Proof-Carrying Authentication*

Andrew W. Appel and Edward W. Felten
Secure Internet Programming Laboratory
Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

August 9, 1999

Abstract

We have designed and implemented a general and powerful distributed authentication framework based on higher-order logic. Authentication frameworks — including Taos, SPKI, SDSI, and X.509 — have been explained using logic. We show that by starting with the logic, we can implement these frameworks, all in the same concise and efficient system. Because our logic has no decision procedure — although proof checking is simple — users of the framework must submit proofs with their requests.

1 Introduction

Distributed authentication frameworks allow sharing of access to resources across administrative boundaries in a distributed system. The main abstractions they support are name-to-key bindings, access control, and delegation. Statements in these frameworks are represented as data structures that are often digitally signed to ensure their integrity.

Several authentication frameworks exist; we mention a few here as examples. The Taos operating system provided support for secure remote procedure call and data structures to represent authority and identity [6]. X.509 [15] is a widely-used standard for expressing and using digital certificates. SPKI [4] and SDSI [14] (since merged under the joint name SPKI) were reactions to the perceived complexity of X.509; in both cases the 'S' stands for 'simple.' PolicyMaker [3] is a language for expressing security policies; it can be applied to distributed security policies. Kerberos [12], unlike the other frameworks, uses symmetric-key encryption to authenticate users. Each framework has a differerent semantics and offers a different kind of flexibility.

Formal logic has been used successfully to explain authentication frameworks and protocols, most notably in the design of the Taos distributed operating system [1, 6]. The designers of Taos started by constructing an elegant and expressive logic of authentication as an extension of propositional calculus. They proved this logic sound — any provable statement is true in all models — which makes the logic an attractive basis for constructing a system. However, since they wanted a server presented with a request to be able to decide whether to grant the request, they chose to implement

*To appear in *6th ACM Conference on Computer and Communications Security*, November 1999.

only a decidable subset of their authentication logic. In a decidable logic, there is an algorithm for determining the truth (or falsehood) of any statement.

Using a simple and decidable logic would have had advantages: it's easier to prove metatheorems such as "there's no way Alice can access the file `bar`." Why, then, do we use an undecidable logic? Previous approaches have taken a set of basic inference rules and added application-specific inference rules to make application-specific logics; but the new rules must be trusted, i.e. proved sound. By allowing quantification over predicates, we can use a single set of inference rules, with application-specific rules proved as lemmas; therefore, the application-specific rules can be used to prove access requests to servers that know only the basic logic. However, logics with quantification over predicates – higher-order logics – tend to be undecidable: there is no general algorithm for producing proofs of all true statements.

Still, a server presented with a request must be able to figure out what to do. We solve this problem by analogy with proof-carrying code [9]: the client desiring access must construct a proof, and the server will simply check that proof. Even in an undecidable logic, proof *checking* can be simple and efficient. We put the burden of proof on the requester. We will show several different strategies by which requesters can construct proofs, for example by picking an application-specific decidable subset of our general logic.

Each of the existing frameworks has chosen a particular set of concepts and abstractions: a particular form of delegation, a key-binding mechanism, certain access control rules, and so on. Although these choices are as reasonable as any, no set of choices is right for everybody. By using higher-order logic, our framework allows the specification and use of new abstractions and new variants on the existing abstractions.

Since all the application-specific logics are expressed using the same general inference rules, they can safely interoperate: We can take one theorem proved using the SPKI definitions, and another proved using Taos definitions, and combine them to prove access even to a server that has seen neither set of definitions before.

Although we have expressed SPKI in our logic, we don't really recommend the use of SPKI 5-tuples for authorization; the operators we outline in sections 4 and 5 seem more natural.

2 An Example

Suppose three principals, a client Alice, a file server Bob, and a certification authority Charlie, are interacting across a network. (See Figure 1.) Bob receives a request to "read foo." Bob's access control list says that a principal called Alice can read foo — but is the request from Alice? Bob trusts Charlie to guarantee key-to-name bindings, and Bob knows that Charlie's key is K_c. Alice has obtained a certificate signed by K_c that "K_a is Alice's key," and uses K_a to sign "read foo." Armed with all of this information, Bob can safely grant the request to read foo.

Our framework can express this as follows. We can treat "read foo" as an uninterpreted atom, meaning that the logic doesn't know what it means although the participants do. Digital signing is a primitive of our logic, so (K_a signed *read foo*) is a statement of the logic.

Each principal is modeled as the set of formulas that she will admit as true. Thus, $\mathit{Alice}(\forall x. x \to x)$ means that *Alice* is willing to claim that $(\forall x. x \to x)$. That's not admitting much, as the statement is a tautology! But in fact any principal is required to admit any statement provable from other statements she admits.

We translate the statement "Alice wants to read foo" as $\mathit{Alice}(\mathit{read\ foo})$. We translate "read foo if Alice says to" as follows:

$$\mathit{Alice}(\mathit{read\ foo}) \ \to\ \mathit{read\ foo}.$$

Charlie's certification that K_a is Alice's key

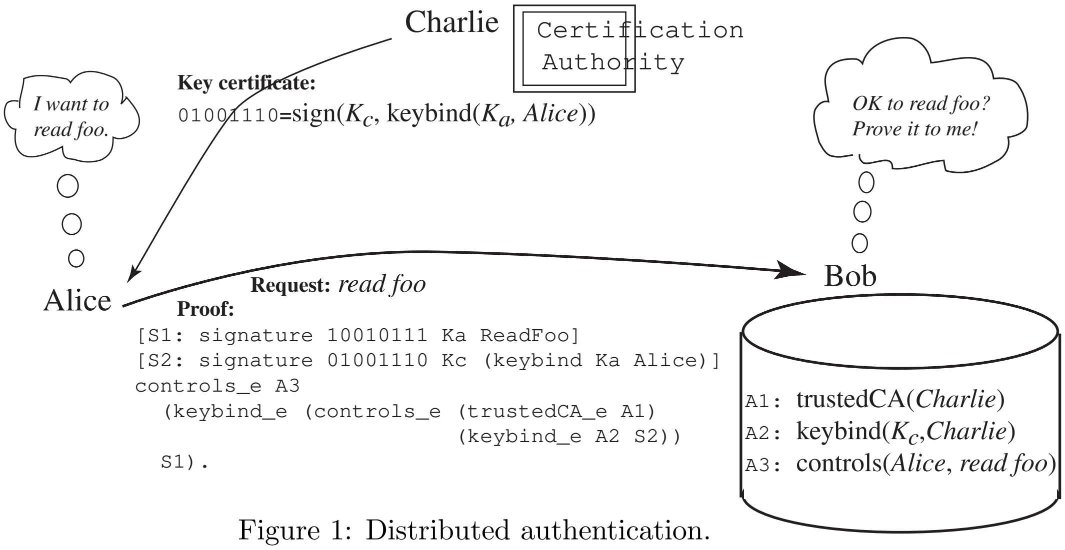

Figure 1: Distributed authentication.

we translate as

$$K_c \text{ signed } (\forall F. (K_a \text{ signed } F) \rightarrow (\mathit{Alice}(F))).$$

Bob's knowledge of Charlie's key is stated as

$$\forall F.\ (K_c \text{ signed } F) \rightarrow \mathit{Charlie}(F).$$

Finally, Bob's trust in Charlie is expressed as a memorandum to himself:

$$\begin{array}{l} \forall k. \forall p. \\ \mathit{Charlie}((\forall S. (k \text{ signed } S) \rightarrow p(S))) \rightarrow \\ \quad \forall S. (k \text{ signed } S) \rightarrow p(S). \end{array}$$

These statements are expressed in a higher-order logic (i.e., one in which we can quantify over formulas and predicates), with inference rules given in Figure 3.

Since these statements are rather unwieldy, it is only natural for the participants to have agreed upon some definitions:

keybind(k, p) stands for $\forall S. (k \text{ signed } S) \rightarrow p(S)$. That is, k is p's key, so that if k signs some statement S then p should be considered to believe it.

p controls S stands for $p(S) \rightarrow S$. That is, p "controls" the statement S, so if p says S then S will be considered true.

trustedCA(c) stands for $\forall k. \forall p. c$ controls keybind(k, p), meaning that c is a trusted certification authority: if c says some key binding, we can trust that key binding.

These particular definitions may be well-suited to our example transaction, but they are hardly likely to be general enough for the real world. The strength of our approach is that it does not "build in" a fixed set of definitions but allows the participants in each application to define and use their own abbreviations.

Some useful lemmas follow immediately from these definitions. For example, we can prove the lemma *controls_e,* which states that if p controls S and p says S, then S is true:

$$\frac{p \text{ controls } S \qquad p(S)}{S}\,\text{controls\_e}$$

Additional lemmas allow one to use statements by a trusted certification authority, and to make inferences from key bindings:

$$\frac{\text{trustedCA}(c)}{c \text{ controls keybind}(k, p)}\,\text{trustedCA\_e}$$

$$\frac{\text{keybind}(k, p) \qquad k \text{ signed } S}{p(S)}\,\text{keybind\_e}$$

2.1 Constructing a proof.

Bob starts with the following assumptions in his security database:

$$A_1 \quad = \quad \text{trustedCA}(\mathit{Charlie})$$

$$\dfrac{A \text{ controls } \mathit{read\ foo} \qquad \dfrac{\dfrac{\dfrac{\text{trustedCA}(C)}{C \text{ controls keybind}(K_a, A)}\text{trustedCA\_e} \qquad \dfrac{\text{keybind}(K_c, C) \qquad K_c \text{ signed keybind}(K_a, A)}{C(\text{keybind}(K_a, A))}\text{keybind\_e}}{\text{keybind}(K_a, A)}\text{controls\_e} \qquad K_a \text{ signed } \mathit{read\ foo}}{A(\mathit{read\ foo})}\text{keybind\_e}}{\mathit{read\ foo}}\text{controls\_e}$$

Figure 2: Proof that Alice can read foo.

$$\begin{aligned} A_2 &= \text{keybind}(K_c, \mathit{Charlie}) \\ A_3 &= \mathit{Alice} \text{ controls } \mathit{read\ foo} \end{aligned}$$

Using these assumptions, and the lemmas shown above, Alice can prove to Bob that he should allow her to read foo. Before Alice can do this, Bob must publish his assumptions – his security policy – so that she can construct a valid proof. Some parts of the policy – such as "*Alice* controls *read foo*" – he may not wish to broadcast to the whole world, but he should at least tell each client the assumptions relevant to her.

The proof is illustrated in Figure 2. It has five premises: the security policy A_1, A_2, A_3 and two digital signatures sent by Alice with the proof. Alice can sign the statement *read foo* with her own key K_a; she must ask Charlie to sign the statement keybind(K_a, A) with his own key K_c.

The proof is checked as follows. First we check the two digital signatures to establish the facts K_a signed *read foo* and K_c signed keybind(K_a, A). From assumption A_1 we prove C controls keybind(K_a, A) by the lemma trustedCA_e. From A_2 and a digital signature we prove C(keybind(K_a, A)) by the keybind_e lemma. Now from C controls keybind(K_a, A) and C(keybind(K_a, A)) we prove keybind(K_a, A) by the controls_e lemma. From this and a signature we prove A(*read foo*) by the keybind_e lemma. Finally, from assumption A_3 and A(*read foo*) we prove *read foo* by controls_e.

The definitions, and the proofs of the lemmas, can be included with the proof for the benefit of any participant who was not already familiar with the definitions and their consequences.

3 The Logic

We are using a higher-order logic with the type *form* of formulas and a base type, *string*, which will be used to represent keys, signatures, local names, and for many other purposes. The inference rules of the logic are given in Figure 3; except for the last four rules, it is standard higher-order logic. The last four inference rules allow reasoning about digital signatures and statements by principals derived from signatures and names (strings). For reasoning about time, we also need the type *integer* and rules for arithmetic, which we do not show here.

The type *form* → *form* – which is a predicate on formulas – represents the *worldview* of an actor in the system: a principal (an individual or machine), a group of principals, or some other such combination. We will define a *principal* P as a worldview that has the properties

$$\forall F.\, F \to P(F) \qquad \text{and}$$
$$\forall F \forall G.\, P(F) \land P(F \to G) \to P(G)$$

that is, if F is true then P admits it, and if G is a consequence of other things P admits, then P admits G.

A principal may be constructed from a string by the built-in function $\mathcal{N}()$. For any string s the inference rules *name_r* and *name_imp_e* guarantee that the worldview $\mathcal{N}(s)$ satisfies the properties required of principals.

We represent a cryptographic key as a string in the X.509 SubjectPublicKeyInfo format, that is, an AlgorithmIdentifer followed by a bit

$$\frac{A \quad B}{A \wedge B}\text{and\_i} \qquad \frac{A \wedge B}{A}\text{and\_e1} \qquad \frac{A \wedge B}{B}\text{and\_e2}$$

$$\frac{A}{A \vee B}\text{or\_i1} \qquad \frac{B}{A \vee B}\text{or\_i2}$$

$$\frac{A \vee B \quad \begin{array}{c}[A]\\ C\end{array} \quad \begin{array}{c}[B]\\ C\end{array}}{C}\text{or\_e}$$

$$\frac{\begin{array}{c}[A]\\ B\end{array}}{A \to B}\text{imp\_i} \qquad \frac{A \to B \quad A}{B}\text{imp\_e}$$

$$\frac{A(Y) \qquad Y \text{ not occurring in } \forall x.A(x)}{\forall x.A(x)}\text{forall\_i}$$

$$\frac{\forall x.A(x)}{A(T)}\text{forall\_e}$$

$$\frac{X = Z \quad H(Z)}{H(X)}\text{congr} \qquad \frac{}{X = X}\text{refl}$$

$$\frac{F}{\mathcal{N}(s)(F)}\text{name\_i}$$

$$\frac{\mathcal{N}(s)(F) \quad \mathcal{N}(s)(F \to G)}{\mathcal{N}(s)(G)}\text{name\_imp\_e}$$

$$\frac{\text{digital\_signature}(s, k, F)}{\mathcal{N}(k)(F)}\text{signed}$$

Figure 3: Inference rules of the logic. Quantification and equality (e.g., $\forall x.\exists y.x = y$) are polymorphic: the quantified variables may be of base type (such as *string*) or formulas, functions, or predicates. The logic also has operators and inference rules for integer arithmetic, existential quantification, and falsehood, which are not shown here.

string that is the actual key [15]. The rule *signed* says that if s is the digital signature with key k of statement F, then $\mathcal{N}(k)(F)$: the principal $\mathcal{N}(k)$ must admit any statement signed by k. The informal k signed F stands for the formal $\mathcal{N}(k)(F)$.

There is an infinite family of *digital_signature* axioms, one for each triple s, k, F where s is the digital signature with key k of the representation of the formula F. The proof-checking infrastructure must be able to check digital signatures using one or more encryption algorithms.

Because definitions such as "keybind" and "controls" are not part of our trusted computing base – such definitions are part of application-specific customizable protocols – each server must be able to mechanically check all proofs of their properties. As we will explain in Section 8, we have implemented a proof-checker for our logic (as will be necessary for the operation of a server in a distributed authentication system). *Every lemma and theorem that we state in this paper has been mechanically verified.*

4 Using the Logic

Since a worldview is just a predicate on formulas, worldviews can exhibit a wide variety of behaviors. This fact can be useful, as we will see below, but it also has its drawbacks. To simplify things, it is useful to define two special classes of worldviews, and to prove some lemmas about the behavior of these kinds of worldviews.

Principals. The first special class of worldviews includes those made by the $\mathcal{N}()$ operator, and other worldviews that behave like them; these satisfy

$$\begin{aligned}&\text{prin}(P) \equiv \\ &\quad (\forall F.\, (F \to P(F))) \\ &\quad \wedge(\forall F \forall G.(P(F) \to P(F \to G) \to P(G))).\end{aligned}$$

Intuitively, $\text{prin}(P)$ means that we can assume that P admits all tautologies, and that P

admits any formula that can be deduced from other formulas it admits. The following two lemmas are provable:

$$\frac{\text{prin}(P) \qquad F}{P(F)}\text{prin\_taut}$$

$$\frac{\text{prin}(P) \quad P(F) \quad P(F \to G)}{P(G)}\text{prin\_imp\_e}$$

From the *name_r* and *name_imp_e* inference rules, we can prove the lemma

$$\frac{}{\text{prin}(\mathcal{N}(s))}\text{n\_is\_prin}$$

We can use function composition on worldviews:

$$(A \circ B)(F) \equiv A(B(F)).$$

$A \circ B$ can be thought of as "A quoting B". Composition of two principals is a principal:

$$\frac{\text{prin}(A) \qquad \text{prin}(B)}{\text{prin}(A \circ B)}\text{prin\_comp}$$

We can also define a $\wedge$_w operator on worldviews, so that $A \wedge \_\text{w}B$ admits a formula whenever both A and B admit it.

$$(A \wedge \_\text{w}B)(F) \equiv A(F) \wedge B(F).$$

We can then prove the lemma

$$\frac{\text{prin}(A) \qquad \text{prin}(B)}{\text{prin}(A \wedge \_\text{w}B)}\text{prin\_}\wedge$$

along with lemmas saying that $\wedge$_w is associative and commutative.

Similarly, we can define a $\vee$_w operator:

$$(A \vee \_\text{w}B)(F) \equiv A(F) \vee B(F).$$

The $\vee$_w operator can be used to construct groups of worldviews in which any member can speak on behalf of the group. $\vee$_w is associative and commutative. However, if prin(A) and prin(B), this does *not* imply prin($A \vee \_\text{w}B$). (For example, we could have $A(F)$ and $B(F \to G)$, but neither $A(G)$ nor $B(G)$.) Groups evidently satisfy a weaker property than prin(); we write this property as

$$\text{group}(A) \equiv \forall F.(F \to A(F)).$$

We can then prove the following lemmas:

$$\frac{\text{prin}(A)}{\text{group}(A)}\text{prin\_is\_grp}$$

$$\frac{\text{group}(A) \qquad F}{A(F)}\text{grp\_taut}$$

$$\frac{\text{group}(A) \qquad \text{group}(B)}{\text{group}(A \circ B)}\text{grp\_comp}$$

$$\frac{\text{group}(A) \qquad \text{group}(B)}{\text{group}(A \wedge \_\text{w}B)}\text{grp\_}\wedge$$

$$\frac{\text{group}(A) \qquad \text{group}(B)}{\text{group}(A \vee \_\text{w}B)}\text{grp\_}\vee$$

In addition to these classes of worldviews, any application is free to define its own classes.

4.1 Says and Quoting

The system as described so far is adequate, but some may find the representation of principals as functions confusing. In order to make the system more palatable to these users, we can define operators to abstract away the representation of worldviews.

We do this by defining a *says* operator:

$$A \text{ says } F \equiv A(F).$$

Some simple lemmas about *says* follow:

$$\frac{\text{group}(A) \qquad F}{A \text{ says } F}\text{says\_taut}$$

$$\frac{\text{prin}(A) \quad A \text{ says } F \quad A \text{ says } (F \to G)}{A \text{ says } G}\text{says\_imp\_e}$$

We can then define a quoting operator $|$ as

$$A|B \equiv A \circ B,$$

and it follows that

$$\frac{}{(A|B) \text{ says } F \leftrightarrow A \text{ says } (B \text{ says } F)}\text{quote\_ident}.$$

The $\wedge\_w$ and $\vee\_w$ operators will lead to the identities

$$\overline{(A \wedge \_wB) \text{ says } F \leftrightarrow (A \text{ says } F) \wedge \_w(B \text{ says } F)}$$

$$\overline{(A \vee \_wB) \text{ says } F \leftrightarrow (A \text{ says } F) \vee (B \text{ says } F)}$$

If we use a logical framework that support abstract data types, we could choose to hide the representation of principals, says, and quoting, and simply expose the set of lemmas they satisfy.

5 Application-Specific Operators

We expect that the designers of specific applications will often have in mind their own sets of operators and rules for manipulating them. Our system allows users to define new operators with manipulation rules, so it can implement application-specific operators. In this section, we give as an example a set of operators we have found useful.

Any application using our system is free to define whatever operators it likes and prove lemmas about them. This kind of extensibility is the main advantage of our logic-based approach.

Controls. First we define the *controls* operator[1]

$$A \text{ controls } F \equiv ((A \text{ says } F) \rightarrow F).$$

Informally, A controls F means that A can make F true just by saying it. We can prove a simple lemma

$$\frac{A \text{ controls } F \qquad A \text{ says } F}{F}\text{controls\_r}\quad .$$

We can then prove a lemma that says control can be handed off from one principal to another:

$$\frac{\text{prin}(A) \quad A \text{ controls } F \quad A \text{ says } (B \text{ controls } F)}{B \text{ controls } F}\quad .$$

[1]This definition differs from the one given in Section 2 in order to account for the *says* abstraction.

Speaksfor. The operator

$$A \text{ speaksfor } B \equiv \forall x.(A \text{ says } x \rightarrow B \text{ says } x)$$

says that everything said by A is also said by B. Generally, if A speaks for B, then A can exercise any rights that B has.

Names. The ability to translate any string S into a principal $\mathcal{N}(S)$ gives us the ability to define name-spaces. For example, we can define a *localname* operator as

$$\mathcal{LN}(A, S) \equiv A|\mathcal{N}(S).$$

Now A can give principal B the right to speak for $\mathcal{LN}(A, S)$ by making the statement

$$A \text{ says } (B \text{ speaksfor } \mathcal{N}(S)).$$

Now (assuming prin(A)) it follows that B speaksfor $\mathcal{LN}(A, S)$[2].

We can also use local names to implement roles, with $\mathcal{LN}(A, \texttt{role}:\texttt{admin})$ referring to A's administrative role; A could then make a statement F on behalf of the role, such as (A says ($\mathcal{N}(\texttt{role}:\texttt{admin})$ says F)).

5.1 Access Control

Suppose a file server machine M stores a file f, and M wants to limit who can read f. (We assume prin(M).) This can be implemented by requiring each read request on f to carry a proof of (M says *read*(f)). M can give another principal A permission to read the file by making the statement (M says (A controls *read*(f))). Now if A makes the request (A says *read*(f)), A can use the request, along with the statement made by M, to prove M says *read*(f)[3].

[2]Proof sketch: Suppose B says F. It follows by says_taut that A says (B says F). Now we have two statements that are said by A: B says F and B speaksfor $\mathcal{N}(S)$. These two statements together imply $\mathcal{N}(S)$ says F, so when they are both said by A, and since prin(A) holds, we can infer that A says ($\mathcal{N}(S)$ says F). By the definition of *localname*, this equals $\mathcal{LN}(A, S)$ says F.

[3]To prove this result, we start with A says *read*(f), and then (using prin(M)) apply the says_taut lemma to

5.2 Public-Key Infrastructure

Public-key certificates and their use can be implemented in our system as well. A certification authority C can issue a certificate binding key K_a to principal A:

$$(\text{cert } C\ K_a\ A) \equiv C \text{ says } (\mathcal{N}(K_a) \text{ speaksfor } A).$$

This certificate is pointless unless we trust C to issue certificates; this trust can be encoded as

$$\text{trustedCA}(C) \equiv \forall k \forall p.\, C \text{ controls } (\mathcal{N}(k) \text{ speaksfor } p)$$

Now we can prove lemmas such as

$$\frac{\text{trustedCA}(C) \qquad (\text{cert } C\ K_a\ A) \qquad K_a \text{ signed } F}{A \text{ says } F}$$

indicating that our definitions are consistent with one model of public-key infrastructure.

Of course, other definitions may make sense to other people. The strength of our system is that it allows anyone to make their own definitions.

5.3 Expiration and Revocation

The primitive now() maps strings to integers; now(m) gives the current time as measured on the clock of the host whose name is m. We make no assumptions about how different clocks are related, though any principal can make statements and form beliefs about how any two clocks are related.

When Bob checks the proof of any statement, he will have an assumption in his assumption database of the form,

$$\text{now}(\texttt{bob.com}) = 83487$$

if 83487 is the current time.

deduce M says (A says *read*(f)). Then we start with M says A controls *read*(f) and expand the definition of controls to deduce M says ((A says *read*(f)) $\rightarrow$ *read*(f)). The proof is completed by combining the results of the previous two sentences, using the *says_imp_e* lemma (and the assumption prin(M)) to deduce M says *read*(f).

Statements can be given limited periods of validity. For example, the statement

$$A \text{ says } ((\text{now}(\texttt{Bob}) < T) \rightarrow F),$$

implies A says F until Bob's clock reaches T; after that it is vacuous. Of course, a statement that expires might be renewed with a later expiration time, and statements might come with hints about how to renew them. In order for Alice to generate a proof that will be valid on Bob's machine, she should first learn how much clock skew there is between her machine and Bob's. Also, if Charlie trusts Bob to keep the clock skew between Bob and Charlie under 5 seconds, at least until tomorrow, he can issue the statement,

$$\begin{aligned}&K_c \text{ signed } (\text{now}(\texttt{charlie.com}) < 100929 \ \rightarrow \\ &\qquad |\text{now}(\texttt{charlie.com}) - \text{now}(\texttt{bob.com})| < 5)\end{aligned}$$

Such certificates about clock skew can help Alice formulate a proof that her key-certificate (signed by Charlie) is still valid with respect to Bob's clock.

An alternative to expiration is revocation. For example, the statement

$$A \text{ says } (\neg\text{revoked}(F) \rightarrow F)$$

says that A says F, unless F has been revoked. A could periodically send out a revocation list; for example

$$\begin{aligned}&A \text{ says } ((\text{now}(A) < T) \rightarrow \\ &\quad \forall x.(\text{revoked}(x) \rightarrow (x = S_1 \lor x = S_2 \lor x = S_3))).\end{aligned}$$

From this we can infer that (until A's clock reaches T) statements other than S_1, S_2, and S_3 have not been revoked. By making T only a short time in the future, we can achieve the effect of on-line revocation list checking. As usual, the cost of frequent revocation list updates has to be balanced against the time required for revocations to propagate through the system.

6 Alternative Versions of Says

The definition of says given above seems natural, but some applications may want a version of says that behaves differently. These applications are free to define and use the primitives they want. We now give two examples of alternate definitions of says.

6.1 Says without prin()

Some users may find it more natural to reason about a world where rules like says_taut and says_imp_e hold for *all* principals. This can be achieved by defining a new operator:

$$A \text{ saysp } F \equiv F \vee (\text{prin}(A) \wedge A(F)).$$

Two lemmas follow:

$$\frac{F}{A \text{ saysp } F}$$

$$\frac{A \text{ saysp } F \qquad A \text{ saysp } (F \rightarrow G)}{A \text{ saysp } G}$$

We can then proceed to define new versions of the other operators such as *controls*, and to prove the corresponding lemmas.

6.2 Says without Delegation

Our first definition of *says* allows any principal to delegate any rights it may have. Some applications may not like this, so we may want to define a version of *says* that does not allow arbitrary delegation. To do this, we define a "says directly" operator:

$$A \text{ saysdir } F \equiv \exists K.((A = \mathcal{N}(K)) \wedge (K \text{ signed } F)).$$

Now if B says $((A \text{ saysdir } F) \rightarrow F)$, B has delegated to A all of B's rights to F, but A cannot delegate these rights further, since the only way A can exercise these rights is to directly sign a request to use them.

We can define a "says with optional delegation" operator:

$$A \text{ saysopt}_D\ F \equiv (A \text{ saysdir } F) \vee (D \wedge (A \text{ says } F)).$$

Now A can delegate its rights to B:

$$A \text{ says } ((B \text{ saysopt}_D\ F) \rightarrow F)$$

and B can delegate these rights to third parties if and only if D is true.

There are many variants of *says*, and our system allows each application to define and use the variant that best suits its needs.

7 Encoding Other Authentication Frameworks

Our system is general enough to encode other distributed authentication frameworks. We encode a framework by expressing its primitives in a set of definitions, and then proving the framework's processing rules as lemmas. By providing a single language in which the semantics of several frameworks can be expressed, we provide a way for those frameworks to interoperate with users of our system.

By describing different frameworks in a single logic, we provide a way for those frameworks to interoperate in a principled way. For example, if the semantics of SPKI certificates and Taos certificates are clearly specified in our logic, then certificates from both frameworks can be used together in the same proof, as long as the requester can show they satisfy the server's requirements.

Interoperability has another advantage: it facilitates the deployment of new frameworks. If frameworks can interoperate, then a new framework does not need near-universal deployment in order to attract users. A new framework can be used by a few people at first, while those people exploit interoperation to work with the rest of the world.

7.1 SPKI

As an example, we now describe how to encode the SPKI [4] framework. *Certificates* are the main data structure in SPKI; a certificate can encode a name-to-key binding or a name-to-privileges binding.

The SPKI specification describes how every certificate can be translated into a *5-tuple* data structure, and it gives rules for combining 5-tuples to deduce new 5-tuples, and for deciding whether a particular 5-tuple is sufficient evidence to allow access to a resource. In the 5-tuple (I, S, D, T, V)

I is the key that issued the certificate;

S is the subject of the certificate, which could be a key, a name, or a group;

D is a delegation flag, saying whether the rights associated with the certificate may be delegated;

T is a tag, a data structure that describes a request or set of requests; and

V says when the certificate is valid, giving a not-valid-before and a not-valid-after time.

Space does not permit a full description of SPKI semantics; see the SPKI documents [4] for full details.

Encoding SPKI To encode SPKI, we encode the 5-tuple data structure and the rules for manipulating 5-tuples. Since converting certificates to 5-tuples is a straightforward (though tedious) translation process, we could also encode this conversion in our logic.

In practice, one would want to model the SPKI data structures in great detail, for example, by expressing the tag-matching rules in logic. We simplify the model here for brevity.

We give the following definition:

$$\begin{aligned}&\text{tuple}(I, S, D, T, V) \equiv V \rightarrow \\ &\quad I \text{ says } (\forall x.((S \text{ saysopt}_D \text{ ok}(x)) \rightarrow \\ &\qquad\qquad\qquad\qquad T(x) \rightarrow \text{ok}(x))).\end{aligned}$$

Here I is the issuing principal; S is the subject principal, which might be a group; D is the delegation flag; the tag T is represented as a predicate on strings (a predicate which admits any string iff that string will successfully match the tag); and the validity interval is represented as a simple predicate V. $\text{ok}(x)$ represents the assertion the the operation specified by x should be permitted; we encode it as the placeholder $\text{ok}(x) \equiv (\mathcal{N}(ok)|\mathcal{N}(x))$ says false.

Now we can prove SPKI's rules for manipulating 5-tuples as lemmas. To give a simple example, we can prove

$$\frac{\begin{array}{c}\text{prin}(I) \quad \text{prin}(J) \\ \text{tuple}(I, J, \text{true}, A, V) \quad \text{tuple}(J, S, D, A, V)\end{array}}{\text{tuple}(I, S, D, A, V)}$$

We could also go about proving the other rules for reasoning about 5-tuples, including a rule for extracting a useful result from a SPKI chain:

$$\frac{\begin{array}{c}J \text{ signed ok}(X) \quad \text{prin}(I) \quad V \quad T(X) \\ \text{tuple}(I, \mathcal{N}(J), D, T, V)\end{array}}{\text{ok}(X)}$$

Having done this, we could conclude that any client who could have gotten approval for operation X on server I in SPKI will be able to prove I says $\text{ok}(X)$. SPKI provides a way to define local access policies such as

$$\text{ok}(\texttt{tag (pkpfs /foo read)}) \rightarrow \textit{read /foo}$$

Although we have not yet done so, we believe that other distributed authentication frameworks could be encoded in a similar way.

8 Implementation

Proofs must be produced by the client requesting services and checked by the server, so there must be a machine-readable and -checkable notation for theorem and proof. We use a higher-order logic implemented in Twelf [13], an implementation of the Edinburgh Logical Framework [5]. Research in proof-carrying code [11] has shown that the Logical Framework (LF) is an excellent notation for explicit proofs that are to be transmitted and then checked with a minimal trusted computing base. The algorithm for checking LF proofs is as simple as programming-language type checking, and in fact the Touchstone system for proof-carrying code [11] includes a simple proof-checker written in the C programming language.

An earlier version of our system was implemented in λProlog instead of Twelf, using

```
controls_e:  pf (controls @ A @ F) -> pf (A @ F) -> pf F.
keybind_e:  pf (keybind @ K @ W) -> pf (digital_signature S K F) -> pf (W @ F).
trusted_ca_e:  pf (trusted_ca @ Ca) -> pf (controls @ Ca @ (keybind @ K @ A)).

abc: pf (trusted_ca @ Charlie) ->
     pf (keybind @ Kc @ Charlie) ->
     pf (controls @ Alice @ ReadFoo) ->
     pf (digital_signature B10010111 Ka ReadFoo) ->
     pf (digital_signature B01001110 Kc (keybind @ Ka @ Alice)) ->
  pf ReadFoo =

 [A1][A2][A3][S1][S2]
 controls_e A3 (keybind_e (controls_e (trusted_ca_e A1) (keybind_e A2 S2)) S1).
```

Figure 4: Twelf representation of "read foo" theorem and its proof.

higher-order logic with lemmas and definitions as described by Appel and Felty [2].

8.1 Proof representation

Figure 4 shows the Twelf code that implements the theorem that Alice proves to Bob in the example of Section 2.1. The first three clauses are the statements of the supporting lemmas; here we have omitted their proofs.

The name of the theorem, `abc`, is followed by a statement of the theorem: given proofs of the five premises, we get a proof of `ReadFoo`.

After the `=` sign comes the proof of the theorem. The variables `[A1][A2][A3][S1][S2]` stand for the five premises; the square brackets indicate that they are formal parameters of the proof, and the body of the proof may refer to them by name. We have chosen the name `A1,A2,A3` for the first three premises to correspond to the names used in Section 2.1. The variables `S1,S2` stand for the two digital-signature premises.

The last line shows the proof tree, written out in Twelf notation. It is the same tree shown in Figure 2, but in a form more suited to machine processing.

Another illustration is the *controls_e* lemma (Figure 5). The first line declares the type of the name `controls`, as a predicate, i.e. a function taking a worldview and a formula and returning a formula. Next, it defines the name `controls` to stand for the (higher-order) formula $\lambda a \lambda f. (a\, f) \rightarrow f$. We have made functions (λ) explicit in our logic using the `lam` operator, and function-application explicit using the `@` operator.

The `controls_e` lemma states that, given a proof that A controls F, and a proof that A says F, we can construct a proof of F. The proof first uses another lemma, `def2_e`, to expand the definition of `controls` in the premise `P1`; then the implication-elimination rule `imp_e` to complete the proof.

The Twelf notation will allow the recipient of a proof (and associated lemmas) to check the validity of each lemma, and then to use the lemma in checking the proof of the main theorem (and of other lemmas).

8.2 Measurements

The implementation of our logic in Twelf is quite concise. Such a checker could be used as the core of a distributed authentication system. The logical inference rules are specified in 46 lines, excluding rules for arithmetic, which can be specified in another dozen or so.

The Twelf proof-checker itself [13] is implemented in Standard ML [8]: the parser is 2549 lines of commented code, and the proof-checking algorithm is 1540 lines. The parser need not be considered part of the trusted computing base, since a broken parser cannot

```
controls: tm (worldview arrow form arrow form) = lam[A] lam[F] (A @ F) imp F.

controls_e: pf (controls @ A @ F) -> pf (A @ F) -> pf F = [P1][P2] imp_e (def2_e P1) P2.
```

Figure 5: Representation of `controls` and `controls_e`.

cause an invalid proof to be accepted by the checker. Necula [11] has also implemented an LF proof-checker; its current size is about 2000 lines of commented C code [7].

Clients who want to generate proofs will need application-specific definitions and lemmas, which might amount to several hundred lines of Twelf, but this will be outside the trusted computing base – the lemmas can be independently checked by the recipients of proofs.

Proofs are really just s-expressions extended with a primitive notion of binding; that is, trees whose operators are names of inference rules and lemmas. Each use of a lemma with n arguments should add approximately n words to the size of the proof, and each statement-and-proof of a lemma should add a number of words proportional to the size of the lemma and its proof. A good approximation to the size, in words, of the representation of a proof is the number of non-punctuation tokens in the fully explicit form of its Twelf syntax. We can measure the size of the proof illustrated in Figure 1:

Concept	Definition	Lemma	Proof
controls	21	29	33
keybind	20	39	42
trustedCA	32	15	33
Main theorem		84	123

Since Bob's security database contains statements of the form trustedCA($Charlie$), he presumably also has copies of the relevent definitions, lemmas, and proofs. Thus, Alice can simply send a 84-word theorem and 123-word proof to justify *read foo*. But if other lemmas turn out to be helpful in structuring the proof, they can be represented in a very reasonable size, as the table shows. These numbers are gross overestimates of what can be achieved in practice; Necula [10] has shown methods of reducing the redundancy in LF proofs and cutting their size by large factors.

Some servers – such as programmable disk controllers or active-network routers – are so specialized that they will not even want to have a full proof checker. In this case, they can rely on certificates. Suppose Doris the disk controller trusts Bob to check proofs for her; she can rely on the single inference rule

$$\frac{K_b \text{ signed } (\mathit{Doris} \text{ says } F)}{F}\text{trust\_bob}$$

Now if Alice wants service *format disk* from Doris, she can submit a proof of (Doris says *format disk*) to Bob, who will check it and issue the certificate.

Bob should not sign time-dependent statements such as now(`bob.com`) = 1998 which can become false. He can avoid this by not putting any such assumptions into his database while checking proofs of statements that he is being asked to sign.

9 Conclusion

Higher-order logic allows application-specific modal logics to be defined and proved in a simple and general framework. The apparent disadvantage of such a logic — undecidability — can be overcome by submitting a proof with each authentication request, in the same way that proof-carrying code submits proofs of safety with programs.

We have demonstrated that proofs can be small, that proof checking is simple to implement, and that existing authentication frameworks can be expressed as application-specific definitions and lemmas in our logic. Although proof generation is undecidable in general, we have shown by example that in cases of interest, the requester will have a good idea why she should be able to access a resource.

References

[1] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin. A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] Andrew W. Appel and Amy Felty. Lightweight Lemmas in Lambda Prolog. In *16th International Conference on Logic Programming*. MIT Press, November 1999.

[3] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Distributed Trust Management. In *Proc. of 17th IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[4] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple Public Key Certificate. Internet Draft draft-ietf-spki-cert-structure-05.txt, 1998.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, January 1993. To appear. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[6] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[7] Peter Lee. personal communication, 1999.

[8] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[9] George C. Necula. Proof-Carrying Code. In *Procedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, January 1997.

[10] George C. Necula and Peter Lee. Efficient Representation and Validation of Proofs. In *In Proceedings of the 13th Annual Symposium on Logic in Computer Science*, 1998.

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

[12] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, September 1994.

[13] Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[14] Ron Rivest and Butler Lampson. SDSI – A Simple Distributed Security Infrastructure. September 1996.

[15] International Telecommunications Union. ITU-T Recommendation X.509: The Directory: Authentication Framework. Technical Report X.509, ITU, www.itu.int, 1997.