A Trustworthy Proof Checker

Andrew W. Appel* Princeton University
Neophytos Michael* Princeton University
Aaron Stump† Stanford University
Roberto Virga* Princeton University

Princeton CS TR-648-02, April 16, 2002

Abstract

Proof-Carrying Code (PCC) and other applications in computer security require machine-checkable proofs of properties of machine-language programs. The main advantage of the PCC approach is that the amount of code that must be explicitly trusted is very small: it consists of the logic in which predicates and proofs are expressed, the safety predicate, and the proof checker. We have built a minimal-TCB checker, and we explain its design principles, and the representation issues of the logic, safety predicate, and safety proofs. We show that the trusted code in such a system can indeed be very small. In our current system the TCB is less than 2,700 lines of code (an order of magnitude smaller even than other PCC systems) which adds to our confidence of its correctness.

1 Introduction

Machine-verified proofs have applications in computer security, program verification, and the formalization of mathematics. In these applications, but especially in security applications such as proof-carrying code and proof-carrying authorization, the proof checker is an essential component of the trusted computing base: a bug in the proof checker can be a security hole in the larger system. Therefore, the checker must be trustworthy: it must be small, simple, readable, and based on well-understood engineering and mathematical principles.

In contrast, theorem provers are often large and ugly, as required by the incompleteness results of Gödel and Turing: no prover of bounded size is sufficiently general, but one can always hack more features into the prover until it proves the desired class of theorems. It is difficult to fully trust such software, so some proving systems use technical means to ensure that buggy provers cannot produce invalid proofs: the abstract data type *theorem* of LCF [13], or the proof-witness objects of Coq [7] or Twelf [19]. With these means, only a small part of a large system must be examined and trusted.

How large is the proof checker that must be examined and trusted? To answer this question we have tried the experiment of constructing and measuring the *smallest possible* useful proof checker for some real application. Our checker receives, checks the safety of, and executes, proof-carrying code: machine code for the Sparc with an accompanying proof of safety. The proof is in higher-order logic represented in LF notation.

The checker would also be directly useful for proof-carrying authorization [3, 8], that is, checking proofs of authentication and permission according to some distributed policy.

A useful measure of the effort required to examine, understand, and trust a program is its size in (non-blank, non-comment) lines of source code. Although there may be much variation in effort and complexity per line of code, a crude quantitative measure is better than none. It is also necessary to count, or otherwise account for, any compiler, libraries, or supporting software used to execute the program, and we address this issue explicitly.

The *trusted computing base* (TCB) of a proof-carrying code system consists of all code that must be explicitly trusted as correct by the user of the system. In our case the TCB consists of two pieces: first, the specification of the safety predicate in higher-order logic, and second, the proof checker, a small C program that checks proofs, loads, and executes safe programs.

In his investigation of Java-enabled browsers [10], Ed Felten found that the first-generation implementations averaged one security-relevant bug per 3,000 lines of source code [12]. These browsers, as mobile-code host platforms that depend on static checking for security, exemplify the kind of application for which proof-carrying code is well suited. Wang and Appel [6] measured the TCBs of various Java Virtual Machines at between 50,000 and 200,000

*This research was supported in part by DARPA award F30602-99-1-0519.
†This research was supported DARPA/Air Force contract F33615-00-C-1693 and NSF contract CCR-9806889.

lines of code. The SpecialJ JVM [9] uses proof-carrying code to reduce the TCB to 36,000 lines.

In this work, we show how to reduce the size of the TCB to under 2,700 lines, and by basing those lines on a well understood logical framework, we have produced a checker which is small enough so that it can be manually verified; and as such it can be relied upon to accept only valid proofs. Since this small checker "knows" only about machine instructions, and nothing about the programming language being compiled and its type system, the semantic techniques for generating the proofs that the TCB will check can be involved and complex [2], but the checker doesn't.

2 The LF logical framework

For a proof checker to be simple and correct, it is helpful to use a well designed and well understood representation for logics, theorems, and proofs. We use the LF logical framework.

LF [14] provides a means for defining and presenting logics. The framework is general enough to represent a great number of logics of interest in mathematics and computer science (for instance: first-order, higher-order, intuitionistic, classical, modal, temporal, relevant and linear logics, and others). The framework is based on a general treatment of syntax, rules, and proofs by means of a typed first-order λ-calculus with dependent types. The LF type system has three levels of terms: objects, types, and kinds. Types classify objects and kinds classify families of types. The formal notion of definitional equality is taken to be $\beta\eta$-conversion.

A logical system is presented by a signature, which assigns types and kinds to a finite set of constants that represent its syntax, its judgments, and its rule schemes. The LF type system ensures that object-logic terms are well formed. At the proof level, the system is based on the *judgments-as-types* principle: judgments are represented as types, and proofs are represented as terms whose type is the representation of the theorem they prove. Thus, there is a correspondence between type-checked terms and theorems of the object logic. In this way proof checking of the object logic is reduced to type checking of the LF terms.

For developing our proofs, we use Twelf [19], an implementation of LF by Frank Pfenning and his students. Twelf is a sophisticated system with many useful features: in addition to an LF type checker, it contains a type-reconstruction algorithm that permits users to omit many explicit parameters, a proof-search algorithm (which is like a higher-order Prolog interpreter), constraint regimes (e.g., linear programming over the exact rational numbers), mode analysis of parameters, a meta-theorem prover, a pretty-printer, a module system, a configuration system, an interactive Emacs mode, and more. We have found many of these features useful in proof development, but Twelf is certainly not a minimal proof checker. However, since Twelf does construct explicit proof objects internally, we can extract these objects to send to our minimal checker.

In LF one declares the operators, axioms, and inference rules of an *object logic* as constructors. For example, we can declare a fragment of first-order logic with the type `form` for formulas and a dependent type constructor `pf` for proofs, so that for any formula `A`, the type `pf(A)` contains values that are proofs of `A`. Then, we can declare an "implies" constructor `imp` (infix, so it appears between its arguments), so that if `A` and `B` are formulas then so is `A imp B`. Finally, we can define introduction and elimination rules for `imp`.[1]

```
form : type.
pf   : form -> type.
imp  : form -> form -> form.
%infix right 10 imp.
imp_i: (pf A -> pf B) -> pf (A imp B).
imp_e: pf (A imp B) -> pf A -> pf B.
```

All the above are defined as constructors. In general, constructors have the form *name* : τ and declare that *name* is a value of type τ.

It is easy to declare inconsistent object-logic constructors. For example, `invalid: pf A` is a constructor that acts as a proof of any formula, so using it we could easily prove the false proposition:

```
logic_inconsistent : pf (false) = invalid.
```

So the object logic should be designed carefully and must be trusted.

Once the object logic is defined, theorems can be proved. We can prove for instance that implication is transitive:

```
imp_trans: pf (A imp B) ->
             pf (B imp C) ->
              pf (A imp C) =
   [p1: pf (A imp B)]
   [p2: pf (B imp C)]
   imp_i [p3: pf A]
   imp_e p2 (imp_e p1 p3).
```

[1]Here, for example, the `imp_i` axiom states that if you have an LF function that can transform proofs of `A` to proofs of `B`, then applying the rule produces a proof of `A imp B`. The rule `imp_e` (generally known as *modus ponens*) goes the other way: given a proof of `A imp B` and a proof of `A`, applying the rule produces a proof of `B`.

In general, definitions (including predicates and theorems) have the form $name : \tau = exp$, which means that $name$ is now to stand for the value exp whose type is τ. In this example, the exp is a function with formal parameters `p1` and `p2`, and with body `imp_i [p3] imp_e p2 (imp_e p1 p3)`.

Definitions need not be trusted, because the type-checker can verify whether exp does have type τ. In general, if a proof checker is to check the proof P of theorem T in a logic $\mathcal{L}$, then the constructors (operators and axioms) of $\mathcal{L}$ must be given to the checker in a trusted way (i.e., the adversary must not be free to install inconsistent axioms). The statement of T must also be trusted (i.e., the adversary must not be free to substitute an irrelevant or vacuous theorem). The adversary provides only the proof P, and then the checker does the proof checking (i.e., it type-checks in the LF type system the definition $t : T = P$, for some arbitrary name t).

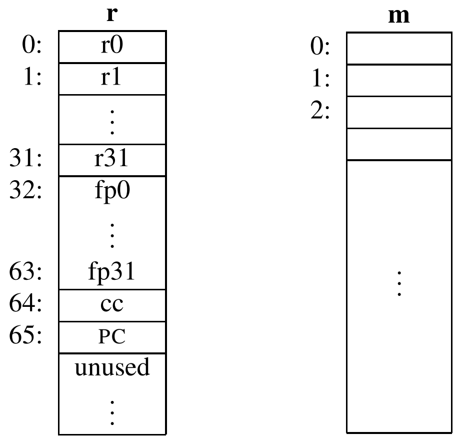
3 Application: Proof-carrying code

Our checker is intended to serve a purpose: to check safety theorems about machine-language programs. It is important to include application-specific portions of the checker in our measurements to ensure that we have adequately addressed all issues relating to interfacing to the real world.

The most important real-world-interface issue is, "is the proved theorem meaningful?" An accurate checker does no good if it checks the wrong theorem. As we will explain, the specification of the safety theorem is larger than all the other components of our checker combined!

Given a machine-language program P, that is, a sequence of integers that code for machine instructions (on the Sparc, in our case), the theorem is, "when run on the Sparc, P never executes an illegal instruction, nor reads or writes from memory outside a given range of addresses." To formalize this theorem it is necessary to formalize a description of instruction execution on the Sparc processor. We do this in higher-order logic augmented with arithmetic.

In our model [15], a machine state comprises a *register bank* and a *memory*, each of which is a function from integers (register numbers and addresses) to integers (contents). Every register of the instruction-set architecture (ISA) must be assigned a number in the register bank: the general registers, the floating-point registers, the condition codes, and the program counter. Where the ISA does not specify a number (such as for the PC) or when the numbers for two registers conflict (such as for the floating point and integer registers) we use an arbitrary unused index:



A single step of the machine is the execution of one instruction. We can specify instruction execution by giving a step relation $(r, m) \mapsto (r', m')$ that maps the prior state (r, m) to the next state (r', m') that holds after the execution of the machine instruction.

For example, to describe the add instruction $r_1 \leftarrow r_2 + r_3$ we might start by writing,

$$\begin{aligned}&(r, m) \mapsto (r', m') \;\equiv\\ &\quad r'(1) = r(2) + r(3) \wedge (\forall x \neq 1.\, r'(x) = r(x))\\ &\quad \wedge\; m' = m\end{aligned}$$

In fact, we can parameterize the above on the three registers involved and define $\mathrm{add}(i, j, k)$ as the following predicate on four arguments (r, m, r', m'):

$$\begin{aligned}&\mathrm{add}(i, j, k) \;\equiv\\ &\quad \lambda r, m, r', m'.\; r'(i) = r(j) + r(k)\\ &\qquad\qquad \wedge (\forall x \neq i.\, r'(x) = r(x)) \wedge\; m' = m\end{aligned}$$

Similarly, for the load instruction $r_i \leftarrow m[r_j + c]$ we define its semantics to be the predicate:

$$\begin{aligned}&\mathrm{load}(i, j, c) \;\equiv\\ &\quad \lambda r, m, r', m'.\; r'(i) = m(r(j) + c)\\ &\qquad\qquad \wedge (\forall x \neq i.\, r'(x) = r(x)) \;\wedge\; m' = m\end{aligned}$$

To enforce memory safety policies, we will modify the definition of $\mathrm{load}(i, j, c)$ to require that the loaded address is legal [2], but we omit those details here.

But we must also take into account instruction fetch and decode. Suppose, for example, that the add instruction is encoded as a 32-bit word, containing a 6-bit field with opcode 3 denoting *add,* a 5-bit field denoting the destination register i, and 5-bit fields denoting the source registers j, k:

3	i	j	0	k
26	21	16	5	0

The load instruction might be encoded as:

12	i	j	c
26	21	16	0

Then we can say that some number w decodes to an instruction *instr* iff,

$$
\begin{aligned}
&\mathrm{decode}(w, \mathit{instr}) \;\equiv \\
&\quad (\exists i, j, k. \\
&\quad\quad 0 \le i < 2^5 \;\wedge\; 0 \le j < 2^5 \;\wedge\; 0 \le k < 2^5 \;\wedge \\
&\quad\quad w = 3 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + k \cdot 2^0 \;\wedge \\
&\quad\quad \mathit{instr} = \mathrm{add}(i, j, k)) \\
&\quad \vee (\exists i, j, c. \\
&\quad\quad 0 \le i < 2^5 \;\wedge\; 0 \le j < 2^5 \;\wedge\; 0 \le c < 2^{16} \;\wedge \\
&\quad\quad w = 12 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + c \cdot 2^0 \;\wedge \\
&\quad\quad \mathit{instr} = \mathrm{load}(i, j, \text{sign-extend}(c))) \\
&\quad \vee \;\ldots
\end{aligned}
$$

with the ellipsis denoting the many other instructions of the machine, which must also be specified in this formula.

We have shown [15] how to scale this idea up to the instruction set of a real machine. Real machines have large but semi-regular instruction sets; instead of a single global disjunction, the decode relation can be factored into operands, addressing modes, and so on. Real machines don't use integer arithmetic, they use modular arithmetic, which can itself be specified in our higher-order logic. Some real machines have multiple program counters (e.g., Sparc) or variable-length instructions (e.g., Pentium), and these can also be accommodated.

Our description of the decode relation is heavily factored by higher-order predicates (this would not be possible without higher-order logic). We have specified the execution behavior of a large subset of the Sparc architecture, and we have built a prototype proof-generating compiler that targets that subset. For proof-carrying code, it is sufficient to specify a subset of the machine architecture; any unspecified instruction will be treated by the safety policy as illegal. While this may be inconvenient for compilers that want to generate that instruction, it does ensure that safety cannot be compromised.

4 Specifying safety

Our step relation $(r, m) \mapsto (r', m')$ is deliberately partial; some states have no successor state. In these states the program counter $r(\mathrm{PC})$ points to an illegal instruction. Using this partial step relation, we can define safety. A safe program is one that will never execute an illegal instruction; that is, a given state is safe if, for any state reachable in the Kleene closure of the step relation, there is a successor state:

$$
\begin{aligned}
&\text{safe-state}(r, m) \;\equiv \\
&\qquad \forall r', m'.\; (r, m) \mapsto^* (r', m') \;\Rightarrow \\
&\qquad\quad \exists r'', m''.\; (r', m') \mapsto (r'', m'')
\end{aligned}
$$

A program is just a sequence of integers (each representing a machine instruction); we say that a program p is loaded at a location l in memory m if

$$\mathrm{loaded}(p, m, l) \;\equiv\; \forall i \in \mathrm{dom}(p).\;\; m(i + l) = p(i)$$

Finally (assuming that programs are written in position-independent code), a program is *safe* if, no matter where we load it in memory, we get a safe state:

$$
\begin{aligned}
&\mathrm{safe}(p) \;\equiv \\
&\quad \forall r, m, \mathit{start}.\;\; \mathrm{loaded}(p, m, \mathit{start}) \;\wedge\; r(\mathrm{PC}) = \mathit{start} \\
&\qquad\qquad\qquad \Rightarrow \text{safe-state}(r, m)
\end{aligned}
$$

Let `;` be a "cons" operator for sequences of integers (easily definable in higher-order logic); then for some program

```
8420 ; 2837 ; 2938 ; 2384 ; nil
```

the safety theorem is simply

```
safe ( 8420 ; 2837 ; 2938 ; 2384 ; nil )
```

and, given a proof P, the LF definition that must be type-checked is

```
t: pf(safe(8420;2837;2938;2384;nil)) = P.
```

Though we wrote in section 2 that definitions need not be trusted because they can be type-checked, this is not strictly true. Any definition used in the statement of the theorem must be trusted, because the wrong definition will lead to the proof of the wrong theorem. Thus, all the definitions leading up to the definition of `safe` (including `add`, `load`, `safe-state`, `step`, `loaded`, etc.) must be part of the trusted checker. Since we have approximately 1,600 lines of such definitions, and they are a component of our "minimal" checker, one of the most important issues we faced is the representation of these definitions; we will discuss this in Section 7.

On the other hand, a large proof will contain hundreds of internal definitions. These are predicates and internal lemmas of the proof (not of the statement of the theorem), and are at the discretion of the proof provider. Since each is type checked by the checker before it is used in further definitions and proofs, they don't need to be trusted.

In the table below we show the various pieces needed for the specification of the safety theorem in our logic. Every piece in this table is part of the TCB. Column two shows the number of lines of Twelf code needed for the specification and column three the number of definitions in that specification. The first two lines show the size of the logical and arithmetic connectives (in which theorems are specified) as well as the size of the logical and arithmetic axioms (using which theorems are proved). The Sparc specification has two components, a "syntactic" part (the decode relation) and a semantic part (the definitions of *add, load,* etc.); these are shown in the next two lines. The size of the safety predicate is shown last.

Safety Specification	*Lines*	*Definitions*
Logic	135	61
Arithmetic	160	94
Machine Syntax	460	334
Machine Semantics	1,005	692
Safety Predicate	105	25
Total	1,865	1,206

From this point on we will refer to everything in the table as the *safety specification* or simply the *specification*.

5 Eliminating redundancy

Typically an LF signature will contain much redundant information. Consider for example the rules for `imp` presented previously; in fully explicit form, their representation in LF is as follows:

```
imp_i: {A : form}{B : form}
       (pf A -> pf B) -> pf (A imp B).
imp_e: {A : form}{B : form}
       pf (A imp B) -> pf A -> pf B.
```

The fact that both `A` and `B` are formulas can be easily inferred by the fact they are given as arguments to the constructor `imp`, which has been previously declared as an operator over formulas.

On the one hand, eliminating redundancy from the representation of proofs benefits both proof size and type-checking time. On the other hand, it requires performing term reconstruction, and thus it may dramatically increase the complexity of type checking, driving us away from our goal of building a minimal checker.

Twelf deals with redundancy by allowing the user to declare some parameters as *implicit*. More precisely, all variables which are not quantified in the declaration are automatically assumed implicit. Whenever an operator is used, Twelf's term reconstruction will try to determine the correct substitution for all its implicit arguments. For example, in type-checking the lemma

```
imp_refl : pf (A imp A) =
   imp_i ([p : pf A] p).
```

Twelf will automatically reconstruct the two implicit arguments of `imp_i` to be both equal to `A`.

While Twelf's notion of implicit arguments is effective in eliminating most of the redundancy, type reconstruction adds considerable complexity to the system. Another drawback of Twelf's type reconstruction is its reliance on unification, which for higher-order terms is in general undecidable. Because of this, type checking of some valid proofs may fail.

Necula's LF_i [17] uses partial type reconstruction and a simple algorithm to determine which of the arguments can be made implicit. Implicit arguments are omitted in the representation, and replaced by placeholders.

Oracle-based checking [18] reduces the proof size even further by allowing the erasure of subterms whose reconstruction is not uniquely determined. Specifically, in cases when the reconstruction of a subterm is not unique, but there is a finite (and usually small) list of candidates, it stores an oracle for the right candidate number instead of storing the entire subterm.

All the techniques mentioned above are based on the idea that it is acceptable to make the typechecker "smarter" (and therefore larger, and more likely to contain bugs) in order to minimize proof size. In this work, however, we are interested in a minimal proof checker, and so we looked for a way to eliminate some redundancy from the LF representation, without adding too many lines of code to the TCB.

We internally represent LF terms as directed acyclic graphs (DAGs). This representation gives us a simple way to reduce redundancy by structure sharing. We save space, since common subterms are stored just once; we also save time, because testing for equality of two terms is performed by reference first.

A node in the DAG can be one of ten possible types: one for kinds, five for ordinary LF terms, and four for arithmetic expressions. Each node may store up to three integers, `arg1`, `arg2`, and `type`. This last one, if present, will always point to the sub-DAG representing the type of the expression.

	arg1	arg2	type	
n	U	U	U	kind
c	U	U	M	constant
v	M	M	M	variable
a	M	M	O	application
p	M	M	O	product
l	M	M	O	abstraction
#	M	U	O	number
+	M	M	O	addition proof obj
*	M	M	O	mult proof obj
/	M	M	O	div proof obj

M = mandatory
O = optional
U = unused

The content of `arg1` and `arg2` is used in different ways for different node types. For all nodes representing arithmetic expressions ('#', '+', '*', and '/'), these contain integer values; in particular, for the nodes corresponding to meta-level numbers ('#') only `arg1` is used. For products and abstractions ('p' and 'l'), `arg1` points to the bound variable, and `arg2` to the term where the binding takes place. For variable nodes ('v'), `arg1` and `arg2` are used to make sure that the variable always occurs within the scope of a quantifier. For application nodes ('a'), `arg1` and `arg2` point to the function and its argument, respectively. Finally, constant declaration nodes ('c'), and kind declaration nodes ('n') use neither of the two.

For a concrete example, consider the LF signature:

```
form : type.
pf   : form -> type.
imp  : form -> form -> form.
```

Remembering that -> above is just a shorthand for the product of types, we present below the DAG representation of this signature. We "flattened" the DAG into a numbered list, and, for clarity, we also added a comment on the right showing the corresponding LF term.

```
 1| n 0 0 0   ; type Kind
 2| c 0 0 1   ; form: type
 3| v 0 0 2   ; x: form
 4| p 3 1 0   ; {x: form} type
 5| c 0 0 4   ; pf: {x: form} type
 6| v 0 0 2   ; y: form
 7| p 6 2 0   ; {y: form} form
 8| v 0 0 2   ; x: form
 9| p 8 7 0   ; {x: form}{y: form} form
10| c 0 0 9   ; imp: {x: form}{y: form} form
```

6 Dealing with arithmetic

Since our proofs reason about encodings of machine instructions (opcode calculations) and integer values manipulated by programs, the problem of representing arithmetic within our system is a critical one. A purely logical representation based on 0, successor and predecessor is not suitable to us, since it would cause proof size to explode.

The latest releases of Twelf offer extensions that deal natively with infinite-precision integers and rationals. While these extensions are very powerful and convenient to use, they offer far more than we need, and because of their generality they have a very complex implementation (the rational extension alone is 1,950 lines of Standard ML). What we actually would like for our checker is an extension built in the same spirit as those, but much simpler and lighter.

There are essentially two properties that we require from such an extension:

1. LF terms for all the numbers we use; moreover, the size of the LF term for n should be constant and independent of n.

2. Proof objects for single-operation arithmetic facts such as "$10 + 2 = 12$"; again, we require that such proof objects have constant size.

Our arithmetic extensions to the checker are the smallest and simplest ones to satisfy (1) and (2) above. We add the `word32` type to the TCB, (representing integers in the range $[0, 2^{32} - 1]$) as well as the following axioms:

```
word32 : type.
+ : word32 -> word32 -> word32 -> type.
* : word32 -> word32 -> word32 -> type.
/ : word32 -> word32 -> word32 -> type.
```

We also modify the checker to accept arithmetic terms such as:

```
567    : word32.
456+25 : + 456 25 481.
32*4   : * 32 4 128.
56/5   : / 56 5 11.
```

This extension does not modify in any way the standard LF type checking: we could have obtained the same result (although much more inefficiently) if we added all these constants to the trusted LF signature by hand. However, granting them special treatment allowed us to save literally millions of lines in the axioms in exchange for an extra 55 lines in the checker.

To embed and use these new constants in our object logic, we also declare:

```
const: word32 -> tm num.
eval_plus:
  + A B C ->
    pf (eq (plus (const A) (const B))
           (const C)).
eval_times:
  * A B C ->
    pf (eq (times (const A) (const B))
           (const C)).
eval_div:
  / M N Q ->
    pf ((geq (const M)
             (times (const N) (const Q)))
        and
        (not
           (geq (const M)
                (times (const N)
                       (plus one
                             (const Q)))))).
```

This embedding from `word32` to numbers in our object logic is not surjective. Numbers in our object logic are still unbounded; `word32` merely provides us with handy names for the ones used most often.

With this "glue" to connect object logic to meta logic, numbers and proofs of elementary arithmetic properties, are just terms of size two. For example, the representation of 5634 in the object logic, is `const 5634`, which, by virtue of these additional axioms, can be easily verified to be a valid term of type `num`.

7 Representing axioms and trusted definitions

Since we can represent axioms, theorems, and proofs as DAGs, it might seem that we need neither a parser nor a pretty-printer in our minimal checker. In principle, we could provide our checker with an initial trusted DAG representing the axioms and the theorem to be proved, and then it could receive and check an untrusted DAG representing the proof. The trusted DAG could be represented in the C language as an initialized array of graph nodes.

This might work if we had a very small number of axioms and trusted definitions, and if the statement of the theorem to be proved were very small. We would have to read and trust the initialized-array statements in C, and understand their correspondence to the axioms (etc.) as we would write them in LF notation. For a sufficiently small DAG, this might be simpler than reading and trusting a parser for LF notation.

However, even a small set of operators and axioms (especially once the axioms of arithmetic are included) requires hundreds of graph nodes. In addition, as explained in Section 4, our trusted definitions include the machine-instruction step relation of the Sparc processor. These 1,865 lines of Twelf expand to 22,270 DAG nodes. Clearly it is impossible for a human to directly read and trust a graph that large.

Therefore, we require a parser or pretty-printer in the minimal checker; we choose to use a parser. Our C program will parse the 1,865 lines of axioms and trusted definitions, translating the LF notation into DAG nodes. The axioms and definitions are also part of the C program: they are a constant string to which the parser is applied on startup.

This parser is 428 lines of C code; adding these lines to the minimal checker means our minimal checker can use 1,865 lines of LF instead of 22,270 lines of graph-node initializers, clearly a good tradeoff.

Our parser accepts valid LF expressions, written in the same syntax used by Twelf. In addition to those, it will also parse nonnegative numbers below 2^{32} and type them as `word32` objects.

Two extra features we added to our parser are infix operators (the fixity information is passed to the parser using the same `%infix` directive used by Twelf), and single-line comments. While these are not strictly necessary, implementing them did not significantly affect the complexity of the parser, and their use greatly improved the readability of the specification, and therefore its trustworthiness.

7.1 Encoding higher-order logic in LF

Our encoding of higher-order logic in LF follows that of Harper et al. [14] and is shown in figure 1. The *constructors* generate the syntax of the object logic and the *axioms* generate its proofs. A meta-logical type is `type` and an object-logic type is `tp`. Object-logic types are constructed from `form` (the type of formulas), `num` (the type of integers), and the `arrow` constructor. So the object-logic predicate `even?`, for instance, would have object type `(num arrow form)`. The LF term `tm` maps an object type to a meta type, so an object-level term of type `T` has type (`tm T`) in the meta logic.

Abstraction in the object logic is expressed by the `lam` term. The term `(lam [x] (F x))` is the object-logic function that maps `x` to `(F x)`. Application for such lambda terms is expressed via the `@` operator. The quantifier `forall` is defined to take as input a meta-level (LF) function of type `(tm T -> tm form)` and produce a `tm form`. The use of the LF functions here makes it easy to perform substitution when a proof of `forall` needs to be discharged, since equality in LF is just $\beta\eta$-conversion.

Notice that most of the standard logical connectives are absent from figure 1. This is because we can produce

```
                    Logic Constructors

tp         :  type.
tm         :  tp -> type.
form       :  tp.
num        :  tp.
arrow      :  tp -> tp -> tp.
pf         :  tm form -> type.
lam        :  (tm T1 -> tm T2) -> tm (T1 arrow T2).
@          :  tm (T1 arrow T2) -> tm T1 -> tm T2.
forall     :  (tm T -> tm form) -> tm form.
imp        :  tm form -> tm form -> tm form.

                    Logic Axioms

beta_e     :  pf (P ((lam F) @ X)) -> pf (P (F X)).
beta_i     :  pf (P (F X)) -> pf (P (lam F) @ X).
imp_i      :  (pf A -> pf B) -> pf (A imp B).
imp_e      :  pf (A imp B) -> pf A -> pf B.
forall_i   :  ({X : tm T} pf (A X)) -> pf (forall A).
forall_e   :  pf (forall A) -> {X : tm T} pf (A X).
```

Figure 1: Higher Order Logic in Twelf

them as definitions from the constructors we already have. For instance, the `and` and `or` connectives can be defined as follows:

```
and = [A][B] forall [C]
             (A imp B imp C) imp C.
or = [A][B] forall [C]
       (A imp C) imp (B imp C) imp C.
```

It is easy to see that the above formulae are equivalent to the standard definitions of `and` and `or`. We can likewise define introduction and elimination rules for all such logic constructors. These rules are proven as lemmas and need not be trusted. Object-level equality[2] is also easy to define:

```
eq : tm T -> tm T -> tm form =
 [A : tm T][B : tm T]
  forall [P] P @ B imp P @ A.
```

This states that objects `A` and `B` are considered equal iff any predicate `P` that holds on `B` also holds on `A`.

[2] The equality predicate is polymorphic in `T` in a sense to be made precise later. The objects `A` and `B` have object type `T` and so they could be `nums`, `forms` or even object level functions (`arrow` types). The object type `T` is implicit in the sense that when we use the `eq` predicate we do not have to specify it; Twelf can automatically infer it. So internally, the meta-level type of `eq` is not what we have specified above but the following:

```
eq : {T : tp} tm T -> tm T -> tm form = ...
```

We will have more to say about this in section 7.2.

Terms of type (`pf A`) are terms representing proofs of object formula `A`. Such terms are constructed using the axioms of figure 1. Axioms `beta_e` and `beta_i` are used to prove β-equivalence in the object logic. The first states that for any meta-level predicate `P` (of type `tm T -> tm form`), if `P` holds on the term `((lam F) @ X)` then it also holds on `(F X)`. Axiom `beta_i` takes one in the other direction. Axioms `imp_i` and `imp_e` transform a meta-level proof function to the object level and vice-versa. Finally, `forall_i` introduces the `forall` statement if it is presented with a meta-level function from `X` to `pf (A X)`, and `forall_e` discharges a `forall` by applying the meta-level function `A` to an instance `X` of its domain to produce `pf (A X)`. Notice how the higher-order abstract syntax of LF makes this discharging (and the term substitution of `X` in the body of `A`) completely painless for the designer of the logic.

As an example of a simple proof we show how to prove the reflexivity lemma `refl`, which states that all objects equal themselves. By the definition of `eq`, if `X` is to equal itself, we must to show that for any predicate `P` if `(P @ X)` then `(P @ X)` – which is obvious. Here is the proof:

```
refl : pf (eq X X) =
 forall_i [P] imp_i [q : pf (P @ X)] q.
```

7.2 "Polymorphic" programming in Twelf

ML-style implicit polymorphism allows one to write a function usable at many different argument types, and ML-style type inference does this with a low syntactic overhead. We are writing proofs, not programs, but we would still like to have polymorphic predicates and polymorphic lemmas. LF is not polymorphic in the ML sense, but Harper et al. [14] show how to use LF's dependent type system to get the effect and (most of) the convenience of implicit parametric polymorphism with an encoding trick, which we will illustrate with an example.

Suppose we wish to write the lemma `congr` that would allow us to substitute equals for equals:

```
congr : {H : type -> tm form}
 pf (eq X Z) -> pf (H Z)
             -> pf (H X) = ...
```

The lemma states that for any predicate `H`, if `H` holds on `Z` and `Z` $=$ `X` then `H` also holds on `X`. Unfortunately this does not work in LF since LF does not allow polymorphism. Fortunately though, there is way to get polymorphism at the object level. We rewrite `congr` as:

```
congr : {H : tm T -> tm form}
 pf (eq X Z) -> pf (H Z)
             -> pf (H X) = ...
```

and this is now acceptable to Twelf. Function `H` now judges objects of meta-type `(tm T)` for any object-level type `T`, and so `congr` is now "polymorphic" in `T`. We can apply it on any object-level type, such as `num`, `form`, `num arrow num`, `num arrow form`, etc. This solution is general enough to allow us to express any polymorphic term or lemma with ease. Axioms `forall_i` and `forall_e` in figure 1 are likewise polymorphic in `T`.

7.3 How to write explicit Twelf

In the definition of lemma `congr` above, we have left out many explicit parameters since Twelf's type-reconstruction algorithm can infer them. The actual LF type of the term `congr` is:

```
congr : {T : tp}{X : tm T}{Z : tm T}
        {H : tm T -> tm form}
 pf (_eq T X Z) -> pf (H Z)
                -> pf (H X) = ...
```

Type reconstruction in Twelf is extremely useful, especially in a large system like ours, where literally hundreds of definitions and lemmas have to be stated and proved.

Our safety specification was originally written to take advantage of Twelf's ability to infer missing arguments. Before proof checking can begin, this specification needs to be fed to our proof checker. In choosing then what would be in our TCB we had to decide between the following alternatives:

1. Keep the implicitly-typed specification in the TCB and run it through Twelf to produce an explicit version (with no missing arguments or types). This explicit version would be fed to our proof checker. This approach allows the specification to remain in the implicit style. Also our proof checker would remain simple (with no type reconstruction/inference capabilities) but unfortunately we now have to add to the TCB Twelf's type-reconstruction and unification algorithms, which are about 5,000 lines of ML code.

2. Run the implicitly typed specification through Twelf to get an explicit version. Now instead of trusting the implicit specification and Twelf's type-reconstruction algorithms, we keep them out of the TCB and proceed to manually verify the explicit version. This approach also keeps the checker simple (without type-reconstruction capabilities). Unfortunately the explicit specification produced by Twelf explodes in size from 1,700 to 11,000 lines, and thus the code that needs to be verified correct is huge. The TCB would grow by a lot.

3. Rewrite the trusted definitions in an explicit style. Now we do not need type reconstruction in the TCB (the problem of choice 1), and if the rewrite from the implicit to the explicit style can avoid the size explosion (the problem of choice 2), then we have achieved the best of both worlds.

Since neither of choices 1 and 2 above were consistent with our goal of a small trusted computing base, we followed choice 3 and rewrote the trusted definitions in an explicit style while managing to avoid the size explosion. The new safety specification is only 1,865 lines of explicitly-typed Twelf. It contains no terms with implicit arguments – everything is explicit and every quantified variable is explicitly typed. Thus, we do not need a type-reconstruction/type-inference algorithm in the proof checker. The rewrite solves the problem while maintaining the succinctness and brevity of the original TCB, the penalty of the explicit style being an increase in size of 124 lines. The remainder of this section explains the problem in detail and the method we used to bypass it.

To see why there is such an enormous difference in size (1,700 lines vs 11,000) between the implicit specification and its explicit representation generated by Twelf's

Object Logic Abstraction/Application

```
fld2 = [T1:tp][T2:tp][T3:tp][T4:tp]
 lam6 (arrow T1 (arrow T2 form))
      (arrow T3 (arrow T2 form))
      (arrow T2 form)
      (arrow T1 (arrow T3 T4))
      T4 T2 form
  [f0][f1][p_pi][icons][ins][w]
  (@ T2 form p_pi w) and
  (exists2 T1 ([x:tm T1] T3)
   ([g0:tm T1] [g1:tm T3]
    (@ T2 form
     (&& T2 (@ T1 (arrow T2 form) f0 g0)
        (@ T3 (arrow T2 form) f1 g1)) w) and
   (eq T4 ins (@ T3 T4
               (@ T1 (arrow T3 T4)
                  icons g0) g1)))).
```

Meta Logic Abstraction/Application

```
fld2 = [T1:tp][T2:tp][T3:tp][T4:tp]
 [f0][f1][p_pi][icons][ins][w]
 (p_pi w) and
 (exists2 [g0:tm T1][g1:tm T3]
  (f0 g0 && f1 g1) w) and
  (eq ins (icons g0 g1)).
```

Figure 2: Abstraction & Application in the Object versus Meta Logic.

type-reconstruction algorithm, consider the following example. Let `F` be a two-argument object-level predicate `F : tm (num arrow num arrow form)` (typical case when describing unary operators in an instruction set). When such a predicate is applied, as in `(F @ X @ Y)`, Twelf has to infer the implicit arguments to the two instances of operator `@`. The explicit representation of the application then becomes:

```
@ num form (@ num (num arrow form) F X) Y
```

It is easy to see how the explicit representation explodes in size for terms of higher order. Since the use of higher-order terms was essential in achieving maximal factoring in the machine descriptions [15], the size of the explicit representation quickly becomes unmanageable.

Here is another more concrete example from the `decode` relation of section 3. This one shows how the abstraction operator `lam` suffers from the same problem. The predicate below (given in implicit form) is used in specifying the syntax of all Sparc instructions of two arguments.

```
fld2 = lam6 [f0][f1][p_pi][icons][ins][w]
 p_pi @ w and
 exists2 [g0][g1] (f0 @ g0 && f1 @ g1) @ w and
                  eq ins (icons @ g0 @ g1).
```

Predicates `f0` and `f1` specify the input and the output registers, `p_pi` decides the instruction opcode, `icons` is the instruction constructor, `ins` is the instruction we are decoding, and `w` is the machine-code word.[3] In explicit form this turns into what we see on the left-hand side of figure 2 – an explicitly typed definition 16 lines long.

The way around this problem is the following: We avoid using object-logic predicates whenever possible. This way we need not specify the types on which object-logic application and abstraction are used. For example, the `fld2` predicate above now becomes what we see on the right-hand side of figure 2. This new predicate has shrunk in size by more than half.

Sometimes moving predicates to the meta-logic is not possible. For instance, we represent *instructions* as predicates from machine states to machine states (see section 3). Such predicates must be in the object logic since we need to be able to use them in quantifiers (`exists [ins : tm instr]` ...). Thus, we face the problem of having to supply all the implicit types when defining such predicates and when applying them. But since these types are always fixed we can factor the partial applications and avoid the repetition. So, for example, when defining some Sparc machine instruction as in:

```
i_anyInstr = lam2 [rs : tnum][rd : tnum]
 lam4 registers memory
      registers memory form
   [r : tregs][m : tmem]
   [r' : tregs][m' : tmem]
    ...
```

we define the predicate `instr_lam` as:

[3] As we mentioned before, our specifications are highly factored and this is an example of such factoring – any instruction of two arguments can be specified using this predicate. To define, for instance, the predicate that decides the `FMOVs` (floating-point move) instruction on the Sparc, we would say:

```
ins_MOVs = fld2 @ f_fs2 @ f_fd @ p_FMOVs @ i_FMOVs.
```

where the arguments are as described above.

```
instr_lam = lam4 registers memory
                 registers memory form.
```

and then use it in defining each of the 250 or so Sparc instructions as below:

```
i_anyInstr = [rs : tnum][rd : tnum]
 instr_lam [r : tregs][m : tmem]
          [r' : tregs][m' : tmem]
  ...
```

This technique turns out to be very effective because our machine syntax and semantics specifications were highly factored to begin with [15].

So by moving to the meta logic and by clever factoring we have moved the TCB from implicit to explicit style with only a minor increase in size. Now we don't have to trust a complicated type-reconstruction/type-inference algorithm. What we feed to our proof checker is precisely the set of axioms we explicitly trust.

7.4 The implicit layer

When we are building proofs, we still wish to use the implicit style because of its brevity and convenience. For this reason we have built an implicit layer on top of our explicit TCB. This allows us to write proofs and definitions in the implicit style and LF's $\beta\eta$-conversion takes care of establishing meta-level term equality. For instance, consider the object-logic application operator `_@` given below:

```
_@ : {T1 : tp}{T2 : tp}
        tm (T1 arrow T2) ->
          tm T1 -> tm T2.
```

In the implicit layer we now define a corresponding application operator `@` in terms of `_@` as follows:

```
@ : tm (T1 arrow T2) ->
     tm T1 -> tm T2 = _@ T1 T2.
```

In this term the type variables `T1` and `T2` are implicit and need not be specified when `@` is used. Because `@` is a definition used only in proofs (not in the statement of the safety predicate), it does not have to trusted.

8 The proof checker

The total number of lines of code that form our checker is 2,668. Of these, 1,865 are used to represent the LF signature containing the core axioms and definition, which is stored as a static constant string.

The remaining 803 lines of C source code, can be broken down as follows:

Component	*Lines*
Error messaging	14
Input/Output	29
Parser	428
DAG creation and manipulation	111
Type checking and term equality	167
Main program	54
Total	803

In designing the components listed above, we make sure not to use any library functions. Libraries often have bugs, and by avoiding their use we eliminate the possibility that some adversary may exploit one of these bugs to disable or subvert proof checking.

8.1 Trusting the C compiler

We hasten to point out that these 803 lines of C need to be compiled by a C compiler, and so it would appear that this compiler would need to be included in our TCB. The C compiler could have bugs that may potentially be exploited by an adversary to circumvent proof checking. More dangerously perhaps, the C compiler could have been written by the adversary so while compiling our checker, it could insert a Thompson-style [21] Trojan horse into the executable of the checker.

All proof verification systems suffer from this problem. One solution (as suggested by Pollack [20]) is that of independent checking: write the proof checker in a widely used programming language and then use different compilers of that language to compile the checker. Then run all your checkers on the proof in question. This is similar to the way mathematical proofs are "verified" as such by mathematicians today.

The small size of our checker suggests another solution. Given enough time one may read the output of the C compiler (assembly language or machine code) and verify that this output faithfully implements the C program given to the compiler. Such an examination would be tedious but it is not out of the question for a C program the size of our checker, and it could be carried out if such a high level of assurance was necessary. Such an investigation would certainly uncover Thompson-style Trojan horses inserted by a malicious compiler. This approach would not be feasible for the JVMs mentioned in the introduction; they are simply too big.

8.2 Proof-checking measurements

In order to test the proof checker, and measure its performance, we wrote a small Standard ML program that converts Twelf internal data structures into DAG format, and dumps the output of this conversion to a file, ready for consumption by the checker.

We performed our measurements on a sample proof of nontrivial size. In its original formulation, this proof is 6,367 lines of Twelf, and makes extensive use of implicit arguments. Converted to its fully explicit form, its size expands to 49,809 lines. Its DAG representation consists of 177,425 nodes.

Checking the sample proof consists of the steps shown below; timings for each step are also shown:

1. Parsing the axioms and definitions in the TCB and generating a DAG for them: 0.06 secs.

2. Loading the proof from disk: 0.02 secs.

3. Checking the DAG for well-formedness: 0.03 secs.

4. Type checking the proof term: 79.94 secs.

The measurements above were made on a 1 GHz Pentium III PC with 256MB of memory. During type checking of this proof the number of temporary nodes generated is 1,115,768. Most of the time during type checking is spent in performing substitutions in reductions to weak head normal form (WHNF). We believe that reduction to WHNF may not be necessary in all cases, and that by performing some analysis in the checker we could obtain improvements in both memory usage and type-checking time, without a significant increase in the size of the checker.

9 Future work

The DAG representation of proofs is quite large, and we would like to do better. One approach would be to compress the DAGs in some way; another approach is to use a compressed form of the LF syntactic notation. However, we believe that the most promising approach is neither of these.

Our proofs of program safety are structured as follows: first we prove (by hand, and check by machine) many structural lemmas about machine instructions and semantics of types [4, 5, 1]. Then we use these lemmas to prove (by hand, as derived lemmas) the rules of a low-level typed assembly language (TAL). Our TAL has several important properties:

1. Each TAL operator corresponds to exactly 0 or 1 machine instructions (0-instruction operators are coercions that serve as hints to the TAL typechecker and do nothing at runtime).

2. The TAL rules prescribe Sparc instruction encodings as well as the more conventional [16] register names, types, and so on.

3. The TAL typing rules are syntax-directed, so type-checking a TAL expression is decidable by a simple tree-walk without backtracking.

4. The TAL rules can be expressed as a set of Horn clauses.

Although we use higher-order logic to state and prove lemmas leading up to the proofs of the TAL typing rules, and we use higher-order logic in the proofs of the TAL rules, we take care to make all the statements of the TAL rules first-order Horn clauses. Consider such a clause,

```
head :- goal1 , goal2 , goal3.
```

In LF (using our object logic) we could express this as a lemma:

```
n : pf (goal3) -> pf (goal2) ->
     pf (goal1) -> pf (head) =
        proof.
```

Inside *proof* there may be higher-order abstract syntax, quantification, and so on, but the `goals` are all Prolog-style. The name `n` identifies the clause.

Our compiler (that produced the Sparc machine code) does so using a series of typed intermediate languages, the last of which is our TAL. Our prover constructs the safety proof for the Sparc program by "executing" the TAL Horn clauses as a logic program, using the TAL expression as input data. The proof is then a tree of clause names, corresponding to the TAL typing derivation.

We can make a checker that takes smaller proofs by just implementing a simple Prolog interpreter (without backtracking, since TAL is syntax-directed). But the we would need to trust the Prolog interpreter and the Prolog program itself (all the TAL rules). This is similar to what Necula [17] and Morrisett et al. [16] do. The problem is that in a full-scale system, the TAL comprises about a thousand fairly complex rules. Necula and Morrisett have given informal (i.e., mathematical) proofs of the soundness of their type systems for prototype languages, but no machine-checked proof, and no proof of a full-scale system.

The solution, we believe, is to use the technology we have described in the previous sections of this paper to check the derivations of the TAL rules from the axioms of logic (and the specification of the Sparc). Then, we can add a simple (non-backtracking) Prolog interpreter to our minimal checker, which will no longer be minimal: we estimate that this interpreter will add 200–300 lines of C code.

The proof producer (adversary) will first send to our checker, as a DAG, the definitions of Horn clauses for the TAL rules, which will be LF-typechecked. Then, the "proofs" sent for machine-language programs will be in the form of TAL expressions, which are much smaller than the proof DAGs we measured in section 8.

A further useful extension would be to implement oracle-based checking [18]. In this scheme, a stream of "oracle bits" guides the application of a set of Horn clauses, so that it would not be necessary to send the TAL expression – it would be re-derived by consulting the oracle. This would probably give the most concise safety proofs for machine-language programs, and the implementation of the oracle-stream decoder would not be too large. Again, in this solution the Horn clauses are first checked (using a proof DAG), and then they can be used for checking many successive TAL programs.

Although this approach seems very specific to our application in proof-carrying code, it probably applies in other domains as well. Our semantic approach to distributed authentication frameworks [3] takes the form of axioms in higher-order logic, which are then used to prove (as derived lemmas) first-order protocol-specific rules. While in that work we did not structure those rules as Horn clauses, more recent work in distributed authentication [11] does express security policies as sets of Horn clauses. By combining the approaches, we could have our checker first verify the soundness of a set of rules (using a DAG of higher-order logic) and then interpret these rules as a Prolog program.

10 Conclusion

Proof-carrying code has a number of technical advantages over other approaches to the security problem of mobile code. We feel that the most important of these is the fact that the trusted code of such a system can be made small. We have quantified this and have shown that in fact the trusted code can be made orders of magnitude smaller than in competing systems (JVMs). We have also analyzed the representation issues of the logical specification and shown how they relate to the size of the safety predicate and the proof checker. In our system the trusted code itself is based on a well understood and analyzed logical framework, which adds to our confidence of its correctness.

References

[1] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *In Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, July 2002.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[3] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*. ACM Press, November 1999.

[4] 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, New York, January 2000. ACM Press.

[5] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, to appear.

[6] Andrew W. Appel and Daniel C. Wang. Jvm tcb: Measurements of the trusted computing base of java virtual machines. Technical Report CS-TR-647-02, Princeton University, April 2002.

[7] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, , Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1998.

[8] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of USENIX Security*, August 2002.

[9] 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)*, New York, June 2000. ACM Press.

[10] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java security: Web browers and beyond. In Dorothy E. Denning and Peter J. Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*. ACM Press (New York, New York), October 1997.

[11] John DeTreville. Binder, a logic-based security language. In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, page (to appear), May 2002.

[12] Edward W. Felten. Personal communication, April 2002.

[13] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.

[14] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[15] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, pages 7–24, Berlin, June 2000. Springer-Verlag. LNAI 1831.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, New York, January 1998. ACM Press.

[17] 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.

[18] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154. ACM Press, January 2001.

[19] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*, Berlin, July 1999. Springer-Verlag.

[20] Robert Pollack. How to believe a machine-checked proof. In Sambin and Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1996.

[21] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.