd  : stexp -> exp -> pred.
exec    : instr -> exp -> pred -> type.
```

The main judgment in this case is

$$\Gamma; \Delta \vdash \texttt{exec}\ I\ X\ VC$$

which, as before, should be interpreted as in linear type theory. The proof term M is again omitted. The program and machine characteristics are stored in the unrestricted context, Γ. Register values, the new memory mechanism, and the current depth are stored in the linear context, Δ. The intended meaning is that under this state, examination of the assembly program from instruction I at address X results in verification condition VC.

In the case of a memory read, we perform the read and register update as in normal execution, and we also add a `saferd` component to the VC.

```
    exmemr  :    (rlook R2 M & mlook MEM & upr1 R1 (sel MEM M) X VC)
              -o exec (memr R1 R2) X ((saferd MEM M) /\ VC).
```

We generally will not know the outcome of a conditional operation; therefore, we must take both routes when we encounter a condCOP statement. We want to guarantee termination; moreover, we would prefer that running time be proportional to the amount of code instead of the actual running time of the program. Both of these are treated by careful handling of functions and loops.

To deal with functions, we require that every function be annotated with sets of preconditions and postconditions. We then generate a separate VC for each function in the program and build the overall VC by forming the conjunction of these smaller statements. To examine a function by itself, we need to initialize the relevant components of the state. We wish to show that the function is safe beginning from any state that satisfies the given preconditions, so we introduce variables for values and modify the VC to assume that these variables satisfy certain conditions related to the function preconditions. Thus we have created half of an implication that will be completed by the remainder of the function.

Upon reaching the end of the function, we close the implication by adding a final set of conditions that requires the final state to conform to the function's postconditions. The annotation for a function also includes a set of registers (the *callee-save register set* that are guaranteed to be preserved across the function call. Statements enforcing this preservation are also added at this time.

On a function call within the function currently being analyzed, we do not want to transfer control to the called function — this is the reason that we have required preconditions, postconditions, and the callee-save register set. We simply add conditions requiring the current state to conform to the function's preconditions. Then we introduce new variables as values for any location that is not contained in the callee-save register set and then assume that the postconditions hold on this new state, again writing half of an implication into the VC, exactly as done at the beginning of the function.

Loops require much of the same machinery. We require that the destination of any backward jump instruction to be a loop annotation. This annotation contains a loop invariant as well as a set of registers that may be modified by the loop. Anytime we reach a loop annotation, we always add to the VC statements enforcing the state to conform to the invariant. On reaching the invariant for the first time, we introduce variables for any location that may be changed by the loop corresponding to the invariant and continue. On subsequent visits to the invariant, we check that only the allowable registers have been modified and then end VC construction. This will usually be the branch of a condCOP statement that continues the loop, with the other branch continuing towards the end of the function.

8 Conclusion and Future Work

We have implemented two execution models for safe assembly language, a post-execution safety checker, a verification condition generator and the inference rules of the predicate calculus in the logical framework. The framework itself already provides the proof checker. This completes a specification of the key trusted component of the PCC architecture.

The specification in the framework has a number of attractive features: it is concise and at a high-level of abstraction. It can be executed using the logic programming interpretation of the linear logical framework. The features of the framework which made this natural encoding possible include the linear operators for machine state, and the dependent types and proof terms for the external safety check.

However, there are also some shortcomings of the framework. The absence of arithmetic means binary numbers have to be painfully coded. Furthermore, the current LLF implementation provides no tracing or other debugging tools which makes it difficult to find and correct errors in a specification. Finally, the absence of a module system requires a lot of programming by cut-and-paste which should not be necessary. To give in idea of the size of the implementation: the unsafe SAL specification occupies about 250 lines, safe SAL execution about 300. The external safety checker consist of about 200 lines, while the VCGen implementation is about 500 lines long.

The next step will be to formalize the meta-theoretic proofs of correctness for the architectures. In particular, we would like to verify that the unsafe execution of a program whose verification condition has been proven satisfies the explicit safety check. We would also like to investigate other uses of linearity in the PCC system. For example, it may be possible to use linearity in the verification condition itself, which could be one way of reducing proof size.

References

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, University of Pennsylvania, May 1995.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, January 1997.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994.

[MJ93] Steven McCanne and Van Jacobsen. The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference*, pages 259–269. USENIX Association, January 1993.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Nec97] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[Nec98] George C. Necula. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)*, pages 229–243, Seattle, Washington, October 1996.

[NL98a] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Keith D. Cooper, editor, *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montreal, Canada, June 1998. ACM Press.

[NL98b] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, Trento, Italy, June 1999. Springer-Verlag LNCS. To appear.