Temporal Logic for Proof-Carrying Code

Andrew Bernard and Peter Lee*

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
(andrewb|petel)@cs.cmu.edu

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 Introduction

Proof-carrying code [11] (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.

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

Using PCC, a *code producer* provides an untrusted program to a *code consumer*. A trusted *enforcement mechanism* checks the program against one or more security policies before it is allowed to run.

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 [11]). We will argue here that *temporal logic* [8] 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* [1], 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 $\Box(\mathtt{pc} \geq 0 \wedge \mathtt{pc} < 100)$ 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 [7]. 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.

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 system 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. Due to space limitations, we must omit many important details in this paper, but a complete development will be available as a technical report [2].

2 Temporal Logic

2.1 Syntax

The syntax of our logic (see Figure 1) 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.[1] 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.

Times	t	$::= 0 \mid t_1 + 1$
Rigidities	ρ	$::= +_{\mathsf{r}} \mid -_{\mathsf{r}}$
Expressions	e^τ	$::= a^\tau \mid x^\tau \mid f^{\tau_1 \times \cdots \times \tau_k \to \tau}(e_1^{\tau_1}, \ldots, e_k^{\tau_k})$
Propositions	p	$::= R^{\tau_1 \times \cdots \times \tau_k \to o}(e_1^{\tau_1}, \ldots, e_k^{\tau_k}) \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid p_1 \supset p_2$ $\mid \forall x^\tau{:}\rho.\ p_1 \mid \exists x^\tau{:}\rho.\ p_1 \mid \Box p_1 \mid \bigcirc p_1 \mid p_1 \,\mathcal{U}\, p_2$
Core Judgments	J	$::= t_1 \geq t_2 \mid e{:}\rho \mid p{:}\rho \mid p \circledcirc t \mid p \circledcirc [t_1, t_2)$
Contexts	Γ	$::= \cdot \mid \Gamma,\ J$

Fig. 1. Abstract Syntax (Temporal Logic)

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 [8], 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

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

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.
- $\bigcirc p$ 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.

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.

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.

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).

Valuation A *valuation function* $\mathcal{V}$ assigns values to expressions (see the companion technical report). Thus, $\mathcal{V}(t)$ is the value of time expression t as a natural number. $\mathcal{V}_\phi$ 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.

Satisfaction 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 $\pi : +_\mathsf{r}$ when π is rigid.

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 2 for representative connectives: the complete definition can be found in the technical report); 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.
- $e : \rho$ holds when e denotes a sequence with rigidity ρ.
- $p : \rho$ holds when the truth/falsity of p has rigidity ρ.
- $p \circledcirc t$ holds when p is true at time t.
- $p \circledcirc [t_1, t_2)$ ("p is true over t_1 to t_2") holds when p is true at all times in the half-open interval $[t_1, t_2)$.

Thus, $\phi \vDash p \circledcirc t$ ("ϕ satisfies p at time t") holds if p is true of ϕ at time t.

$\phi \vDash R(e_1, \ldots, e_k) \circledcirc t$	iff $\langle \mathcal{V}_\phi(e_1)(\mathcal{V}(t)), \ldots, \mathcal{V}_\phi(e_k)(\mathcal{V}(t)) \rangle \in \mathcal{J}(R)$
$\phi \vDash p_1 \wedge p_2 \circledcirc t$	iff $\phi \vDash p_1 \circledcirc t$ and $\phi \vDash p_2 \circledcirc t$
$\phi \vDash \forall x^\tau : \rho.\ p \circledcirc t$	iff $\phi[a^\tau \mapsto \pi^\tau] \vDash [a^\tau / x^\tau]\, p \circledcirc t$ for some a^τ not appearing in p and all π^τ such that $\pi^\tau : \rho$
$\phi \vDash p_1 \,\mathcal{U}\, p_2 \circledcirc t$	iff $\phi \vDash p_2 \circledcirc t_2$ for some t_2 such that $\phi \vDash t_2 \geq t$ and $\phi \vDash p_1 \circledcirc [t, t_2)$
$\phi \vDash p \circledcirc [t_1, t_2)$	iff $\phi \vDash p \circledcirc t$ for all t such that $\phi \vDash t \geq t_1$ and $\phi \vDash t_2 \geq t + 1$

Fig. 2. The Satisfaction Relation (Excerpt)

2.3 Proof System

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

A *context* Γ is a collection of hypothetical judgments that weaken provability. For example, $a : +_r \vdash [a/x]\, p \circledcirc t$ 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).

We present our proof system in the technical report; we only claim here that it is sound with respect to the semantics.

Proposition 1 (Soundness). *$\phi \vDash J$ if $\phi \vDash \Gamma$ and $\Gamma \vdash J$*

Proof. See the technical report.

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`; $Val^{\texttt{wd}}$ is an initial subrange of the natural numbers. 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 $\texttt{r}_j$ 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[2]; programs are instruction sequences:

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

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 $\texttt{r}_0$ with its own factorial:

```
r1 ← 1              // r1 is current counter
r2 ← 1              // r2 is current product
r3 ← 1              // r3 is always one
r4 ← r1 gtw r0      // r4 is nonzero iff r1 > r0
cond neq0w r4, 3    // skip 3 when r4 is nonzero
r2 ← r2 mulw r1     // accumulate product
r1 ← r1 addw r3     // increment counter
cond truew r0, −5   // always skip back 5
r0 ← r2             // replace r0
halt
```

Our calling convention starts execution at the first instruction; `halt` is an abbreviation for `cond truew` $\texttt{r}_0, -1$. Program length ($|\Phi|$) and subscript (Φ_i) are defined in the obvious way.

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. $\texttt{selw}^{\texttt{mapw}\times\texttt{wd}\rightarrow\texttt{wd}}$ (apply map) and $\texttt{updw}^{\texttt{mapw}\times\texttt{wd}\times\texttt{wd}\rightarrow\texttt{mapw}}$ (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 $\texttt{selu}^{\texttt{mapu}\times\texttt{ureg}\rightarrow\texttt{wd}}$ (select register) and

[2] 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.

$\texttt{updu}^{\texttt{mapu}\times\texttt{ureg}\times\texttt{wd}\to\texttt{mapu}}$ (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.[3]

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.

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. The *execution set* Σ_p of a proposition p is the set of

[3] 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").

executions that satisfy it at time zero ($\Sigma_p = \{\sigma \mid \sigma \vDash p \circledcirc 0\}$). Given a security-property p, an execution σ does not violate security if and only if $\sigma \in \Sigma_p$. 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 3 for representative instructions: the complete definition can be found in the companion technical report). $i_1 \dot{+} i_2$ 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.

$\Phi \rhd s \to s'$	
$\Phi_{s(\texttt{pc})}$	s'
$r_1 \leftarrow r_2$	$s[\texttt{pc} \mapsto s(\texttt{pc}) \dot{+} 1][\texttt{u} \mapsto s(\texttt{u})[r_1 \mapsto s(\texttt{u})(r_2)]]$
$\texttt{cond}\ cop\ r_1, i_1$	$\begin{cases} s[\texttt{pc} \mapsto s(\texttt{pc}) \dot{+} 1 \dot{+} i_1] & \text{if } s(\texttt{u})(r_1) \in cop \\ s[\texttt{pc} \mapsto s(\texttt{pc}) \dot{+} 1] & \text{if } s(\texttt{u})(r_1) \notin cop \end{cases}$

Fig. 3. The Transition Relation (Excerpt)

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

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* (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.

For temporal-logic PCC, we provide a proof of $\vdash p_{\mathsf{sp}} \circledcirc 0$ 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 4 specifies two such rules (see the companion technical report for the remaining 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.

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

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

$$\frac{}{\Gamma \vdash p_{\mathsf{trans\_mv}} \circledcirc t}\ \mathsf{trans\_mv} \qquad \frac{}{\Gamma \vdash p_{\mathsf{trans\_cond}} \circledcirc t}\ \mathsf{trans\_cond}$$

Fig. 4. Encoding the Transition Relation (Excerpt)

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 2 (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* $\phi|_{Reg} \in \Sigma_{\mathcal{J}(\mathtt{pm})}$

Proof. By the definition of the transition relation and the definitions of valuation and satisfaction (see the technical report for details).

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 3 (Enforcement Soundness). $\Sigma_{\mathcal{J}(\mathtt{pm})} \subseteq \Sigma_{p_{\mathsf{sp}}}$ *if* $\vdash p_{\mathsf{sp}} \circledcirc 0$

Proof.

for all $\sigma \in \Sigma_{\mathcal{J}(\mathtt{pm})}$	
for all ϕ such that $\phi\|_{Reg} = \sigma$	
$\phi \vDash p_{\mathsf{sp}} \circledcirc 0$	Proposition 2 and Proposition 1
$\sigma \vDash p_{\mathsf{sp}} \circledcirc 0$	Def. $\sigma \vDash p \circledcirc t$
$\sigma \in \Sigma_{p_{\mathsf{sp}}}$	Def. Σ_p

□

Let $\Phi = \mathcal{J}(\mathtt{pm})$. The code producer provides a derivation of $\vdash p_{\mathsf{sp}} \circledcirc 0$ along with Φ; we use a trusted proof checker (*e.g*, Necula [14]) to verify its correctness. From Proposition 3, 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 [14] (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 [3, 14, 10] 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 [18, 5, 4].

Our certification mechanism generates derivations of judgments of the form $\vdash \mathtt{pc} = 0 \wedge p_{\mathsf{pre}} \supset \Box p_{\mathsf{safe}} \circledcirc 0$, where p_{pre} and p_{safe} are assertions (*i.e.*, $p_{\mathsf{sp}} \equiv \mathtt{pc} = 0 \wedge p_{\mathsf{pre}} \supset \Box p_{\mathsf{safe}}$); an *assertion* is a proposition that contains no temporal operators. This class of security properties represents a slight generalization of the *invariance properties* [8], 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 [11] to our machine model to fix the strategy of our proof generator (see the companion technical report for details).

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

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

We call this the *essential safety policy* [7]. 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 [14]; 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 $\vdash VC_{p_{\mathsf{pre}},\mathcal{I}} \circledcirc 0$.

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 [8]. However, instead of using the usual global invariance rule [8], 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 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 [9] achieve a similar reduction by factoring invariants using predicate transformers.

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

Proposition 4 (Relative Completeness). *There is an algorithm that derives* $\vdash \mathtt{pc} = 0 \wedge p_{\mathsf{pre}} \supset \Box p_{\mathsf{safe}} \circledcirc 0$ *from* $\vdash VC_{p_{\mathsf{pre}},\mathcal{I}} \circledcirc 0$*, where* p_{safe} *is the essential safety policy*

Proof. In the technical report, we specify an algorithm for deriving a temporal security proof from the proof of a corresponding first-order VC.

We have implemented a prototype proof generator for the abstract RISC processor as a logic program in the Twelf [15] meta-logical framework, along with a simulator for the enforcement mechanism. Small experiments based on a binary encoding of temporal logic suggest that the size of the temporal proof skeleton is less than five times the code size; this overhead is in addition to the first-order proof and the prelude. Though such proofs are relatively large by current standards [13], the experiments indicate that our approach is practical.

We are currently in the midst of implementing an experimental framework based on the SpecialJ certifying compiler for Java [3]. The SpecialJ compiler produces certified x86 executable code from Java class files; our new framework generates temporal-logic proofs from this certified code. When completed, this framework will allow us to obtain more comprehensive measurements of proofs sizes for Java programs. Initial results are so far consistent with the results of our earlier experiments.

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 specification language for security policies, including "expressive" [18, 16] 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.[4] However, the choice of notation 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 [14], but at the cost of additional trusted code and restrictions on control flow. It would be more satisfying to develop an untrusted mechanism based on new certification techniques, and thereby continue to use the same simple enforcement mechanism we have presented here. However, in order to prove the specification of a recursive procedure, we must be able to assume provisionally that the specification holds—thus, we may need fixed-point operators to encode the general case of mutually-recursive procedures.

We plan to adapt instrumentation techniques for security automata [16] to the certification problem. Security automata can specify all safety properties, and program transformations exist [6, 18] 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

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

fundamentally difficult, though it requires a substantial effort (see Appel and Felty [1], 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 [14].

Necula and Lee [12] pioneered the use of PCC for resource bounds. Appel and Felty [1] 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) [10] 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 [18] 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 [5] developed a TAL that enforces resource bounds. Crary, Walker, and Morrisett [4] 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) [17] 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 [6] for enforcing security policies encoded in a security-automata language.

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

1. 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.
2. Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. Technical Report CMU-CS-02-130, Carnegie Mellon University, School of Computer Science, 2002. In preparation.

3. 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.
4. 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.
5. 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.
6. 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.
7. Dexter Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, Computer Science Department, January 1998.
8. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1991.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Available as Technical Report CMU-CS-98-154.
15. 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.
16. Fred B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, Computer Science Department, July 1999.
17. 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.
18. 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.