Temporal Logic for Proof-Carrying Code

Andrew Bernard Peter Lee[1]

August 29, 2002

CMU-CS-02-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Proof-carrying code (PCC) is a framework for ensuring that untrusted programs are safe to install and execute. When using PCC, untrusted programs are required to contain a proof that allows the program text to be checked efficiently for safe behavior. In this paper, we lay the foundation for a potential engineering improvement to PCC. Specifically, we present a practical approach to using temporal logic to specify security policies in such a way that a PCC system can enforce them.

[1]This work has been partially supported by the National Science Foundation under grant CCR-0121633.

Keywords: proof-carrying code, temporal logic, formal methods

1 Introduction

Proof-carrying code [Nec97] (PCC) is a framework for ensuring that untrusted programs are safe to install and execute. When using PCC, untrusted programs are required to contain a proof that allows the program text to be checked efficiently for safe behavior. PCC can check optimized object code, and a program checker is relatively easy to implement. These advantages, among others, make PCC an attractive scheme for enabling a network of computers to distribute software safely. In this paper, we lay the foundation for a potential engineering improvement to PCC. Specifically, we present a practical approach to using temporal logic to specify security policies in such a way that a PCC system can enforce them. The PCC system would furthermore be "universal," in the sense of not needing to be modified or extended for each new security policy, as long as each such policy can be specified in temporal logic. This approach additionally enables us to replace a substantial portion of the program-checking software with formal specifications, but at the cost of larger proofs.

A central component of a PCC program checker is the *security policy*, which defines the precise notion of "safety" that the host system demands of all untrusted code. In the work cited above, a major portion of the security policy is given by a verification-condition (VC) generator that in practice takes the form of a manually constructed computer program (written, in this particular case, in the C programming language). While this is an expedient approach that is also consistent with the desire to implement PCC as an operating system service, it does not necessarily lead to a trustworthy checker, nor does it permit easy adaptation of the checker to new security policies.

To motivate the problems addressed by this research, consider how we might design a PCC-based personal digital assistant (PDA). The PDA can be enhanced by new programs, with the proviso that each such program is checked by PCC before it is installed, thereby ensuring that the PDA (a *code consumer*) will not cease to work because of faulty or malicious software. Untrusted extensions are provided by a *code producer*; we will focus on two for the moment:

- The *alarm clock* runs continuously, but only for brief intervals. It updates the display once per second and emits a special sound, when appropriate.
- The *synchronizer* runs only when the user "docks" the PDA. The synchronizer ensures that the PDA is consistent with a desktop computer.

Figure 1 contains a diagram of this design. A trusted *enforcement mechanism* checks each program against several distinct security policies before it is allowed to run. A *memory-safety* policy protects the operating system and libraries from corruption. Additional *resource-bound* policies place limits on the system resources that programs can consume. The memory-safety policy is common to all programs, but the resource-bound policies are tailored to individual programs.

The alarm clock needs little memory to run, but runs continuously for an unlimited period of time; it is usually waiting in between clock ticks. We thus assign to the alarm clock the *wait-frequency* policy that limits it to a small number of instructions before invoking the `wait` system call. The *small-heap-bound* policy constrains the alarm clock to only a small amount of dynamic memory. The *instruction-bound* policy requires the synchronizer to terminate after executing a number of instructions proportional to the size of the PDA's address book. The *large-heap-bound* policy constrains the synchronizer to a large amount of dynamic memory (also proportional to the address-book size); because the synchronizer will terminate in a limited time frame, we know that its dynamic memory will be released soon.

A typical implementation of this design would require a separate enforcement mechanism for each distinct security policy. Unfortunately, it is relatively difficult to tailor an enforcement mechanism to a

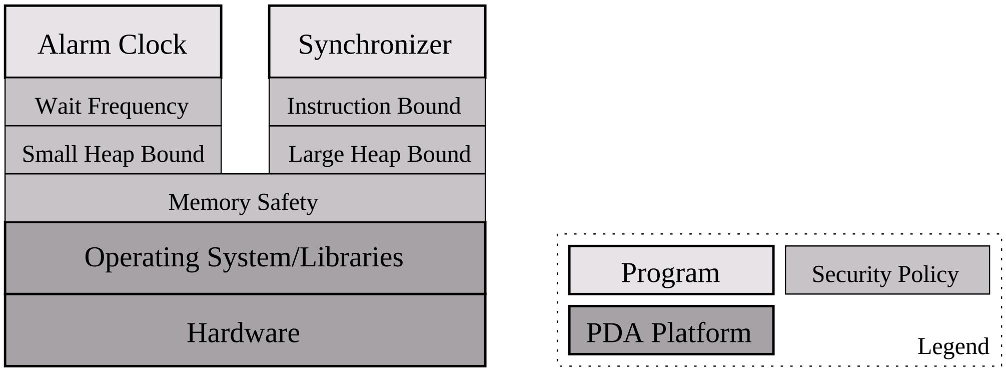

Figure 1: Secure PDA

new security policy (in general), especially if we expect to change the policy over time or if we want to vary it for different programs. On the one hand, we can try to incorporate all the security policies at once into a single mechanism, but this leaves us with a complex mass of code that is difficult to reuse in new situations. On the other hand, we can implement a separate mechanism for each security policy, but this is potentially inefficient because each mechanism must examine the program and its proof.

We would prefer that security policies were instead parameters of a single universal enforcement mechanism. We could then develop policy and mechanism independently, and reuse a single implementation for an unlimited number of applications. We first attempted to address this problem by extending a standard enforcement mechanism with a security-policy interpreter [BL01]—unfortunately, this approach entails considerable complexity. In this paper, we present an alternative approach that uses a simpler enforcement mechanism, but at the expense of larger security proofs.

Until now, our PCC implementations have encoded security proofs in first-order logic, and the enforcement mechanism included a trusted VC generator that essentially encoded the security policy in a C implementation (*e.g.*, Necula [Nec97]). We will argue here that *temporal logic* [MP91, Eme90, CGP99] has certain advantages over first-order logic for PCC. Using temporal logic, we can remake the VC generator as an *untrusted* component and thereby allow the security policy to be separated from the enforcement mechanism. This also provides the crucial advantage of reducing the amount of software in the trusted computing base, though as we shall see, this advantage comes at the cost of larger proofs. In this respect, our approach resembles *foundational PCC* [App01, AF00], although, unlike foundational PCC, our code producer and consumer must agree on a shared notion of type safety.

A temporal logic is characterized by its temporal operators: they enable us to distinguish the different times at which a proposition is true. In this paper, we will identify time with the CPU clock and regard propositions as statements about machine states. For example, the proposition

$$\mathtt{pc} = 0 \supset \bigcirc(\mathtt{pc} = 1)$$

asserts that "if the program counter is 0 now, then it will be 1 in the next state." We can also specify security policies in temporal logic. For example, the proposition

$$\Box(\mathtt{pc} \geq 0 \land \mathtt{pc} < 100)$$

asserts that "the program counter is always between zero and 100," but we can also interpret this as the requirement "the program counter must always be between zero and 100"—a specification for a simple form of control-flow safety [Koz98]. We will exploit this duality to reap a practical benefit.

For a PCC system based on first-order logic, the enforcement mechanism generates a proposition from the program and the security policy together—the security proof is a proof of this proposition. For temporal-logic PCC, the enforcement mechanism recognizes the program as a formal term, and the operational semantics of the host machine is encoded as a set of trusted inference rules. We can then encode the security policy directly—the security proof shows that the security policy is a consequence of running the program from a set of initial conditions. Notice that the security policy is independent of the enforcement mechanism, but we require no additional mechanism to interpret it.

We want to be confident that the security policy is correct: this confidence is difficult to obtain for a security policy in C code. In contrast, temporal logic has a clear semantics, and security policies are comparatively compact.

Temporal logic can express a wide variety of security policies [MP90], including type-safety, resource-bound, and liveness policies. For example,

$$(\mathtt{n} = 0 \wedge \Box(\bigcirc(\mathtt{n}) = \mathtt{n} + 1)) \supset \Box(\mathtt{n} \geq 1000 \supset \Phi_{\mathtt{pc}} = \mathtt{halt})$$

is an encoding of an instruction bound. Read this proposition as "for any n such that n is initially zero and increases by one at each cycle,[1] we must halt by the time $\mathtt{n}$ reaches 1000."

As we shall see, we can implement a simple enforcement mechanism for temporal-logic PCC at the cost of increasing proof sizes. This can be a favorable trade-off, because we are shifting work from a trusted component to an untrusted one. Initial experiments show that the size increase relative to a first-order proof is a small multiple of the code size.

The body of this paper lays a theoretical foundation for temporal-logic PCC. Section 2 outlines a first-order temporal logic that is suitable for PCC security proofs. Section 3 defines an abstract RISC processor for which our framework is intended. Section 4 details how the machine semantics is encoded and why it is sound. Section 5 shows we can systematically obtain efficient temporal type-safety proofs from first-order type-safety proofs. Finally, in Section 6 we examine related work and suggest future improvements.

2 Temporal Logic

We use a linear-time first-order temporal logic that resembles classical temporal logic [MP91]. However, instead of developing an axiomatization of this logic we follow Davies [Dav96] and Simpson [Sim94] and construct a natural-deduction system based on explicit times [BPW01]. We use a natural-deduction system to enable integration with other PCC systems, and because the orthogonal treatment of connectives facilitates incremental extensions and restrictions. The extension of Davies' system to additional temporal operators is straightforward; the extension to first-order quantifiers requires more effort to accommodate both rigid and flexible variables (see Section 2.1).

[1] Here we use $\bigcirc()$ as an abbreviation for a more complex expression (see Section 4 for examples of incrementing parameters).

$$
\begin{array}{lll}
\text{Times} & t & ::= 0 \mid t_1 + 1 \\
\text{Rigidities} & \rho & ::= +_{\mathsf{r}} \mid -_{\mathsf{r}} \\
\text{Parameter Lists} & \alpha & ::= \cdot \mid \alpha_1, a \\
\text{Expressions} & e^\tau & ::= a^\tau \mid x^\tau \mid f^{\tau_1 \times \cdots \times \tau_k \to \tau}(e_1^{\tau_1}, \ldots, e_k^{\tau_k}) \\
\text{Propositions} & p & ::= R^{\tau_1 \times \cdots \times \tau_k \to o}(e_1^{\tau_1}, \ldots, e_k^{\tau_k}) \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid p_1 \supset p_2 \\
 & & \quad \mid \forall x^\tau : \rho.\ p_1 \mid \exists x^\tau : \rho.\ p_1 \mid \Box p_1 \mid \Diamond p_1 \mid \bigcirc p_1 \mid p_1 \,\mathcal{U}\, p_2 \mid p_1 \,\mathcal{W}\, p_2 \\
\text{Core Judgments} & J & ::= t_1 \geq t_2 \mid p : +_{\mathsf{l}} (a) \mid e : \rho\ (\alpha) \mid p : \rho\ (\alpha) \mid p \,@\, t \mid p \,@[t_1, t_2) \mid p \,@[t, \infty) \\
\text{Contexts} & \Gamma & ::= \cdot \mid \Gamma,\ J
\end{array}
$$

Figure 2: Abstract Syntax (Temporal Logic)

2.1 Syntax

The syntax of our logic (see Figure 2) is based on disjoint countably infinite sets of parameters and variables; a *parameter* a is always free in a proposition, whereas a *variable* x is normally bound.[2] This is a many-sorted logic, so each parameter or variable is annotated with an explicit type τ, of which there are countably many; types have no internal structure. We often omit type annotations when they can be inferred. Primitive functions and relations are named by a countable set of *constants* (f and R, respectively). Constants are also annotated with types: $\tau_1 \times \cdots \times \tau_k \to \tau$ is the annotation of a function from k parameters to a value of type τ, whereas $\tau_1 \times \cdots \times \tau_k \to o$ is the annotation of a relation on k parameters. Constant values c^τ are nullary functions, whereas constant propositions (*i.e.*, $\top$, $\bot$) are nullary relations. There is a binary equality relation for each type. This is a first-order logic, so functions and relations appear only as constants.

Expressions e^τ are constructed from parameters, variables, and applications of constant functions; τ is the type of e. The simple type system for our logic is built into the syntax: ill-typed expressions are not well formed.

Following Manna and Pnueli [MP91], some expressions are *rigid*: it is syntactically evident that a rigid expression has the same value at all times. A *flexible* expression may (but need not) have different values at different times. For example, the constant 5 is rigid, whereas the stack pointer register is flexible. Variables also have rigidity: rigidities must match when a variable is instantiated. We declare the rigidity ρ of a variable when the variable is bound: $+_{\mathsf{r}}$ denotes a rigid variable, whereas $-_{\mathsf{r}}$ denotes a flexible variable. A rigid expression contains only rigid variables and parameters.

Propositions p include a selection of the usual connectives and quantifiers of first-order logic, plus the following temporal operators:

- $\Box p$ holds iff p holds at all future times.
- $\Diamond p$ holds iff p holds at some future time.
- $\bigcirc p$ holds iff p holds at the next future time.
- $p_1 \,\mathcal{U}\, p_2$ holds iff p_2 holds at some future time, and p_1 holds until then.

[2]The syntactic distinction between parameters and variables simplifies inference rules.

- $p_1 \mathcal{W} p_2$ holds iff p_1 holds until the first future time at which p_2 holds, but p_2 need never hold.

A rigid proposition has only rigid parameters (bound variables may be flexible).

Some propositions are associated with a *time expression* t; we count time in unary notation: 0 denotes the earliest possible time (*e.g.*, the start of execution), and $t+1$ denotes the time immediately following time t.

$[e_1/x]\,e$ is the usual *substitution* of expression e_1 for variable x in expression e. For substitution to be well formed, e_1 must have the same type as x, and e_1 must be closed (*i.e.*, it must not contain variables); e need not be closed. $[e/x]\,p$ is the usual extension, where e must be closed, but p need not be.

$$
\begin{array}{ll}
[e/x]\,a & = a \\
[e/x]\,x & = e \\
[e/x]\,x_1 & = x_1 \text{ if } x \neq x_1 \\
[e/x]\,f(e_1,\ldots,e_k) & = f([e/x]\,e_1,\ldots,[e/x]\,e_k)
\end{array}
$$

$$
\begin{array}{ll}
[e/x]\,R(e_1,\ldots,e_k) & \equiv R([e/x]\,e_1,\ldots,[e/x]\,e_k) \\
[e/x]\,(p_1 \wedge p_2) & \equiv [e/x]\,p_1 \wedge [e/x]\,p_2 \\
[e/x]\,(p_1 \vee p_2) & \equiv [e/x]\,p_1 \vee [e/x]\,p_2 \\
[e/x]\,(p_1 \supset p_2) & \equiv [e/x]\,p_1 \supset [e/x]\,p_2 \\
[e/x]\,\forall x.\ p & \equiv \forall x.\ p \\
[e/x]\,\forall x_1.\ p & \equiv \forall x_1.\ [e/x]\,p \text{ if } x \neq x_1 \\
[e/x]\,\exists x.\ p & \equiv \exists x.\ p \\
[e/x]\,\exists x_1.\ p & \equiv \exists x_1.\ [e/x]\,p \text{ if } x \neq x_1 \\
[e/x]\,\Box p & \equiv \Box [e/x]\,p \\
[e/x]\,\Diamond p & \equiv \Diamond [e/x]\,p \\
[e/x]\,\bigcirc p & \equiv \bigcirc [e/x]\,p \\
[e/x]\,(p_1\,\mathcal{U}\,p_2) & \equiv [e/x]\,p_1\,\mathcal{U}[e/x]\,p_2 \\
[e/x]\,(p_1\,\mathcal{W}\,p_2) & \equiv [e/x]\,p_1\,\mathcal{W}[e/x]\,p_2
\end{array}
$$

Substitution has the following properties:

Proposition 2.1 (Absence) $[e_1/x]\,e = e$ *if* x *does not appear in* e

Proposition 2.2 (Elimination) x *does not appear in* $[e_1/x]\,e$

Proposition 2.3 (Exchange) $[e_1/x_1]\,[e_2/x_2]\,e = [e_2/x_2]\,[e_1/x_1]\,e$ *if* $x_1 \neq x_2$

Proofs by induction on the structure of e.

Proposition 2.4 (Idempotency) $[e_1/x]\,[e_2/x]\,e = [e_2/x]\,e$

Proof by Absence and Elimination.

Proposition 2.5 (Idempotency) $[e_1/x]\,[e_2/x]\,p \equiv [e_2/x]\,p$

Proposition 2.6 (Exchange) $[e_1/x_1]\,[e_2/x_2]\,p \equiv [e_2/x_2]\,[e_1/x_1]\,p$ *if* $x_1 \neq x_2$

Proofs by induction on the structure of p.

2.2 Semantics

We define a formal model for our temporal logic. Each expression is assigned the infinite sequence of values that the expression takes over time. A satisfaction relation determines whether a given proposition holds at a given time. This model is similar to the usual models of temporal logic.

2.2.1 Definitions

Val^τ is the set of values v^τ of type τ. A *sequence* π^τ is mapping from natural numbers (representing times) to values of type τ. An *environment* ϕ maps each parameter to a sequence of its type.

We assume an *interpretation function* $\mathcal{J}$ mapping each constant to its value, which may be a simple value (nullary functions), a total function (other functions), or a set of tuples (relations). We assume that $\mathcal{J}$ is defined as follows for the basic constants:

$$\begin{aligned} \mathcal{J}(\top) &= \{\langle\rangle\} \\ \mathcal{J}(\bot) &= \emptyset \\ \mathcal{J}(=^\tau) &= \{\langle v, v\rangle \mid v \in Val^\tau\} \end{aligned}$$

2.2.2 Valuation

A *valuation function* $\mathcal{V}$ assigns values to expressions. Thus, $\mathcal{V}(t)$ is the value of time expression t as a natural number:

$$\begin{aligned} \mathcal{V}(0) &= 0 \\ \mathcal{V}(t+1) &= \mathcal{V}(t) + 1 \end{aligned}$$

$\mathcal{V}_\phi$ evaluates expressions to sequences of the same type in the environment ϕ; e must be closed for $\mathcal{V}_\phi(e)$ to be well formed:

$$\begin{aligned} \mathcal{V}_\phi(a) &= \phi(a) \\ \mathcal{V}_\phi(f(e_1, \ldots, e_k)) &= j \mapsto \mathcal{J}(f)(\mathcal{V}_\phi(e_1)(j), \ldots, \mathcal{V}_\phi(e_k)(j)) \end{aligned}$$

Let Seq^τ be the set of all sequences of type τ. Valuation has the following properties:

Proposition 2.7 (Type Preservation) $\mathcal{V}_\phi(e^\tau) \in Seq^\tau$

Proof by induction on the structure of e^τ.

Proposition 2.8 (Renaming) $\mathcal{V}_{\phi[a_1 \mapsto \pi]}([a_1/x]\, e) = \mathcal{V}_{\phi[a_2 \mapsto \pi]}([a_2/x]\, e)$
if a_1 and a_2 do not appear in e and $[a_1/x]\, e$ is closed

Proposition 2.9 (Independence) $\mathcal{V}_{\phi[a \mapsto \pi]}(e) = \mathcal{V}_\phi(e)$
if a does not appear in e and e is closed

Proposition 2.10 (Past/Future Independence) $\mathcal{V}_{\phi[a \mapsto \pi]}(e)(j) = \mathcal{V}_\phi(e)(j)$
if $\pi(j) = \phi(a)(j)$ and e is closed

Proposition 2.11 (Extraction) $\mathcal{V}_\phi([e_1/x]\, e) = \mathcal{V}_{\phi[a \mapsto \mathcal{V}_\phi(e_1)]}([a/x]\, e)$
if a does not appear in e and $[e_1/x]\, e$ is closed

Proofs by induction on the structure of e.

2.2.3 Satisfaction

A proposition is *local* in a parameter if it is only sensitive to the current-time value of the parameter (*e.g.*, we can vary the value of the parameter at any other time without affecting the state of the proposition). A sequence is *rigid* if it has the same value at all times; the value of a rigid expression is always a rigid sequence, but the converse does not always hold. We write $\pi : \rho$ when π has rigidity ρ:

$$\pi : \rho \text{ iff } \rho = +_{\mathsf{r}} \text{ implies } \pi(j_1) = \pi(j_2) \text{ for all } j_1, j_2$$

A *core judgment* J encodes a property of an environment. The *satisfaction* relation $\vDash$ defines when a core judgment holds for a particular environment (see Figure 3); the judgment must be closed for satisfaction to be well formed. We informally describe each core judgment:

- $t_1 \geq t_2$ holds when t_1 denotes the same time as t_2 or a later time than t_2.
- $p\!:\!+_{\mathsf{l}}\,(a)$ holds when p is local in a.
- $e\!:\!\rho\,(\alpha)$ holds when e denotes a sequence with rigidity ρ.
- $p\!:\!\rho\,(\alpha)$ holds when p is a proposition with rigidity ρ.
- $p \,@\, t$ holds when p is true at time t.
- $p \,@[t_1, t_2)$ ("p is true over t_1 to t_2") holds when p is true at all times in the half-open interval $[t_1, t_2)$.
- $p \,@[t, \infty)$ ("p is true from t") holds when p is true at t and all times later than t.

α is a list of parameters whose sequences are "shifted" to the current comparison time when rigidity is considered; shifting parameters are introduced by quantifiers. For example, $\exists x\!:\!-_{\mathsf{r}}.\; x = 5$ is considered to be a rigid proposition, even though x is a flexible variable. We often abbreviate $e\!:\!\rho\,(\cdot)$ as $e\!:\!\rho$ and $p\!:\!\rho\,(\cdot)$ as $p\!:\!\rho$.

Thus, $\phi \vDash p \,@\, t$ ("ϕ satisfies p at time t") holds if p is true of ϕ at time t. We say that a proposition is *valid* at a given time if and only if it is satisfied by all environments at that time; a proposition is *valid* (in general) if and only if it is valid at all times.

We use the following notation for shifting environments and sequences:

$$\phi_{k..|\alpha} = a \mapsto \begin{cases} (\phi(a))_{k..} & \text{if } a \in \alpha \\ \phi(a) & \text{otherwise} \end{cases}$$

$$\pi_{k..} \quad = j \mapsto \begin{cases} \pi(0) & \text{if } j + k < 0 \\ \pi(j + k) & \text{otherwise} \end{cases}$$

Rigidity, shifting, and satisfaction have the following properties:

Proposition 2.12 (Equivalence) $\phi \vDash e\!:\!\rho$ *iff* $\mathcal{V}_\phi(e) : \rho$

Proof by definition of $\vDash$.

Proposition 2.13 (Rigidity) $\pi_{k..} : \rho$ *if* $\pi : \rho$

Proof by definition of $\pi_{k..}$.

$$\phi \models t_1 \geq t_2 \text{ iff } \mathcal{V}(t_1) \geq \mathcal{V}(t_2)$$

$$\phi \models p : +_{\mathsf{l}}\,(a) \quad \text{iff} \quad \phi \models p \mathbin{@} t \text{ implies } \phi[a \mapsto \pi] \models p \mathbin{@} t \text{ for all } \pi \text{ such that } \pi(\mathcal{V}(t)) = \phi(a)(\mathcal{V}(t))$$

$$\phi \models e : \rho\,(\alpha) \quad \text{iff} \quad \rho = +_{\mathsf{r}} \text{ implies } \mathcal{V}_{\phi}(e)(j_1) = \mathcal{V}_{\phi_{(j_1 - j_2)..|\alpha}}(e)(j_2) \text{ for all } j_1, j_2$$

$$\phi \models p : \rho\,(\alpha) \quad \text{iff} \quad \rho = +_{\mathsf{r}} \text{ implies } \phi \models p \mathbin{@} t_1 \text{ implies } \phi_{(\mathcal{V}(t_1) - \mathcal{V}(t_2))..|\alpha} \models p \mathbin{@} t_2 \text{ for all } t_1, t_2$$

$$\begin{array}{lll}
\phi \models R(e_1, \ldots, e_k) \mathbin{@} t & \text{iff} & \langle \mathcal{V}_{\phi}(e_1)(\mathcal{V}(t)), \ldots, \mathcal{V}_{\phi}(e_k)(\mathcal{V}(t)) \rangle \in \mathcal{J}(R) \\
\phi \models p_1 \wedge p_2 \mathbin{@} t & \text{iff} & \phi \models p_1 \mathbin{@} t \text{ and } \phi \models p_2 \mathbin{@} t \\
\phi \models p_1 \vee p_2 \mathbin{@} t & \text{iff} & \phi \models p_1 \mathbin{@} t \text{ or } \phi \models p_2 \mathbin{@} t \\
\phi \models p_1 \supset p_2 \mathbin{@} t & \text{iff} & \phi \models p_1 \mathbin{@} t \text{ implies } \phi \models p_2 \mathbin{@} t \\
\phi \models \forall x^{\tau} : \rho.\ p \mathbin{@} t & \text{iff} & \phi[a^{\tau} \mapsto \pi^{\tau}] \models [a^{\tau}/x^{\tau}]\, p \mathbin{@} t \text{ for some } a^{\tau} \text{ not appearing in } p \\
 & & \text{and all } \pi^{\tau} \text{ such that } \pi^{\tau} : \rho \\
\phi \models \exists x^{\tau} : \rho.\ p \mathbin{@} t & \text{iff} & \phi[a^{\tau} \mapsto \pi^{\tau}] \models [a^{\tau}/x^{\tau}]\, p \mathbin{@} t \text{ for some } a^{\tau} \text{ not appearing in } p \\
 & & \text{and some } \pi^{\tau} \text{ such that } \pi^{\tau} : \rho \\
\phi \models \Box p_1 \mathbin{@} t & \text{iff} & \phi \models p_1 \mathbin{@} t_1 \text{ for all } t_1 \text{ such that } \phi \models t_1 \geq t \\
\phi \models \Diamond p_1 \mathbin{@} t & \text{iff} & \phi \models p_1 \mathbin{@} t_1 \text{ for some } t_1 \text{ such that } \phi \models t_1 \geq t \\
\phi \models \bigcirc p \mathbin{@} t & \text{iff} & \phi \models p \mathbin{@} t + 1 \\
\phi \models p_1\, \mathcal{U}\, p_2 \mathbin{@} t & \text{iff} & \phi \models p_2 \mathbin{@} t_2 \text{ for some } t_2 \text{ such that } \phi \models t_2 \geq t \text{ and } \phi \models p_1 \mathbin{@} [t, t_2) \\
\phi \models p_1\, \mathcal{W}\, p_2 \mathbin{@} t & \text{iff} & \text{either } \phi \models p_1 \mathbin{@} [t, \infty) \\
 & & \text{or } \phi \models p_2 \mathbin{@} t_2 \text{ for some } t_2 \text{ such that } \phi \models t_2 \geq t \text{ and } \phi \models p_1 \mathbin{@} [t, t_2)
\end{array}$$

$$\phi \models p \mathbin{@} [t_1, t_2) \quad \text{iff} \quad \phi \models p \mathbin{@} t \text{ for all } t \text{ such that } \phi \models t \geq t_1 \text{ and } \phi \models t_2 \geq t + 1$$

$$\phi \models p \mathbin{@} [t_1, \infty) \quad \text{iff} \quad \phi \models p \mathbin{@} t \text{ for all } t \text{ such that } \phi \models t \geq t_1$$

Figure 3: The Satisfaction Relation

Proposition 2.14 (Cancellation) $(\pi_{k..})_{(-k)..}(j) = \pi(j)$ *if* $j \geq k$

Proof by definition of $\pi_{k..}$.

Proposition 2.15 (Cancellation) $(\phi_{k..|\alpha})_{(-k)..|\alpha}(a)(j) = \phi(a)(j)$ *if* $j \geq k$

Proof by Proposition 2.14 and definition of $\phi_{k..|\alpha}$.

Proposition 2.16 (Renaming) $\phi[a_1 \mapsto \pi] \vDash [a_1/x]\,p \,@\, t$ *iff* $\phi[a_2 \mapsto \pi] \vDash [a_2/x]\,p \,@\, t$
if a_1 *and* a_2 *do not appear in* p *and* $[a_1/x]\,p$ *is closed*

Proposition 2.17 (Independence) $\phi[a \mapsto \pi] \vDash p \,@\, t$ *iff* $\phi \vDash p \,@\, t$
if a *does not appear in* p *and* p *is closed*

Proposition 2.18 (Past Independence) $\phi[a \mapsto \pi] \vDash p \,@\, t$ *iff* $\phi \vDash p \,@\, t$
if $\pi(j) = \phi(a)(j)$ *for all* $j \geq \mathcal{V}(t)$ *and* p *is closed*

Proposition 2.19 (Extraction) $\phi \vDash [e/x]\,p \,@\, t$ *iff* $\phi[a \mapsto \mathcal{V}_\phi(e)] \vDash [a/x]\,p \,@\, t$
if a *does not appear in* p *and* $[e/x]\,p$ *is closed*

Proofs by induction on the structure of p.

2.3 Proof System

The *provability relation* $\vdash$ asserts that there is a proof that a particular core judgment holds. Note that provability for locality and rigidity is efficiently decidable.

A *context* Γ is a collection of hypothetical judgments that weaken provability. For example, $a : +_{\mathsf{r}} \vdash [a/x]\,p \,@\, t$ asserts that it is provable that $[a/x]\,p$ holds at time t, assuming that a is rigid. An environment satisfies a context ($\phi \vDash \Gamma$) when it satisfies each judgment in the context (the context must be closed).

Context satisfaction is defined as follows:

$$\begin{array}{lll} \phi \vDash \cdot & & \\ \phi \vDash \Gamma, J & \text{iff} & \phi \vDash \Gamma \text{ and } \phi \vDash J \end{array}$$

It has the following property:

Proposition 2.20 (Independence) $\phi[a \mapsto \pi] \vDash \Gamma$ *iff* $\phi \vDash \Gamma$
if a *does not appear in* Γ *and* Γ *is closed*

Proof by induction on the structure of Γ.

We now present our proof system.

2.4 Inference Rules

The hypothesis rule lets us use a hypothesis as a conclusion in a derivation:

$$\frac{}{\Gamma_1,\, J,\, \Gamma_2 \vdash J}\;\mathsf{hyp}$$

$$\frac{}{\Gamma \vdash t \geq t} \geq \mathsf{iref} \quad \frac{\Gamma \vdash t_1 \geq t_2}{\Gamma \vdash t_1 + 1 \geq t_2} \geq \mathsf{i} \quad \frac{\Gamma \vdash t' \geq t_0 \quad \Gamma \vdash J \quad \Gamma, t \geq t_0, [t/t_0]\, J \vdash [t+1/t_0]\, J}{\Gamma \vdash [t'/t_0]\, J} \geq \mathsf{e}^t$$

$$\frac{\Gamma \vdash t_1 \geq t_2 \quad \Gamma \vdash t_2 \geq t_3}{\Gamma \vdash t_1 \geq t_3} \mathsf{trans}_{\geq} \quad \frac{\Gamma \vdash t_1 \geq t_2 \quad \Gamma \vdash t_2 \geq t_1 + 1}{\Gamma \vdash J} \mathsf{asym}_{\geq}$$

$$\frac{\Gamma, t_1 \geq t_2 \vdash J \quad \Gamma, t_2 \geq t_1 + 1 \vdash J}{\Gamma \vdash J} \mathsf{lin}_{\geq}$$

Figure 4: Inference Rules (Time)

$$\frac{}{\Gamma \vdash p :+_{\mathsf{l}} (a)} \mathsf{ix}_{\mathsf{l}}^{a \notin p} \quad \frac{}{\Gamma \vdash R(e_1, \ldots, e_k) :+_{\mathsf{l}} (a)} R\mathsf{i}_{\mathsf{l}}$$

$$\frac{\Gamma \vdash p_1 :+_{\mathsf{l}} (a) \quad \Gamma \vdash p_2 :+_{\mathsf{l}} (a)}{\Gamma \vdash p_1 \wedge p_2 :+_{\mathsf{l}} (a)} \wedge\mathsf{i}_{\mathsf{l}} \quad \frac{\Gamma \vdash p_1 :+_{\mathsf{l}} (a) \quad \Gamma \vdash p_2 :+_{\mathsf{l}} (a)}{\Gamma \vdash p_1 \vee p_2 :+_{\mathsf{l}} (a)} \vee\mathsf{i}_{\mathsf{l}} \quad \frac{\Gamma \vdash p_1 :+_{\mathsf{l}} (a) \quad \Gamma \vdash p_2 :+_{\mathsf{l}} (a)}{\Gamma \vdash p_1 \supset p_2 :+_{\mathsf{l}} (a)} \supset \mathsf{i}_{\mathsf{l}}$$

$$\frac{\Gamma \vdash [a'/x]\, p :+_{\mathsf{l}} (a)}{\Gamma \vdash \forall x : \rho.\ p :+_{\mathsf{l}} (a)} \forall\mathsf{i}_{\mathsf{l}}^{a'} \quad \frac{\Gamma \vdash [a'/x]\, p :+_{\mathsf{l}} (a)}{\Gamma \vdash \exists x : \rho.\ p :+_{\mathsf{l}} (a)} \exists\mathsf{i}_{\mathsf{l}}^{a'}$$

$$\frac{\Gamma \vdash [a/x]\, p :+_{\mathsf{l}} (a) \quad \Gamma \vdash e = e' \circledast t \quad \Gamma \vdash [e/x]\, p \circledast t}{\Gamma \vdash [e'/x]\, p \circledast t} \mathsf{e}_{\mathsf{l}}^{a}$$

Figure 5: Inference Rules (Locality)

The inference rules in Figure 4 allow us to derive judgments on time; these rules are standard properties of the natural numbers. The induction rule $\geq \mathsf{e}$ permits us to infer that a judgment holds at an arbitrary future time if it holds now, and if it is preserved at each future time step. When a parameter (or time variable) appears as a superscript of an inference-rule label, it should be understood to mean that the parameter is "fresh" (*i.e.*, it does not appear in the conclusion of the rule).

The inference rules in Figure 5 allow us to infer locality: any parameter that does not appear in the scope of a temporal operator is local. The rule e_{l} declares that equality at the current time is sufficient to perform substitutions into local positions.[3]

The inference rules in Figure 6 allow us to infer rigidity. The rule $\mathsf{flexi}_{\mathsf{r}}$ declares that all expressions are flexible (*e.g.*, rigid expressions are also flexible). The rule $f\mathsf{i}_{\mathsf{r}}$ declares that rigidity is preserved by constant functions. The rule e_{r} lets us "transport" rigid propositions through time.

The inference rules for connectives (see Figure 7) are straightforward adaptations of the standard introduction and elimination rules. Note that the rule $\vee\mathsf{e}$ does not require the time of the first premise to match the time of the conclusion; the generalization of this rule to conclusions on interval judgments can be derived from within the system. We can show that the introduction and elimination rules for these connectives are locally sound and complete by adapting the standard reductions and expansions [Pfe99].

We can adapt the standard introduction and elimination rules for quantifiers by explicitly considering the rigidity of the appropriate parameter (see Figure 7).

[3] Note that this rule is unsound if the logic is extended to include next-time expressions [MP91].

$$\frac{}{\Gamma \vdash e : -_{\mathsf{r}}\,(\alpha)}\ -_{\mathsf{r}}\mathsf{i} \qquad \frac{}{\Gamma \vdash a : \rho\,(a)}\ \mathsf{ip_r} \qquad \frac{\Gamma \vdash e : \rho\,(\alpha)}{\Gamma \vdash e : \rho\,(\alpha_1, \alpha, \alpha_2)}\ \mathsf{weak_r}$$

$$\frac{\Gamma \vdash e_1 : \rho\,(\alpha) \quad \ldots \quad \Gamma \vdash e_k : \rho\,(\alpha)}{\Gamma \vdash f(e_1, \ldots, e_k) : \rho\,(\alpha)}\ f\mathsf{i_r} \qquad \frac{\Gamma \vdash e_1 : \rho\,(\alpha) \quad \ldots \quad \Gamma \vdash e_k : \rho\,(\alpha)}{\Gamma \vdash R(e_1, \ldots, e_k) : \rho\,(\alpha)}\ R\mathsf{i_r}$$

$$\frac{\Gamma \vdash p_1 : \rho\,(\alpha) \quad \Gamma \vdash p_2 : \rho\,(\alpha)}{\Gamma \vdash p_1 \wedge p_2 : \rho\,(\alpha)}\ \wedge\mathsf{i_r} \qquad \frac{\Gamma \vdash p_1 : \rho\,(\alpha) \quad \Gamma \vdash p_2 : \rho\,(\alpha)}{\Gamma \vdash p_1 \vee p_2 : \rho\,(\alpha)}\ \vee\mathsf{i_r} \qquad \frac{\Gamma \vdash p_1 : \rho\,(\alpha) \quad \Gamma \vdash p_2 : \rho\,(\alpha)}{\Gamma \vdash p_1 \supset p_2 : \rho\,(\alpha)}\ \supset\mathsf{i_r}$$

$$\frac{\Gamma \vdash [a/x]\,p : \rho\,(\alpha, a)}{\Gamma \vdash \forall x : \rho'.\ p : \rho\,(\alpha)}\ \forall\mathsf{i}_{\mathsf{r}}^{a} \qquad \frac{\Gamma \vdash [a/x]\,p : \rho\,(\alpha, a)}{\Gamma \vdash \exists x : \rho'.\ p : \rho\,(\alpha)}\ \exists\mathsf{i}_{\mathsf{r}}^{a}$$

$$\frac{\Gamma \vdash p : \rho\,(\alpha)}{\Gamma \vdash \Box p : \rho\,(\alpha)}\ \Box\mathsf{i_r} \qquad \frac{\Gamma \vdash p : \rho\,(\alpha)}{\Gamma \vdash \Diamond p : \rho\,(\alpha)}\ \Diamond\mathsf{i_r} \qquad \frac{\Gamma \vdash p : \rho\,(\alpha)}{\Gamma \vdash \bigcirc p : \rho\,(\alpha)}\ \bigcirc\mathsf{i_r}$$

$$\frac{\Gamma \vdash p_1 : \rho\,(\alpha) \quad \Gamma \vdash p_2 : \rho\,(\alpha)}{\Gamma \vdash p_1\,\mathcal{U}\,p_2 : \rho\,(\alpha)}\ \mathcal{U}\mathsf{i_r} \qquad \frac{\Gamma \vdash p_1 : \rho\,(\alpha) \quad \Gamma \vdash p_2 : \rho\,(\alpha)}{\Gamma \vdash p_1\,\mathcal{W}\,p_2 : \rho\,(\alpha)}\ \mathcal{W}\mathsf{i_r}$$

$$\frac{\Gamma \vdash p : +_{\mathsf{r}} \quad \Gamma \vdash p \,@\, t}{\Gamma \vdash p \,@\, t'}\ \mathsf{e_r}$$

Figure 6: Inference Rules (Rigidity)

The introduction and elimination rules for temporal operators are based on Davies [Dav96] (see Figure 7). The rule $\Box\mathsf{i}$ permits us to infer that a proposition is true at all future times if we can prove it at any arbitrary future time.[4] The rules $\Box\mathsf{e}$ and $\Diamond\mathsf{i}$ follow directly from the definitions of $\Box$ and $\Diamond$, respectively. The rule $\Diamond\mathsf{e}$ resembles $\exists\mathsf{e}$: given $\Diamond p_1 \,@\, t_1$, we can derive $p \,@\, t$ if we can derive $p \,@\, t$ under the assumption that p_1 holds at some arbitrary point in the future. The introduction and elimination rules for the temporal operators are also locally sound and complete [BPW01].

The standard equality rules are based on Necula [Nec98] (see Figure 9). The rule $\mathsf{congr}_=$ must be weakened to account for the case in which p contains temporal operators: we must show that e and e' are equal at all times; this rule complements $\mathsf{e_l}$. The rule $\mathsf{some}_=$ allows us to introduce a parameter (usually rigid) that is equal to the current value of an expression (usually flexible).

We can now show that our inference rules are sound with respect to the formal model of Section 2.2. We presume that additional domain-specific axioms (*e.g.*, the theory of natural numbers, machine operations) are valid.

Proposition 2.21 (Soundness) $\phi \vDash J$ *if* $\phi \vDash \Gamma$ *and* $\Gamma \vdash J$

Proof by induction on the derivation of $\Gamma \vdash J$.

[4] We take a small liberty here by treating meta variables such as t as time parameters—this treatment does not complicate the LF encoding.

$$\frac{\Gamma \vdash p_1 \mathbin{@} t \quad \Gamma \vdash p_2 \mathbin{@} t}{\Gamma \vdash p_1 \wedge p_2 \mathbin{@} t} \wedge\mathsf{i} \qquad \frac{\Gamma \vdash p_1 \wedge p_2 \mathbin{@} t}{\Gamma \vdash p_1 \mathbin{@} t} \wedge\mathsf{el} \qquad \frac{\Gamma \vdash p_1 \wedge p_2 \mathbin{@} t}{\Gamma \vdash p_2 \mathbin{@} t} \wedge\mathsf{er}$$

$$\frac{\Gamma \vdash p_1 \mathbin{@} t}{\Gamma \vdash p_1 \vee p_2 \mathbin{@} t} \vee\mathsf{il} \qquad \frac{\Gamma \vdash p_2 \mathbin{@} t}{\Gamma \vdash p_1 \vee p_2 \mathbin{@} t} \vee\mathsf{ir} \qquad \frac{\Gamma \vdash p_1 \vee p_2 \mathbin{@} t \quad \Gamma,\, p_1 \mathbin{@} t \vdash p' \mathbin{@} t' \quad \Gamma,\, p_2 \mathbin{@} t \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'} \vee\mathsf{e}$$

$$\frac{\Gamma,\, p_1 \mathbin{@} t \vdash p_2 \mathbin{@} t}{\Gamma \vdash p_1 \supset p_2 \mathbin{@} t} \supset\mathsf{i} \qquad \frac{\Gamma \vdash p_1 \supset p_2 \mathbin{@} t \quad \Gamma \vdash p_1 \mathbin{@} t}{\Gamma \vdash p_2 \mathbin{@} t} \supset\mathsf{e}$$

$$\frac{\Gamma,\, a\!:\!\rho \vdash [a/x]\, p \mathbin{@} t}{\Gamma \vdash \forall x\!:\!\rho.\ p \mathbin{@} t} \forall\mathsf{i}^a \qquad \frac{\Gamma \vdash \forall x\!:\!\rho.\ p \mathbin{@} t \quad \Gamma \vdash e\!:\!\rho}{\Gamma \vdash [e/x]\, p \mathbin{@} t} \forall\mathsf{e}$$

$$\frac{\Gamma \vdash e\!:\!\rho \quad \Gamma \vdash [e/x]\, p \mathbin{@} t}{\Gamma \vdash \exists x\!:\!\rho.\ p \mathbin{@} t} \exists\mathsf{i} \qquad \frac{\Gamma \vdash \exists x\!:\!\rho.\ p \mathbin{@} t \quad \Gamma,\, a\!:\!\rho,\, [a/x]\, p \mathbin{@} t \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'} \exists\mathsf{e}^a$$

$$\frac{\Gamma,\, t_1 \geq t \vdash p_1 \mathbin{@} t_1}{\Gamma \vdash \Box p_1 \mathbin{@} t} \Box\mathsf{i}^{t_1} \qquad \frac{\Gamma \vdash \Box p_1 \mathbin{@} t \quad \Gamma \vdash t_1 \geq t}{\Gamma \vdash p_1 \mathbin{@} t_1} \Box\mathsf{e}$$

$$\frac{\Gamma \vdash t_1 \geq t \quad \Gamma \vdash p_1 \mathbin{@} t_1}{\Gamma \vdash \Diamond p_1 \mathbin{@} t} \Diamond\mathsf{i} \qquad \frac{\Gamma \vdash \Diamond p_1 \mathbin{@} t \quad \Gamma,\, t_1 \geq t,\, p_1 \mathbin{@} t_1 \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'} \Diamond\mathsf{e}^{t_1}$$

$$\frac{\Gamma \vdash p \mathbin{@} t+1}{\Gamma \vdash \bigcirc p \mathbin{@} t} \bigcirc\mathsf{i} \qquad \frac{\Gamma \vdash \bigcirc p \mathbin{@} t}{\Gamma \vdash p \mathbin{@} t+1} \bigcirc\mathsf{e}$$

$$\frac{\Gamma \vdash t_2 \geq t \quad \Gamma \vdash p_1 \mathbin{@} [t, t_2) \quad \Gamma \vdash p_2 \mathbin{@} t_2}{\Gamma \vdash p_1 \,\mathcal{U}\, p_2 \mathbin{@} t} \mathcal{U}\mathsf{i} \qquad \frac{\Gamma \vdash p_1 \,\mathcal{U}\, p_2 \mathbin{@} t \quad \Gamma,\, t_2 \geq t,\, p_1 \mathbin{@} [t, t_2),\, p_2 \mathbin{@} t_2 \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'} \mathcal{U}\mathsf{e}^{t_2}$$

$$\frac{\Gamma \vdash p_1 \mathbin{@} [t, \infty)}{\Gamma \vdash p_1 \,\mathcal{W}\, p_2 \mathbin{@} t} \mathcal{W}\mathsf{i1} \qquad \frac{\Gamma \vdash t_2 \geq t \quad \Gamma \vdash p_1 \mathbin{@} [t, t_2) \quad \Gamma \vdash p_2 \mathbin{@} t_2}{\Gamma \vdash p_1 \,\mathcal{W}\, p_2 \mathbin{@} t} \mathcal{W}\mathsf{i2}$$

$$\frac{\Gamma \vdash p_1 \,\mathcal{W}\, p_2 \mathbin{@} t \quad \Gamma,\, p_1 \mathbin{@} [t, \infty) \vdash p' \mathbin{@} t' \quad \Gamma,\, t_2 \geq t,\, p_1 \mathbin{@} [t, t_2),\, p_2 \mathbin{@} t_2 \vdash p' \mathbin{@} t'}{\Gamma \vdash p' \mathbin{@} t'} \mathcal{W}\mathsf{e}^{t_2}$$

Figure 7: Inference Rules (Instants)

$$\frac{\Gamma,\, t \geq t_1,\, t_2 \geq t+1 \vdash p \mathbin{@} t}{\Gamma \vdash p \mathbin{@} [t_1, t_2)} \mathsf{ovi}^t \qquad \frac{\Gamma \vdash p \mathbin{@} [t_1, t_2) \quad \Gamma \vdash t \geq t_1 \quad \Gamma \vdash t_2 \geq t+1}{\Gamma \vdash p \mathbin{@} t} \mathsf{ove}$$

$$\frac{\Gamma,\, t \geq t_1 \vdash p \mathbin{@} t}{\Gamma \vdash p \mathbin{@} [t_1, \infty)} \mathsf{fmi}^t \qquad \frac{\Gamma \vdash p \mathbin{@} [t_1, \infty) \quad \Gamma \vdash t \geq t_1}{\Gamma \vdash p \mathbin{@} t} \mathsf{fme}$$

Figure 8: Inference Rules (Intervals)

$$\frac{}{\Gamma \vdash e = e \circledast t}\ \mathsf{ref}_{=} \qquad \frac{\Gamma \vdash e = e' \circledast t_1 \quad \Gamma \vdash [e/x]\,p \circledast t}{\Gamma \vdash [e'/x]\,p \circledast t}\ \mathsf{congr}^{t_1}_{=} \qquad \frac{\Gamma,\, a = e \circledast t_1,\, a{:}\rho \vdash p \circledast t}{\Gamma \vdash p \circledast t}\ \mathsf{some}^{a}_{=}$$

$$\frac{\Gamma,\, e_1 = e_2 \circledast t_1 \vdash p \circledast t \quad \Gamma,\, e_1 \neq e_2 \circledast t_1 \vdash p \circledast t}{\Gamma \vdash p \circledast t}\ \mathsf{case}_{=} \qquad \frac{\Gamma \vdash e_1 = e_2 \circledast t_1 \quad \Gamma \vdash e_1 \neq e_2 \circledast t_1}{\Gamma \vdash p \circledast t}\ \mathsf{contr}_{=}$$

Figure 9: Inference Rules (Equality)

3 Machine Model

We define an idealized RISC processor that will provide a foundation for the remainder of this paper. This processor operates on "words" of some fixed size (*e.g.* 32-bit numbers). There are a small number of general-purpose registers that each contain a single word, a word-sized program counter, and a memory register that contains a mapping from words to words. The processor executes a program that is simply a sequence of instructions. We assume that the program is in a separate memory and thereby protected from modification: we do not address self-modifying code in this paper.

3.1 Instruction Set

A *machine word* i is a value of type `wd`; $Val^{\mathtt{wd}}$ is an initial subrange of the natural numbers. i_{max} is the largest word. Words are inherently unsigned, but negative numbers can be simulated by signed operators using a suitable convention (*e.g.*, two's complement). A *register token* r identifies a general-purpose register; each register token $\mathtt{r}_j$ is a value of type `ureg`. We designate a small, machine-dependent subset of the total functions from pairs of words to words as *executable operators* eop (type `eop`). A *conditional operator* cop (type `cop`) is a selected unary word relation. The exact set of operators is unimportant, as long as it includes modular addition.

We use a small RISC instruction set[5]; programs are instruction sequences:

$$\begin{array}{lll} \text{Instructions} & I & ::= r_1 \leftarrow i_1 \mid r_1 \leftarrow r_2 \mid r_1 \leftarrow r_2\ eop_1\ r_3 \\ & & \quad \mid \mathtt{cond}\ cop_1\ r_1, i_1 \mid r_1 \leftarrow \mathtt{m}(r_2) \mid \mathtt{m}(r_1) \leftarrow r_2 \\ \text{Programs} & \Phi & ::= \cdot \mid I;\ \Phi \end{array}$$

An instruction I is a value of type `inst`, a program Φ is a value of type `prog`. For example, the following program replaces register $\mathtt{r}_0$ with its own factorial:

$$\begin{array}{ll} \mathtt{r}_1 \leftarrow 1 & //\ \mathtt{r}_1 \text{ is current counter} \\ \mathtt{r}_2 \leftarrow 1 & //\ \mathtt{r}_2 \text{ is current product} \\ \mathtt{r}_3 \leftarrow 1 & //\ \mathtt{r}_3 \text{ is always one} \\ \mathtt{r}_4 \leftarrow \mathtt{r}_1\ \mathtt{gtw}\ \mathtt{r}_0 & //\ \mathtt{r}_4 \text{ is nonzero iff } \mathtt{r}_1 > \mathtt{r}_0 \\ \mathtt{cond\ neq0w}\ \mathtt{r}_4, 3 & //\ \text{skip 3 when } \mathtt{r}_4 \text{ is nonzero} \\ \mathtt{r}_2 \leftarrow \mathtt{r}_2\ \mathtt{mulw}\ \mathtt{r}_1 & //\ \text{accumulate product} \\ \mathtt{r}_1 \leftarrow \mathtt{r}_1\ \mathtt{addw}\ \mathtt{r}_3 & //\ \text{increment counter} \\ \mathtt{cond\ truew}\ \mathtt{r}_0, -5 & //\ \text{always skip back 5} \\ \mathtt{r}_0 \leftarrow \mathtt{r}_2 & //\ \text{replace } \mathtt{r}_0 \\ \mathtt{halt} & \end{array}$$

[5]The instruction set does not include procedure call instructions, but it is a simple matter to add an indirect jump instruction that will support the usual RISC calling conventions; this does not complicate the enforcement mechanism.

Our calling convention starts execution at the first instruction; `halt` is an abbreviation for

$$\texttt{cond}\,\texttt{truew}\,\texttt{r}_0, -1$$

Program length ($|\Phi|$) and subscript (Φ_i) are defined in the obvious way:

$$\begin{array}{llll} |\cdot| & = 0 & (I;\,\Phi)_0 & = I \\ |I;\,\Phi| & = |\Phi| + 1 & (I;\,\Phi)_{i+1} & = \Phi_i \end{array}$$

We model a general-purpose register file as a single value of type `mapu`, mapping from register tokens to words. Memory is modeled by a total function from words to words (type `mapw`).

3.2 Syntax

We now specify how our machine model is incorporated into the logic.

The constants $0^{\texttt{wd}}$, $1^{\texttt{wd}}$, ... denote words; n is an arbitrary word constant. $\texttt{selw}^{\texttt{mapw}\times\texttt{wd}\to\texttt{wd}}$ (apply map) and $\texttt{updw}^{\texttt{mapw}\times\texttt{wd}\times\texttt{wd}\to\texttt{mapw}}$ (update map) are function constants; for example, $\texttt{updw}(\texttt{m}, 3, 4)$ denotes the same map as `m`, except that address 3 is mapped to 4. The constants $\texttt{selu}^{\texttt{mapu}\times\texttt{ureg}\to\texttt{wd}}$ (select register) and $\texttt{updu}^{\texttt{mapu}\times\texttt{ureg}\times\texttt{wd}\to\texttt{mapu}}$ (update register) operate on register files. There are no operations yielding register tokens, just designated constants (c_r).

We associate a constant $c^{\texttt{eop}}$ with each executable operator, and likewise with each conditional operator; $\texttt{addw}^{\texttt{eop}}$ denotes addition. $\texttt{appe}^{\texttt{eop}\times\texttt{wd}\times\texttt{wd}\to\texttt{wd}}$ is a function constant that applies an executable operator, and $\texttt{appc}^{\texttt{cop}\times\texttt{wd}\to o}$ is a relation constant that applies a conditional operator; we ordinarily elide these constants in the interest of readability and use infix notation for executable operators (*e.g.*, $e_1\,\texttt{addw}\,e_2$ stands for $\texttt{appe}(\texttt{addw}, e_1, e_2)$). $\texttt{compl}^{\texttt{cop}\to\texttt{cop}}$ is a function constant that complements a conditional operator (*e.g.*, $\texttt{compl}(\texttt{eq0w}) = \texttt{neq0w}$).

Identifiers for the special-purpose registers are chosen from parameters; the interpretation of these parameters is constrained by the machine model. *Reg* is the set of all register parameters (note that these are *not* register tokens). `pc` (the program counter) is a parameter of type `wd`, `u` (the contents of the register file) is a parameter of type `mapu`, and `m` (the contents of memory) is a parameter of type `mapw`. Propositions can express properties of machine states: for example, $\texttt{selu}(\texttt{u}, \texttt{r}_0) \neq 0^{\texttt{wd}}$ asserts that general-purpose register $\texttt{r}_0$ is not zero.

Our logic encompasses instructions and programs by means of constant functions. For example, $\texttt{imv}^{\texttt{ureg}\times\texttt{ureg}\to\texttt{inst}}$ constructs a move instruction from two register tokens, $\texttt{len}^{\texttt{prog}\to\texttt{wd}}$ returns the length of a program, and $\texttt{fetch}^{\texttt{prog}\times\texttt{wd}\to\texttt{inst}}$ extracts a particular instruction from a program. The logic is coupled to a particular untrusted program by means of the constant $\texttt{pm}^{\texttt{prog}}$: $\mathcal{J}(\texttt{pm})$ is the program whose first instruction is at address zero of the program memory.[6]

Intuitively, a value of type `prog` is "object code," and an expression of type `prog` is "assembly code." Instruction expressions enable us to model the operational semantics of our abstract machine directly in temporal logic (see Section 4) and are also useful for specifying security policies.

3.3 Semantics

Our operational semantics defines a set of executions for each program.

[6] Because the program code is presumably ready to be run by the code consumer, we use `pm` as a "stand in" to avoid replicating the program inside the proof. Alternatively, the program code could be stored in the proof and extracted by the code consumer after proof checking (*i.e.*, "code-carrying proof").

A *state* s maps each register to a value of its type; a state is simply a snapshot of the machine at a particular time. An *execution* σ is an infinite sequence of states representing the trace of a computation. Finite executions are represented by repeating the final state infinitely (this is the effect of the `halt` instruction).

We can turn an environment into an execution (see Section 2.2) by sampling each register at each time; $\phi|_{Reg}$ is the execution for environment ϕ:

$$\phi|_{Reg} = \sigma \text{ such that } \sigma_j = a \mapsto \phi(a)(j) \text{ for all } j \text{ and } a \in Reg$$

We call $\phi|_{Reg}$ the *erasure* of ϕ (*i.e.*, non-register parameters are "erased"). An execution σ *satisfies* a proposition p at time t ($\sigma \vDash p \,@\, t$) if all environments that erase to σ satisfy p at t:

$$\sigma \vDash p \,@\, t \text{ iff } \phi \vDash p \,@\, t \text{ for all } \phi \text{ such that } \phi|_{Reg} = \sigma$$

The *execution set* Σ_p of a proposition p is the set of executions that satisfy it at time zero ($\Sigma_p = \{\sigma \mid \sigma \vDash p \,@\, 0\}$). We can treat temporal logic as a formal security-property[7] language [BL01]: given a security-property p, an execution σ does not violate security if and only if $\sigma \in \Sigma_p$.

Now, the standard connectives of temporal logic allow us to combine security properties in a modular way: for example, $\Sigma_{p_1 \wedge p_2}$ is the intersection of Σ_{p_1} and Σ_{p_2} (*i.e.*, the program must simultaneously satisfy both p_1 and p_2). Disjunction can similarly be interpreted as the union of execution sets (*i.e.*, the code producer can choose which of two possible security properties to satisfy). Additionally, we can universally quantify a *flexible* history parameter (such as `n` in the instruction bound example from Section 1) to specify that the parameter is local to a given security property; this ensures that the parameter will not be interpreted inconsistently when the security property is combined with other security properties. We discuss security properties further in Section 4.

We now specify a transition relation between states for any given program: $\Phi \rhd s \to s'$ asserts that there is a valid transition from state s to state s' when executing program Φ (see Figure 10). $i_1 \dot{+} i_2$ abbreviates $\mathcal{J}(\texttt{addw})(i_1, i_2)$ in this figure. The notation $\psi[v_1 \mapsto v_2]$ is the redefinition of the mapping ψ such that v_1 is mapped to v_2:

$$\mathrm{dom}(\psi[v_1 \mapsto v_2]) = \mathrm{dom}\,\psi \cup \{v_1\}$$
$$(\psi[v_1 \mapsto v_2])(v_3) = \begin{cases} v_2 & \text{if } v_1 = v_3 \\ \psi(v_3) & \text{otherwise} \end{cases}$$

The execution set of a program (*i.e.*, its possible behavior) comprises all executions with valid transitions ($\Sigma_\Phi = \{\sigma \mid \Phi \rhd \sigma_j \to \sigma_{j+1} \text{ for all } j \geq 0\}$).

[7] A *security property* is a security policy that corresponds to an execution set [Sch99, AS86].

$\Phi \triangleright s \rightarrow s'$	
$\Phi_{s(\mathtt{pc})}$	s'
$r_1 \leftarrow i_1$	$s[\mathtt{pc} \mapsto s(\mathtt{pc}) \dot{+} 1][\mathtt{u} \mapsto s(\mathtt{u})[r_1 \mapsto i_1]]$
$r_1 \leftarrow r_2$	$s[\mathtt{pc} \mapsto s(\mathtt{pc}) \dot{+} 1][\mathtt{u} \mapsto s(\mathtt{u})[r_1 \mapsto s(\mathtt{u})(r_2)]]$
$r_1 \leftarrow r_2\ eop_1\ r_3$	$s[\mathtt{pc} \mapsto s(\mathtt{pc}) \dot{+} 1][\mathtt{u} \mapsto s(\mathtt{u})[r_1 \mapsto eop_1(s(\mathtt{u})(r_2), s(\mathtt{u})(r_3))]]$
$\mathtt{cond}\ cop_1\ r_1, i_1$	$\begin{cases} s[\mathtt{pc} \mapsto s(\mathtt{pc}) \dot{+} 1 \dot{+} i_1] & \text{if } s(\mathtt{u})(r_1) \in cop \\ s[\mathtt{pc} \mapsto s(\mathtt{pc}) \dot{+} 1] & \text{if } s(\mathtt{u})(r_1) \notin cop \end{cases}$
$r_1 \leftarrow \mathtt{m}(r_2)$	$s[\mathtt{pc} \mapsto s(\mathtt{pc}) \dot{+} 1][\mathtt{u} \mapsto s(\mathtt{u})[r_1 \mapsto s(\mathtt{m})(s(\mathtt{u})(r_2))]]$
$\mathtt{m}(r_1) \leftarrow r_2$	$s[\mathtt{pc} \mapsto s(\mathtt{pc}) \dot{+} 1][\mathtt{m} \mapsto s(\mathtt{m})[s(\mathtt{u})(r_1) \mapsto s(\mathtt{u})(r_2)]]$

Figure 10: The Transition Relation

Finally, we give formal interpretations of the various constants associated with our machine model:

$$
\begin{aligned}
\mathcal{J}(0^{\mathtt{wd}}) &= 0 \\
\mathcal{J}(1^{\mathtt{wd}}) &= 1 \\
&\vdots \\
\mathcal{J}(\mathtt{selw}) &= (v, i) \mapsto v(i) \\
\mathcal{J}(\mathtt{updw}) &= (v, i, i') \mapsto v[i \mapsto i'] \\
\mathcal{J}(\mathtt{selu}) &= (v, r) \mapsto v(r) \\
\mathcal{J}(\mathtt{updu}) &= (v, r, i) \mapsto v[r \mapsto i] \\
\mathcal{J}(\mathtt{addw}) &= (i_1, i_2) \mapsto (i_1 + i_2) \bmod (i_{\mathsf{max}} + 1) \\
\mathcal{J}(\mathtt{appe}) &= (eop, i_1, i_2) \mapsto eop(i_1, i_2) \\
\mathcal{J}(\mathtt{appc}) &= \{\langle cop, i \rangle \mid i \in cop\} \\
\mathcal{J}(\mathtt{compl}) &= cop \mapsto Val^{\mathtt{wd}} \setminus cop
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{J}(\mathtt{len}) &= \Phi \mapsto |\Phi| \\
\mathcal{J}(\mathtt{fetch}) &= (\Phi, i) \mapsto \Phi_i \\
\mathcal{J}(\mathtt{imvi}) &= (r_1, i_1) \mapsto (r_1 \leftarrow i_1) \\
\mathcal{J}(\mathtt{imv}) &= (r_1, r_2) \mapsto (r_1 \leftarrow r_2) \\
\mathcal{J}(\mathtt{ieop}) &= (eop, r_1, r_2, r_3) \mapsto (r_1 \leftarrow r_2\ eop\ r_3) \\
\mathcal{J}(\mathtt{icond}) &= (cop, r_1, i_1) \mapsto (\mathtt{cond}\ cop\ r_1, i_1) \\
\mathcal{J}(\mathtt{iload}) &= (r_1, r_2) \mapsto (r_1 \leftarrow \mathtt{m}(r_2)) \\
\mathcal{J}(\mathtt{istore}) &= (r_1, r_2) \mapsto (\mathtt{m}(r_1) \leftarrow r_2)
\end{aligned}
$$

4 Enforcement

We now address the code consumer's principal concern: how do I tell if my system is secure when I execute an untrusted program?

Current PCC enforcement mechanisms are implemented in the C programming language and generate a *verification condition* [Kin71] (VC) that is true only if the program does not violate the security policy; an LF type checker establishes that the security proof is a correct proof of the VC. We argue in an earlier report [BL01] that the VC generator should interpret a security policy specification instead of "hard coding" a security policy. However, temporal logic is expressive enough to encode security properties directly; we therefore do not need a special language.

For temporal-logic PCC, we provide a proof of $\vdash p_{\mathsf{sp}} \mathbin{@} 0$ instead of a proof of a VC. p_{sp} is a *security property* that must hold for the system to be secure. p_{sp} is specified by the code consumer directly; the definition of satisfaction can be used to verify that it has the intended meaning.

Contrast this approach with a first-order PCC system, in which the code producer proves a VC derived from the security property by a trusted analysis. In our system, the code producer proves the security property directly from a formal encoding of the abstract machine's transition relation. To show that our enforcement mechanism is sound, we need only show that the encoded transition relation is valid (see Section 4.2).

4.1 Encoding the Transition Relation

We provide one inference rule for each instruction type; Figure 11 specifies these rules.

In each rule, we identify the current-time values of the registers with the rigid variables xpc, xu, and xm. Then, for any program that contains an instruction of the appropriate type at the current program counter, we provide new values of the registers at the next time instant. Rigid variables name the previous-time values of the registers inside the $\bigcirc$ operator. In the case of rule trans_mv (move register), the program counter is incremented by one, and the general-purpose register r1 is assigned the value of r2 in the register file. In the case of rule trans_cond (conditional branch), a branch is taken if a conditional test succeeds; otherwise, the program counter is simply incremented; the other registers are unchanged by this instruction.

Note that the transition relation does not check that the program has proper control flow, unlike other implementations of PCC. We permit any control flow that has a valid security proof, but the security property will ordinarily require that the program counter stay within the program.

Figure 12 contains rules for inferring properties of constant expressions, and Figure 13 contains rules for the program memory. These rules have easily decidable side conditions that are verified by the proof checker. For any simple value v, $\overline{v}$ is defined as the constant c such that $\mathcal{J}(c) = v$. $\neg cop$ is an abbreviation for $Val^{\mathtt{wd}} \setminus cop$: this is the interpretation of compl.

4.2 Soundness

To show that our enforcement mechanism is sound, we first show that the encoded transition relation is valid for any execution of the untrusted program:

Proposition 4.1 (Transition Soundness) $\phi \vDash p_{\mathsf{trans\_}l} \circledcirc t$
for each $l \in \{\mathsf{mvi}, \mathsf{mv}, \mathsf{eop}, \mathsf{cond}, \mathsf{load}, \mathsf{store}\}$ *if* $\phi|_{Reg} \in \Sigma_{\mathcal{J}(\mathtt{pm})}$

We provide detailed proofs for mv and cond; the other cases are similar.

PROOF:

(case mv)

let $j = \mathcal{V}(t), \sigma = \phi|_{Reg}, s = \sigma_j, s' = \sigma_{j+1}$

$\sigma \in \Sigma_{\mathcal{J}(\mathtt{pm})}$ Prem.

$\mathcal{J}(\mathtt{pm}) \rhd s \to s'$ Def. Σ_Φ

let $a_{\mathtt{pc}}, a_{\mathtt{u}}, a_{\mathtt{m}}, a_{\mathtt{r1}}, a_{\mathtt{r2}} \notin p_{\mathsf{trans\_mv}}$

for all $\pi_{\mathtt{pc}} : +_{\mathsf{r}}, \pi_{\mathtt{u}} : +_{\mathsf{r}}, \pi_{\mathtt{m}} : +_{\mathsf{r}}, \pi_{\mathtt{r1}} : +_{\mathsf{r}}, \pi_{\mathtt{r2}} : +_{\mathsf{r}}$

let $\phi' = \phi[a_{\mathtt{pc}} \mapsto \pi_{\mathtt{pc}}][a_{\mathtt{u}} \mapsto \pi_{\mathtt{u}}][a_{\mathtt{m}} \mapsto \pi_{\mathtt{m}}][a_{\mathtt{r1}} \mapsto \pi_{\mathtt{r1}}][a_{\mathtt{r2}} \mapsto \pi_{\mathtt{r2}}]$

$\phi' \vDash a_{\mathtt{pc}} = \mathtt{pc} \wedge a_{\mathtt{u}} = \mathtt{u} \wedge a_{\mathtt{m}} = \mathtt{m} \circledcirc t$ Hyp.

$\phi' \vDash a_{\mathtt{pc}} = \mathtt{pc} \circledcirc t \quad \phi' \vDash a_{\mathtt{u}} = \mathtt{u} \circledcirc t \quad \phi' \vDash a_{\mathtt{m}} = \mathtt{m} \circledcirc t$ Def. $\vDash$

$\langle \mathcal{V}_{\phi'}(a_{\mathtt{pc}})(j), \mathcal{V}_{\phi'}(\mathtt{pc})(j)\rangle, \langle \mathcal{V}_{\phi'}(a_{\mathtt{u}})(j), \mathcal{V}_{\phi'}(\mathtt{u})(j)\rangle, \langle \mathcal{V}_{\phi'}(a_{\mathtt{m}})(j), \mathcal{V}_{\phi'}(\mathtt{m})(j)\rangle \in \mathcal{J}(=)$ Def. $\vDash$

$\mathcal{V}_{\phi'}(a_{\mathtt{pc}})(j) = \mathcal{V}_{\phi'}(\mathtt{pc})(j) \quad \mathcal{V}_{\phi'}(a_{\mathtt{u}})(j) = \mathcal{V}_{\phi'}(\mathtt{u})(j) \quad \mathcal{V}_{\phi'}(a_{\mathtt{m}})(j) = \mathcal{V}_{\phi'}(\mathtt{m})(j)$ Def. $\mathcal{J}$

$\phi'(a_{\mathtt{pc}})(j) = \phi'(\mathtt{pc})(j) \quad \phi'(a_{\mathtt{u}})(j) = \phi'(\mathtt{u})(j) \quad \phi'(a_{\mathtt{m}})(j) = \phi'(\mathtt{m})(j)$ Def. $\mathcal{V}$

$\phi' \vDash \mathtt{fetch}(\mathtt{pm}, \mathtt{pc}) = \mathtt{imv}(a_{\mathtt{r1}}, a_{\mathtt{r2}}) \circledcirc t$ Hyp.

$\langle \mathcal{V}_{\phi'}(\mathtt{fetch}(\mathtt{pm}, \mathtt{pc}))(j), \mathcal{V}_{\phi'}(\mathtt{imv}(a_{\mathtt{r1}}, a_{\mathtt{r2}}))(j)\rangle \in \mathcal{J}(=)$ Def. $\vDash$

$$p_{\mathsf{trans\_mvi}} \equiv \forall \mathtt{xpc}{:}+_{\mathsf{r}}.\ \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \forall \mathtt{r1}{:}+_{\mathsf{r}}.\ \forall \mathtt{i1}{:}+_{\mathsf{r}}.\\ \mathtt{xpc} = \mathtt{pc} \wedge \mathtt{xu} = \mathtt{u} \wedge \mathtt{xm} = \mathtt{m} \supset \mathtt{fetch}(\mathtt{pm},\mathtt{pc}) = \mathtt{imvi}(\mathtt{r1},\mathtt{i1})\\ \supset\bigcirc(\mathtt{pc} = \mathtt{xpc}\ \mathtt{addw}\ 1 \wedge \mathtt{u} = \mathtt{updu}(\mathtt{xu},\mathtt{r1},\mathtt{i1}) \wedge \mathtt{m} = \mathtt{xm})$$

$$p_{\mathsf{trans\_mv}} \equiv \forall \mathtt{xpc}{:}+_{\mathsf{r}}.\ \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \forall \mathtt{r1}{:}+_{\mathsf{r}}.\ \forall \mathtt{r2}{:}+_{\mathsf{r}}.\\ \mathtt{xpc} = \mathtt{pc} \wedge \mathtt{xu} = \mathtt{u} \wedge \mathtt{xm} = \mathtt{m} \supset \mathtt{fetch}(\mathtt{pm},\mathtt{pc}) = \mathtt{imv}(\mathtt{r1},\mathtt{r2})\\ \supset\bigcirc(\mathtt{pc} = \mathtt{xpc}\ \mathtt{addw}\ 1 \wedge \mathtt{u} = \mathtt{updu}(\mathtt{xu},\mathtt{r1},\mathtt{selu}(\mathtt{xu},\mathtt{r2})) \wedge \mathtt{m} = \mathtt{xm})$$

$$p_{\mathsf{trans\_eop}} \equiv \forall \mathtt{xpc}{:}+_{\mathsf{r}}.\ \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \forall \mathtt{eop1}{:}+_{\mathsf{r}}.\ \forall \mathtt{r1}{:}+_{\mathsf{r}}.\ \forall \mathtt{r2}{:}+_{\mathsf{r}}.\ \forall \mathtt{r3}{:}+_{\mathsf{r}}.\\ \mathtt{xpc} = \mathtt{pc} \wedge \mathtt{xu} = \mathtt{u} \wedge \mathtt{xm} = \mathtt{m} \supset \mathtt{fetch}(\mathtt{pm},\mathtt{pc}) = \mathtt{ieop}(\mathtt{eop1},\mathtt{r1},\mathtt{r2},\mathtt{r3})\\ \supset\bigcirc(\mathtt{pc} = \mathtt{xpc}\ \mathtt{addw}\ 1 \wedge \mathtt{u} = \mathtt{updu}(\mathtt{xu},\mathtt{r1},\mathtt{selu}(\mathtt{xu},\mathtt{r2})\ \mathtt{eop1}\ \mathtt{selu}(\mathtt{xu},\mathtt{r3})) \wedge \mathtt{m} = \mathtt{xm})$$

$$p_{\mathsf{trans\_cond}} \equiv \forall \mathtt{xpc}{:}+_{\mathsf{r}}.\ \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \forall \mathtt{cop1}{:}+_{\mathsf{r}}.\ \forall \mathtt{r1}{:}+_{\mathsf{r}}.\ \forall \mathtt{i1}{:}+_{\mathsf{r}}.\\ \mathtt{xpc} = \mathtt{pc} \wedge \mathtt{xu} = \mathtt{u} \wedge \mathtt{xm} = \mathtt{m} \supset \mathtt{fetch}(\mathtt{pm},\mathtt{pc}) = \mathtt{icond}(\mathtt{cop1},\mathtt{r1},\mathtt{i1})\\ \supset\bigcirc\begin{pmatrix} \ (\mathtt{cop1}(\mathtt{selu}(\mathtt{xu},\mathtt{r1})) \supset \mathtt{pc} = \mathtt{xpc}\ \mathtt{addw}\ 1\ \mathtt{addw}\ \mathtt{i1}) \\ \wedge((\mathtt{compl}(\mathtt{cop1}))(\mathtt{selu}(\mathtt{xu},\mathtt{r1})) \supset \mathtt{pc} = \mathtt{xpc}\ \mathtt{addw}\ 1) \\ \wedge \mathtt{u} = \mathtt{xu} \wedge \mathtt{m} = \mathtt{xm} \end{pmatrix}$$

$$p_{\mathsf{trans\_load}} \equiv \forall \mathtt{xpc}{:}+_{\mathsf{r}}.\ \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \forall \mathtt{r1}{:}+_{\mathsf{r}}.\ \forall \mathtt{r2}{:}+_{\mathsf{r}}.\\ \mathtt{xpc} = \mathtt{pc} \wedge \mathtt{xu} = \mathtt{u} \wedge \mathtt{xm} = \mathtt{m} \supset \mathtt{fetch}(\mathtt{pm},\mathtt{pc}) = \mathtt{iload}(\mathtt{r1},\mathtt{r2})\\ \supset\bigcirc(\mathtt{pc} = \mathtt{xpc}\ \mathtt{addw}\ 1 \wedge \mathtt{u} = \mathtt{updu}(\mathtt{xu},\mathtt{r1},\mathtt{selw}(\mathtt{xm},\mathtt{selu}(\mathtt{xu},\mathtt{r2}))) \wedge \mathtt{m} = \mathtt{xm})$$

$$p_{\mathsf{trans\_store}} \equiv \forall \mathtt{xpc}{:}+_{\mathsf{r}}.\ \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \forall \mathtt{r1}{:}+_{\mathsf{r}}.\ \forall \mathtt{r2}{:}+_{\mathsf{r}}.\\ \mathtt{xpc} = \mathtt{pc} \wedge \mathtt{xu} = \mathtt{u} \wedge \mathtt{xm} = \mathtt{m} \supset \mathtt{fetch}(\mathtt{pm},\mathtt{pc}) = \mathtt{istore}(\mathtt{r1},\mathtt{r2})\\ \supset\bigcirc(\mathtt{pc} = \mathtt{xpc}\ \mathtt{addw}\ 1 \wedge \mathtt{u} = \mathtt{xu} \wedge \mathtt{m} = \mathtt{updw}(\mathtt{xm},\mathtt{selu}(\mathtt{xu},\mathtt{r1}),\mathtt{selu}(\mathtt{xu},\mathtt{r2})))$$

$$\frac{}{\Gamma \vdash p_{\mathsf{trans\_mvi}} \mathbin{@} t}\ \mathsf{trans\_mvi} \qquad \frac{}{\Gamma \vdash p_{\mathsf{trans\_mv}} \mathbin{@} t}\ \mathsf{trans\_mv} \qquad \frac{}{\Gamma \vdash p_{\mathsf{trans\_eop}} \mathbin{@} t}\ \mathsf{trans\_eop}$$

$$\frac{}{\Gamma \vdash p_{\mathsf{trans\_cond}} \mathbin{@} t}\ \mathsf{trans\_cond} \qquad \frac{}{\Gamma \vdash p_{\mathsf{trans\_load}} \mathbin{@} t}\ \mathsf{trans\_load} \qquad \frac{}{\Gamma \vdash p_{\mathsf{trans\_store}} \mathbin{@} t}\ \mathsf{trans\_store}$$

Figure 11: Encoding the Transition Relation

$$\frac{}{\Gamma \vdash \overline{i_1}\ \overline{eop}\ \overline{i_2} = \overline{i'} \mathbin{@} t}\ \mathsf{const\_eop}\ \text{if } i' = eop(i_1, i_2) \qquad \frac{}{\Gamma \vdash \overline{cop}(\overline{i}) \mathbin{@} t}\ \mathsf{const\_cop}\ \text{if } i \in cop$$

$$\frac{}{\Gamma \vdash \mathtt{compl}(\overline{cop}) = \overline{cop'} \mathbin{@} t}\ \mathsf{const\_compl}\ \text{if } cop' = \neg cop$$

Figure 12: Rules for Constant Expressions

$$\frac{}{\Gamma \vdash \texttt{len(pm)} = \bar{i} \circledast t}\ \textsf{pm\_len} \quad \text{if } |\mathcal{J}(\texttt{pm})| = i$$

$$\frac{}{\Gamma \vdash \texttt{fetch}(\texttt{pm}, \bar{i}) = \texttt{imvi}(\overline{r_1}, \overline{i_1}) \circledast t}\ \textsf{pm\_mvi} \quad \text{if } \mathcal{J}(\texttt{pm})_i = (r_1 \leftarrow i_1)$$

$$\frac{}{\Gamma \vdash \texttt{fetch}(\texttt{pm}, \bar{i}) = \texttt{imv}(\overline{r_1}, \overline{r_2}) \circledast t}\ \textsf{pm\_mv} \quad \text{if } \mathcal{J}(\texttt{pm})_i = (r_1 \leftarrow r_2)$$

$$\frac{}{\Gamma \vdash \texttt{fetch}(\texttt{pm}, \bar{i}) = \texttt{ieop}(\overline{eop_1}, \overline{r_1}, \overline{r_2}, \overline{r_3}) \circledast t}\ \textsf{pm\_eop} \quad \text{if } \mathcal{J}(\texttt{pm})_i = (r_1 \leftarrow r_2\ eop_1\ r_3)$$

$$\frac{}{\Gamma \vdash \texttt{fetch}(\texttt{pm}, \bar{i}) = \texttt{icond}(\overline{cop_1}, \overline{r_1}, \overline{i_1}) \circledast t}\ \textsf{pm\_cond} \quad \text{if } \mathcal{J}(\texttt{pm})_i = (\texttt{cond}\ cop_1\ r_1, i_1)$$

$$\frac{}{\Gamma \vdash \texttt{fetch}(\texttt{pm}, \bar{i}) = \texttt{iload}(\overline{r_1}, \overline{r_2}) \circledast t}\ \textsf{pm\_load} \quad \text{if } \mathcal{J}(\texttt{pm})_i = (r_1 \leftarrow \texttt{m}(r_2))$$

$$\frac{}{\Gamma \vdash \texttt{fetch}(\texttt{pm}, \bar{i}) = \texttt{istore}(\overline{r_1}, \overline{r_2}) \circledast t}\ \textsf{pm\_store} \quad \text{if } \mathcal{J}(\texttt{pm})_i = (\texttt{m}(r_1) \leftarrow r_2)$$

Figure 13: Rules for the Program Memory

$\mathcal{V}_{\phi'}(\texttt{fetch}(\texttt{pm}, \texttt{pc}))(j) = \mathcal{V}_{\phi'}(\texttt{imv}(a_{\texttt{r1}}, a_{\texttt{r2}}))(j)$	Def. $\mathcal{J}$
$\mathcal{J}(\texttt{fetch})(\mathcal{V}_{\phi'}(\texttt{pm})(j), \mathcal{V}_{\phi'}(\texttt{pc})(j)) = \mathcal{J}(\texttt{imv})(\mathcal{V}_{\phi'}(a_{\texttt{r1}})(j), \mathcal{V}_{\phi'}(a_{\texttt{r2}})(j))$	Def. $\mathcal{V}$
$\mathcal{J}(\texttt{fetch})(\mathcal{J}(\texttt{pm}), \phi(\texttt{pc})(j)) = \mathcal{J}(\texttt{imv})(\pi_{\texttt{r1}}(j), \pi_{\texttt{r2}}(j))$	Def. $\mathcal{V}$
let $r_1 = \pi_{\texttt{r1}}(j), r_2 = \pi_{\texttt{r2}}(j)$	
$\mathcal{J}(\texttt{pm})_{\phi(\texttt{pc})(j)} = (r_1 \leftarrow r_2)$	Def. $\mathcal{J}$
$\mathcal{J}(\texttt{pm})_{s(\texttt{pc})} = (r_1 \leftarrow r_2)$	Def. $\phi\vert_{Reg}$
$s' = s[\texttt{pc} \mapsto s(\texttt{pc}) \dot{+} 1][\texttt{u} \mapsto s(\texttt{u})[r_1 \mapsto s(\texttt{u})(r_2)]]$	Def. $\Phi \rhd s \to s'$
$\mathcal{V}_{\phi'}(\texttt{pc})(j+1) = \phi'(\texttt{pc})(j+1)$	Def. $\mathcal{V}$
$= \phi(\texttt{pc})(j+1)$	
$= s'(\texttt{pc})$	Def. $\phi\vert_{Reg}$
$= s(\texttt{pc}) \dot{+} 1$	
$= \phi(\texttt{pc})(j) \dot{+} 1$	Def. $\phi\vert_{Reg}$
$= \phi'(\texttt{pc})(j) \dot{+} 1 = \phi'(a_{\texttt{pc}})(j) \dot{+} 1 = \pi_{\texttt{pc}}(j) \dot{+} 1$	
$= \pi_{\texttt{pc}}(j+1) \dot{+} 1$	Def. $\pi : +_{\mathsf{r}}$
$= \phi'(a_{\texttt{pc}})(j+1) \dot{+} 1$	
$= \mathcal{V}_{\phi'}(a_{\texttt{pc}})(j+1) \dot{+} \mathcal{V}_{\phi'}(1)(j+1)$	Def. $\mathcal{V}$
$= \mathcal{V}_{\phi'}(a_{\texttt{pc}}\ \texttt{addw}\ 1)(j+1)$	Def. $\mathcal{V}$
$\mathcal{V}_{\phi'}(\texttt{u})(j+1) = \mathcal{V}_{\phi'}(\texttt{updu}(a_{\texttt{u}}, a_{\texttt{r1}}, \texttt{selu}(a_{\texttt{u}}, a_{\texttt{r2}})))(j+1)$	similar
$\mathcal{V}_{\phi'}(\texttt{m})(j+1) = \mathcal{V}_{\phi'}(a_{\texttt{m}})(j+1)$	similar
$\phi' \vDash \texttt{pc} = a_{\texttt{pc}}\ \texttt{addw}\ 1 \circledast t+1$	Def. $\mathcal{J}, \vDash$
$\phi' \vDash \texttt{u} = \texttt{updu}(a_{\texttt{u}}, a_{\texttt{r1}}, \texttt{selu}(a_{\texttt{u}}, a_{\texttt{r2}})) \circledast t+1$	Def. $\mathcal{J}, \vDash$
$\phi' \vDash \texttt{m} = a_{\texttt{m}} \circledast t+1$	Def. $\mathcal{J}, \vDash$
let $p' \equiv \texttt{pc} = a_{\texttt{pc}}\ \texttt{addw}\ 1 \wedge \texttt{u} = \texttt{updu}(a_{\texttt{u}}, a_{\texttt{r1}}, \texttt{selu}(a_{\texttt{u}}, a_{\texttt{r2}})) \wedge \texttt{m} = a_{\texttt{m}}$	
$\phi' \vDash \bigcirc p' \circledast t$	Def. $\vDash$
$\phi' \vDash a_{\texttt{pc}} = \texttt{pc} \wedge a_{\texttt{u}} = \texttt{u} \wedge a_{\texttt{m}} = \texttt{m} \supset \texttt{fetch}(\texttt{pm}, \texttt{pc}) = \texttt{imv}(a_{\texttt{r1}}, a_{\texttt{r2}}) \supset \bigcirc p' \circledast t$	Def. $\vDash$
$\phi \vDash p_{\textsf{trans\_mv}} \circledast t$	Def. $\vDash$

□

PROOF:

(case cond)

let $j = \mathcal{V}(t), \sigma = \phi|_{Reg}, s = \sigma_j, s' = \sigma_{j+1}$

$\sigma \in \Sigma_{\mathcal{J}(\mathtt{pm})}$ Prem.

$\mathcal{J}(\mathtt{pm}) \triangleright s \to s'$ Def. Σ_Φ

let $a_{\mathtt{pc}}, a_{\mathtt{u}}, a_{\mathtt{m}}, a_{\mathtt{cop1}}, a_{\mathtt{r1}}, a_{\mathtt{i1}} \notin p_{\mathsf{trans\_cond}}$

for all $\pi_{\mathtt{pc}} : +_{\mathsf{r}}, \pi_{\mathtt{u}} : +_{\mathsf{r}}, \pi_{\mathtt{m}} : +_{\mathsf{r}}, \pi_{\mathtt{cop1}} : +_{\mathsf{r}}, \pi_{\mathtt{r1}} : +_{\mathsf{r}}, \pi_{\mathtt{i1}} : +_{\mathsf{r}}$

let $\phi' = \phi[a_{\mathtt{pc}} \mapsto \pi_{\mathtt{pc}}][a_{\mathtt{u}} \mapsto \pi_{\mathtt{u}}][a_{\mathtt{m}} \mapsto \pi_{\mathtt{m}}][a_{\mathtt{cop1}} \mapsto \pi_{\mathtt{cop1}}][a_{\mathtt{r1}} \mapsto \pi_{\mathtt{r1}}][a_{\mathtt{i1}} \mapsto \pi_{\mathtt{i1}}]$

$\phi' \models a_{\mathtt{pc}} = \mathtt{pc} \wedge a_{\mathtt{u}} = \mathtt{u} \wedge a_{\mathtt{m}} = \mathtt{m} \circledast t$ Hyp.

$\phi'(a_{\mathtt{pc}})(j) = \phi'(\mathtt{pc})(j) \quad \phi'(a_{\mathtt{u}})(j) = \phi'(\mathtt{u})(j) \quad \phi'(a_{\mathtt{m}})(j) = \phi'(\mathtt{m})(j)$ see mv proof

$\phi' \models \mathtt{fetch}(\mathtt{pm}, \mathtt{pc}) = \mathtt{icond}(a_{\mathtt{cop1}}, a_{\mathtt{r1}}, a_{\mathtt{i1}}) \circledast t$ Hyp.

let $cop_1 = \pi_{\mathtt{cop1}}(j), r_1 = \pi_{\mathtt{r1}}(j), i_1 = \pi_{\mathtt{i1}}(j)$

$\mathcal{J}(\mathtt{pm})_{s(\mathtt{pc})} = (\mathtt{cond}\; cop_1\; r_1, i_1)$ see mv proof

$\phi' \models a_{\mathtt{cop1}}(\mathtt{selu}(a_{\mathtt{u}}, a_{\mathtt{r1}})) \circledast t+1$ Hyp.

$\langle \mathcal{V}_{\phi'}(a_{\mathtt{cop1}})(j+1), \mathcal{V}_{\phi'}(\mathtt{selu}(a_{\mathtt{u}}, a_{\mathtt{r1}}))(j+1)\rangle \in \mathcal{J}(\mathtt{appc})$ Def. $\models$

$\langle \mathcal{V}_{\phi'}(a_{\mathtt{cop1}})(j+1), \mathcal{J}(\mathtt{selu})(\mathcal{V}_{\phi'}(a_{\mathtt{u}})(j+1), \mathcal{V}_{\phi'}(a_{\mathtt{r1}})(j+1))\rangle \in \mathcal{J}(\mathtt{appc})$ Def. $\mathcal{V}$

$\langle \pi_{\mathtt{cop1}}(j+1), \mathcal{J}(\mathtt{selu})(\pi_{\mathtt{u}}(j+1), \pi_{\mathtt{r1}}(j+1))\rangle \in \mathcal{J}(\mathtt{appc})$ Def. $\mathcal{V}$

$\langle cop_1, \mathcal{J}(\mathtt{selu})(\pi_{\mathtt{u}}(j), r_1)\rangle \in \mathcal{J}(\mathtt{appc})$ Def. $\pi : +_{\mathsf{r}}$

$\mathcal{J}(\mathtt{selu})(\pi_{\mathtt{u}}(j), r_1) \in cop_1$ Def. $\mathcal{J}$

$\pi_{\mathtt{u}}(j)(r_1) \in cop_1$ Def. $\mathcal{J}$

$\pi_{\mathtt{u}}(j) = \phi'(a_{\mathtt{u}})(j) = \phi'(\mathtt{u})(j) = \phi(\mathtt{u})(j)$

$s(\mathtt{u})(r_1) \in cop_1$ Def. $\phi|_{Reg}$

$s' = s[\mathtt{pc} \mapsto s(\mathtt{pc}) \dot{+} 1 \dot{+} i_1]$ Def. $\Phi \triangleright s \to s'$

$\mathcal{V}_{\phi'}(\mathtt{pc})(j+1) = \mathcal{V}_{\phi'}(a_{\mathtt{pc}}\; \mathtt{addw}\; 1\; \mathtt{addw}\; a_{\mathtt{i1}})(j+1)$ see mv proof

$\phi' \models \mathtt{pc} = a_{\mathtt{pc}}\; \mathtt{addw}\; 1\; \mathtt{addw}\; a_{\mathtt{i1}} \circledast t+1$ Def. $\mathcal{J}, \models$

$\phi' \models a_{\mathtt{cop1}}(\mathtt{selu}(a_{\mathtt{u}}, a_{\mathtt{r1}})) \supset \mathtt{pc} = a_{\mathtt{pc}}\; \mathtt{addw}\; 1\; \mathtt{addw}\; a_{\mathtt{i1}} \circledast t+1$ Def. $\models$

$\phi' \models (\mathtt{compl}(a_{\mathtt{cop1}}))(\mathtt{selu}(a_{\mathtt{u}}, a_{\mathtt{r1}})) \supset \mathtt{pc} = a_{\mathtt{pc}}\; \mathtt{addw}\; 1 \circledast t+1$ similar

$\phi' \models \mathtt{u} = a_{\mathtt{u}} \circledast t+1$ see mv proof

$\phi' \models \mathtt{m} = a_{\mathtt{m}} \circledast t+1$ see mv proof

let $p' \equiv \quad (a_{\mathtt{cop1}}(\mathtt{selu}(a_{\mathtt{u}}, a_{\mathtt{r1}})) \supset \mathtt{pc} = a_{\mathtt{pc}}\; \mathtt{addw}\; 1\; \mathtt{addw}\; a_{\mathtt{i1}})$
$\quad \wedge((\mathtt{compl}(a_{\mathtt{cop1}}))(\mathtt{selu}(a_{\mathtt{u}}, a_{\mathtt{r1}})) \supset \mathtt{pc} = a_{\mathtt{pc}}\; \mathtt{addw}\; 1)$
$\quad \wedge \mathtt{u} = a_{\mathtt{u}} \wedge \mathtt{m} = a_{\mathtt{m}}$

$\phi' \models \bigcirc p' \circledast t$ Def. $\models$

$\phi' \models a_{\mathtt{pc}} = \mathtt{pc} \wedge a_{\mathtt{u}} = \mathtt{u} \wedge a_{\mathtt{m}} = \mathtt{m} \supset \mathtt{fetch}(\mathtt{pm}, \mathtt{pc}) = \mathtt{icond}(a_{\mathtt{cop1}}, a_{\mathtt{r1}}, a_{\mathtt{i1}}) \supset \bigcirc p' \circledast t$ Def. $\models$

$\phi \models p_{\mathsf{trans\_cond}} \circledast t$ Def. $\models$

□

Now, let p_{sp} be a security property. The following proposition establishes that the system is secure with respect to any program that has a security proof:

Proposition 4.2 (Enforcement Soundness) $\Sigma_{\mathcal{J}(\mathtt{pm})} \subseteq \Sigma_{p_{\mathsf{sp}}}$ *if* $\vdash p_{\mathsf{sp}} \circledast 0$

PROOF:

for all $\sigma \in \Sigma_{\mathcal{J}(\mathtt{pm})}$

for all ϕ such that $\phi|_{Reg} = \sigma$

$\phi \models p_{\mathsf{sp}} \circledast 0$ Proposition 4.1 and Proposition 2.21

$\sigma \vDash p_{\mathsf{sp}} \circledast 0$ Def. $\sigma \vDash p \circledast t$
$\sigma \in \Sigma_{p_{\mathsf{sp}}}$ Def. Σ_p
$\square$

Let $\Phi = \mathcal{J}(\mathtt{pm})$. The code producer provides a derivation of $\vdash p_{\mathsf{sp}} \circledast 0$ along with Φ; we use a trusted proof checker (*e.g*, Necula [Nec98]) to verify its correctness. From Proposition 4.2, we conclude $\Sigma_\Phi \subseteq \Sigma_{p_{\mathsf{sp}}}$: no execution of Φ violates p_{sp}.

5 Certification

We now address the code producer's principal concern: how do I generate a security proof for my program such that it will satisfy the code consumer?

Of course, as a last resort, the code producer can always write proofs by hand, but this approach is feasible only for small programs. Practical systems for PCC rely on a certifying compiler [Nec98] (a *certification mechanism*) to produce a security proof in the normal course of compiling a program. We would like to have temporal-logic certifying compilers.

Unfortunately, certification appears to be significantly harder than enforcement: existing certifying compilers [CLN+00, Nec98, MWCG98] provide proofs of type safety only for relatively standard type systems. In this section, we restrict our attention to programs without procedure calls and provide an algorithm for transforming the output of a first-order PCC compiler into a temporal-logic proof of type safety. This limits our choice of security policies, but note that type safety is an essential starting point for any practical PCC system, and that type systems exist for many "expressive" security policies [Wal00, CW00, CWM99].

Our certification mechanism generates derivations of judgments of the form

$$\vdash \mathtt{pc} = 0 \wedge p_{\mathsf{pre}} \supset \square p_{\mathsf{safe}} \circledast 0$$

where p_{pre} and p_{safe} are assertions ; an *assertion* is a proposition that contains no temporal operators. This class of security properties represents a slight generalization of the *invariance properties* [MP91], and includes all type safety properties. Intuitively, an invariance property requires us to prove that some assertion (*i.e.*, p_{safe}) holds at all times. We generalize this class by allowing the code producer to assume that the program counter is zero and that a precondition assertion (*i.e.*, p_{pre}) holds at the start of execution.

In addition to object code, existing certifying compilers for PCC produce a set of loop invariants and a proof of a first-order VC. A *loop invariant* is an assertion that holds at the head of each loop; a complete set of loop invariants ensures that the VC generator will terminate, even if the program does not. For temporal-logic PCC, we pass the object code, loop invariants, and first-order proof to an *ad hoc* proof generation algorithm that produces a temporal-logic security proof. The *ad hoc* proof generator mimics the operation of the VC generator; both are untrusted components in our system.

In order to obtain efficient temporal-logic proofs, we factor fixed sequences of inferences into derived rules that are introduced by the *prelude* of the proof. The prelude is identical for all programs compiled by the same compiler, and is thus a constant overhead. We call the temporal-logic component of the security proof a *proof skeleton*. The proof skeleton is constructed by the application of derived rules; the derivations of the derived rules (in the prelude) are first checked by the proof checker. The "leaves" of the original first-order proof are embedded in the temporal proof skeleton, after purely structural rules are stripped away.

5.1 VC Generation

We first adapt Necula's VC generator [Nec97] to our machine model to fix the strategy of our proof generator (see Figure 14) .

For certifying control-flow safety and memory safety, p_{safe} is

$$\begin{array}{l} \mathtt{neq0}(\mathtt{len}(\mathtt{pm})\ \mathtt{gtu}\ \mathtt{pc}) \\ \wedge(\forall \mathtt{r1}{:}+_{\mathsf{r}}.\ \forall \mathtt{r2}{:}+_{\mathsf{r}}.\ \mathtt{fetch}(\mathtt{pm},\mathtt{pc}) = \mathtt{iload}(\mathtt{r1},\mathtt{r2}) \supset \mathtt{saferd}(\mathtt{m},\mathtt{selu}(\mathtt{u},\mathtt{r2}))) \\ \wedge(\forall \mathtt{r1}{:}+_{\mathsf{r}}.\ \forall \mathtt{r2}{:}+_{\mathsf{r}}.\ \mathtt{fetch}(\mathtt{pm},\mathtt{pc}) = \mathtt{istore}(\mathtt{r1},\mathtt{r2}) \\ \qquad\qquad\qquad\qquad \supset \mathtt{safewr}(\mathtt{m},\mathtt{selu}(\mathtt{u},\mathtt{r1}),\mathtt{selu}(\mathtt{u},\mathtt{r2}))) \end{array}$$

We call this the *essential safety policy* [Koz98]. It allows the program counter to range over the entire program. The constants `saferd` and `safewr` denote arbitrary relations that encode the memory safety policy [Nec98]; the VC proves that these relations hold for each possible program state.

Let $VC_{p_{\mathsf{pre}},\mathcal{I}}$ be the VC for program $\mathcal{J}(\mathtt{pm})$, precondition p_{pre}, and loop invariants $\mathcal{I}$. The certifying compiler produces $\mathcal{I}$ along with a proof of $\vdash VC_{p_{\mathsf{pre}},\mathcal{I}} @ 0$.

$\mathcal{I}$ is a partial function from words (addresses) to propositions: if $i \in \mathrm{dom}\,\mathcal{I}$, then $\mathcal{I}(i)$ is the loop invariant for address i. Typically, $i \in \mathrm{dom}\,\mathcal{I}$ if there is any backward branch to i. The three registers are replaced by the variables `xpc`, `xu`, and `xm` respectively in $\mathcal{I}(i)$ and p_{pre} in order to simplify the VC generator.

$VC_{p_{\mathsf{pre}},\mathcal{I}}$ is derived by symbolically executing the program code on the variables `xu` and `xm`. The symbolic evaluator reaches a loop invariant after executing some finite number of instructions; it checks that this invariant holds with the new symbolic register values. The left conjunct of *VC* checks that we get to state where a loop invariant holds if we start from a state satisfying the precondition. The right conjunct of *VC* checks that each loop invariant leads to another loop invariant.

The function *SE* checks for a loop invariant at the current instruction and substitutes the current symbolic state into the invariant, if one is found. The function *SESafe* checks that the current symbolic state does not violate the safety policy and proceeds with the next instruction. The function *SENext* iterates the current symbolic state according to the current instruction. We use *SESafe* instead of *SE* in the right conjunct of *VC* to ensure that at least one instruction is checked before reaching a new loop invariant.

5.2 Proof Generation

The proof generator extends first-order proofs to temporal invariance proofs by mimicking the operation of the VC generator in temporal logic. In effect, the proof skeleton is a trace of a particular run of the VC generator encoded in the language of temporal logic. The proof of control-flow safety is encoded in the proof skeleton itself; other properties are demonstrated by the first-order proof. Our proof generator is not a search algorithm: given a well-formed first-order proof, a temporal proof is always found in time directly proportional to the size of the VC. Note that because our enforcement mechanism does not depend on the VC generator, we are free to change VC generators at any time, even after the enforcement mechanism has been widely deployed.

It should not be surprising that we can reduce temporal invariance proofs to first-order proofs, because this is a well-known technique for verifying reactive systems [MP91]. However, instead of using the usual *general invariance rule* [MP91], we instead show that some loop invariant always recurs after a finite amount of time, and that the system is safe in the meantime: this is essentially the function of the VC

$$
\begin{aligned}
VC_{p_{\mathsf{pre}},\mathcal{I}} \equiv \forall \texttt{xu}\!:\!+_{\mathsf{r}}.\ \forall \texttt{xm}\!:\!+_{\mathsf{r}}.\quad & ([0/\texttt{xpc}]\, p_{\mathsf{pre}} \supset \mathit{SE}_{\mathcal{I}}(0, \texttt{xu}, \texttt{xm})) \\
& \wedge \textstyle\bigwedge_{i \in \mathrm{dom}\, \mathcal{I}} ([\bar{i}/\texttt{xpc}]\, \mathcal{I}(i) \supset \mathit{SESafe}_{\mathcal{I}}(i, \texttt{xu}, \texttt{xm}))
\end{aligned}
$$

$$
\mathit{SE}_{\mathcal{I}}(i, e_{\texttt{u}}, e_{\texttt{m}}) \equiv \begin{cases} [\bar{i}/\texttt{xpc}]\,[e_{\texttt{u}}/\texttt{xu}]\,[e_{\texttt{m}}/\texttt{xm}]\,\mathcal{I}(i) & \text{if } i \in \mathrm{dom}\,\mathcal{I} \\ \mathit{SESafe}_{\mathcal{I}}(i, e_{\texttt{u}}, e_{\texttt{m}}) & \text{otherwise} \end{cases}
$$

$\mathcal{J}(\texttt{pm})_i$	$\mathit{SESafe}_{\mathcal{I}}(i, e_{\texttt{u}}, e_{\texttt{m}})$
$r_1 \leftarrow i_1$ $r_1 \leftarrow r_2$ $r_1 \leftarrow r_2\ \mathit{eop}_1\ r_3$ $\texttt{cond}\ \mathit{cop}_1\ r_1, i_1$	$\mathit{SENext}_{\mathcal{I}}(i, e_{\texttt{u}}, e_{\texttt{m}})$
$r_1 \leftarrow \texttt{m}(r_2)$	$\texttt{saferd}(e_{\texttt{m}}, \texttt{selu}(e_{\texttt{u}}, \overline{r_2})) \wedge \mathit{SENext}_{\mathcal{I}}(i, e_{\texttt{u}}, e_{\texttt{m}})$
$\texttt{m}(r_1) \leftarrow r_2$	$\texttt{safewr}(e_{\texttt{m}}, \texttt{selu}(e_{\texttt{u}}, \overline{r_1}), \texttt{selu}(e_{\texttt{u}}, \overline{r_2})) \wedge \mathit{SENext}_{\mathcal{I}}(i, e_{\texttt{u}}, e_{\texttt{m}})$

$\mathcal{J}(\texttt{pm})_i$	$\mathit{SENext}_{\mathcal{I}}(i, e_{\texttt{u}}, e_{\texttt{m}})$
$r_1 \leftarrow i_1$	$\mathit{SE}_{\mathcal{I}}(i \dot{+} 1, \texttt{updu}(e_{\texttt{u}}, \overline{r_1}, \overline{i_1}), e_{\texttt{m}})$
$r_1 \leftarrow r_2$	$\mathit{SE}_{\mathcal{I}}(i \dot{+} 1, \texttt{updu}(e_{\texttt{u}}, \overline{r_1}, \texttt{selu}(e_{\texttt{u}}, \overline{r_2})), e_{\texttt{m}})$
$r_1 \leftarrow r_2\ \mathit{eop}_1\ r_3$	$\mathit{SE}_{\mathcal{I}}(i \dot{+} 1, \texttt{updu}(e_{\texttt{u}}, \overline{r_1}, \texttt{selu}(e_{\texttt{u}}, \overline{r_2})\ \overline{\mathit{eop}_1}\ \texttt{selu}(e_{\texttt{u}}, \overline{r_3})), e_{\texttt{m}})$
$\texttt{cond}\ \mathit{cop}_1\ r_1, i_1$	$(\overline{\mathit{cop}_1}(\texttt{selu}(e_{\texttt{u}}, \overline{r_1})) \supset \mathit{SE}_{\mathcal{I}}(i \dot{+} 1 \dot{+} i_1, e_{\texttt{u}}, e_{\texttt{m}}))$ $\wedge(\overline{\neg \mathit{cop}_1}(\texttt{selu}(e_{\texttt{u}}, \overline{r_1})) \supset \mathit{SE}_{\mathcal{I}}(i \dot{+} 1, e_{\texttt{u}}, e_{\texttt{m}}))$
$r_1 \leftarrow \texttt{m}(r_2)$	$\mathit{SE}_{\mathcal{I}}(i \dot{+} 1, \texttt{updu}(e_{\texttt{u}}, \overline{r_1}, \texttt{selw}(e_{\texttt{m}}, \texttt{selu}(e_{\texttt{u}}, \overline{r_2}))), e_{\texttt{m}})$
$\texttt{m}(r_1) \leftarrow r_2$	$\mathit{SE}_{\mathcal{I}}(i \dot{+} 1, e_{\texttt{u}}, \texttt{updw}(e_{\texttt{m}}, \texttt{selu}(e_{\texttt{u}}, \overline{r_1}), \texttt{selu}(e_{\texttt{u}}, \overline{r_2})))$

Figure 14: VC Generation

generator. This property can be encoded easily enough by appealing to the "until" operator:

$$\Box(p_{\mathcal{I}} \supset p_{\mathsf{safe}} \wedge \bigcirc(p_{\mathsf{safe}} \, \mathcal{U} \, p_{\mathcal{I}}))$$

where $p_{\mathcal{I}}$ is the disjunction of all loop invariants. If we combine this with a derivation of

$$\mathtt{pc} = 0 \wedge p_{\mathsf{pre}} \supset p_{\mathsf{safe}} \, \mathcal{U} \, p_{\mathcal{I}}$$

we can derive p_{sp} through a constant number of temporal inferences.

We now realize two benefits: our safety proofs are considerably smaller than the equivalent global invariance proofs, and we obtain a correspondence with the VC generator that is close enough to embed a first-order proof directly. The reduction in proof size is brought about by specifying an invariant only for each loop head, rather than for each reachable instruction. Michael and Appel [MA00] achieve a similar reduction by factoring invariants using predicate transformers.

By realizing this strategy as an algorithm, we obtain the following result:

Proposition 5.1 (Relative Completeness) *There is an algorithm that derives*
$\vdash \mathtt{pc} = 0 \wedge [Reg]\, p_{\mathsf{pre}} \supset \Box p_{\mathsf{safe}} \circledast 0$
from $\vdash VC_{p_{\mathsf{pre}},\mathcal{I}} \circledast 0$*, where* p_{safe} *is the essential safety policy*

PROOF:
We provide an interdependent set of skeleton derivations that together refine a proof of a first-order VC into a corresponding temporal security proof. A *skeleton derivation* (*e.g.*, *SkStart*) is a derivation of some conclusion from one or more premises, but is not a derived rule because the structure of the derivation can depend on the premise(s) at which it is instantiated. These derivations constitute an algorithm for deriving the temporal security proof. In Figure 15, we show the premise(s) and conclusion of each skeleton derivation in diagrammatic form; in Section 5.2.1, we specify the inferences that make up the derivations. The skeleton derivations are based on the application of derived rules, which are specified in Figure 16 through Figure 19.

The result derivation is well formed by construction: we only need to show that we do not fail and we do not loop forever. We cannot loop forever because the VC is finite and is always decreased by any cycle in the algorithm. We can infer that the algorithm will not fail by examining which case checks could fail: the only such failures occur when $e_{\mathtt{u}}$ or $e_{\mathtt{m}}$ is not rigid. But, $e_{\mathtt{u}}$ and $e_{\mathtt{m}}$ are both initially instantiated to rigid parameters, and each successive instantiation contains no flexible expressions, so $e_{\mathtt{u}}$ and $e_{\mathtt{m}}$ are always rigid. □

The remainder of this section is based on the following notational abbreviations:

$$\begin{aligned} [Reg]\, p &\equiv [\mathtt{pc}/\mathtt{xpc}]\,[\mathtt{u}/\mathtt{xu}]\,[\mathtt{m}/\mathtt{xm}]\, p \\ p_{\mathcal{I}} &\equiv \textstyle\bigvee_{i \in \mathrm{dom}\, \mathcal{I}} \mathtt{pc} = \overline{i} \wedge [Reg]\, \mathcal{I}(i) \end{aligned}$$

$\mathcal{J}(\mathtt{pm})_i$	$InvNext(i, e_{\mathtt{u}}, e_{\mathtt{m}})$
$r_1 \leftarrow i_1$	$\mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = \mathtt{updu}(e_{\mathtt{u}}, \overline{r_1}, \overline{i_1}) \wedge \mathtt{m} = e_{\mathtt{m}}$
$r_1 \leftarrow r_2$	$\mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = \mathtt{updu}(e_{\mathtt{u}}, \overline{r_1}, \mathtt{selu}(e_{\mathtt{u}}, \overline{r_2})) \wedge \mathtt{m} = e_{\mathtt{m}}$
$r_1 \leftarrow r_2 \; eop_1 \; r_3$	$\mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = \mathtt{updu}(e_{\mathtt{u}}, \overline{r_1}, \mathtt{selu}(e_{\mathtt{u}}, \overline{r_2}) \; \overline{eop_1} \; \mathtt{selu}(e_{\mathtt{u}}, \overline{r_3})) \wedge \mathtt{m} = e_{\mathtt{m}}$
$\mathtt{cond} \; cop_1 \; r_1, i_1$	$(\mathtt{pc} = \overline{i \dot{+} 1 \dot{+} i_1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \wedge \overline{cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})))$ $\vee(\mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \wedge \overline{\neg cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})))$
$r_1 \leftarrow \mathtt{m}(r_2)$	$\mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = \mathtt{updu}(e_{\mathtt{u}}, \overline{r_1}, \mathtt{selw}(e_{\mathtt{m}}, \mathtt{selu}(e_{\mathtt{u}}, \overline{r_2}))) \wedge \mathtt{m} = e_{\mathtt{m}}$
$\mathtt{m}(r_1) \leftarrow r_2$	$\mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = \mathtt{updw}(e_{\mathtt{m}}, \mathtt{selu}(e_{\mathtt{u}}, \overline{r_1}), \mathtt{selu}(e_{\mathtt{u}}, \overline{r_2}))$

$$
\begin{array}{lll}
\mathit{Loc}_\Gamma(e_{\mathtt{u}}, e_{\mathtt{m}}, e'_{\mathtt{pc}}, e'_{\mathtt{u}}, p) & \text{iff} & \Gamma \vdash [a/\mathtt{xpc}]\,[e_{\mathtt{u}}/\mathtt{xu}]\,[e_{\mathtt{m}}/\mathtt{xm}]\,p :+_{\mathsf{l}}(a) \\
 & & \text{and } \Gamma \vdash [e'_{\mathtt{pc}}/\mathtt{xpc}]\,[a/\mathtt{xu}]\,[e_{\mathtt{m}}/\mathtt{xm}]\,p :+_{\mathsf{l}}(a) \\
 & & \text{and } \Gamma \vdash [e'_{\mathtt{pc}}/\mathtt{xpc}]\,[e_{\mathtt{u}'}/\mathtt{xu}]\,[a/\mathtt{xm}]\,p :+_{\mathsf{l}}(a) \\
\mathit{Rig}_\Gamma(e_1, \ldots, e_k, p_1, \ldots, p_{k'}) & \text{iff} & \Gamma \vdash e_1 :+_{\mathsf{r}} \text{ and } \ldots \text{ and } \Gamma \vdash e_k :+_{\mathsf{r}} \\
 & & \text{and } \Gamma \vdash p_1 :+_{\mathsf{r}} \text{ and } \ldots \text{ and } \Gamma \vdash p_{k'} :+_{\mathsf{r}}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{IsMvi}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{r1}}, e_{\mathtt{i1}}) & \text{iff} & \Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t \\
 & & \text{and } \Gamma \vdash \mathtt{fetch}(\mathtt{pm}, e_{\mathtt{pc}}) = \mathtt{imvi}(e_{\mathtt{r1}}, e_{\mathtt{i1}}) \mathbin{@} t \\
 & & \text{and } \Gamma \vdash e_{\mathtt{pc}} :+_{\mathsf{r}} \text{ and } \ldots \text{ and } \Gamma \vdash e_{\mathtt{i1}} :+_{\mathsf{r}} \\
\mathit{IsMv}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}) & \text{iff} & \Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t \\
 & & \text{and } \Gamma \vdash \mathtt{fetch}(\mathtt{pm}, e_{\mathtt{pc}}) = \mathtt{imv}(e_{\mathtt{r1}}, e_{\mathtt{r2}}) \mathbin{@} t \\
 & & \text{and } \Gamma \vdash e_{\mathtt{pc}} :+_{\mathsf{r}} \text{ and } \ldots \text{ and } \Gamma \vdash e_{\mathtt{r2}} :+_{\mathsf{r}} \\
\mathit{IsEop}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{eop1}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}, e_{\mathtt{r3}}) & \text{iff} & \Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t \\
 & & \text{and } \Gamma \vdash \mathtt{fetch}(\mathtt{pm}, e_{\mathtt{pc}}) = \mathtt{ieop}(e_{\mathtt{eop1}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}, e_{\mathtt{r3}}) \mathbin{@} t \\
 & & \text{and } \Gamma \vdash e_{\mathtt{pc}} :+_{\mathsf{r}} \text{ and } \ldots \text{ and } \Gamma \vdash e_{\mathtt{r3}} :+_{\mathsf{r}} \\
\mathit{IsCond}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{cop1}}, e_{\mathtt{r1}}, e_{\mathtt{i1}}) & \text{iff} & \Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t \\
 & & \text{and } \Gamma \vdash \mathtt{fetch}(\mathtt{pm}, e_{\mathtt{pc}}) = \mathtt{icond}(e_{\mathtt{cop1}}, e_{\mathtt{r1}}, e_{\mathtt{i1}}) \mathbin{@} t \\
 & & \text{and } \Gamma \vdash e_{\mathtt{pc}} :+_{\mathsf{r}} \text{ and } \ldots \text{ and } \Gamma \vdash e_{\mathtt{i1}} :+_{\mathsf{r}} \\
\mathit{IsLoad}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}) & \text{iff} & \Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t \\
 & & \text{and } \Gamma \vdash \mathtt{fetch}(\mathtt{pm}, e_{\mathtt{pc}}) = \mathtt{iload}(e_{\mathtt{r1}}, e_{\mathtt{r2}}) \mathbin{@} t \\
 & & \text{and } \Gamma \vdash e_{\mathtt{pc}} :+_{\mathsf{r}} \text{ and } \ldots \text{ and } \Gamma \vdash e_{\mathtt{r2}} :+_{\mathsf{r}} \\
\mathit{IsStore}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}) & \text{iff} & \Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t \\
 & & \text{and } \Gamma \vdash \mathtt{fetch}(\mathtt{pm}, e_{\mathtt{pc}}) = \mathtt{istore}(e_{\mathtt{r1}}, e_{\mathtt{r2}}) \mathbin{@} t \\
 & & \text{and } \Gamma \vdash e_{\mathtt{pc}} :+_{\mathsf{r}} \text{ and } \ldots \text{ and } \Gamma \vdash e_{\mathtt{r2}} :+_{\mathsf{r}}
\end{array}
$$

$$
\mathit{SafePC}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{len}}) \quad \text{iff} \quad \Gamma \vdash \mathtt{len}(\mathtt{pm}) = e_{\mathtt{len}} \mathbin{@} t \text{ and } \Gamma \vdash e_{\mathtt{len}} \,\mathtt{gtu}\, e_{\mathtt{pc}} = 1 \mathbin{@} t
$$

Note that *Loc* and *Rig* can be decided efficiently, and we can reduce proof sizes considerably by eliding their derivations. Note also that because p_{pre} and each $\mathcal{I}(i)$ are assertions with registers replaced by variables, they are local on all variables, and are rigid whenever they are instantiated with rigid variables.

5.2.1 Skeleton Derivations

In this section, we enumerate the inferences that comprise each skeleton derivation. The inferences constitute a specification for a proof-generation algorithm. Context weakening steps are not shown, because they are provided by our logical framework.

At the beginning of each derivation, we identify the premises with "Prem.". "App." is an abbreviation for "apply rule", and "Def." is an abbreviation for "by definition." We arrange conjunction and disjunction comprehensions into balanced trees so that conjunction eliminations and disjunction introductions take a logarithmic number of inferences.

The proof skeleton maintains the current machine state as a set of equalities between registers and rigid expressions (*e.g.*, $\mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}}$). These equalities permit us to discharge implications in and extract conclusions from the proof of the first-order VC.

$$\begin{array}{c} \vdash VC_{p_{\mathsf{pre}},\mathcal{I}} @\, 0 \\ \vdots\ \mathit{SkStart} \\ \vdash \mathtt{pc} = 0 \land [\mathit{Reg}]\, p_{\mathsf{pre}} \supset \Box p_{\mathsf{safe}} @\, 0 \end{array}$$

$$\begin{array}{c} \begin{array}{cc} \Gamma \vdash \bigvee_{i \in I} \mathtt{pc} = \bar{i} \land [\mathit{Reg}]\, \mathcal{I}(i) @\, t & \Gamma \vdash \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \bigwedge_{i \in I} ([\bar{i}/\mathtt{xpc}]\, \mathcal{I}(i) \supset \mathit{SESafe}_{\mathcal{I}}(i, \mathtt{xu}, \mathtt{xm})) @\, 0 \\ \vdots & \vdots \end{array} \\ \vdots\ \mathit{SkSplitStart} \\ \Gamma \vdash p_{\mathsf{safe}} \land \bigcirc (p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}}) @\, t \end{array}$$

$$\begin{array}{c} \begin{array}{cc} \Gamma \vdash \mathit{InvNext}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @\, t & \Gamma \vdash \mathit{SENext}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @\, 0 \\ \vdots & \vdots \end{array} \\ \vdots\ \mathit{SkSplit} \\ \Gamma \vdash p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}} @\, t \end{array}$$

$$\begin{array}{c} \begin{array}{cc} \Gamma \vdash \mathtt{pc} = \bar{i} \land \mathtt{u} = e_{\mathtt{u}} \land \mathtt{m} = e_{\mathtt{m}} @\, t & \Gamma \vdash \mathit{SE}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @\, 0 \\ \vdots & \vdots \end{array} \\ \vdots\ \mathit{SkNext} \\ \Gamma \vdash p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}} @\, t \end{array}$$

$$\begin{array}{c} \Gamma \vdash \mathtt{pc} = \bar{i} \land \mathtt{u} = e_{\mathtt{u}} \land \mathtt{m} = e_{\mathtt{m}} @\, t \\ \vdots\ \mathit{SkTrans} \\ \Gamma \vdash \mathit{InvNext}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @\, t + 1 \end{array}$$

$$\begin{array}{c} \begin{array}{cc} \Gamma \vdash \mathtt{pc} = \bar{i} \land \mathtt{u} = e_{\mathtt{u}} \land \mathtt{m} = e_{\mathtt{m}} @\, t & \Gamma \vdash \mathit{SESafe}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @\, 0 \\ \vdots & \vdots \end{array} \\ \vdots\ \mathit{SkSafe} \\ \Gamma \vdash p_{\mathsf{safe}} @\, t \end{array}$$

$$\begin{array}{c} \Gamma \vdash \mathit{SESafe}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @\, t \\ \vdots\ \mathit{SkSkip} \\ \Gamma \vdash \mathit{SENext}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @\, t \end{array}$$

Figure 15: Skeleton-Derivation Diagrams

$$\frac{\Gamma' \vdash p_1 \,\mathcal{U}\, p_2 \mathbin{@} t_0 \quad \Gamma,\, t \geq t_0,\, p_2 \mathbin{@} t \vdash p_1 \wedge \bigcirc(p_1 \,\mathcal{U}\, p_2) \mathbin{@} t}{\Gamma \vdash p_0 \supset \Box p_1 \mathbin{@} t_0}\ \textsf{sk\_safe}^{t,a_{\mathtt{u}},a_{\mathtt{m}}}$$

$$\text{where } \Gamma' = (\Gamma,\, p_0 \mathbin{@} t_0,\, a_{\mathtt{u}} = \mathtt{u} \mathbin{@} t_0,\, a_{\mathtt{u}} \mathbin{:} +_{\mathsf{r}},\, a_{\mathtt{m}} = \mathtt{m} \mathbin{@} t_0,\, a_{\mathtt{m}} \mathbin{:} +_{\mathsf{r}})$$

$$\frac{\Gamma' \vdash p_1 \mathbin{@} t \quad \Gamma' \vdash p_1 \,\mathcal{U}\, p_2 \mathbin{@} t+1}{\Gamma \vdash p_1 \wedge \bigcirc(p_1 \,\mathcal{U}\, p_2) \mathbin{@} t}\ \textsf{sk\_first}^{a_{\mathtt{u}},a_{\mathtt{m}}} \quad \text{where } \Gamma' = (\Gamma,\, a_{\mathtt{u}} = \mathtt{u} \mathbin{@} t,\, a_{\mathtt{u}} \mathbin{:} +_{\mathsf{r}},\, a_{\mathtt{m}} = \mathtt{m} \mathbin{@} t,\, a_{\mathtt{m}} \mathbin{:} +_{\mathsf{r}})$$

$$\frac{\Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge p \mathbin{@} t \quad \Gamma \vdash e_{\mathtt{u}} = \mathtt{u} \mathbin{@} t \quad \Gamma \vdash e_{\mathtt{m}} = \mathtt{m} \mathbin{@} t}{\Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t}\ \textsf{sk\_init}$$

$$\frac{\Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge [\mathit{Reg}]\, p \mathbin{@} t \quad \Gamma \vdash e_{\mathtt{u}} = \mathtt{u} \mathbin{@} t \quad \Gamma \vdash e_{\mathtt{m}} = \mathtt{m} \mathbin{@} t \quad \mathit{Loc}_{\Gamma}(\mathtt{u}, \mathtt{m}, e_{\mathtt{pc}}, e_{\mathtt{u}}, p)}{\Gamma \vdash [e_{\mathtt{pc}}/\mathtt{xpc}]\, [e_{\mathtt{u}}/\mathtt{xu}]\, [e_{\mathtt{m}}/\mathtt{xm}]\, p \mathbin{@} t}\ \textsf{sk\_inst}$$

$$\frac{\Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t \quad \Gamma \vdash p' \mathbin{@} t_0 \quad \mathit{Rig}_{\Gamma}(p') \quad \mathit{Loc}_{\Gamma}(e_{\mathtt{u}}, e_{\mathtt{m}}, \mathtt{pc}, \mathtt{u}, p)}{\Gamma \vdash \mathtt{pc} = e_{\mathtt{pc}} \wedge [\mathit{Reg}]\, p \mathbin{@} t}\ \textsf{sk\_loop}$$

$$\text{where } p' \equiv [e_{\mathtt{pc}}/\mathtt{xpc}]\, [e_{\mathtt{u}}/\mathtt{xu}]\, [e_{\mathtt{m}}/\mathtt{xm}]\, p$$

Figure 16: Derived Rules for Managing Invariants

$$\frac{\mathit{IsMvi}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{r1}}, e_{\mathtt{i1}}) \quad \Gamma \vdash e_{\mathtt{pc}}\ \mathtt{addw}\ 1 = e'_{\mathtt{pc}} \mathbin{@} t+1}{\Gamma \vdash \mathtt{pc} = e'_{\mathtt{pc}} \wedge \mathtt{u} = \mathtt{updu}(e_{\mathtt{u}}, e_{\mathtt{r1}}, e_{\mathtt{i1}}) \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t+1}\ \textsf{sk\_mvi}$$

$$\frac{\mathit{IsMv}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}) \quad \Gamma \vdash e_{\mathtt{pc}}\ \mathtt{addw}\ 1 = e'_{\mathtt{pc}} \mathbin{@} t+1}{\Gamma \vdash \mathtt{pc} = e'_{\mathtt{pc}} \wedge \mathtt{u} = \mathtt{updu}(e_{\mathtt{u}}, e_{\mathtt{r1}}, \mathtt{selu}(e_{\mathtt{u}}, e_{\mathtt{r2}})) \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t+1}\ \textsf{sk\_mv}$$

$$\frac{\mathit{IsEop}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{eop1}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}, e_{\mathtt{r3}}) \quad \Gamma \vdash e_{\mathtt{pc}}\ \mathtt{addw}\ 1 = e'_{\mathtt{pc}} \mathbin{@} t+1}{\Gamma \vdash \mathtt{pc} = e'_{\mathtt{pc}} \wedge \mathtt{u} = \mathtt{updu}(e_{\mathtt{u}}, e_{\mathtt{r1}}, \mathtt{selu}(e_{\mathtt{u}}, e_{\mathtt{r2}})\ e_{\mathtt{eop1}}\ \mathtt{selu}(e_{\mathtt{u}}, e_{\mathtt{r3}})) \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t+1}\ \textsf{sk\_eop}$$

$$\frac{\mathit{IsCond}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{cop1}}, e_{\mathtt{r1}}, e_{\mathtt{i1}}) \quad \Gamma \vdash p'_{\mathtt{pc}} \mathbin{@} t+1 \quad \Gamma \vdash p''_{\mathtt{pc}} \mathbin{@} t+1 \quad \Gamma \vdash p'_{\mathtt{cop1}} \mathbin{@} t+1}{\Gamma \vdash \begin{array}{l} (\mathtt{pc} = e'_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \wedge e_{\mathtt{cop1}}(\mathtt{selu}(e_{\mathtt{u}}, e_{\mathtt{r1}}))) \\ \vee(\mathtt{pc} = e''_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \wedge e'_{\mathtt{cop1}}(\mathtt{selu}(e_{\mathtt{u}}, e_{\mathtt{r1}}))) \end{array} \mathbin{@} t+1}\ \textsf{sk\_cond}$$

$$\begin{array}{l} \text{where } p'_{\mathtt{pc}} \equiv e_{\mathtt{pc}}\ \mathtt{addw}\ 1\ \mathtt{addw}\ e_{\mathtt{i1}} = e'_{\mathtt{pc}} \\ \text{and } p''_{\mathtt{pc}} \equiv e_{\mathtt{pc}}\ \mathtt{addw}\ 1 = e''_{\mathtt{pc}} \\ \text{and } p'_{\mathtt{cop1}} \equiv \mathtt{compl}(e_{\mathtt{cop1}}) = e'_{\mathtt{cop1}} \end{array}$$

$$\frac{\mathit{IsLoad}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}) \quad \Gamma \vdash e_{\mathtt{pc}}\ \mathtt{addw}\ 1 = e'_{\mathtt{pc}} \mathbin{@} t+1}{\Gamma \vdash \mathtt{pc} = e'_{\mathtt{pc}} \wedge \mathtt{u} = \mathtt{updu}(e_{\mathtt{u}}, e_{\mathtt{r1}}, \mathtt{selw}(e_{\mathtt{m}}, \mathtt{selu}(e_{\mathtt{u}}, e_{\mathtt{r2}}))) \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t+1}\ \textsf{sk\_load}$$

$$\frac{\mathit{IsStore}_{\Gamma,t}(e_{\mathtt{pc}}, e_{\mathtt{u}}, e_{\mathtt{m}}, e_{\mathtt{r1}}, e_{\mathtt{r2}}) \quad \Gamma \vdash e_{\mathtt{pc}}\ \mathtt{addw}\ 1 = e'_{\mathtt{pc}} \mathbin{@} t+1}{\Gamma \vdash \mathtt{pc} = e'_{\mathtt{pc}} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = \mathtt{updw}(e_{\mathtt{m}}, \mathtt{selu}(e_{\mathtt{u}}, e_{\mathtt{r1}}), \mathtt{selu}(e_{\mathtt{u}}, e_{\mathtt{r2}})) \mathbin{@} t+1}\ \textsf{sk\_store}$$

Figure 17: Derived Rules for Evaluating Instructions

$$\frac{\mathit{IsMvi}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{u}}, e_{\texttt{m}}, e_{\texttt{r1}}, e_{\texttt{i1}}) \quad \mathit{SafePC}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{len}})}{\Gamma \vdash p_{\mathsf{safe}} \mathbin{@} t} \;\textsf{sk\_safe\_mvi}$$

$$\frac{\mathit{IsMv}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{u}}, e_{\texttt{m}}, e_{\texttt{r1}}, e_{\texttt{r2}}) \quad \mathit{SafePC}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{len}})}{\Gamma \vdash p_{\mathsf{safe}} \mathbin{@} t} \;\textsf{sk\_safe\_mv}$$

$$\frac{\mathit{IsEop}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{u}}, e_{\texttt{m}}, e_{\texttt{eop1}}, e_{\texttt{r1}}, e_{\texttt{r2}}, e_{\texttt{r3}}) \quad \mathit{SafePC}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{len}})}{\Gamma \vdash p_{\mathsf{safe}} \mathbin{@} t} \;\textsf{sk\_safe\_eop}$$

$$\frac{\mathit{IsCond}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{u}}, e_{\texttt{m}}, e_{\texttt{cop1}}, e_{\texttt{r1}}, e_{\texttt{i1}}) \quad \mathit{SafePC}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{len}})}{\Gamma \vdash p_{\mathsf{safe}} \mathbin{@} t} \;\textsf{sk\_safe\_cond}$$

$$\frac{\mathit{IsLoad}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{u}}, e_{\texttt{m}}, e_{\texttt{r1}}, e_{\texttt{r2}}) \quad \mathit{SafePC}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{len}}) \quad \Gamma \vdash p_{\texttt{saferd}} \mathbin{@} t_0 \quad \Gamma \vdash p_{\texttt{saferd}} :+_{\mathsf{r}}}{\Gamma \vdash p_{\mathsf{safe}} \mathbin{@} t} \;\textsf{sk\_safe\_load}$$

where $p_{\texttt{saferd}} \equiv \texttt{saferd}(e_{\texttt{m}}, \texttt{selu}(e_{\texttt{u}}, e_{\texttt{r2}}))$

$$\frac{\mathit{IsStore}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{u}}, e_{\texttt{m}}, e_{\texttt{r1}}, e_{\texttt{r2}}) \quad \mathit{SafePC}_{\Gamma,t}(e_{\texttt{pc}}, e_{\texttt{len}}) \quad \Gamma \vdash p_{\texttt{safewr}} \mathbin{@} t_0 \quad \Gamma \vdash p_{\texttt{safewr}} :+_{\mathsf{r}}}{\Gamma \vdash p_{\mathsf{safe}} \mathbin{@} t} \;\textsf{sk\_safe\_store}$$

where $p_{\texttt{safewr}} \equiv \texttt{safewr}(e_{\texttt{m}}, \texttt{selu}(e_{\texttt{u}}, e_{\texttt{r1}}), \texttt{selu}(e_{\texttt{u}}, e_{\texttt{r2}}))$

Figure 18: Derived Rules for Instruction Safety

$$\frac{\Gamma \vdash [a/x]\, p :+_{\mathsf{l}} (a) \quad \Gamma \vdash e' = e \mathbin{@} t \quad \Gamma \vdash [e/x]\, p \mathbin{@} t}{\Gamma \vdash [e'/x]\, p \mathbin{@} t} \;\textsf{el\_sym}^{a}$$

$$\frac{\Gamma \vdash p_2 \mathbin{@} t}{\Gamma \vdash p_1 \,\mathcal{U}\, p_2 \mathbin{@} t} \;\textsf{u\_now} \qquad \frac{\Gamma \vdash p_1 \mathbin{@} t \quad \Gamma \vdash p_1 \,\mathcal{U}\, p_2 \mathbin{@} t+1}{\Gamma \vdash p_1 \,\mathcal{U}\, p_2 \mathbin{@} t} \;\textsf{u\_next}$$

Figure 19: Generic Derived Rules

Note that in our implementation, we have special cases for unconditional branches (*e.g.*, `truew`) that do not consider the case that never holds. We do not show these special cases here in the interest of simplifying the presentation.

The proof rule *AndEL* is equivalent to applying $\wedge\mathsf{el}$, except in the case where the last inference of the target derivation is $\wedge\mathsf{i}$, in which case we perform a local reduction and simply use the left premise of the $\wedge\mathsf{i}$ rule. *AndER*, *ImpE*, and *AllE* are similar, except that we must substitute a derivation or expression in the latter two cases. Some care must be taken when substituting derivations to avoid duplicating large parts of the proof tree. In practice, we use an embedded "cut" rule to ensure that any derivation that might be substituted is always a variable.

SkStart $\Longrightarrow$

$\vdash VC_{p_{\mathsf{pre}},\mathcal{I}} @ 0$	Prem.
let $\Gamma = (\cdot,\, \mathtt{pc} = 0 \wedge [\mathit{Reg}]\, p_{\mathsf{pre}} @ 0,\, a_{\mathtt{u}} = \mathtt{u} @ 0,\, a_{\mathtt{u}}{:}+_{\mathsf{r}},\, a_{\mathtt{m}} = \mathtt{m} @ 0,\, a_{\mathtt{m}}{:}+_{\mathsf{r}})$	
$\Gamma \vdash \mathtt{pc} = 0 \wedge [\mathit{Reg}]\, p_{\mathsf{pre}} @ 0 \quad \Gamma \vdash a_{\mathtt{u}} = \mathtt{u} @ 0 \quad \Gamma \vdash a_{\mathtt{m}} = \mathtt{m} @ 0$	App. hyp
$\Gamma \vdash \mathtt{pc} = 0 \wedge \mathtt{u} = a_{\mathtt{u}} \wedge \mathtt{m} = a_{\mathtt{m}} @ 0$	App. $\mathsf{sk\_init}$
$\mathit{Loc}_{\Gamma}(\mathtt{u}, \mathtt{m}, 0, a_{\mathtt{u}}, p_{\mathsf{pre}})$	Def. p_{pre}
$\Gamma \vdash [0/\mathtt{xpc}]\,[a_{\mathtt{u}}/\mathtt{xu}]\,[a_{\mathtt{m}}/\mathtt{xm}]\, p_{\mathsf{pre}} @ 0$	App. $\mathsf{sk\_inst}$
$\mathit{Rig}_{\Gamma}(a_{\mathtt{u}}, a_{\mathtt{m}})$	
$\Gamma \vdash \mathit{SE}_{\mathcal{I}}(0, a_{\mathtt{u}}, a_{\mathtt{m}}) @ 0$	$\mathit{AllE} \times 2, \mathit{AndEL}, \mathit{ImpE}$
$\Gamma \vdash p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}} @ 0$	*SkNext*
let $t \notin \Gamma, \Gamma' = (\cdot,\, t \geq 0,\, p_{\mathcal{I}} @ t)$	
$\Gamma' \vdash p_{\mathcal{I}} @ t$	App. hyp
$\Gamma' \vdash \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \bigwedge_{i \in \mathrm{dom}\, \mathcal{I}} ([\bar{i}/\mathtt{xpc}]\, \mathcal{I}(i) \supset \mathit{SESafe}_{\mathcal{I}}(i, \mathtt{xu}, \mathtt{xm})) @ 0$	$\mathit{AllE} \times 2, \mathit{AndER}$, App. $\forall\mathsf{i} \times 2$
$\Gamma' \vdash p_{\mathsf{safe}} \wedge \bigcirc(p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}}) @ t$	*SkSplitStart*
$\vdash \mathtt{pc} = 0 \wedge [\mathit{Reg}]\, p_{\mathsf{pre}} \supset \Box p_{\mathsf{safe}} @ 0$	App. $\mathsf{sk\_safe}^{t, a_{\mathtt{u}}, a_{\mathtt{m}}}$

SkSplitStart $\Longrightarrow$

$\Gamma \vdash \bigvee_{i \in I} \mathtt{pc} = \bar{i} \wedge [\mathit{Reg}]\, \mathcal{I}(i) @ t$	Prem.
$\Gamma \vdash \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \bigwedge_{i \in I} ([\bar{i}/\mathtt{xpc}]\, \mathcal{I}(i) \supset \mathit{SESafe}_{\mathcal{I}}(i, \mathtt{xu}, \mathtt{xm})) @ 0$	Prem.
case: $I = \emptyset$	
$\quad \Gamma \vdash p_{\mathsf{safe}} \wedge \bigcirc(p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}}) @ t$	App. $\perp\mathsf{e}$
case: $I = \{i\}$	
$\quad$ let $a_{\mathtt{u}}, a_{\mathtt{m}} \notin \Gamma, \Gamma' = (\Gamma,\, a_{\mathtt{u}} = \mathtt{u} @ t,\, a_{\mathtt{u}}{:}+_{\mathsf{r}},\, a_{\mathtt{m}} = \mathtt{m} @ t,\, a_{\mathtt{m}}{:}+_{\mathsf{r}})$	
$\quad \Gamma' \vdash a_{\mathtt{u}} = \mathtt{u} @ t \quad \Gamma' \vdash a_{\mathtt{m}} = \mathtt{m} @ t$	App. hyp
$\quad \Gamma' \vdash \mathtt{pc} = \bar{i} \wedge \mathtt{u} = a_{\mathtt{u}} \wedge \mathtt{m} = a_{\mathtt{m}} @ t$	App. $\mathsf{sk\_init}$
$\quad \mathit{Loc}_{\Gamma'}(\mathtt{u}, \mathtt{m}, \bar{i}, a_{\mathtt{u}}, \mathcal{I}(i))$	Def. $\mathcal{I}$
$\quad \Gamma' \vdash [\bar{i}/\mathtt{xpc}]\,[a_{\mathtt{u}}/\mathtt{xu}]\,[a_{\mathtt{m}}/\mathtt{xm}]\, \mathcal{I}(i) @ t$	App. $\mathsf{sk\_inst}$
$\quad \mathit{Rig}_{\Gamma'}(a_{\mathtt{u}}, a_{\mathtt{m}}, [\bar{i}/\mathtt{xpc}]\,[a_{\mathtt{u}}/\mathtt{xu}]\,[a_{\mathtt{m}}/\mathtt{xm}]\, \mathcal{I}(i))$	Def. $\mathcal{I}$
$\quad \Gamma' \vdash \mathit{SESafe}_{\mathcal{I}}(i, a_{\mathtt{u}}, a_{\mathtt{m}}) @ 0$	$\mathit{AllE} \times 2, \mathit{ImpE}$
$\quad \Gamma' \vdash p_{\mathsf{safe}} @ t$	*SkSafe*
$\quad \Gamma' \vdash \mathit{InvNext}(i, a_{\mathtt{u}}, a_{\mathtt{m}}) @ t + 1$	*SkTrans*
$\quad \Gamma' \vdash \mathit{SENext}_{\mathcal{I}}(i, a_{\mathtt{u}}, a_{\mathtt{m}}) @ 0$	*SkSkip*
$\quad \Gamma' \vdash p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}} @ t + 1$	*SkSplit*
$\quad \Gamma \vdash p_{\mathsf{safe}} \wedge \bigcirc(p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}}) @ t$	App. $\mathsf{sk\_first}^{a_{\mathtt{u}}, a_{\mathtt{m}}}$
case: $I = I_1 \cup I_2$	
$\quad$ let $\Gamma_1 = (\Gamma,\, \bigvee_{i \in I_1} \mathtt{pc} = \bar{i} \wedge [\mathit{Reg}]\, \mathcal{I}(i) @ t)$	

$\Gamma_1 \vdash \bigvee_{i \in I_1} \mathtt{pc} = \bar{i} \wedge [Reg]\,\mathcal{I}(i) \,@\, t$	App. hyp
$\Gamma_1 \vdash \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \bigwedge_{i \in I_1}([\bar{i}/\mathtt{xpc}]\,\mathcal{I}(i) \supset \mathit{SESafe}_{\mathcal{I}}(i, \mathtt{xu}, \mathtt{xm})) \,@\, 0$	*AllE* $\times$ 2, *AndEL*, App. $\forall\mathsf{i} \times 2$
$\Gamma_1 \vdash p_{\mathsf{safe}} \wedge \bigcirc(p_{\mathsf{safe}}\,\mathcal{U}\,p_{\mathcal{I}}) \,@\, t$	*SkSplitStart*
let $\Gamma_2 = (\Gamma, \bigvee_{i \in I_2} \mathtt{pc} = \bar{i} \wedge [Reg]\,\mathcal{I}(i) \,@\, t)$	
$\Gamma_2 \vdash \bigvee_{i \in I_2} \mathtt{pc} = \bar{i} \wedge [Reg]\,\mathcal{I}(i) \,@\, t$	App. hyp
$\Gamma_2 \vdash \forall \mathtt{xu}{:}+_{\mathsf{r}}.\ \forall \mathtt{xm}{:}+_{\mathsf{r}}.\ \bigwedge_{i \in I_2}([\bar{i}/\mathtt{xpc}]\,\mathcal{I}(i) \supset \mathit{SESafe}_{\mathcal{I}}(i, \mathtt{xu}, \mathtt{xm})) \,@\, 0$	*AllE* $\times$ 2, *AndER*, App. $\forall\mathsf{i} \times 2$
$\Gamma_2 \vdash p_{\mathsf{safe}} \wedge \bigcirc(p_{\mathsf{safe}}\,\mathcal{U}\,p_{\mathcal{I}}) \,@\, t$	*SkSplitStart*
$\Gamma \vdash p_{\mathsf{safe}} \wedge \bigcirc(p_{\mathsf{safe}}\,\mathcal{U}\,p_{\mathcal{I}}) \,@\, t$	App. $\vee\mathsf{e}$

SkSplit $\Longrightarrow$

$\Gamma \vdash \mathit{InvNext}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \,@\, t \quad \Gamma \vdash \mathit{SENext}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \,@\, 0$	Prem.
case: $\mathcal{J}(\mathtt{pm})_i = (\mathtt{cond}\ cop_1\ r_1, i_1)$	
case: $Rig_{\Gamma}(e_{\mathtt{u}})$	
let $\Gamma_1 = (\Gamma, \mathtt{pc} = \overline{i \dot{+} 1 \dot{+} i_1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \wedge \overline{cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})) \,@\, t)$	
$\Gamma_1 \vdash \mathtt{pc} = \overline{i \dot{+} 1 \dot{+} i_1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \wedge \overline{cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})) \,@\, t$	App. hyp
$\Gamma_1 \vdash \mathtt{pc} = \overline{i \dot{+} 1 \dot{+} i_1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \,@\, t$	App. $\wedge\mathsf{el}$
$\Gamma_1 \vdash \overline{cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})) \,@\, t$	App. $\wedge\mathsf{er}$
$Rig_{\Gamma_1}(\overline{cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})))$	
$\Gamma_1 \vdash \overline{cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})) \,@\, 0$	App. $\mathsf{e_r}$
$\Gamma_1 \vdash \mathit{SE}_{\mathcal{I}}(i \dot{+} 1 \dot{+} i_1, e_{\mathtt{u}}, e_{\mathtt{m}}) \,@\, 0$	*AndEL*, *ImpE*
$\Gamma_1 \vdash p_{\mathsf{safe}}\,\mathcal{U}\,p_{\mathcal{I}} \,@\, t$	*SkNext*
let $\Gamma_2 = (\Gamma, \mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \wedge \overline{\neg cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})) \,@\, t)$	
$\Gamma_2 \vdash \mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \wedge \overline{\neg cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})) \,@\, t$	App. hyp
$\Gamma_2 \vdash \mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \,@\, t$	App. $\wedge\mathsf{el}$
$\Gamma_2 \vdash \overline{\neg cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})) \,@\, t$	App. $\wedge\mathsf{er}$
$Rig_{\Gamma_2}(\overline{\neg cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})))$	
$\Gamma_2 \vdash \overline{\neg cop_1}(\mathtt{selu}(e_{\mathtt{u}}, \overline{r_1})) \,@\, 0$	App. $\mathsf{e_r}$
$\Gamma_2 \vdash \mathit{SE}_{\mathcal{I}}(i \dot{+} 1, e_{\mathtt{u}}, e_{\mathtt{m}}) \,@\, 0$	*AndER*, *ImpE*
$\Gamma_2 \vdash p_{\mathsf{safe}}\,\mathcal{U}\,p_{\mathcal{I}} \,@\, t$	*SkNext*
$\Gamma \vdash p_{\mathsf{safe}}\,\mathcal{U}\,p_{\mathcal{I}} \,@\, t$	App. $\vee\mathsf{e}$
case: not $Rig_{\Gamma}(e_{\mathtt{u}})$	
fail	
case: $\mathcal{J}(\mathtt{pm})_i \neq (\mathtt{cond}\ cop_1\ r_1, i_1)$	
$\Gamma \vdash \mathtt{pc} = \overline{i \dot{+} 1} \wedge \mathtt{u} = e'_{\mathtt{u}} \wedge \mathtt{m} = e'_{\mathtt{m}} \,@\, t$	Def. *InvNext*
$\Gamma \vdash \mathit{SE}_{\mathcal{I}}(i \dot{+} 1, e'_{\mathtt{u}}, e'_{\mathtt{m}}) \,@\, 0$	Def. *SENext*
$\Gamma \vdash p_{\mathsf{safe}}\,\mathcal{U}\,p_{\mathcal{I}} \,@\, t$	*SkNext*

SkNext $\Longrightarrow$

$\Gamma \vdash \mathtt{pc} = \bar{i} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \,@\, t \quad \Gamma \vdash \mathit{SE}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \,@\, 0$	Prem.
case: $i \in \mathrm{dom}\,\mathcal{I}$	
case: $Rig_{\Gamma}(e_{\mathtt{u}}, e_{\mathtt{m}})$	
$Rig_{\Gamma}([\bar{i}/\mathtt{xpc}]\,[e_{\mathtt{u}}/\mathtt{xu}]\,[e_{\mathtt{m}}/\mathtt{xm}]\,\mathcal{I}(i))$	Def. $\mathcal{I}$
$Loc_{\Gamma}(e_{\mathtt{u}}, e_{\mathtt{m}}, \mathtt{pc}, \mathtt{u}, \mathcal{I}(i))$	Def. $\mathcal{I}$

$\Gamma \vdash \mathtt{pc} = \overline{i} \wedge [Reg]\,\mathcal{I}(i) \mathbin{@} t$	App. sk_loop
$\Gamma \vdash p_{\mathcal{I}} \mathbin{@} t$	App. $\vee$i $\times \lceil \log_2 \lvert \mathrm{dom}\,\mathcal{I} \rvert \rceil$
$\Gamma \vdash p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}} \mathbin{@} t$	App. u_now
case: not $Rig_{\Gamma}(e_{\mathtt{u}}, e_{\mathtt{m}})$	
fail	
case: $i \notin \mathrm{dom}\,\mathcal{I}$	
$\Gamma \vdash p_{\mathsf{safe}} \mathbin{@} t$	*SkSafe*
$\Gamma \vdash InvNext(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \mathbin{@} t+1$	*SkTrans*
$\Gamma \vdash SENext_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \mathbin{@} 0$	*SkSkip*
$\Gamma \vdash p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}} \mathbin{@} t+1$	*SkSplit*
$\Gamma \vdash p_{\mathsf{safe}}\, \mathcal{U}\, p_{\mathcal{I}} \mathbin{@} t$	App. u_next

SkTrans $\Longrightarrow$

$\Gamma \vdash \mathtt{pc} = \overline{i} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} \mathbin{@} t$	Prem.
case: $Rig_{\Gamma}(e_{\mathtt{u}}, e_{\mathtt{m}})$	
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow i_1)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \overline{i}) = \mathtt{imvi}(\overline{r_1}, \overline{i_1}) \mathbin{@} t$	App. pm_mvi
$\Gamma \vdash \overline{i}\ \mathtt{addw}\ 1 = \overline{i \dot{+} 1} \mathbin{@} t+1$	App. const_eop
$\Gamma \vdash InvNext(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \mathbin{@} t+1$	App. sk_mvi
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow r_2)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \overline{i}) = \mathtt{imv}(\overline{r_1}, \overline{r_2}) \mathbin{@} t$	App. pm_mv
$\Gamma \vdash \overline{i}\ \mathtt{addw}\ 1 = \overline{i \dot{+} 1} \mathbin{@} t+1$	App. const_eop
$\Gamma \vdash InvNext(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \mathbin{@} t+1$	App. sk_mv
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow r_2\ eop_1\ r_3)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \overline{i}) = \mathtt{ieop}(\overline{eop_1}, \overline{r_1}, \overline{r_2}, \overline{r_3}) \mathbin{@} t$	App. pm_eop
$\Gamma \vdash \overline{i}\ \mathtt{addw}\ 1 = \overline{i \dot{+} 1} \mathbin{@} t+1$	App. const_eop
$\Gamma \vdash InvNext(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \mathbin{@} t+1$	App. sk_eop
case: $\mathcal{J}(\mathtt{pm})_i = (\mathtt{cond}\ cop_1\ r_1, i_1)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \overline{i}) = \mathtt{icond}(\overline{cop_1}, \overline{r_1}, \overline{i_1}) \mathbin{@} t$	App. pm_cond
$\Gamma \vdash \overline{i}\ \mathtt{addw}\ 1 = \overline{i \dot{+} 1} \mathbin{@} t+1$	App. const_eop
$\Gamma \vdash \overline{i \dot{+} 1}\ \mathtt{addw}\ \overline{i_1} = \overline{i \dot{+} 1 \dot{+} i_1} \mathbin{@} t+1$	App. const_eop
$\Gamma \vdash \overline{i}\ \mathtt{addw}\ 1\ \mathtt{addw}\ \overline{i_1} = \overline{i \dot{+} 1 \dot{+} i_1} \mathbin{@} t+1$	App. el_sym
$\Gamma \vdash \mathtt{compl}(\overline{cop_1}) = \overline{\neg cop_1} \mathbin{@} t+1$	App. const_compl
$\Gamma \vdash InvNext(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \mathbin{@} t+1$	App. sk_cond
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow \mathtt{m}(r_2))$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \overline{i}) = \mathtt{iload}(\overline{r_1}, \overline{r_2}) \mathbin{@} t$	App. pm_load
$\Gamma \vdash \overline{i}\ \mathtt{addw}\ 1 = \overline{i \dot{+} 1} \mathbin{@} t+1$	App. const_eop
$\Gamma \vdash InvNext(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \mathbin{@} t+1$	App. sk_load
case: $\mathcal{J}(\mathtt{pm})_i = (\mathtt{m}(r_1) \leftarrow r_2)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \overline{i}) = \mathtt{istore}(\overline{r_1}, \overline{r_2}) \mathbin{@} t$	App. pm_store
$\Gamma \vdash \overline{i}\ \mathtt{addw}\ 1 = \overline{i \dot{+} 1} \mathbin{@} t+1$	App. const_eop
$\Gamma \vdash InvNext(i, e_{\mathtt{u}}, e_{\mathtt{m}}) \mathbin{@} t+1$	App. sk_store
case: not $Rig_{\Gamma}(e_{\mathtt{u}}, e_{\mathtt{m}})$	
fail	

SkSafe $\Longrightarrow$

$\Gamma \vdash \mathtt{pc} = \bar{i} \wedge \mathtt{u} = e_{\mathtt{u}} \wedge \mathtt{m} = e_{\mathtt{m}} @ t \quad \Gamma \vdash \mathit{SESafe}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @ 0$	Prem.
$\Gamma \vdash \overline{\mathtt{len}(\mathtt{pm}) = \lvert\mathcal{J}(\mathtt{pm})\rvert} @ t$	App. pm_len
$\Gamma \vdash \overline{\lvert\mathcal{J}(\mathtt{pm})\rvert}\ \mathtt{gtu}\ \bar{i} = 1 @ t$	App. const_eop
case: $\mathit{Rig}_{\Gamma}(e_{\mathtt{u}}, e_{\mathtt{m}})$	
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow i_1)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \bar{i}) = \mathtt{imvi}(\overline{r_1}, \overline{i_1}) @ t$	App. pm_mvi
$\Gamma \vdash p_{\mathsf{safe}} @ t$	App. sk_safe_mvi
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow r_2)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \bar{i}) = \mathtt{imv}(\overline{r_1}, \overline{r_2}) @ t$	App. pm_mv
$\Gamma \vdash p_{\mathsf{safe}} @ t$	App. sk_safe_mv
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow r_2\ \mathit{eop}_1\ r_3)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \bar{i}) = \mathtt{ieop}(\overline{\mathit{eop}_1}, \overline{r_1}, \overline{r_2}, \overline{r_3}) @ t$	App. pm_eop
$\Gamma \vdash p_{\mathsf{safe}} @ t$	App. sk_safe_eop
case: $\mathcal{J}(\mathtt{pm})_i = (\mathtt{cond}\ \mathit{cop}_1\ r_1, i_1)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \bar{i}) = \mathtt{icond}(\overline{\mathit{cop}_1}, \overline{r_1}, \overline{i_1}) @ t$	App. pm_cond
$\Gamma \vdash p_{\mathsf{safe}} @ t$	App. sk_safe_cond
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow \mathtt{m}(r_2))$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \bar{i}) = \mathtt{iload}(\overline{r_1}, \overline{r_2}) @ t$	App. pm_load
$\Gamma \vdash \mathtt{saferd}(e_{\mathtt{m}}, \mathtt{selu}(e_{\mathtt{u}}, \overline{r_2})) @ 0$	*AndEL*
$\Gamma \vdash p_{\mathsf{safe}} @ t$	App. sk_safe_load
case: $\mathcal{J}(\mathtt{pm})_i = (\mathtt{m}(r_1) \leftarrow r_2)$	
$\Gamma \vdash \mathtt{fetch}(\mathtt{pm}, \bar{i}) = \mathtt{istore}(\overline{r_1}, \overline{r_2}) @ t$	App. pm_store
$\Gamma \vdash \mathtt{safewr}(e_{\mathtt{m}}, \mathtt{selu}(e_{\mathtt{u}}, \overline{r_1}), \mathtt{selu}(e_{\mathtt{u}}, \overline{r_2})) @ 0$	*AndEL*
$\Gamma \vdash p_{\mathsf{safe}} @ t$	App. sk_safe_store
case: not $\mathit{Rig}_{\Gamma}(e_{\mathtt{u}}, e_{\mathtt{m}})$	
fail	

SkSkip $\Longrightarrow$

$\Gamma \vdash \mathit{SESafe}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @ t$	Prem.
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow i_1)$	
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow r_2)$	
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow r_2\ \mathit{eop}_1\ r_3)$	
case: $\mathcal{J}(\mathtt{pm})_i = (\mathtt{cond}\ \mathit{cop}_1\ r_1, i_1)$	
$\Gamma \vdash \mathit{SENext}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @ t$	Def. *SESafe*
case: $\mathcal{J}(\mathtt{pm})_i = (r_1 \leftarrow \mathtt{m}(r_2))$	
case: $\mathcal{J}(\mathtt{pm})_i = (\mathtt{m}(r_1) \leftarrow r_2)$	
$\Gamma \vdash \mathit{SENext}_{\mathcal{I}}(i, e_{\mathtt{u}}, e_{\mathtt{m}}) @ t$	*AndER*

5.2.2 Implementation

We have implemented a prototype proof generator for the x86 processor (as well as the abstract RISC processor) as logic programs in the Twelf [PS99] meta-logical framework, along with a simulator for the enforcement mechanism. Our x86 proof generator is compatible with the SpecialJ certifying compiler for Java [CLN+00]. The SpecialJ compiler produces certified x86 executable code from Java class files; our new framework generates temporal-logic proofs from this certified code.

The x86 proof generator is considerably more complicated than the system presented here, though both are based on the same proof-generation strategy. We plan to publish a future paper in which details of the x86 infrastructure will be provided.

To enable compact certificates, we additionally attach a *decoding* to each temporal-logic proof that specifies a binary-to-LF translation. This enables the binary encoding of the proof to be customized to the certification strategy. Initial experiments indicate that the total certificate size is between four and seven times the program size. Though such proofs are relatively large by current standards [NR01], the experiments suggest that our approach is practical.

6 Conclusion

The contributions of this research are threefold:

- A temporal-logic framework for PCC that is parameterized by formal security properties
- An enforcement mechanism for security properties that is simple to implement and easy to verify
- A certification mechanism for type safety that adapts existing certifying compilers to temporal logic

Our contributions are practical applications of proven techniques for program verification: our challenge lies principally in engineering efficient security proofs and in minimizing the complexity of the trusted enforcement mechanism.

Our approach offers these benefits:

- Temporal logic is a suitable language for specifying security policies, including "expressive" [Wal00, Sch99] safety properties and liveness properties. Thus, we can specify security policies directly without a special interpreter, and without having to write any C code.
- Enforcement is simple—we minimize the amount of trusted code by moving the VC generator out of the code consumer. Soundness of the enforcement mechanism is a direct consequence of the abstract machine semantics.
- Enforcement is also flexible—the enforcement mechanism adapts to different VC generators as a matter of course. Additionally, it does not anticipate and thereby restrict control flow; an indirect jump, for example, can branch to any address that is proven safe.

These advantages come at a cost, however, because our security proofs require a temporal proof skeleton in addition to first-order security proofs; in practice, we expect the proof skeleton to grow linearly with the size of the program.

We should acknowledge that temporal logic is not a fundamental requirement of our approach: for example, temporal logic can be translated into first-order logic with explicit time parameters, and state transition relations can mimic temporal operators by transitive closure.[8] However, the choice of notation

[8]We conjecture, however, that an explicit representation of a state transition is needed to make the VC generator into an untrusted component.

for PCC has practical consequences, because formalisms that are equivalent in a foundational sense may not enable equally compact security proofs. Temporal logic is well established as a specification language, but only further experiments will reveal whether it is a good notation for a PCC implementation.

6.1 Future Work

Our machine model does not have a procedure mechanism: we might adapt the procedure mechanism from Necula [Nec98], but at the cost of additional trusted code and restrictions on control flow. Instead, we have developed an untrusted mechanism based on new certification techniques, and thus we can continue to use the same simple enforcement mechanism we have presented here. Our current x86 implementation can certify nonrecursive procedures using the standard calling convention. In order to prove the specification of a recursive procedure using our current technique, we must be able to assume provisionally that the specification holds—we are currently investigating how such a proof rule might be incorporated into our current framework.

We plan to adapt instrumentation techniques for security automata [Sch99] to the certification problem. Security automata can specify all safety properties, and program transformations exist [ES00, Wal00] that will guarantee in many cases that such properties hold. A security automaton that has been threaded through a program by instrumentation is known is an *inline reference monitor* (IRM). Adding an IRM transformation to our certification mechanism would considerably broaden the class of security properties that we can automatically certify.

Our enforcement mechanism can be extended to check self-modifying code by encoding the processor's instruction decoder as a formal relation. This is not fundamentally difficult, though it requires a substantial effort (see Appel and Felty [AF00], for example). PCC certification for self-modifying code, however, is still largely unexplored, and we would be incurring a significant cost for standard programs by requiring additional proofs of instruction decodings.

6.2 Related Work

We touch here only on work related to security policies for untrusted software. For a more comprehensive PCC bibliography, we refer the reader to Necula [Nec98].

Necula and Lee [NL98] pioneered the use of PCC for resource bounds. Appel and Felty [AF00] argue that we should rely upon an encoding of the machine semantics in higher-order logic and derive an untrusted type system from it; the proof checker should be the only trusted component. Interesting safety properties can be specified by extending the machine model. In some respects, our work represents a less radical step in a similar direction: the enforcement mechanism disassembles the program, but does not to analyze its control flow or generate a VC.

The enforcement mechanism for *typed assembly language* (TAL) [MWCG98] is a type checker that does not accept unsafe programs; type annotations accompany program instructions. A TAL compiler translates a well-typed source program into a well-typed object program. Walker [Wal00] developed a TAL based on security automata; this version of TAL is novel because, like our system, the security policy is separate from the enforcement mechanism. Additionally, Walker provides an IRM transformation for ensuring that the security policy is always satisfied. Crary and Weirich [CW00] developed a TAL that enforces resource bounds. Crary, Walker, and Morrisett [CWM99] developed a TAL to enforce security policies based on a capability calculus; this calculus can ensure the safety of explicit deallocation.

Software fault isolation (SFI) [WLAG93, ALLW96] instruments a program so that it cannot violate a built-in memory safety policy. *Security automata SFI implementation* (SASI) is an SFI-based tool developed by Erlingsson and Schneider [ES00, ES99] for enforcing security policies encoded in a security-automata language.

The security policy of the Java Development Kit (JDK) 1.2 Security Model [GMPS97] is partially specified through configuration files. A *policy file* specifies which permissions a program receives based on predefined attributes (*e.g.*, its origin or digital signature). Other researchers (*e.g.*, PoET [ES00], J-Kernel [HCC+97], Naccio [ET99]) have developed extensions for more expressive security policies.

Acknowledgements

We thank Frank Pfenning for his guidance during the development of our temporal logic. We also thank Fred B. Schneider, Bob Harper, and the anonymous referees for helpful comments on earlier drafts of this paper.

References

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, Boston, MA, January 2000.

[ALLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 127–136, Philadelphia, PA, May 1996.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings, 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, June 2001.

[AS86] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical Report TR86-727, Cornell University, Computer Science Department, January 1986.

[BL01] Andrew Bernard and Peter Lee. Enforcing formal security properties. Technical Report CMU-CS-01-121, Carnegie Mellon University, School of Computer Science, April 2001.

[BPW01] Andrew Bernard, Frank Pfenning, and M. Angela Weiss. Natural deduction for temporal systems. Unpublished manuscript, 2001.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proceedings of the ACM SIGPLAN '00 conference on programming language design and implementation*, pages 95–107, Vancouver, BC Canada, June 2000.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, Boston, MA, January 2000.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, TX, January 1999.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, 1990.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, September 1999.

[ES00] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[ET99] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, pages 32–45, Oakland, CA, May 1999.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[HCC+97] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. Technical Report TR97-1660, Cornell University, Computer Science Department, December 1997.

[Kin71] J. C. King. Proving programs to be correct. *IEEE Transactions on Computers*, 20(11):1331–1336, November 1971.

[Koz98] Dexter Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, Computer Science Department, January 1998.

[MA00] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher order logic. In *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, June 2000.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In Cynthia Dwork, editor, *Proceedings of the 9th Annual ACM Symposium on Principles of Distribted Computing*, pages 377–408, Québec City, Québec, Canada, August 1990. ACM Press.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1991.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, CA, January 1998.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

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

[NL98] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[NR01] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, London, UK, January 2001.

[Pfe99] Frank Pfenning. Logical frameworks. In *Handbook of Automated Reasoning*, pages 1–82. Elsevier Science Publishers, 1999.

[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)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Sch99] Fred B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, Computer Science Department, July 1999.

[Sim94] Alex K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

[Wal00] David Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267, Boston, MA, January 2000.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.