Models for Security Policies in Proof-Carrying Code

Andrew W. Appel and Edward W. Felten
Dept. of Computer Science
Princeton University

March 29, 2001

Abstract

When an untrusted machine-language program runs on a host with resources that must be protected, the security policy specifies how the program may interact with the host. Proof-carrying code is a framework for automatically verifying that a program complies with a security policy. We show how to specify a wide variety of security policies, how to ensure that these specifications are consistent, how to develop compilers that produce compliant machine code and proofs of compliance, and how to reason about whether the host environment correctly implements its side of the policy.

1 Introduction

The early proof-carrying code (PCC) system designed by Necula and Lee [8] enforced a simple memory safety policy which imposed limits on which memory addresses the program could read, write, and execute. PCC uses a precise model of the machine's instruction set to prove properties of a program.

It was soon evident, however, that PCC could be applied in principle to a wider range of policies, including such rules as "no more than K instructions executed between coroutine yields," rules regarding memory safety in situations where the underlying platform made available calls to allocate and deallocate memory, and rules describable in terms of reference monitors or security automata [10]. All of these policies describe interactions between the program and its environment (which includes the host on which the program runs).

Though policies like these are easy to describe in words, it is not always obvious how to implement them in the framework of PCC. How can we be sure that a formalization of such a policy makes sense and matches our intuition about what the policy means?

We argue that the right way to do this is to construct a model for the state of the environment (hidden behind an API). This allows us to think more clearly about the inference rules of the safety policy, and indeed to prove them as theorems.

2 Examples

To motivate our discussion, we now present several policies of the type we would like to be able to enforce.

Periodic Coroutine Yields. In this example, we require that the program never execute more than some constant number K of instructions without making a call to a `yield` function. This provides a guarantee that other coroutines can make progress.

Memory Allocation and Deallocation. This example extends the standard memory-safety policy by adding library (or operating system) calls that allocate and deallocate memory.

Security Automaton for Correct Sequencing of API Calls. Following the work of Schneider [10], this example requires that the sequence of API calls made by the program correspond to the transitions of a security automaton. A

security automaton is a state machine that makes a transition every time the program makes a call to a built-in library API. If an illegal transition is attempted, the automaton will terminate the program instead of permitting the API call. Our implementation will allow wide latitude to code producers in choosing between static, dynamic, and hybrid methods of enforcing the automaton's required behavior. In particular, instead of requiring the trusted side of the API to implement a state machine, we can mechanically check that the client side faithfully implements it; then, in fact, the client's compiler can optimize away part or all of the security automaton while still preserving verifiable compliance. We come to praise security automata, and to bury them inside the client.

Limiting the Number of Locks Held. In this example, the program may acquire and release locks by making library calls, and we require that the program never hold more than some constant number K of locks (in fact, this policy can be expressed by a security automaton).

All Output Must Be Logged. This example requires the program to write a copy of all of its output to a special "log" output stream. This example (and the next one) cannot be expressed using security automata.

Packet Forwarding. Our final example allows the program to send and receive network packets, but every packet that is sent must be identical to some packet that was previously received. We could also specify that no packet is forwarded more than once.

3 A cautionary tale

Necula's Touchstone compiler [9] uses a safety policy in which there is a predicate readable(m,x), saying that address x is readable in memory m. Essentially, there is also a predicate safeExecute(r,m,p) meaning that it is safe to execute the program fragment p in the program state with registers r and memory m. A typical axiom of this system is

$$\frac{\text{safeExecute}(r[i{:=}m(x)],m,p) \quad \text{readable}(m,x)}{\text{safeExecute}(r,m,\ (r(i) \leftarrow m(x)); p)}$$

That is, if it's safe to execute p in a register bank where $r(i)$ has been set to the contents of $m(x)$, and if location x is readable, then it's safe to execute the load instruction $r(i) \leftarrow m(x)$ and continue with p.

Another rule says that if we can execute the *malloc* API call to get more memory. That is, if it is safe to execute a program `p` provided that register `r[i]` points to a block of allocated memory, then even from a state in which `r[i]` doesn't point to allocated memory, it is safe to execute the program `malloc(n);p`, which calls malloc and then executes `p`.

$$\frac{\begin{array}{c} n \geq 0 \\ \text{readable}(m,b\ldots b+n-1) \rightarrow \text{safeExecute}(r[i{:=}b],m,p)) \end{array}}{\text{safeExecute}(r,m,\ (r(i) \leftarrow \text{malloc}(n)); p)}$$

These are seemingly reasonable rules. But the analysis of such axioms that constitute a security policy is not easy: are the axioms consistent? What properties do they have?

One of Necula's breakthrough achievements in formulating the notion of proof-carrying code was to axiomatize the system very concretely and at a very low level – that of individual machine instructions and memory locations. But we will argue that his specification is actually too abstract! By writing an even more concrete axiomatization we make it easier to prove properties of the interface between the PCC host and the untrusted program.

For example, in Necula's Touchstone compiler the readable(m,x) predicate takes two arguments, the memory and the address in question. A year later, in Necula (and others') "Special J" Java compiler [2] the readable(x) takes only the address as an argument. Which version is best? Does the memory argument serve any purpose? What's the rationale for designing security policies?

The short answer is that if there is a *malloc* API function then the readable predicate should take the extra argument. We will first present our models of security policies, and then return to this example.

4 Axiomatization of instruction execution

We model a machine state as a register bank r and a memory m, each of which is a function from integer to integer [1, 6]. The register bank contains not only ordinary general-purpose registers $r(0), r(1), \ldots, r(15)$ (e.g., if there are 16 of them) but also dedicated registers such as the program counter and condition codes which we arbitrarily assign numbers (e.g., the program counter is $r(17)$; we define pc $= 17$).

We specify machine instruction decoding and execution via a single-step relation $(r,m) \mapsto (r',m')$, meaning that if r describes the contents of registers (including program counter) and m describes the memory, and one instruction executes, then the new state will be described by r' and m'.

To illustrate the specification of the step relation, we will use a Toy machine, a 16-bit word-addressed machine with simple instruction encodings [1].

	OPCODE	
add	0 d s_1 s_2	$r_d \leftarrow r_{s_1} + r_{s_2}$
addi	1 d s c	$r_d \leftarrow r_s \pm c$
load	2 d s c	$r_d \leftarrow m(r_s \pm c)$
store	3 s_1 s_2 c	$m(r_{s_2} \pm c) \leftarrow r_{s_1}$
jump	4 d s c	$r_d \leftarrow pc$; $pc \leftarrow r_s \pm c$
bgt	5 s_1 s_2 c	if $r_{s_1} > r_{s_2}$ then $pc \leftarrow pc \pm c$

On a von Neumann machine, each instruction is represented in memory by an integer. Our *decode* relation (Figure 2) is a predicate on two arguments (v,i) and says that v is the encoding of instruction i. Finally, the step relation $\mapsto$ expresses the idea of instruction fetch, instruction decode, and instruction execution.

Our step relation is purposely partial; that is, not every state r,m has a legal next step. For example, if $m(r(\text{pc})) = 7000_{16}$, an illegal opcode, there is no r',m' such that $r,m \mapsto r',m'$. Using this idea, we can express a safety predicate:

$$\text{safe}(r,m) = \forall r',m'.\ r,m \mapsto^* r',m' \ \rightarrow \\ \exists r'',m''.\ r',m' \mapsto r'',m''$$

That is, a machine state is safe if, in any state reachable by executing instructions from that state, there's always a legal next step.

A *program* is just a sequence of integers $p_0, p_1, ..., p_{n-1}$ (that code for instructions and data) to be loaded at a start address *start*. The initial condition is that the program is loaded and the program counter points to the entry point:

$$\text{initial}(p,r,m) = \\ r(\text{pc}) = start \wedge \forall i \in \text{dom}(p).\ m(start+i) = p_i.$$

The simplest possible security policy – that the program never executes an illegal instruction – is axiomatized by our definition of *safe*. A program p conforms to the policy if

$$\text{safeprog}(p) = \forall r,m.\ \text{initial}(p,r,m) \rightarrow \text{safe}(r,m)$$

The step relation $\overset{\text{d}}{\mapsto}$ defined arithmetically as in Figure 2 may be a subset of what the machine can legally execute; we denote the latter relation by $\mapsto$. We may wish to avoid axiomatizing the rare and complicated instructions that compilers don't generate in practice. By making this choice, we lose the ability to prove safe any program that executes those instructions, but any program we prove safe with respect to $\overset{\text{d}}{\mapsto}$ is also safe with respect to $\mapsto$.

An axiom relates $\overset{\text{d}}{\mapsto}$ to $\mapsto$:

$$\text{StepAxiom} : \forall r,m,r',m',r'',m''. \\ (r,m) \overset{\text{d}}{\mapsto} (r',m') \ \rightarrow \\ ((r,m) \mapsto (r',m') \wedge \\ ((r,m) \mapsto (r'',m'') \ \rightarrow \ r' = r'' \wedge m' = m''))$$

This says that if $\overset{\text{d}}{\mapsto}$ makes a step, then $\mapsto$ makes the same step and no other. But if $\overset{\text{d}}{\mapsto}$ makes no step, then $\mapsto$ is free to make any step.

5 Encoding security policies

Suppose we want to require that the program writes only to memory locations in a specified range. We can define a predicate writable(x); the security policy is that the program writes only to those addresses. We also require that the program reads only from addresses permitted by readable(x). To axiomatize this policy, we modify $\overset{\text{d}}{\mapsto}$ by changing the definition of the load and store instruction:

$$\mathrm{upd}(f,d,x,f') = \forall z.\,(d = z \wedge f'(z) = x) \vee (d \neq z \wedge f'(z) = f(z))$$
$$\mathrm{pc} = 17$$

$$\mathrm{add}(d,s_1,s_2)(r,m,r',m') = \mathrm{upd}(r,d,r(s_1) +_{16} r(s_2),r') \wedge m = m'.$$
$$\mathrm{addi}(d,s,c)(r,m,r',m') = \mathrm{upd}(r,d,r(s) +_{16} \mathrm{sext}(c),r') \wedge m = m'$$
$$\mathrm{load}(d,s,c)(r,m,r',m') = \mathrm{upd}(r,d,m(r(s) +_{16} \mathrm{sext}(c)),r') \wedge m = m'$$
$$\mathrm{store}(s_1,s_2,c)(r,m,r',m') = \mathrm{upd}(m,r(s_2) +_{16} \mathrm{sext}(c),r(s_1),m') \wedge r = r'$$
$$\mathrm{jump}(d,s,c)(r,m,r',m') = \exists r''.\ \mathrm{upd}(r,\mathrm{pc},r(s) +_{16} \mathrm{sext}(c),r'') \wedge \mathrm{upd}(r'',d,r(\mathrm{pc}),r') \wedge m = m'$$
$$\mathrm{bgt}(s_1,s_2,c)(r,m,r',m') =$$
$$(r(s_1) > r(s_2) \wedge \mathrm{upd}(r,\mathrm{pc},r(\mathrm{pc}) +_{16} \mathrm{sext}(c),r') \wedge m = m') \ \vee \ (r(s_1) \leq r(s_2) \wedge r = r' \wedge m = m')$$

Figure 1: Semantic definition of machine instructions. The symbol $+_{16}$ denotes addition modulo 2^k, and $\mathrm{sext}(c)$ is the sign-extension of c from 4 to 16 bits.

$$\mathrm{format}(w,a,b,c,d) = 0 \leq a < 16 \wedge 0 \leq b < 16 \wedge 0 \leq c < 16 \wedge 0 \leq d < 16 \wedge w = a * 16^3 + b * 16^2 + c * 16 + d.$$

$$\mathrm{decode}_k(v,i) = \ (\exists d,s_1,s_2.\ \mathrm{format}(v,0,d,s_1,s_2) \wedge i = \mathrm{add}_k(d,s_1,s_2))$$
$$\vee \ (\exists d,s_1,c.\ \mathrm{format}(v,1,d,s_1,c) \wedge i = \mathrm{addi}_k(d,s_1,c)) \vee ...$$

Step relation:

$$(r,m) \overset{\mathrm{d}}{\mapsto} (r',m') = \exists i,r''.\mathrm{decode}_{16}(m(r(\mathrm{pc})),i) \wedge \mathrm{upd}(r,\mathrm{pc},r(\mathrm{pc}) +_{16} 1,r'') \wedge i(r'',m,r',m')$$

Figure 2: Instruction decoding and step relation.

$$\begin{aligned}
&\text{load}(d,s,c)(r,m,r',m') = \\
&\quad \text{readable}(r(s)+_{16}\text{sext}(c))\wedge \\
&\quad \text{upd}(r,d,m(r(s)+_{16}\text{sext}(c)),r')\wedge m = m' \\
&\text{store}(s_1,s_2,c)(r,m,r',m') = \\
&\quad \text{writable}(r(s_2)+_{16}\text{sext}(c))\wedge \\
&\quad \text{upd}(m,r(s_2)+_{16}\text{sext}(c),r(s_1),m')\wedge r = r'
\end{aligned}$$

The predicates *readable* and *writable* – originally introduced by Necula [8] – help modularize the security policy. The axioms about machine instructions (in our case, the $\overset{\text{d}}{\mapsto}$ relation and its subsidiary definitions of *load* and *store*) refer to the abstract predicates, and an application-specific policy can then axiomatize these predicates separately.

In effect, the notion of "legal machine instruction" implicit in the definition of $\text{safe}(r,m)$ is an instruction that conforms to the security policy.

Axiomatizing API calls. Suppose the host environment provides a function *print* at location l_{print}. The program is permitted to jump to l_{print} with an integer in register $r(1)$ and a return address in register $r(7)$. The host will then print the integer and jump back to the return address, and in the process will not modify any readable location. The host may modify parts of memory not visible to the program, i.e., parts that the security policy does not permit the program to load. We can axiomatize *print* as follows:

$$\begin{aligned}
&\text{agree}(S,m,m') = \forall x \in S.\ m(x) = m'(x) \\
&\text{PrintAxiom1}:\ \forall(r,m).\ r(\text{pc}) = l_{\text{print}}\ \rightarrow \\
&\quad \exists r',m'.\ (r,m)\mapsto^* (r',m')\wedge r'(\text{pc}) = r(7)\wedge \\
&\qquad \text{agree}(\{0,\ldots,\text{pc}-1\},r,r')\wedge \\
&\qquad \text{agree}(\mathit{readable},m,m')
\end{aligned}$$

This assumes that any call to *print* will return. But perhaps the host has the option of terminating any program at an API call. Then what we want to say is that *print* may or may not return, but if the return address is a "safe" program location, then the call to *print* is safe:

$$\begin{aligned}
&\text{PrintAxiom2}:\ \forall(r,m).\ r(\text{pc}) = l_{\text{print}} \\
&\quad \wedge\ (\forall r',m'.\text{agree}(\text{readable},m,m') \\
&\qquad \wedge r'(pc) = r(7) \\
&\qquad \wedge \text{agree}(\{0,\ldots,\text{pc}-1\},r,r') \\
&\qquad \rightarrow \text{safe}(r',m')) \\
&\quad \rightarrow\ \text{safe(r,m)}
\end{aligned}$$

The *print* call has an effect on the outside world, but it makes no change to the state of the program itself (other than changing the program counter). Of course, most API calls have some effect on the program state. We can specify this by a relation $(r,m)\overset{?}{\mapsto}(r',m')$. For example, the *read* call that reads $n = r(2)$ words into memory starting at the address given by $r(1)$ is specified by the relation:

$$\begin{aligned}
&\text{readRel}(r,m,r',m') = \\
&\quad \text{agree}(\{0,\ldots,\text{pc}-1\},r,r')\wedge \\
&\wedge\,\text{agree}((\text{readable}-\{r(1),\ldots,r(1)+r(2)-1\}),m,m')
\end{aligned}$$

which says that every readable location except the target of the read is unchanged by the call.

Some API calls have preconditions; for example, suppose it is illegal to call *read* with a negative n. We can specify the precondition as a predicate of r,m:

$$\begin{aligned}
&\text{readCond}(r,m) = \\
&\quad r(2)\geq 0 \\
&\wedge\ \ \{r(1),\ldots,r(1)+r(2)-1\}\subset \text{readable}
\end{aligned}$$

Now, given a location, precondition, and relation, we can specify any API call:

$$\begin{aligned}
&\text{API}(l,\mathit{precond},\mathit{rel}) = \\
&\quad \forall r,m.\ r(\text{pc}) = l \\
&\quad \wedge\ \ \mathit{precond}(r,m) \\
&\quad \wedge\ \ (\forall r',m'.\ r'(pc) = r(7)\wedge \mathit{rel}(r,m,r',m') \\
&\qquad\qquad \rightarrow\ \text{safe}(r',m')) \\
&\quad \rightarrow\ \ \text{safe}(r,m)
\end{aligned}$$

Thus, the axiomatization of *read* is

$$\text{ReadAxiom}:\ \text{API}(l_{\text{read}},\text{readCond},\text{readRel})$$

Axiomatizing malloc. Suppose there is an API call *malloc* that the program may use to request n more words of readable and writable memory. We could attempt to specify *malloc* using a precondition and relation by saying that before the call $r(1) = n$, and after the call $r'(1)$ points to a block of readable/writable memory.

This attempt at axiomatizing *malloc* is inadequate:

$$\begin{aligned}
&\text{MallocCond1}(r,m) = \ r(1) \geq 0 \\
&\text{MallocRel1}(r,m,r',m') = \\
&\quad \text{agree}(\{0,\ldots,\text{pc}-1\},r,r') \wedge \\
&\quad \text{agree}(\text{readable},m,m') \wedge \\
&\quad \{r'(1),\ldots,r'(1)+r(1)-1\} \subset (\text{readable} \cap \text{writable})
\end{aligned}$$

We would like to know that the block $\{r'(1),\ldots,r'(1)+n-1\}$ is new memory, that it doesn't overlap with what we already have. This isn't specified in MallocRel1, so it will be difficult to prove that programs are safe, since we won't be able to prove that stores to location $r'(1)$ don't overwrite other data whose invariance we might wish to preserve.

In fact, since the readable predicate doesn't have any arguments except the address tested for readability, it is invariant over machine states: if an address is readable now, it has always been readable and will always be readable. Malloc seems an impossibility.

Necula's Touchstone compiler [9] has the readable predicate take the memory portion m of the machine state (r,m) as an additional argument: SafeRd(m,x). This allows the readable predicate to evolve with machine states, although Necula does not explicitly discuss the rationale for this choice. What's strange is that Necula and Lee's later work, the Special J system, uses a safety policy with an unparametrized *SafeRd*(x), and also has a *malloc* call. This change was made to make proofs simpler: whenever the program does a store to some writable location of memory, in effect changing m to m', one has the burden of proving that for all x, *SafeRd*$(m,x) \ \rightarrow$ *SafeRd*(m',x).

But Special J's model of allocation is not very flexible or extensible. Any address returned by *malloc* has always been readable, it's just that the program could never prove it "until now." Specifying a *free* function that returns memory to the host will be impossible in this model, as will any policy about how many words the program may or must allocate. We speculate that the lack of an explicit model for the SafeRd predicate made it difficult for the designers of Special J to analyze the consequences of the decision to drop the extra argument.

A slight inconsistency. The Special J security policy is consistent as written. But the less-than-obvious model of a *malloc* that does not actually cause more words to be readable/writable can be a minefield for later maintainers of the system who might need to extend the policy.

To illustrate this, we construct a contrived example. Some time in the future, suppose a designer extends Special J's security policy by adding the following two rules:

1. The program may not call *exit* until at least 30 words are readable.

2. The host guarantees that at least 100 words are allocable (via calls to *malloc*).

The intention is that the program will call *malloc* at least once. But with a state-invariant readable predicate, if 30 words will be readable in the future, they are readable now! Therefore the program that calls *exit* immediately can be proved safe.

A more natural way to describe *malloc*, which does not permit such paradoxes, is to parametrize the predicates. That is, let readable(r,m) be the set of readable addresses in the state r,m, and writable(r,m) be the writable addresses. We can describe *malloc* as,

$$\begin{aligned}
&\text{disjUnion}(A,B,C) = A \cup B = C \wedge A \cap B = \emptyset \\
&\text{MallocCond2}(r,m) = \ r(1) \geq 0 \\
&\text{MallocRel2}(r,m,r',m') = \\
&\quad \text{agree}(\text{readable}(r,m),m,m') \\
&\wedge \ \text{agree}(\{0,\ldots,\text{pc}-1\},r,r') \\
&\wedge \ \text{disjUnion}(\text{readable}(r,m),\ \{r'(1),\ldots,r'(1)+r(1)-1\}, \\
&\qquad\qquad \text{readable}(r',m')) \\
&\wedge \ \text{disjUnion}(\text{writable}(r,m),\ \{r'(1),\ldots,r'(1)+r(1)-1\}, \\
&\qquad\qquad \text{writable}(r',m')) \\
&\\
&\text{MallocAxiom} : \text{API}(l_{malloc},\text{MallocCond2},\text{MallocRel2})
\end{aligned}$$

6 Proving consistency

A security policy is a collection of axioms. Generally in logic if a set of axioms is not consistent then it will be possible to prove *false*, and from *false* anything may be proved. For proof-carrying code this means that an inconsistent policy allows any program to run and provides no security. Therefore we are very interested in proving the consistency of our security policies.

A standard way of proving the consistency of a logic is by induction over proofs. That is, we show for each n that any proof of size n does not prove a bad thing. This is the kind of reasoning that type theorists use in proving the soundness of type systems, and that Necula uses in proving the soundness of proof-carrying code [9].

But there is another way. One can define each primitive of logic $\mathcal{A}$ in terms of some simpler logic $\mathcal{L}$, and show that each inference rule of the $\mathcal{A}$ is a theorem of the $\mathcal{L}$. If logic $\mathcal{L}$ is known to be sound, then $\mathcal{A}$ must also be sound. In effect, we use logic $\mathcal{L}$ to describe a model of each formula of logic $\mathcal{A}$.

This is the approach we have used to model type systems for proof-carrying code [1], and we will use it here to model security policies. The logic $\mathcal{L}$ in our case will be Church's higher-order logic, proved sound to everyone's satisfaction in the 1950's.

What are the primitives of our logic $\mathcal{A}$? The step relations $\mapsto, \overset{\mathrm{d}}{\mapsto}$, the predicates readable(r,m,x), writable(r,m,x), safe(r,m), and so on. For each primitive we must provide a definition in Church's logic, and each axiom we must prove as a theorem.

Fortunately, we have already defined $\overset{\mathrm{d}}{\mapsto}$, safe$(r,m)$, agree$(S,m,m')$, and most of the other predicates in $\mathcal{L}$. We must now define $\mapsto$, readable(r,m,x), and writable(r,m,x).

7 The hidden-variables model of machine state

Imagine a computer with memory-protection hardware that can protect ranges of memory with one-byte granularity. Given a program, suppose we could prove that it will never experience a protection violation running on that hardware. Then we could delete the protection hardware from the machine, and instead run the program on stock hardware; the byte-level protection would persist only as a figment of our imagination, a useful fiction that helps us specify the memory policy.

We will model the readable/writable predicates by assuming that the processor has a pair of registers $r(71)$ and $r(72)$ that serve as the memory map. These registers are not architecturally visible to the user-mode program; that is, none of the instructions in the $\overset{\mathrm{d}}{\mapsto}$ relation read or write them. The address x is readable if the corresponding bit in $r(71)$ is 1:

$$\begin{array}{l} \mathrm{rd} = 71 \quad \mathrm{wr} = 72 \\ \mathrm{bit}(x,i) = (x/2^i \bmod 2 = 1) \\ \mathrm{readable}(r,m,x) = \mathrm{bit}(r(\mathrm{rd}),x) \\ \mathrm{writable}(r,m,x) = \mathrm{bit}(r(\mathrm{wr}),x) \end{array}$$

Even though no real machine has a "register 71", the very existence of these formulas can help us prove the consistency of the security policy. If we can prove all the "axioms" about readable/writable as derived lemmas from these definitions, then the policy must be consistent.

Our *readable* and *writable* predicates are parameterized by register-bank r, memory m, and address x. In this example we have not used m in the right-hand side of the definitions. Its pro-forma presence in the formal parameters gives the flexibility to use other models, such as one in which a page table is kept in some portion of the memory itself (presumably in an area not writable by the client).

We now show how to use the hidden-variables model to prove consistency of a policy. We start by defining an APIstep predicate, which expresses what it means to make a system call that either returns or freezes. We model freezing by an infinite loop, though in real life it would be equivalent to terminating the user process.

$$\begin{array}{l} \mathrm{APIstep}(l, precond, rel)(r,m,r',m') = \\ \quad r(\mathrm{pc}) = l \wedge r'(\mathrm{pc}) = r(7) \wedge precond(r,m) \\ \quad \wedge\, (rel(r,m,r',m') \\ \qquad \vee\, (\neg\exists r'', m''.\ rel(r,m,r'',m'')) \wedge r' = r \wedge m' = m) \end{array}$$

Next we define a new MallocRelation that is much like MallocRel2, but permits registers 71 and 72 to change:

$$
\begin{array}{l}
\text{MallocRel3}(r,m,r',m') = \\
\quad \text{agree}(\text{readable}(r,m),m,m') \\
\wedge\ \text{agree}(\{0,\ldots,\text{pc}-1,\text{pc}+1,\ldots,\text{rd}-1, \\
\qquad\qquad \text{wr}+1,\ldots\infty\},r,r') \\
\wedge\ \text{disjUnion}(\text{readable}(r,m),\ \{r'(1),\ldots,r'(1)+r(1)-1\}, \\
\qquad \text{readable}(r',m')) \\
\wedge\ \text{disjUnion}(\text{writable}(r,m),\ \{r'(1),\ldots,r'(1)+r(1)-1\}, \\
\qquad \text{writable}(r',m'))
\end{array}
$$

We define a step relation $\mapsto$ that is an extension of the machine-instruction step relation $\overset{\text{d}}{\mapsto}$.

$$
\begin{array}{l}
(r,m) \mapsto (r',m') = \\
\quad r(\text{pc}) \notin \{l_{\text{print}}, l_{\text{read}}, l_{\text{malloc}}\} \wedge (r,m) \overset{\text{d}}{\mapsto} (r',m') \\
\quad \vee\ \text{APIstep}(l_{\text{print}}, \text{PrintCond}, \text{PrintRel})(r,m,r',m') \\
\quad \vee\ \text{APIstep}(l_{\text{read}}, \text{ReadCond}, \text{ReadRel})(r,m,r',m') \\
\quad \vee\ \text{APIstep}(l_{\text{malloc}}, \text{MallocCond2}, \\
\qquad\qquad \text{MallocRel3})(r,m,r',m')
\end{array}
$$

We assume that $l_{\text{print}}, l_{\text{read}}, l_{\text{malloc}}$ are all distinct addresses.

Now we can prove each of the axioms shown in section 5. For example,

$$\text{MallocAxiom}:\ \text{API}(l_{\text{malloc}}, \text{mallocCond2}, \text{mallocRel2})$$

Proof: We must show

$$
\begin{array}{l}
\forall r,m.\ r(\text{pc}) = l_{\text{malloc}} \\
\quad \wedge\ \ \text{MallocCond2}(r,m) \\
\quad \wedge\ \ (\forall r',m'.\ r'(pc) = r(7) \wedge \text{MallocRel2}(r,m,r',m') \\
\qquad\qquad \rightarrow\ \text{safe}(r',m')) \\
\quad \rightarrow\ \text{safe}(r,m)
\end{array}
$$

Given r,m that satisfy the premises $r(\text{pc}) = l_{\text{malloc}}$, MallocCond2$(r,m)$, and the premise

$$
\begin{array}{l}
\forall r',m'.\ r'(\text{pc}) = r(7)\ \wedge\ \text{MallocRel2}(r,m,r',m') \\
\qquad \rightarrow\ \text{safe}(r',m')
\end{array}
$$

it suffices to show safe(r,m).

Now, either there is a free block of memory of size $r(1)$ or not. If there is, say it is at location b. then let r' be created from r by setting $r'(\text{pc})$ to $r(7)$, setting $r'(1)$ to b, and setting all bits $\{b,\ldots,b+r(1)-1\}$ of $r'(\text{rd})$ and $r'(\text{wr})$ to 1. Let $m' = m$. Then, by the definition MallocRel2, it is the case that MallocRel2(r,m,r',m'), and thus by our premise, safe(r',m'). But now, by the definition of APIstep, we find that APIstep$(l_{\text{malloc}}, \mathit{MallocCond2}, \mathit{MallocRel3})(r,m,r',m')$, and therefore $r,m \mapsto r',m'$. This is the only clause of the step relation that can match the state (r,m). Thus in the state r,m any step leads to a safe state, so therefore r,m is safe.

Now suppose instead that there is no free block of size $r(1)$. By the definition of MallocRel2 we find that MallocRel2(r,m,r,m) and APIstep$(l_{\text{malloc}}, \mathit{MallocCond2}, \mathit{MallocRel3})(r,m,r,m)$. Again, no other clause of the step relation matches, and we have $(r,m) \mapsto (r,m)$. By the definition of safety, we find that an infinite loop is safe, so r,m is safe.

Although this argument is tedious, it is fairly straightforward, because we can prove the soundness of the MallocAxiom without considering all the other rules at the same time.

8 Encodings of each example

Using the hidden-variables approach, we can specify many kinds of security policies.

Periodic Coroutine Yields. The program must not execute more than K of instructions without making an API call. We model this by introducing a instruction-count register $r(\text{ic})$ which is set equal to K by all API calls, and is decremented by the $\overset{\text{d}}{\mapsto}$ clause of the $\mapsto$ relation, as indicated by the newly added *upd* clauses in the step relation:

$$
\begin{array}{l}
(r,m) \mapsto (r',m') = \\
\quad r(\text{pc}) \notin \{l_{\text{print}}, l_{\text{read}}, l_{\text{malloc}}\}\ \wedge\ r(\text{ic}) > 0 \\
\qquad \wedge\ \exists r''.\ \text{upd}(r,\text{ic},r(\text{ic})-1,r'')\ \wedge\ (r'',m) \overset{\text{d}}{\mapsto} (r',m') \\
\vee\ \exists r''.\ \text{APIstep}(l_{\text{print}}, \text{PrintCond}, \text{PrintRel})(r,m,r'',m') \\
\qquad \wedge\ \ \text{upd}(r'',\text{ic},K,r') \\
\vee\ \exists r''.\ \text{APIstep}(l_{\text{print}}, , \ldots,
\end{array}
$$

Memory Allocation and Deallocation. We have shown *malloc*. Specifying a *free* function is straightforward, using the same two hidden variables, $r(\text{rd})$ and $r(\text{wr})$.

Security Automaton for Correct Sequencing of API Calls. According to Schneider's description of security automata [10], whenever the program makes an API call, the host simulates a transition in an automaton to check whether the call is in a legal sequence. In our version, we will statically prove that only legal sequences of calls can occur.

As usual with proof-carrying code, the static proof may rely on dynamic checks made by the program. But the code producer is free to find clever ways to optimize the dynamic bookkeeping and checking, as long as it can still prove statically that the resulting program is safe. Thus we do not constrain the code producer's options regarding how to enforce correct sequencing.

Assume the security automaton is represented by a graph G with integer node labels, and edges labeled by the addresses (l_{read}, l_{print}, etc.) of API functions. A transition from n_1 to n_2 labelled by l is represented by $(n_1, l, n_2) \in G$. We add a hidden variable $r(\text{s})$ to hold the state of the automaton.

Then the APIstep relation can be defined as:

$$\begin{aligned}&\text{APIstep}(l, precond, rel)(r, m, r', m') = \\ &\quad (r(\text{s}), l, r'(\text{s})) \in G \\ &\quad \wedge\ r(\text{pc}) = l \wedge r'(\text{pc}) = r(7) \wedge precond(r, m) \\ &\quad \wedge (rel(r, m, r', m') \\ &\qquad \vee (\neg \exists r'', m''.\ rel(r, m, r'', m'')) \wedge r' = r \wedge m' = m)\end{aligned}$$

where the only addition to our original APIstep is the first line.

For example, suppose that in state r, m the automaton has reached a state $r(\text{s}) = n_1$ where there is no transition $(n_1, l_{\text{print}}, n_2) \in G$ for any n_2. Then $\text{APIstep}(l_{\text{print}}, \text{PrintCond}, \text{PrintRel})$ will be unsatisfiable, and there will be no transition $(r, m) \mapsto (r', m')$.

Limiting the Number of Locks Held. Suppose there is an instruction or an API call that acquires a lock, and another instruction that releases a lock. We can specify the policy that the program may hold no more than K locks simultaneously by introducing a hidden variable $r(\text{locks})$ that tells how many locks are held. Acquiring a lock increments the variable, releasing the lock decrements it. The precondition for making an acquire transition in the $\mapsto$ relation is that $r(\text{locks}) < K$.

All Output Must Be Logged. We can require the program to write a copy of all of its output to a special "log" output stream. One way to do this is to have a hidden variable $r(\text{out})$ that holds all the output sent to the output stream, and another $r(\text{log})$ that holds the history of output to the log. Since our logic can reason about arbitrary-size (i.e., unbounded precision) integers, one "register" can hold the entire contents of a file.

Packet Forwarding. We can allow the program to send and receive network packets, but every packet that is sent must be identical to some packet that was previously received. This kind of policy prevents network routers from "spoofing" recipients by inventing packets. Again, the entire history of packets sent can be kept in a single "register."

In such an example, one need not worry about the efficiency of simulating registers with unbounded integers – for no one is simulating the machine. We are only proving theorems about the execution of this notional machine, and the size of the theorem is independent of the size of the "integer" that might accumulate in the register.

9 Encodings as reference implementations

By demonstrating a step $\mapsto$ relation as a formula in higher-order logic, we can prove each of the "axioms" of our security policy, thereby ensuring the consistency of the policy. But what if the formula is unsatisfiable? That is, what if there are no 4-tuples (r, m, r', m') such that $(r, m) \mapsto (r', m')$? Then our policy is consistent, but trivially so: No program can be proved safe.

Or the policy may be only partially incorrect. For example, suppose $\overset{\text{d}}{\mapsto}$ is correctly specified, but a mistake in the definition of MallocRelation causes just that relation to be unsatisfiable. Then programs that don't use *malloc* can be proved safe, but programs that do use *malloc* cannot be proved safe.

One way to gain confidence in the specifications of the API functions is to implement them. We then show that this reference implementation is a valid model – that all the

axioms are provable as derived lemmas. We will show this method for *malloc*.

We have been discussing host environments that use proof-carrying code (instead of hardware memory protection) to enforce the readable/writable rules. Such a host probably has a data structure describing what parts of memory are considered readable by the client program. This data structure lives in memory, not in some "magic" register $r(\mathrm{rd})$. In particular, it must live in a part of memory not writable by the client program. (Otherwise, the client program could modify this data to give itself permission to read and write all of memory, which is almost certainly a policy violation.)

So let us abandon the infinite-precision registers $r(\mathrm{rd})$ and $r(\mathrm{wr})$, and remodel the readable predicate using a simple in-memory data structure. Let v_{lo} and v_{hi} be two memory addresses that implement this data structure. Initially, the program is loaded at addresses $p_{\mathrm{lo}}, \ldots, p_{\mathrm{hi}} - 1$, with $v_{\mathrm{lo}} < v_{\mathrm{hi}} < l_{\mathrm{malloc}} < p_{\mathrm{lo}}$, and in the initial memory, $m(v_{\mathrm{lo}}) = m(v_{\mathrm{hi}}) = p_{\mathrm{hi}}$.

We can now define

$$\mathrm{readable}(r,m,x) = m(v_{\mathrm{lo}}) \le x < m(v_{\mathrm{hi}})$$

$$\mathrm{writable}(r,m,x) = m(v_{\mathrm{lo}}) \le x < m(v_{\mathrm{hi}})$$

We modify the step relation $\mapsto$ to remove the $\mathrm{APIstep}(l_{\mathrm{malloc}}, \ldots, \ldots)$ clause, and we remove the restriction in the first clause that $r(\mathrm{pc}) \neq l_{\mathrm{malloc}}$. That is, when $r(\mathrm{pc}) = l_{\mathrm{malloc}}$ the ordinary instruction-execution $\overset{\mathrm{d}}{\mapsto}$ is permitted to apply.

At location l_{malloc} we put actual machine instructions that will implement the malloc function. That is, the function receives a value n in register $r(1)$, adds n to $m(v_{\mathrm{hi}})$, and sets $r(1)$ to the old value of $m(v_{\mathrm{hi}})$.

Our obligation now is to prove the MallocAxiom with this new step relation. This will be trivial but tedious. However, such a proof now gives us confidence that *malloc* is correctly specified, since an implementation we understand matches the specification.

The malloc function we have described and the two-word data structure that goes with it are much simpler than the malloc function that a real implementation would use. But it is not meant to be a real implementation; it is meant only as a reference implementation. Having proven the consistency of a security policy, we can then delete the hidden-variable model, and treat the (proven) theorems as axioms.

Validating the host software. Because it is the client, not the host, that comes from an untrusted source, we have been focusing on proving that the client program is safe with respect to a security policy. But even though the host's software is implemented by a trusted party, it can't hurt to prove its correctness. Our semantic model of the policy, especially with a reference implementation as part of the model, can be used to construct such proofs.

10 Compiling with proofs

We have described how to specify the safety theorem that an untrusted program must satisfy. But how is such a theorem to be proved? Program verification is a difficult business, and we must take care not to get stuck in a quagmire. The solution is for the producer of the program to generate the code in a controlled way, by using special compilers.

Memory safety. Necula [8] demonstrated a method to generate programs with proofs of memory safety. First, the program is written in a type-safe source language. Then, it is compiled to machine code using typed intermediate languages. [5] The lowest-level typed intermediate language – called *typed assembly language* (TAL) [7] – has a type system with a soundness proof (if a TAL program type-checks it can't "go wrong"). The type judgements in the type-checking of the TAL program provide the information necessary to construct the loop invariants of the machine-language safety proof. To make our extremely general policy-specification language practical, it is almost essential to use the type-preserving compilers pioneered by Morrisett et al. [7].

***K* instructions between API calls.** For proofs of memory safety we do not rely on the programmer to write memory-safe code — instead, the programmer writes code that type-checks, and the compiler transforms it to memory-safe code. Similarly, we will not rely on the programmer to make API calls at regular intervals — we will make sure the compiler

generates good code. It will still be necessary, to prove that the resulting program obeys the security policy, but the proof can be automatically generated because the automatically inserted code has a regular structure.

Feeley [4] showed how compilers can automatically generate efficient code that does polling: at regular intervals it will perform some action such as testing the contents of a memory location. We can adapt this method as follows. A register (or memory location) c is dedicated by the compiler to hold an instruction count. Whenever the compiled code enters a basic block with n instructions, it decrements c by n and tests $c < 0$. When $c < 0$ it calls the null API function (which we might call *yield*) and resets c to K. Dataflow analysis and other optimization methods described by Feeley can hoist the decrement-and-tests out of loops and out of small functions.

Another to compiling resource-bounded code (and generating proofs) is to use the type system of Crary and Weirich [3].

Number of locks. This policy is easy for the program to self-check. A memory location c tells the number of locks held by the program. The standard library wraps the API call (or instruction) that acquires a lock with code that increments c, and if $c > N$ it halts the program instead of proceeding with the call. The release-lock instruction is similarly wrapped.

Security automaton. There are several approaches to enforcing the policy that API calls must be sequenced according to a security automaton. In one simple approach, we require the program itself to simulate the automaton. A memory location c holds the current state number; another area of memory holds the transition table. Each API function call is wrapped with code that causes c to make the transition from one state to another. Walker [11] explains a general approach to encoding security automata in the type system of the client's type-preserving compiler; this is the enabling client-side technology that allows proofs to be constructed for automaton-based policies, just as TAL enables proofs for memory-safety policies.

Other policies. All the security policies we have discussed are constructively satisfiable: that is, there is an automated method for generating code that is provably safe according to the policy. This is a practically necessary criterion for policy; constructing proofs by hand is not something we can expect of software engineers.

11 Conclusion

In Necula's thesis, the `readable` predicate took the memory as an argument. In the Cedilla Systems alpha product, readable did not take memory as an argument. Lacking the hidden variables model of state, it was difficult to reason about how to formulate the security policy. In hindsight, it is obvious that the predicate must take the state as an argument if there is a malloc function. But without the "hidden variables" model, it wasn't so easy to tell.

The strength of the semantic approach is to make the soundness of the axioms obvious. Although the proofs (such as the one in section 6) can be tedious, we have found the relationship between the semantic definition of the predicate (such as *readable*) and the axioms (such as MallocAxiom) are intuitively clear.

The "hidden variables" approach works so well because the host really does have hidden state that is changed by the API calls. Our hidden variables are just abstractions of that hidden state.

Our specification language for security policies – higher-order logic – is more general than any of the several specific policy languages we have cited: typed assembly language [7], resource bound certification [3], security automata [10], and the security-automata dependent-type system [11]. But since our logic is not decidable, automatically generating proofs of *arbitrary* programs would be impossible. But each of these specific policy languages can be used as a compilation technology to generate nonarbitrary, policy-compliant machine code along with a compliance proof.

References

[1] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, January 2000.

[2] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*. ACM Press, June 2000.

[3] Karl Crary and Stephanie Weirich. Resource bound certification. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198. ACM Press, January 2000.

[4] Marc Feeley. Polling efficiently on stock hardware. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, pages 179–187. ACM Press, 1993.

[5] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. Technical Report CMU-CS-94-185, School of Computer Science, Carnegie Mellon University, September 1994.

[6] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, June 2000.

[7] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

[8] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[9] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[10] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1), February 2000.

[11] David Walker. A type system for expressive security policies. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267. ACM Press, January 2000.