A Proof-Carrying Authorization System

Lujo Bauer Michael A. Schneider Edward W. Felten
Secure Internet Programming Laboratory
Department of Computer Science
Princeton University

Tech Report TR-638-01

April 30, 2001

Abstract

We describe an infrastructure for distributed authorization based on the ideas of proof-carrying authorization (PCA). PCA is more general and more flexible than traditional distributed authorization systems. We extend PCA with the notion of goals and sessions, and add a module system to the proof language. Our framework makes it possible to locate and use pieces of the security policy that have been distributed across arbitrary hosts. We provide a mechanism which allows pieces of the security policy to be hidden from unauthorized clients. As a prototype application we have developed modules that extend a standard web server and a standard web browser to use proof-carrying authorization to control access to web pages. The web browser generates proofs mechanically by iteratively fetching proof components until a proof can be constructed. We provide for iterative authorization, by which a server can require a browser to prove a series of challenges. Our prototype implementation includes a series of optimizations, such as speculative proving and modularizing and caching proofs, which allows proof-carrying authorization to be used with minimal performance and bandwidth overheads.

1 Introduction

Distributed authentication is important in systems that cannot have their access policy determined by a single trustworthy set of administrators. In these systems, different portions of the access policy may be controlled by different parties. The relationships between these parties and their respective areas of control can be quite complex.

Distributed authentication systems [7, 8, 12] can provide a framework under which it is possible to implement these complex policies. One example of such a policy uses a delegation. Suppose a user Alice has access to a file `foo`, which she has been assigned by a system administrator. Alice wants another user, Bob, to have access to the file as well. Rather than requiring the administrator to make policy changes for `foo`, the distributed authentication system could allow Alice to issue Bob a (perhaps temporary or restricted) delegation, "Bob speaks for Alice, signed Alice". Bob could then use this delegation to access `foo`.

Authentication frameworks have sometimes been described using formal logic [6, 10, 1]. In some cases frameworks were developed by first designing an appropriate logic and then building around it a system that supports it [5]. An example of this approach is the Taos operating system [2, 3].

Appel and Felten have recently introduced the idea of proof-carrying authorization[1] (PCA) [4], an authorization framework that is based on a higher-order logic (henceforth referred to as the AF logic). Their

[1] The concept was originally introduced as "proof-carrying authentication," but since it deals with authorizing access rather than authenticating identity, we have with the permission of the original authors renamed it to "proof-carrying authorization."

higher-order logic consists of a standard higher-order logic extended by a very few rules that they deem necessary for defining operators and lemmas suitable for a security logic.

Appel and Felten propose that an authorization framework should be composed of the AF logic and a set of operators and rules that comprise a particular security logic. The operators and rules are expressed in the AF logic; operators as definitions and rules as lemmas that can be proven from these definitions. Since each rule or operator can be expressed directly in the AF logic, systems based on the AF logic can communicate with each other even if they use different operators or inference rules. This allows adding complex security-policy rules that might not have been imagined when the system was designed.

A higher-order logic like the AF logic, however, is not decidable, which means that no decision procedure will always be able to determine the truth of a true statement, even given the axioms that imply it. This makes the AF logic unsuitable for use in traditional distributed authentication frameworks in which the server is given a set of credentials and must decide whether they imply some statement. This problem can be avoided in the server by making it the client's responsibility to generate proofs. The server must now only check that the proof is valid; this is not difficult even in an undecidable logic. Each client can generate proofs using a decidable subset of inference rules specific to its application. The server, using only the common underlying AF logic, can check proofs from all clients, regardless of the inference rules they use.

Building an actual distributed authorization framework based on an AF-style logic raises a number of issues that remain untouched or are not fully addressed by previous work. What set of defined operators can be used to make a practical security logic? Appel and Felten propose several sets, each with its own advantages and disadvantages. What is the minimal set of rules the AF logic needs for these operators to be definable? How does a proof goal correspond to a request to access a resource? If a proof gives its bearer access to a protected resource, how do we ensure that a proof isn't stolen or copied? Can proof-generation be completely automated? – after all, a proof-carrying authorization system is hardly useful if it requires a user to manually construct proofs, in higher order logic, to be authorized to access a web page. Can the security policy – the set of facts necessary to make a proof – be distributed in a way that makes it accessible to legitimate users but not to attackers? If all these questions can be answered constructively, is it possible to build a system that is general enough to be a significant improvement over existing ones and yet also efficient enough to be of practical use?

We present an implementation of a distributed authorization system that answers these questions and demonstrates the feasibility of using proof-carrying authorization in real systems. Our application consists of a web server that allows access to pages only if the web browser can demonstrate that it is authorized to view them. The browser accomplishes this by mechanically constructing a proof of a challenge sent to it by the server. Our system supports arbitrarily complex delegation, the definition of local name spaces, and expiration. We develop a framework that lets the web browser locate and use pieces of the security policy (e.g., delegation statements) that have been distributed across arbitrary hosts, and a system for providing selective access to these pieces.

The infrastructure we implemented is independent of the particular application we chose to build on top of it. The application is only an illustration of how the infrastructure can be used.

2 Example

Let us consider the following scenario. Bob is a professor who teaches CS101. He has put up a web page that has the answers to a midterm exam his class just took. He wants access to the web page to be restricted to students in his class, and he doesn't want the web page to be accessible before 8 P.M.

Alice is a student in Bob's class. It's 9 P.M., and she wants to access the web page (`http://server/midterm.html`) that Bob has put up. Her web browser contacts the server and requests the page `/midterm.html`. The server, seeing that the page is PCA-protected, responds with a request that Alice use an encrypted (HTTPS) connection. Alice's browser switches to HTTPS and again requests `/midterm.html` (figure 1, step 1).

Upon receiving this request, the server constructs a challenge (a statement in the logic) which must be

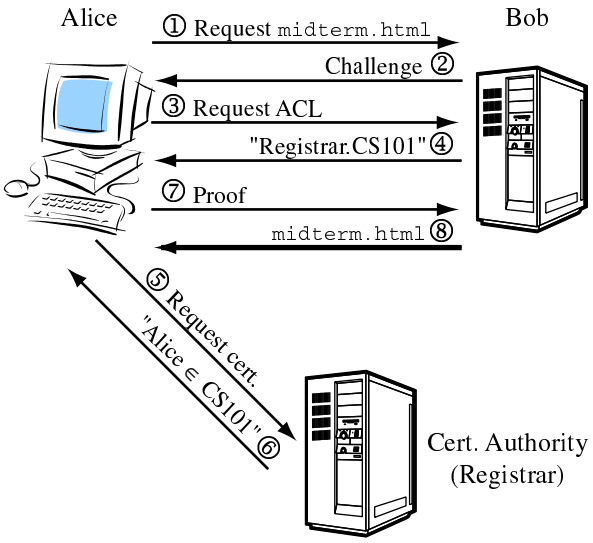

Figure 1. Alice wants to read `midterm.html`.

proven before the requested URL will be returned. The server returns an "Authorization Required" message (figure 1, step 2) which includes the challenge, "*You must prove:* The server says that it's OK to read `/midterm.html`."

When Alice receives the response, she examines the challenge and attempts to construct a proof. Unfortunately, the attempt fails: Alice has no idea how to go about proving that it's OK to read `/midterm.html`. She sends another request to the server: "Please tell me who can read `/midterm.html`." (step 3)

The server's reply (step 4) tells her that all the students taking CS101 (the Registrar has a list of them) may access the page, as long as it's after 8 P.M. Still, that does not give her enough information to construct the proof. She contacts the Registrar (step 5), and from him gets a certificate asserting, "until the end of the semester, Alice is taking CS101." (step 6) Alice has now collected the following facts:

- The server says, "After 8 P.M., everyone taking CS101 may access `/midterm.html`."
- The Registrar says, "Until the end of the semester, Alice is taking CS101."
- The server believes, "The time now after 8:55 P.M." (Alice guesses this, since her own clock shows it is 9:00 P.M.)
- Alice says, "It's OK to read `/midterm.html`." (This is relevant because the server will grant access only to those clients who assert that it's OK to read the requested URL.)

Finally, there is enough information to prove that Alice should be allowed to access the file. Once a proof is generated, Alice sends another request for `/midterm.html` to the prover (step 7). This time she includes in the request the challenge and its proof. The server checks that the proof is valid, and that Alice proved the correct challenge. If both checks succeed, the server returns the requested page (step 8).

3 Logic Design

A proof-carrying authorization system has a core logic (such as the AF logic) with an application-specific logic defined on top of it. The core logic must be sufficiently general to encode a wide range of application-specific logics – that is its primary purpose. On the other hand, it must also contain rules that make it possible to define interesting and useful application-specific logics. For instance, any security logic is likely to need an inference rule that transforms a digital signature into a statement in the logic. There is no convenient way to define such a rule as a provable lemma or definition – so an appropriate axiom should be made part of the core logic.

A standard higher-order logic comprises the majority of the core logic. The choice of the few non-standard rules that we wish to add depends on the functionality we wish the application-specific logics to have. We start the description of our system, therefore, with a discussion of requirements we had for our application-specific logic.

3.1 Application-Specific Operators and Rules

The application-specific security logic consists of operators (e.g., the *speaksfor* operator) and rules that allow us to reason about them (e.g., a transitivity rule for *speaksfor*). In a typical security logic the rules would represent the formal definitions of the operators; in a PCA system, however, both the rules and

$$\frac{A \text{ speaksfor } B \qquad A \text{ says } F}{B \text{ says } F}$$

$$\frac{A \text{ speaksfor } B \qquad B \text{ speaksfor } C}{A \text{ speaksfor } C}$$

$$\frac{A \text{ says } (A.s \text{ says } F)}{A.s \text{ says } F}$$

Figure 2. We want our application-specific operators to be defined so that they interact according to these rules. $A.s$ is shorthand for localname(A,s). s is a string in A's local name space from which A creates the principal localname(A,s).

the operators are expressed in terms of the core logic. Since the rules have to be proven as lemmas, the operators must be defined in such a way that the lemmas are provable. Moreover, the operators must in some cases be defined so that rules other than the intended ones *cannot* be proven. In a system with delegation, for example, we must make sure that rights can be delegated only in strictly allowed ways. Naively defined operators could result in an attacker discovering an unintended way to delegate authority and use this discovery to break the security of a system.

The desirada for our application-specific logic were fairly straightforward: principals should be able to make statements, delegate authority, and create groups or roles (we call them local name spaces). Figure 2 lists these requirements as inference rules. In addition, we wanted principals to be able to draw conclusions based on the things they believe, and we wanted individuals (that is, principals with keys) to be able to draw stronger conclusions than principals that represent roles or groups.

We define the required operators with the goal of being able to prove these rules based on the operators' definitions. A complicated definition may make it easier to prove a particular lemma, but may also make it impossible to prove lemmas that involve several different operators. Too simple a definition, on the other hand, might be insufficient to prove any interesting lemmas about the operator.

A* says *F In addition to the rules in figure 2 it should be true that a principal A *says* any statement that is true. Also, if A *says* the formula X, and the formula Y is true, and X and Y imply formula Z, then A also *says* Z – this allows the principal A to draw conclusions based on its beliefs.

$$A \text{ says } F \;\equiv\; \exists G \,.\, A(G) \wedge (G \rightarrow F)$$

A* speaksfor *B This operator is used for delegation. If principal A speaks for principal B, then anything that A says is spoken with principal B's authority.

$$A \text{ speaksfor } B \;\equiv\; \forall F \,.\, (A \text{ says } F) \rightarrow (B \text{ says } F)$$

A.s The principal $A.s$ (or $localname(A,s)$) is a new principal created in A's local name space from the string s. Principal A controls what $A.s$ says. In our example, the principal *registrar* creates the principal *registrar.cs101*, and signs a formula like 'key("alice") speaksfor (registrar."cs101")' for each student in the class.

$$A.s(F) \;\equiv\; \forall L \,.\, \text{lnlike}(L) \rightarrow L(A)(S)(F)$$

The *lnlike* operator is used to break the recursion in the definition of *localname*. The definition of *lnlike* looks complicated, but is such that $lnlike(L)$ is true for every function L that behaves as a local name should; that is, for every function that generates a principal whose authority A can delegate. *localname* is one of the operators explicitly defined so that it obeys only the set of rules that we require of it; this makes its definition somewhat more complicated and adds complexity to the proofs of lemmas about it.

$$\text{lnlike}(L) \;\equiv\; \forall A,S,F,G \,.\; ((A \text{ says } G) \text{ and } (G \rightarrow (L(A)(S) \text{ says } F))) \rightarrow L(A)(S)(F)$$

In addition to the rules from figure 2, we can prove as lemmas other inference rules that might be helpful for generating proofs. For example,

$$\frac{A \text{ says } F \qquad F \rightarrow G}{A \text{ says } G}\ \text{says\_imp}$$

can be trivially proven from the definition of `says`.

$$\frac{\text{signature}(\mathit{pubkey}, \mathit{fmla}, \mathit{sig})}{\text{Key}(\mathit{pubkey}) \text{ says } \mathit{fmla}} \text{ signed}$$

$$\frac{\text{Key}(A) \text{ says } (F \text{ imp } G) \quad \text{Key}(A) \text{ says } F}{\text{Key}(A) \text{ says } G} \text{ key\_imp\_e}$$

$$\frac{\text{before}(S)(T_1) \quad \mathrm{T}_2 > \mathrm{T}_1}{\text{before}(S)(T_2)} \text{ before\_gt}$$

$$\frac{\text{Key}(\texttt{localhost}) \text{ says before}(X)(T)}{\text{before}(X)(T)} \text{ timecontrols}$$

Figure 3. These inference rules are in addition to the standard rules of higher-order logic. Note that the Key(`localhost`) in the *timecontrols* rule is not universally quantified.

3.2 Core Logic

The base types in our logic are formulas, strings, and integers. Strings are used for representing digital signatures, public keys, and goals, as well as other constructs that we define from the basic ones. Integers are used for describing time.

Our logic has several constructors. (1) The `key` constructor turns a string into a principal. In our system, a principal is just a predicate on formulas. (2) The `goal` constructor takes as an argument a list of strings and returns a formula. Principals in our system prove that they are authorized to perform some action by demonstrating that they can derive a particular `goal` formula stated in the server's challenge. Goals are described at greater length in section 3.3. (3) The `before` constructor generates a term that is meant to describe the temporal state of a host system, usually the server that will be verifying the proof and granting or denying access. `before`$(S)(T)$ means that the time at host S (which is described by a string) has not yet reached T.

Figure 3 shows the inference rules we add to a standard higher-order logic to make it suitable for use as a core logic for a PCA system. We add a *signed* rule that takes as a premise the tuple (*pubkey*, *fmla*, *sig*), where *sig* is the digital signature produced by signing *fmla* with the private key corresponding to *pubkey*, and generates the formula P `says` *fmla*, where P is the principal that corresponds to the public key.[2] An instance of this rule exists for each tuple in which *sig* is a valid signature. Its intuitive meaning is that if a principal's private key has signed a formula, then the principal says that formula.

Unlike the AF logic, our core logic contains no separate inference rule for the introduction of principals; they are introduced only through the statements they sign (i.e., the *signed* rule).

The *key_imp_e* rule ensures that a principal can draw conclusions based on the statements it believes – it allows us to have modus ponens inside *says*. That is, if A says F and A says G and F and G together imply H then A says H. This rule might look out of place in the core logic, since a more complicated definition of *says* – one that we tried in an earlier iteration of our logic was structured similarly to the definition of *localname*, for example – could achieve the same goal. Such a definition, however, would interact badly with the definition of *localname*, making it impossible to prove lemmas that make *localname* useful.

The other rules we add are used for describing time and implementing expiration. Each host introduces a single *before* axiom that describes the current time on the local machine. To allow reasoning about relationships between earlier and later times, we add the rule *before_gt*, which tells us that if the time at host S has not yet reached T_1, then it also has not reached any point T_2 after T_1. The *timecontrols* rule allows the host that is checking the proof to make true the *before* axioms that it *says*. That is, if the host *says* the axiom, then the axiom is true on that system. Rules similar to these two are necessary in any system that has a notion of time.

[2] The reader will note that `says` is not part of the core logic; it is one of the definitions specific to our application. The actual *signed* axiom uses the defining formula, not the abbreviation. This may seem to tie the core logic to the application-specific logic. However, that is not really the case. The connection between the two is merely that the core logic should make it possible to define different and useful application-specific logics. To that end, it is helpful to study possible application-specific logics, as we have done, before settling on a core logic.

3.3 Goals and Sessions

In proof-carrying authorization a client that tries to access a resource is issued a challenge – an artificial logical formula – by the server that owns that resource. The challenge is the goal that the client must prove before it will be granted access to the resource.

Obviously there needs to be a relationship between a resource and the goal statement that represents it. The formula *goal*(`foo`) can be used to represent the right to access file `foo`. But if a proof of *goal*(`foo`) gives its bearer the right to access `foo`, how do we prevent Oscar from stealing a proof that Alice once made and using it as his own? The server will merely check that the proof is correct (i.e., that it is a true statement). It has no mechanism for checking who constructed a proof – such a mechanism, in fact, would be contrary to the idea of proof-carrying authorization.

To prevent clients from using stolen proofs, a server includes in its challenge to each client an identifier (a cryptographically pseudorandom string) that serves to make that client's proof goal unique. This means that clients now prove statements like "I have the authority to read the file `foo` with the unique identifier `sid`" (*goal*(`foo`)(`sid`)) instead of just "I have the authority to read the file `foo`" (*goal*(`foo`)). The identifier prevents clients from using stolen proofs, but it doesn't preclude them from being allowed to reuse an old proof.

If the identifier is a secret shared only by the server and a particular client, the server may elect to reuse the identifier in future challenges; in this case, the client can respond with a previously constructed proof. The period of time during which a server chooses to reuse an identifier we call a session; and the identifier, accordingly, the session identifier.

We discuss the secure transmission of the session identifier, and proofs which might contain it, in section 4.2.

It is important to note that the length of a session may be longer or shorter than a particular exchange between a client and a server. A session is valid until either the server or the client decide to expire the session identifier, which they may do at will.

It is common in security logics to have a *controls* rule to indicate that a client is allowed to control a particular resource. We find that the *key_imp_e* rule, combined with unique goals, gives our system enough power to reason about what principals say that the *controls* rule is no longer needed. To gain access to a resource, a client in our system, instead of showing that it *controls* a resource, proves that the server *says* that the client should have access to the resource.

4 The System Explained: A Narrative

The system we describe can be naturally divided into a client part and a server part (figure 4). The bulk of the client part is a web browser. The rest—the proxy server and the prover—are components that enable the web browser to use the PCA protocol. The browser itself remains unmodified, and our system does not use any features that are unique to a particular browser version.

The server part of our PCA system is built around an unmodified (Apache) web server. The web server is PCA-enabled through the use of a servlet which intercepts and handles all PCA-related requests. The two basic tasks that take place on the server's side during a PCA transaction are generating the proposition that needs to be proved and verifying that the proof provided by the client is correct. Each is performed by a separate component, the proposition generator and the checker, respectively.

Throughout the rest of this section, we will be describing various parts of the system as they are encountered during a transaction like one described in figure 4. As a running example we will use the scenario introduced in section 2. The text of the example will be indented and in italics to offset it from the description of the system.

4.1 Client: Proxy Server

The job of the proxy server is to be the intermediary between a web browser that has no knowledge of the PCA protocol and a web server that is PCA-enabled. An attempt by the browser to access a web page results in a dialogue between the proxy and the server that houses the page. The dialogue is conducted through PCA-enhanced HTTP—HTTP augmented with headers that allow it to convey information needed for authorization using the PCA protocol.

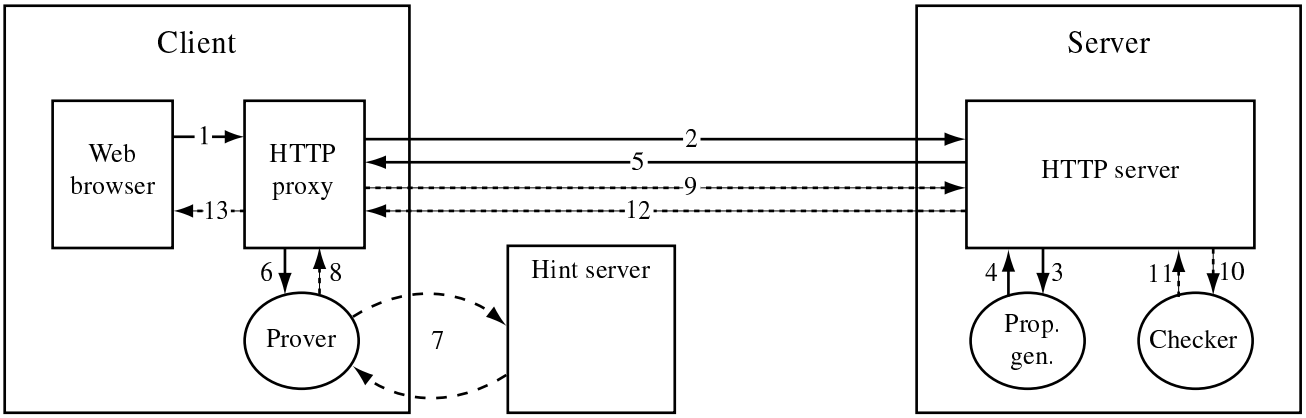

Figure 4. The components of the system.

The browser is completely unaware of this dialogue; it sees only the web page returned at the end.

The proxy is meant to be a substitute for a browser plugin. We decided to use a proxy instead of a plugin because this lets our system be completely browser independent. A production implementation would probably replace the proxy with a plugin. Like a plugin, our proxy is meant to be tightly coupled with the web browser. Unlike traditional web proxies, it is meant to serve a single client, not a set of them. This is because the proxy needs to speak on behalf of a client, perhaps signing statements with the client's private key or identifying itself with the client's public key. If a shared proxy were to be used for this purpose, its ability to access the private information of several clients would be a concern. Also, it would have to authenticate client-side connections so that it would know which client's data, or identity, to use for PCA transactions. Such authentication would be at cross purposes with one of the goals of our system—authorization uncoupled from authentication.

> *When Alice requests to see the page* `http://server/midterm.html`*, her browser forms the request and sends it to the proxy server (figure 4, step 1). The proxy server forwards the request without modifying it.*

4.2 Secure Transmission and Session Identifiers

The session identifier is a shared secret between the client and server. The identifier is used in challenges and proofs (including in digitally signed formulas within the proofs) to make them specific to a single session. This is important because the server caches previously proven challenges and allows clients to present the session identifier as a token that demonstrates that they have already provided the server with a proof.

The session identifier is a string generated by the server using a cryptographic pseudorandom number generator. Our implementation uses an 144-bit value which is then stored using a base-64 encoding. (144 bits was chosen because the value converts evenly into the base-64 encoding.)

Since the session identifier may be sufficient to gain access to a resource, stealing a session identifier, akin to stealing a proof in a system where goals are not unique, compromises the security of the system. In order to keep the session identifier secret, communication between the client and server uses the secure protocol HTTPS instead of normal HTTP in all cases where a session identifier is sent. If the client attempts to make a standard HTTP request for a PCA-protected page, the client is sent a special "Authorization Required" message which directs the client to switch to HTTPS and retry the request.

As an efficiency measure, the client caches locations of PCA-protected pages and automatically uses HTTPS instead of HTTP, shortening the transaction by two messages – the HTTP message that would fail and the reply that directs the client to switch to HTTPS. we assume that if a particular URL is PCA-protected, then any other URL which has the first as a prefix is also PCA-protected. Thus this cache

typically would require one entry per PCA-protected server, rather than one entry per PCA-protected page.

Alice's proxy contacts the server, asking for `midterm.html`*. Since that page is PCA-protected and the proxy used HTTP, the server rejects the request. The proxy switches to HTTPS and sends the same request again.*

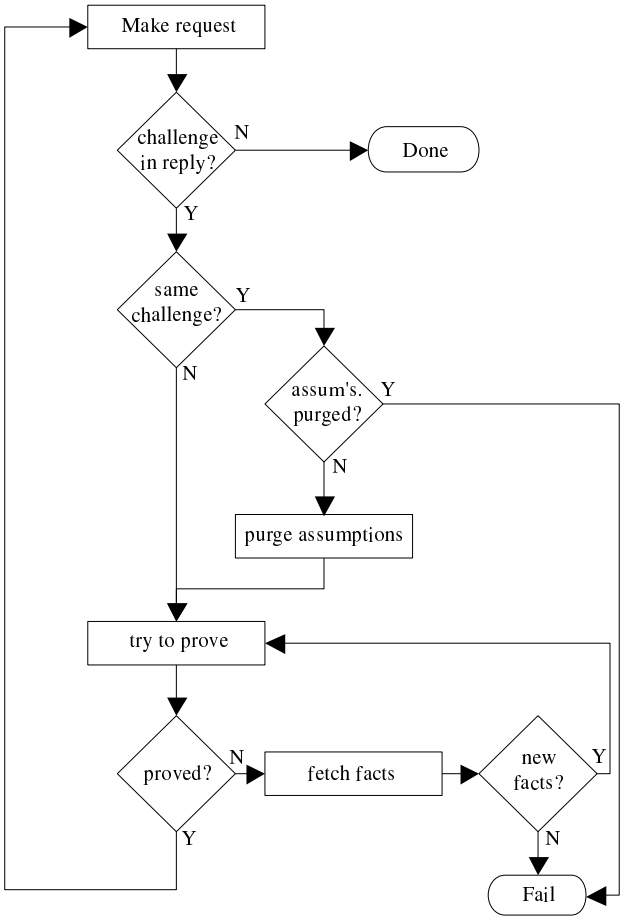

Figure 5. Client flowchart.

4.3 Server: Proposition Generator and Iterative Authorization

When a client attempts to access a PCA-protected web page, the server replies with a statement of the theorem that it wants the client to prove before granting it access. This statement, or proposition, can be generated autonomously; it depends only on the pathname of the file that the client is trying to access and on the syntax of the logic in which it is to be encoded.

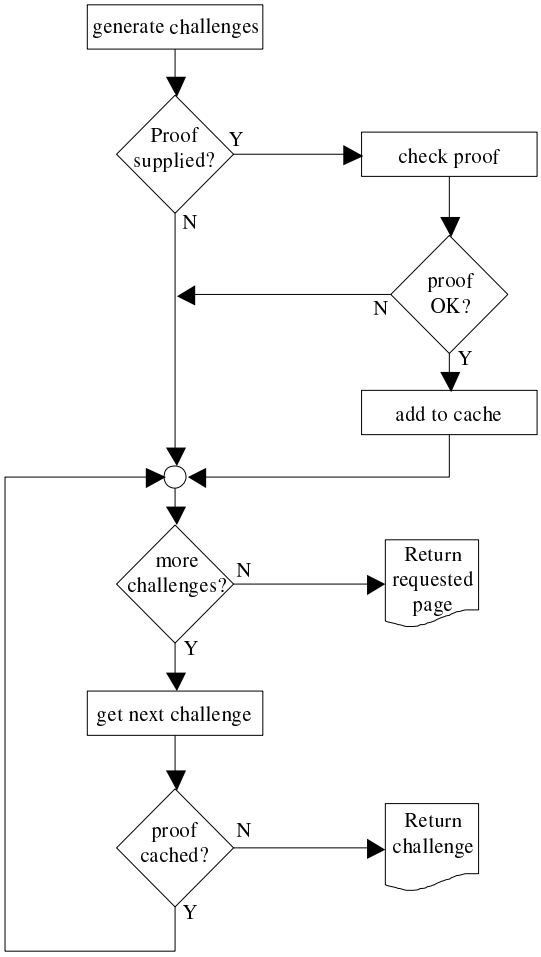

Figure 6. Server flowchart.

The server's *proposition generator* provides the server with a list of propositions. The server returns the first unproven proposition. If the client successfully proves that proposition in a subsequent request, then the server will reply with the next unproven proposition as the challenge. This process of proving and then receiving the next challenge from a list of unproven propositions is called *iterative authorization*. The processes for the client and server are shown in the flowcharts of figure 5 and figure 6.

This process terminates when either the client gives up (e.g. cannot prove one of the propositions) or has successfully proven all of the propositions, in which case access is allowed. If the client presents a proof which fails when the server checks it, it is simply discarded. In this case, the same challenge will be returned twice to the client.

If the client receives the same challenge twice, it knows that although it "successfully" constructed a proof for that challenge, it was rejected by the server. This means that one of the client's assumptions must have been incorrect. The client may choose to discard some assumptions and retry the proof process.

Our prototype application generates a proposition for each directory level of the URL specified in the client's request. Since the server returns identical challenges regardless of whether the requested object exists, returning a challenge reveals no information about the existence of objects on the server.

Isolating the proposition generator from the rest of the server makes it easy to adapt the server for other applications of PCA; using it for another application may require nothing more than changing the proposition generator.

After receiving the second, encrypted request, the server first generates the session ID, "sid". It then passes the request and the ID to the proposition generator. The proposition generator returns a list of propositions that Alice must prove before she is allowed to see `/midterm.html`*:*

```
says @ (key @ "server")
     @ (goal @ "http://server/" @ "sid")

says @ (key @ "server")
     @ (goal @ "http://server/midterm.html"
             @ "sid")
```

The syntax of the example closely follows the actual LF syntax in which the proofs are written: all terms are in prefix notation and application is explicit and denoted by `@`*.*

```
says @ (key @ "server") @ (goal @ X @ Y)
```

is therefore equivalent to "Key(`server`*) says goal(X, Y)."*

For the purposes of this example, we will deal only with the second challenge. In reality, Alice would first have to prove that she is allowed to access `http://server/`*, and only then could she try to prove that she is also allowed to access* `http://server/midterm.html`*.*

A benefit of iterative authorization is that it allows parts of the security policy to be hidden from unauthorized clients. Only when a challenge has been proven will the client be able to access the facts that it needs to prove the next challenge. In the context of our application this means, for example, that a client must prove that it is allowed to access a directory before it can even find out what goal it must prove (and therefore what facts it must gather) to gain access to a particular file in that directory.

4.4 Server: Challenges; Client: Proofs

For each authorization request, the server's proposition generator generates a list of propositions which must be proven before access is granted. Each proposition contains a URL path and a session identifier. The server checks to see if each proposition has been previously proven by the client by checking a cache of previously proven challenges. If all of the propositions have been proven, access is allowed immediately. Otherwise, the first unproven proposition is returned to the client as a challenge. Any other unproven propositions are discarded.

The server constructs a reply with a status code of "Unauthorized". This is a standard HTTP response code (401) [9]. The response includes the required HTTP header field "WWW-Authenticate" with an authentication scheme of "PCA" and the unproven proposition as its single parameter.

Once the client has constructed a proof of the challenge, the client makes another HTTPS request (this can be done with the same TCP connection if allowed by keep-alive) containing the challenge and proof. The challenge is included in an "Authorization" request-header field, and the proof is included in a series of "X-PCA-Proof" request-header fields. The server checks that the proof proves the supplied challenge, adds the challenge to its cache of proven propositions, and then begins the checking process again.

It is sometimes possible for a client to guess what the challenge will be. In that case, it can try to prove the challenge even before the server makes it (we call this *prove-ahead* or speculative proving). The proof can then be sent to the server as part of the original request. If the client guessed correctly, the server will accept the proof without first sending a challenge to the client.

The first proposition in the example is the one stating that the server says that it's OK to read `http://server/`*. The server checks whether it has already*

been proven and moves on to the next one. (Remember that for the purposes of the example we're concentrating only on the second proposition; the authorization process for each is identical.) The next proposition states that the server says it's OK to read `http://server/midterm.html`*. This one hasn't been proven yet, so the server constructs an HTTP response that includes this proposition as a challenge and sends it to Alice. This is step 5 of figure 4.*

4.5 Client: Prover

In the course of a PCA conversation with a server, the proxy needs to generate proofs that will demonstrate to the server that the client should be allowed access to a particular file. This task is independent enough from the rest of the authorization process that it is convenient to abstract it into a separate component. During a PCA conversation the client may need to prove multiple statements; the process of proving each is left to the prover.

The core of the prover in our system is the Twelf logical framework [14]. Proofs are generated by a logic program that uses tactics. The goal that must be proven is encoded as the statement of a theorem. Axioms that are likely to be helpful in proving the theorem are added as assumptions. The logic program generates a derivation of the theorem; this is the "proof" that the proxy sends to the server.

The tactics that define the prover roughly correspond to the inference rules of the application-specific logic. Together with the algorithm that uses them, the tactics comprise a decision procedure that generates proofs – for our system to always find proofs of true statements, this decision procedure must be decidable.

A tactic for proving *A* speaksfor *C* would be to find proofs of *A* speaksfor *B* and *B* speaksfor *C* and then use the transitivity lemma for *speaksfor*. Other tactics might be used to guide the search for these subgoals. The order in which tactics are applied affects their effectiveness. Care must also be taken to avoid situations in which tactics might guide the prover into infinite (or finite but time-consuming) branches that don't necessarily lead to a proof. The prover in our system is able to automatically generate proofs whenever they exist.

If this were a production-strength implementation of a PCA system, we would likely have implemented the theorem prover in Java. The capabilities of Twelf are far greater than what we need; a custom-made theorem prover that had only the required functionality would be more lightweight. It could also be engineered so that a failed attempt to prove a theorem would explain to the user in the user's own terms why access could not be granted. The advantage of Twelf, on the other hand, is that the encoding of the tactics is concise. Since the tactics are closely tied to the logic, Twelf made it much easier to experiment with changes to the logic without having to spend much effort in adapting the theorem prover to it.

In addition to generating the proof of a goal given to it by the proxy, the prover's job is to find all the assumptions that are required by the proof. Assumptions needed to generate a proof might include statements made by the server about who is allowed to access a particular file, statements about clock skew, statements by which principals delegate authority to other principals, or statements of goal. While some of these might be known to the proxy, and would therefore have been provided to the prover, others might need to be obtained from web pages.

Since fetching assumptions from the web is a relatively time-consuming process (hundreds of milliseconds is a long time for a single step of an interactive authorization that should be transparent to the user), the prover caches the assumptions for future use. The prover also periodically discards assumptions which have not been recently used in successful proofs.

4.6 Client: Iterative Proving

The client is responsible for *proof generation.* The client may not always be able to generate a proof of the challenge on the first try. It may need to obtain additional information, such as signed delegations or other facts, before the proof can be completed. The process of fetching additional information and then retrying the proof process is called *iterative proving.* The process does not affect the server, and terminates when a proof is successfully generated.

Proof generation can be divided into two phases. In the first phase, facts are gathered. In the second phase, straightforward prover rules are used to test if these facts are sufficient to prove the challenge. If so, the proof is returned. Otherwise, the phases are repeated,

first gathering additional facts and then reproving, until either a proof is successfully generated, or until no new facts can be found.

The fact-gathering phase involves the client gathering four basic types of facts.

Self-signed Assumptions The first type of facts comes from the client itself. The client can sign statements with its own private key, and these may be useful in constructing proofs. Often, for example, it is necessary for the client to sign part of the challenge itself and use this as an assumption in the proof.

Alice will sign the statement

```
goal @ "http://server/midterm.html" @ "sid"
```

Applying the signature axiom to this statement will yield

```
says @ (key @ "alice")
     @ (goal @ "http://server/midterm.html"
             @ "sid")
```

Armed with this assumption (and no others, so far), Alice tries to prove the challenge. The attempt fails in the client (i.e., no proof is constructed, so nothing is sent to the server); Alice realizes that this assumption by itself isn't sufficient to generate a proof so she tries to collect more facts. (Steps 6 and 8 of figure 4.)

Goal-oriented facts The second type of facts is typically (though not necessarily) provided by the server. While generating propositions and checking proofs are conceptually the two main parts of the server-side PCA infrastructure, a PCA-enabled server may want to carry out a number of other tasks. One of these is managing pieces of the security policy. To generate a proof that it is authorized to access a particular web page, a client will have to know which principals have access to it. Such information, since it describes which principals have direct access to a particular goal, we call goal-oriented facts.

In our implementation, the server keeps this information in access-control lists. Entries from these lists, encoded in a manner that makes them suitable for use as assumptions, are provided to the client on demand. They are not given out indiscriminately, however. Before providing a goal-oriented fact, the server uses an additional PCA exchange to check that the client is authorized to access the fact.

In our system the client queries the server for goal-oriented facts for each challenge it needs to prove. Goals are described by URLs, so the server requires PCA authorization for a directory before it will return the goal-oriented facts that describe access to files/directories inside that directory. The goal-oriented fact that describes access to the root directory is freely returned to any client. In this way, a client is forced to iteratively prove authorization to each directory level on the server.

Since her first attempt at generating a proof didn't succeed, Alice sends a message to the server requesting goal-oriented facts about `http://server/midterm.html`*. Upon receiving the request, the server first checks whether Alice has demonstrated that she has access to* `http://server/`*. It does this by generating a list of assumptions (there will be only a single assumption in the list) and then checking whether Alice has proven it. After determining that Alice is allowed access to the root directory, the server gives to Alice a signed version of the statement*

```
not (before @ "server" @ (8 P.M.))
imp (says @ (localname @ (key @ "registrar")
                         @ "cs101")
          @ (goal @ "http://server/midterm.html"
                  @ "sig"))
imp (goal @ "http://server/midterm.html" @ "sig")
```

Alice translates it into, "Server says: 'If it is not before 8 P.M., and a CS101 student says it's OK to read `midterm.html`*, then it's OK to read* `midterm.html`*.' "*

Fetching the ACL entry from the server is also described by steps 2 through 5 of figure 4.

Server Time In order to generate proofs which include expiring statements, the client must make a guess about the server's clock. The third type of facts is the client's guess about the time which will be showing on the server's clock at the instant of proof checking. If the client makes an incorrect guess, it might successfully generate a proof which is rejected by the server. (An incorrect guess about the server's clock is the only reason for rejecting a properly formed, since it is the only "fact" the the server might not accept.) In this case, the client adjusts its guess about the server's clock and begins the proof generation process again.

In order to use the goal-oriented assumption it received from the server, Alice must also know something about the current time. Since it's 9 P.M. by her clock, she guesses that the server believes that the time is before 9:05 P.M. and after 8:55 P.M. This corresponds to the assumption

```
before @ "server" @ (9:05 P.M.) and
not (before @ "server" @ (8:55 P.M.))
```

Armed with the self-signed assumption, the goal-oriented assumption, and the assumption about time, Alice again tries proving that she can access `midterm.html`*. Again, she discovers that she doesn't have enough facts to construct a proof. She knows that Registrar.CS101 can access the file, but she doesn't know how to extend the access privilege to herself.*

Key-oriented facts The fourth type of facts come from hints that are embedded in keys and that enable facts to be stored on a separate (perhaps centralized or distributed) server. Concatenated with each public key is a list of URLs which contain facts relevant to that key (perhaps maintained by the key-holder). These facts might be signed delegations, for example.

At each fact-fetching step, the client examines all of the keys referenced in all of the facts already fetched. Each key is examined for embedded hints. Then the client fetches new facts from all of these hint URLs. (The client maintains a cache so that hint URLs are not accessed more than once.) In the next iteration (if another iteration is required), these new facts will be examined for additional hint URLs, which will then be fetched. In this way, the client does a breadth-first search for new facts, alternating between searching one additional depth level and attempting to construct a proof with the current set of facts.

Although the proof didn't succeed, Alice can now use the hints from her facts to try to find additional facts that might help the proof. Bob's server's key and the Registrar's key are embedded in the facts Alice has collected. In each key is encoded a URL that describes a location (it can be any location) at which the owner of that key publishes additional facts. Bob's server's key, heretofore given as `key @ "server"` *actually has the form* `key @ "server;http://server/hints/"`*.*

Before giving up, Alice's prover follows these URLs to see if it can find any new facts that might help. This is shown as step 7 of figure 4. Following the hint in the Registrar's key, Alice downloads a signed statement which she translates into the assumption

```
says @ (key @ "registrar") @
  (before @ "registrar" @ (end of semester)
   imp (speaksfor @ (key @ "alice")
          @ (localname @ (key @ "registrar")
                       @ "cs101")))
```

This fact delegates to Alice the right to speak on behalf of Registrar.CS101: "The Registrar says that until the end of the semester, whatever Alice says has the same weight as if Registrar.CS101 said it."

Following the hint in Bob's server's key, Alice obtains a new fact that tells her the clock skew between Bob's server and the Registrar.

Alice now finally has enough facts to generate a proof that demonstrates that she is authorized to read `http://server/midterm.html`*. Alice makes a final request to access* `http://server/midterm.html`*, this time including in it the full proof.*

4.7 Server: Proof Checking

The Theory. After it learns which proposition it must prove, the client generates a proof and sends it to the server. If the proof is correct, the server allows the client to access the requested web page. Proofs are checked using Twelf. The proof provided by the client is encoded as an LF term [11]. The type (in the programming languages sense) of the term is the statement of the proof; the body of the term is the proof's derivation. Checking that the derivation is correct amounts to type checking the term that represents the proof. If the term is well typed, the client has succeeded in proving the proposition.

As is the case for the client, using Twelf for proof checking is overkill, since only the type-checking algorithm is used. The proof checker is part of the trusted computing base of the system. To minimize the likelihood that it contains bugs that could compromise security, it should be as small and simple as possible. Several minimal LF type checkers have already been or will shortly be implemented[13]; one of these could serve as the proof checker for our system.

LF terms can either have explicit type ascriptions or be implicitly typed. An implicitly typed version of the *and_introduction* lemma in our logic has this form:

```
and_i : pf A -> pf B -> pf (A and B)
  = [p1][p2] forall_i ...
```

One with explicitly ascribed types:

```
and_i : {A:tm form} {B:tm form} pf A
         -> pf B
         -> pf (and A B)
  = [A:tm form] [B:tm form]
    [P1:pf A] [P2:pf B] forall_i ...
```

Note that the explicitly-typed version may need to introduce more that one type annotation per variable. This can lead to exponential increase in the size of the proofs. The implicitly-typed version is much more concise, but suffers from a different problem. The type inference algorithm that the server would need to run is undecidable, which could cause correct proofs not to be accepted or the server to the tied up by a complicated proof.

The LF community is currently developing a type checker for semi-explicitly typed LF terms that would solve both problems. Its type-inference algorithm will be decidable, and the level of type ascription it will require will not cause exponential code blowup. Until it becomes available, our system will require proofs to be explicitly typed.

The Practice. Checking the proof provided by the client, however, is not quite as simple as just passing it through an LF type checker. The body of an LF term is the proof of the proposition represented by its type. If the term has only a type ascription but no body, it represents an axiom. That the axiom may type check does not mean that we want to allow it as part of the proof. If we were to do so, the client could respond to a challenge by sending an axiom that asserted the proposition it needed to prove; obviously we wouldn't want to accept this statement as proof of the challenge. In addition, the server must verify that the signature axioms used by the proof actually hold; that is, that any digital signatures are valid and sign well-formed statements.

To solve these problems, the server preprocesses the client's proof before passing it to a type checker. The preprocessor first makes sure that all of the terms that make up the proof have both a type and a body. A proof that contains illegal axioms is rejected.

Next, two special types of axioms are inserted into the proof as necessary. The first type is used to make propositions about digital signatures, and the second type is used to make propositions regarding time. These are required since the proof checker cannot check digital signatures or time statements directly. The client inserts into the proof placeholders for the two types of axioms it can use. The server makes sure that each axiom holds, generates an LF declaration that represents it, and then replaces the placeholder with a reference to the declaration.

For digital signatures, the client inserts into the proof a proposition of the special form "#signature *key, formula, sig*". (Each of the fields is encoded in base 64 for transmission.) The server checks that *sig* is a valid signature made by the key *key* for the formula *formula.* If so, the #signature statement is replaced by an axiom asserting that *key* signed *formula.*

To make statements about time, the client inserts a proposition of the special form "#now". The preprocessing stage replaces the #now with an axiom asserting the current time (in seconds since 1970). Axioms of this form are necessary when signed propositions include an expiration date, for example.

Once the proof has been parsed to make sure it contains no axioms and special axioms of these two forms have been reintroduced, the proof is checked to make sure it actually proves the challenge. (The proof might be a perfectly valid proof of some other challenge!) If this final check succeeds, then the whole proof is passed to an LF type checker; in our case, this is again Twelf.

If all of these checks succeed, then the challenge is inserted into the server's cache of proven propositions. The server will either allow access to the page (if this was the last challenge in the server's list) or return the next challenge to the client.

To avoid re-checking proofs, all correctly proven propositions are cached. Some of them may use time-dependent or otherwise expirable premises–they could be correct when first checked but false later. If such proofs, instead of being retransmitted and rechecked, are found in the cache, their premises must still be checked before authorization is accepted.

The server receives Alice's request for `midterm.html` *and generates a list of propositions that need to be proven before access is granted. Only the last proposition is unproven, and its proof is included in Alice's request. The server expands the #signature and #now propositions, and sends the proof to the type-checker. The proof checks successfully, so the server inserts it in its cache; Alice won't have to prove this proposition again. Finally, the server checks whether Alice proved the correct challenge, which she has. There are no more propositions left to be proven, Alice has successfully proven that she is authorized to read* `http://server/midterm.html`*. The server sends the requested page to Alice.*

5 Module system

Since proofs are meant to be both generated and checked completely automatically, many of the traditional software-engineering reasons for needing a module system are absent, since they are often motivated by making code easier for a human programmer to read. Still, there remain good reasons to allow proofs to be modular. For one, since the trusted computing base of the checker is composed of the smallest possible number of axioms, most of the rules used in constructing the proofs will be lemmas proven from the axioms. Many clients will use these same lemmas in their proofs; most proofs, in fact, are likely to include the same basic set of lemmas. We have added to the proof language a simple module system that allows us to abstract these lemmas from individual proofs. Instead of having to include all the lemmas in each proof, the module system allows them to be imported with a statement like `basiclem = #include http://server/lemmas.elf`. If the lemma `speaksfor_trans`, for example, resides in the `basiclem` module, it can now be referenced from the body of the proof as `basiclem.speaksfor_trans`. Instead individually by each client, abstracting the lemmas into modules allows them to be maintained and published by a third party. A company, for instance, can maintain a single set of lemmas that all its employees can import when trying to prove that they are allowed to access their payroll records.

To make the examples in the previous section more understandable, we have omitted from them references to modules. In reality, each proof sent by a client to a server would be prefixed by a `#include` statement for a module that contained the definitions of, for example, `says`, `speaksfor`, `localname` and the lemmas that manipulate them, as well as more basic lemmas.

Aside from the administrative advantages, an important practical benefit of abstracting lemmas into modules is increased efficiency, both in bandwith consumed during proof transmission and in resources expended for proof checking. Instead of transmitting with each proof several thousands of lines of lemmas, a client merely inserts a `#include` declaration which tells the checker the URL (we currently support only modules that are accessible via HTTP) at which the module containing the lemmas can be found. Before the proof is transmitted from the client to the server, the label under which the module is imported is modified so that it contains the hash of the semantic content (that is, a hash that is somewhat independent of variable names and formatting) of the imported module. This way the checker knows not only where to find the module, but can also verify that the prover and the checker agree on its contents.

When the checker is processing a proof and encounters a `#include` statement, it first checks whether a module with that URL has already been imported. If it has been, and the hash of the previously imported module matches the hash in the proof, then proof checking continues normally and the proof can readily reference lemmas declared in the imported module. If the hashes do not match or the module hasn't been imported, the checker accesses the URL and fetches the module. A module being imported is validated by the checker in the same way that a proof would be. Since they're identified with content hashes, multiple versions of a module with the same URL can coexist in the checker's cache.

Since importing a module is something that is done actively by the server, it raises the possibility of denial-of-service attacks. In designing the checker we have assumed that it is the client's responsibility to make sure that any modules it includes in its proofs are readily accessible. The checker takes appropriate precautions to guard itself against proofs that may contain modules that endlessly import other modules, cyclical import statements, and other similar attacks.

6 Conclusion

In this paper we describe an authorization system for web browsers and web servers that we have built using a proof-carrying authorization framework. Our application is implemented as add-on modules to standard web browsers and web servers and demonstrates that it is feasible to use a proof-carrying authorization framework as a basis for building real systems. We show that such systems, in which the burden of proof is placed on the client, can reap the benefits of using a higher-order security logic (flexibility and extensibility) without being hampered by its traditional weaknesses (undecidability).

We improve upon previous work on proof-carrying authorization by adding to the framework a notion of state and enhancing the PCA logic with goal constructs and and a module system. The additions of state (through what we call sessions) and goals are instrumental in making PCA practical. We also introduce mechanisms that allow servers to provide only selective access to security policies, which was a concept wholly absent from the original work. In addition, we refine the core logic to make it more useful for expressing interesting application-specific logics, and we define a particular application-specific logic that is capable of serving as a security logic for a real distributed authorization system.

Our application allows pieces of the security policy to be distributed across arbitrary hosts. Through the process of iterative proving the client repeatedly fetches proof components until it is able to construct a proof. This mechanism allows the server policy to be arbitrarily complex, controlled by a large number of principals, and spread over an arbitrary network of machines in a secure way. Since proof components can themselves be protected, our system avoids releasing the entire security policy to unauthorized clients. Iterative authorization, or allowing the server to repeatedly challenge the client with new challenges during a single authorization transaction, provides a great deal of flexibility in designing security policies.

Although our system has a great deal of flexibility, we have been successful in reducing the inherent overhead to a minimum, demonstrating that it is possible to use proof-carrying authorization to build an efficient authorization system. To this end, our system uses speculative proving—clients attempt to guess server challenges and generate proofs ahead of time, drastically reducing the exchange between the client and the server. The client also caches proofs and proof components to avoid the expense of fetching them and regenerating the proofs. The server also caches proofs, which avoids the need for a client to produce the same proof each time it tries to access a particular object. A module system in the proof language allows shared lemmas, which comprise the bulk of the proofs, to be transmitted only if the server has not processed them, saving both bandwidth and proof-checking overhead.

Ongoing work includes further development of our prototype application. We will investigate the use of oblivious transfer and other protocols for fetching proof components without revealing unnecessary information and further refine our security logic to reduce its trusted base and increase its generality. In addition to allowing clients to import lemmas from a third party, we would like to devise a method from allowing them to import actual proof rules as well. We are also exploring the idea of using a higher-order logic as a bridge between security logics in a way that would enable authentication frameworks based on different logics to interact and share resources.

References

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, October 1998.

[2] M. Abadi, M. Burrows, B. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] M. Abadi, E. Wobber, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 256–269, Systems Research Center SRC, DEC, Dec. 1993. ACM SIGOPS, ACM Press. These proceedings are also ACM Operating Systems Review, 27,5.

[4] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, Singapore, November 1999.

[5] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *21th*

IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, May 2000.

[6] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust-management system. In *Proceedings of the 2nd Financial Crypto Conference*, volume 1465 of *Lecture Notes in Computer Science*, Berlin, 1998. Springer.

[7] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, May 1998.

[8] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. *SPKI Certificate Theory*, September 1999. RFC2693.

[9] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. IETF - Network Working Group, The Internet Society, June 1999. RFC 2616.

[10] J. Y. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 111–122, Mordano, Italy, June 1999.

[11] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[12] International Telecommunications Union. ITU-T recommendation X.509: The Directory: Authentication Framework. Technical Report X.509, ITU, 1997.

[13] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Oct. 1998. Available as Technical Report CMU-CS-98-154.

[14] F. Pfenning and C. Schürmann. System description: Twelf: A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16-99)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10 1999. Springer.

A Axioms of the Core Logic

Axioms of the higher-order core logic of our PCA system. Except for the last four, they are standard inference rules for higher-order logic.

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

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

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

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

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

$$\frac{\forall x.A(x)}{A(T)}\text{forall\_e} \qquad \frac{}{X = X}\text{refl}$$

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

$$\frac{\text{signature}(\mathit{pubkey}, \mathit{fmla}, \mathit{sig})}{\text{Key}(\mathit{pubkey}) \text{ says } \mathit{fmla}}\ \text{signed}$$

$$\frac{\text{Key}(A) \text{ says } (F \text{ imp } G) \quad \text{Key}(A) \text{ says } F}{\text{Key}(A) \text{ says } G}\ \text{key\_imp\_e}$$

$$\frac{\text{before}(S)(T_1) \quad \text{T}_2 > \text{T}_1}{\text{before}(S)(T_2)}\ \text{before\_gt}$$

$$\frac{\text{Key}(\texttt{localhost}) \text{ says before}(X)(T)}{\text{before}(X)(T)}\ \text{timecontrols}$$