A General and Flexible Access-Control System for the Web

Lujo Bauer* Michael A. Schneider† Edward W. Felten*
Secure Internet Programming Laboratory
Department of Computer Science
Princeton University
{lbauer,schneidr,felten}@cs.princeton.edu

Abstract

We describe the design, implementation, and performance of a new system for access control on the web. To achieve greater flexibility in forming access-control policies – in particular, to allow better interoperability across administrative boundaries – we base our system on the ideas of proof-carrying authorization (PCA). We extend PCA with the notion of goals and sessions, and add a module system to the proof language. Our access-control system 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. Our system is implemented as modules that extend a standard web server and 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 implementation includes a series of optimizations, such as speculative proving, and modularizing and caching proofs, and demonstrates that the goals of generality, flexibility, and interoperability are compatible with reasonable performance.

1 Introduction

After a short period of being not much more than a curiosity, the web quickly became an important medium for discussion, commerce, and business. Instead of holding just information that the entire world could see, web pages also became used to access email, financial records, and other personal or proprietary data that was meant to be viewed only by particular individuals or groups.

This made it necessary to design mechanisms that would restrict access to web pages. The most widely used mechanism is for the user to be prompted for a username and password before he is allowed to see the content of a page [11]. A web administrator decides that a certain page will be visible only if a user enters a correct username/password pair that resides in an appropriate database file on the web server. A successful response will often result in the client's browser being given a cookie; on later visits to the same or related web pages, the cookie will be accepted as proof of the fact that the user has already demonstrated his right to see those pages and he won't be challenged to prove it again [19]. An organization such as a university may require that all people wishing to see a restricted web page first visit a centralized login page which handles authentication for all of the organization's web sites. The cookie placed on the client's browser then contains information which any of the organization's web servers can use to verify that it was legitimately issued by the organization's authentication service. In cases such as this one the functions of authentication (verifying an identity) and authorization (granting access) are separated into two distinct processes.

More modern methods of controlling access to web pages separate these functions even further, not as an optimization, but as a basic element of their design. Increasingly in use are systems in which a certificate, such as a Kerberos ticket [15, 17] or an X.509 certificate [14], is obtained by a user through out-of-band means; a web browser and a web server

*Supported in part by NSF Grant CCR-9870316.
†Supported by a Fannie and John Hertz Graduate Fellowship.

are augmented so that the web browser can pass the certificate to the web server and the web server can use the certificate to authorize the user to access a certain page. The advantage of these mechanisms is that, in addition to providing more secure implementations of protocols similar to basic web authentication, they make it possible for a host of different services to authorize access based on the same token. An organization can now provide a single point of authentication for access to web pages, file systems, and Unix servers.

Though growing increasingly common, most notably due to the use of Kerberos in new versions of the Windows operating system, these systems have not yet gained wide acceptance. This is partly because they don't adequately deal with all the requirements for authorization on the web, so their undeniable advantages may not be sufficient to justify their cost.

One of the chief weaknesses of these systems is that they are not good at providing interoperability between administrative domains, especially when they use different security policies or authorization systems. Having a centralized authentication server that issues each user a certificate works well when there's a large number of web servers which are willing to trust that particular authentication server (at a university, for example), but when such trust is absent (between two universities) they bear no benefit. There have been attempts to build systems that cross this administrative divide [9] but the problem still awaits practical solution.

We have built a system that addresses this issue; in this paper we present its design, implementation, and performance results. Our system even further uncouples authorization from authentication, allowing for superior interoperation across administrative domains and more expressive security policies. Our implementation consists of a web server module and a local web proxy. The server allows access to pages only if the web browser can demonstrate that it is authorized to view them. The browser's local proxy accomplishes this by mechanically constructing a proof of a challenge sent to it by the server. Our system supports arbitrarily complex delegation, and we implement 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. Our system was built for controlling access to web pages, but could relatively easily be extended to encompass access control for other applications (e.g., file systems) as well.

2 Goals and Design

In designing our system for access control of web pages we had several criteria that we wanted to address:

- interoperability and expressivity;
- ease of implementation and integration with web servers and web browsers;
- efficiency;
- convenience to the user;
- applicability to spheres other than web access control.

2.1 Interoperability and Expressivity

Even the most flexible of current systems for web access control are limited in their ability to interoperate across administrative boundaries, especially when they use different security policies or authorization systems. One of the main reasons for this is that though they attempt to separate the functions of authorization and authentication, they overwhelmingly continue to express their security policy – the definition of which entities are authorized to view a certain web page – in terms of the identities of the users. Though the web server often isn't the entity that authenticates a user's identity, basing the security policy on identity makes it very difficult to provide access to users who can't be authenticated by a server in the same administrative domain.

The way we choose to resolve this issue is by making the security policy completely general – access to a page can be described by an arbitrary predicate. This predicate is likely to, but need not, be linked to a verification of identity – it could be that a particular security policy grants access only to people who are able to present the proof of Fermat's last theorem. Since the facts needed to satisfy this arbitrary authorization predicate are likely to include more than just a verification of identity, in our access-control system we replace authentication servers with more general fact servers. In this scenario the problem of deciding whether a particular client should be granted access to a particular web page becomes a general distributed-authentication problem, which we solve by adapting previously developed techniques from that field.

Distributed authentication systems [7, 8, 14] provide a way for implementing and using complex security policies that are distributed across multiple hosts. The methods for distributing and assembling pieces of the security policy can be described using logics [1, 6, 12], and distributed authentication systems have been built by first designing an appropriate logic and the implementing the system around it [2, 3, 5]. The most general of the logics – that is, the one that allows for expressing the widest range of security policies – was recently introduced by Appel and Felten (AF logic) [4]. The AF logic is a higher-order logic that differs from standard ones only by the inclusion of a very few rules that are used for defining operators and lemmas suitable for a security logic.

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 to grant access. 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 – leaving the more complicated task of assembling the proof to the client. The server, using only the common underlying AF logic, can check proofs from all clients, regardless of the method they used to generate the proof or the proof's structure. This technique of proof-carrying authorization (PCA) perfectly satisfies our goal of interoperability – as long as a server bases its access control policy on the AF logic, interoperation with systems in different administrative hierarchies is no more difficult than interoperation with local ones.

2.2 Convenience of Use and Implementation

An important goal for a web access-control system that aspires to be practical is that it be implementable without modification of the existing infrastructure – that is, web browsers and web servers. Our access-control system involves three types of players: web browsers, web servers, and fact servers (which issue tokens that can certify not only successful authentication – as do ordinary authentication servers – but also any other type of fact that they store).

We enable the web browser to understand our authorization protocol by implementing a local web proxy. The proxy intercepts a browser's request for a protected page and then executes the authorization protocol to generate the proof needed for accessing the page; the web browser sees only the result – either the page that the user attempted to access or an appropriate failure message. Each user has a unique cryptographic key held by the proxy. Users' identities are established by name-key certificates stored on fact servers. The use of keys makes it unnecessary to prompt the user for a password, making the authorization process quicker and more transparent to the user.

For tighter integration with the browser and better performance, the proxy could be packaged as a browser plugin. This would make it less portable, however, as a different plugin would have to be written for each type of browser; we did not feel this was within the scope of our prototype implementation.

The web server part of our system is built around an unmodified 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 an authorization 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.

Fact servers hold the facts a client must gather before it can construct a proof. Each fact is a signed statement in the AF logic. We implement an off-line utility for signing statements, which lets us use a standard web server as a fact server. The fact server can also restrict access to the facts it publishes with a servlet, in the same manner as the web server.

2.3 Efficiency

The whole access-control process is completely transparent to a user. To be practical, it must also be efficient. Assembling the facts necessary to construct a proof may involve several transactions over the network. The actual construction of the proof, the cryptographic operations done during the protocol, and proof checking are all potential performance bottlenecks.

Though our system is a prototype and not production-quality, its performance is good enough to make it acceptable in practice. Heavy use of caching limits the need to fetch multiple facts over the network and speculative proving makes it possible to shorten the conversation between the web proxy and the servlet.

2.4 Generality

The best current web authorization mechanisms have the characteristic that they are not limited to providing access control for web pages; indeed, their strength is that they provide a unified method that also regulates access to other resources, such as file systems. Our system, while implemented specifically for access control on the web, can also be extended in this manner. The idea of proof-carrying authorization is not specific to web access control, and the mechanisms we develop, while implemented in a web proxy and a servlet, can easily be modified to provide access control for other resources.

3 Implementation

In this section we use the running example of Alice trying to access `midterm.html` (Figure 1) to describe the implementation of our system in detail. We describe each part of the system when it becomes relevant as we follow the example (the text of which will be indented and italicized).

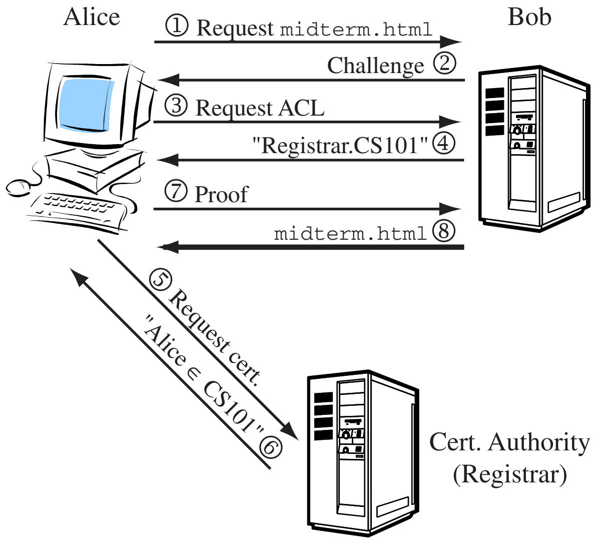

Figure 1. Alice wants to read `midterm.html`. In practice, caching makes most of the messages shown unnecessary.

3.1 Example and Overview

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`.

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

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 server (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).

3.2 Logic

Our authorization system, like other proof-carrying authorization systems, has a core logic with an application-specific logic defined on top of it.

The application-specific logic is used to define the operators and rules useful for writing security policies for web access control – in our case standard operators like *says* and *speaksfor* were sufficient. Unlike in traditional distributed authorization systems, in which these operators would be primitive, in proof-carrying authorization systems these operators are implemented as definitions using the core logic. This is what makes it possible for our system to work seamlessly across administrative domains – as long as they share a common core logic, the operators of any application-specific logic can be regarded merely as abbreviations.

The following are operators of our application-specific logic, their informal definitions, and their encodings in the AF logic.

A* says *F 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.

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

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, it returns true 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.

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

Rules about these operators can be proved as lemmas and are also transient to the core logic. Using the operators we defined, we can now prove rules such as this one, which states that *says* follows implication:

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

The core logic we use is a variant of the AF logic (the rules of which are presented in Appendix A). The only rules that aren't standard rules of higher-order logic are the four that allow us to reason about digital signatures, time, and implication inside the *says* operator.

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

When Alice requests to see the page `http://server/midterm.html`*, her browser forms the request and sends it to the local proxy (Figure 2, step 1). The proxy server forwards the request without modifying it (step 2).*

3.4 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 a 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 server replies with a special "Authorization Required" message which directs the client to switch to HTTPS and retry the request.

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.*

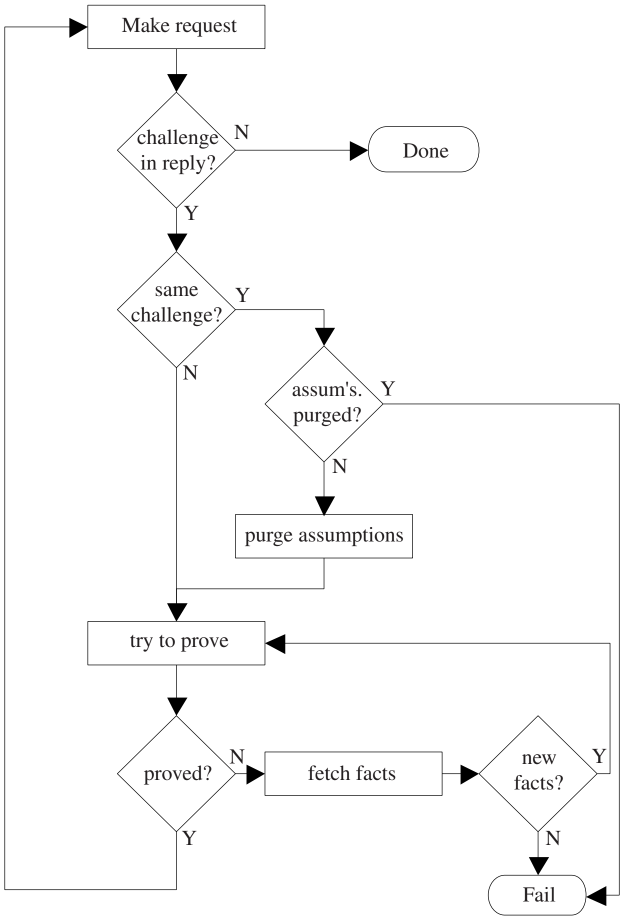

Figure 3. Client flowchart.

3.5 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.

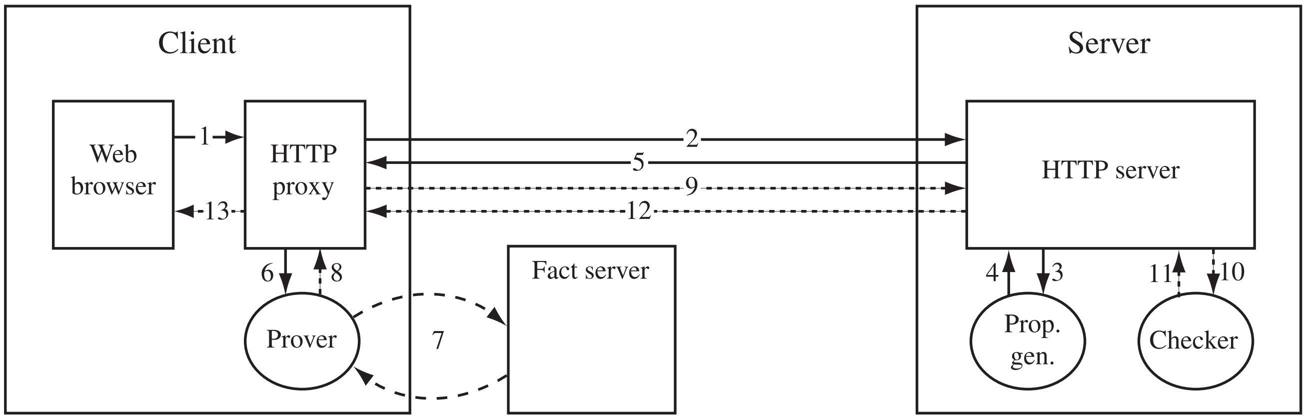

Figure 2. The components of the system.

The server's *proposition generator* provides the server with a list of propositions. The server returns to the client 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 3 and Figure 4.

The process of iterative authorization terminates when either the client gives up (i.e., 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 to the client twice.

If the client receives the same challenge twice, it knows that although it "successfully" constructed a proof for that challenge, the proof 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 system generates a proposition for each directory level of the URL specified in the client's request. This ensures that the client has permission to access the full path (i.e., just like in standard access control for a hierarchical file system). 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`:

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

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

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.

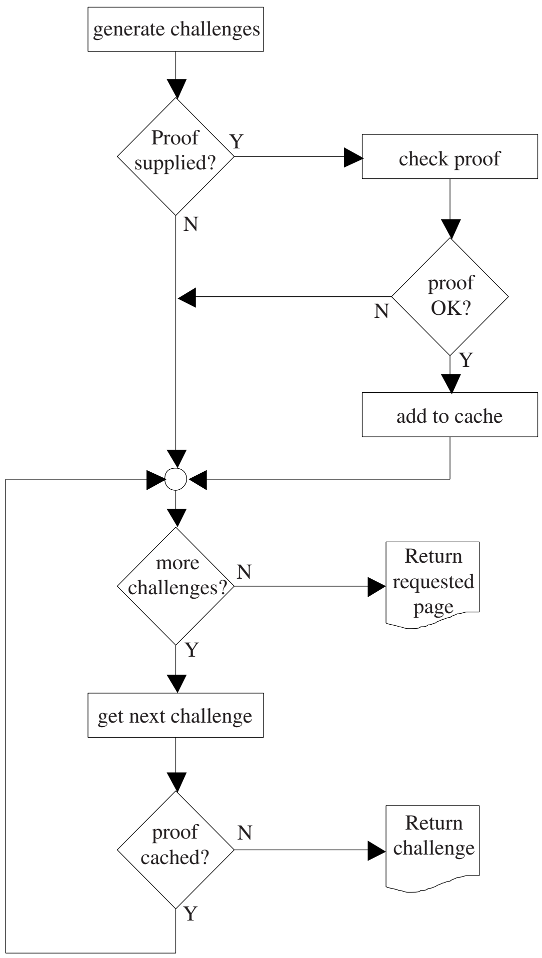

Figure 4. Server flowchart.

3.6 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 and a session identifier. The server checks each proposition to see if it was 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) [10]. 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, it makes another HTTPS request (this can be done with the same TCP connection if allowed by keep-alive) containing the challenge and the 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 for the next proposition.

> *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 2.*

3.7 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 [18]. Proofs are generated automatically 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 lead to a proof. For the restricted set of rules that we are interested in, the prover in our system is able to automatically generate proofs whenever they exist.

As part of 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.

3.8 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*
>
> ```
> (key "alice") says
>    (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 2.)*

Goal-oriented facts The second type of facts is typically (though not necessarily) provided by the web server. While generating propositions and checking proofs are conceptually the two main parts of the server-side 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 whether 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, and 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 for 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 ((localname (key "registrar") "cs101")
     says (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 2.*

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 proof, since it is the only "fact" 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 may contain facts relevant to that key.

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. The client then fetches new facts from all of these hint URLs. If needed, these new facts will be examined for additional hint URLs, which will then be fetched; this process will continue until all needed facts have been found. 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 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 2. Following the hint in the Registrar's key, Alice downloads a signed statement which she translates into the assumption

```
(key "registrar") says
  (before "registrar" (end of semester)
   imp ((key "alice") speaksfor
         (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.*

3.9 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 [13]. 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 [16, 20]; one of these could serve as the proof checker for our system.

LF terms can either have explicit type ascriptions or be implicitly typed. The explicitly-typed version may need to introduce more that one type annotation per variable, which 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 be 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 any digital signatures that are sent with the proof.

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 place holders 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*". 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. 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.

> *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.*

4 Optimizations and Performance Results

4.1 Caching and Modularity

Our authorization protocol involves a number of potentially lengthy operations like transferring data over the network and verifying proofs. We use caching on both the client and the server to alleviate the performance penalty of these operations.

Client-side One of the inevitable side-effects of a security policy that is distributed across multiple hosts is that a client has to communicate with each of them. Delegation statements in the security policy may force this communication to happen sequentially, since a client might fetch one piece of data only to discover that it needs another. While there is little than can be done to improve the worst-case scenario of a series of sequential fetches over the network, subsequent fetches of the same facts can be eliminated by caching them on the client. Some facts that reside in the cache may expire; but since it is easy for the client to check whether they are valid, they can be checked and removed from the cache out-of-band from the proof-generation process.

Server-side 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 proofs are kept cached as long as the session ID with which they are associated is kept alive.

Since all proofs are based on a sparse and basic core logic, they're likely to need many lemmas and definitions for expressing proofs in a concise way. 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 of being managed 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 bandwidth 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.

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.

4.2 Speculative Proving

In our running example the web proxy waited for the server's challenge before it began the process of constructing a proof. In practice, our proxy keeps track of visited web pages that have been protected using PCA. Based on this log, the proxy tries to guess, even before it sends out any data, whether the page that the user is trying to access is PCA protected, and if it is, what the server's challenge is likely to 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. If the web proxy already has all the facts necessary for constructing a proof, this will reduce the amount of communication on the network to a single round trip from the client to the server. This single round trip is necessary in any case, just to fetch the requested web page; in other words, the proof is piggybacked on top of the fetch message.

4.3 Performance Numbers

protocol stage	*ms*
fetch URL attempt without HTTPS	198
fetch URL attempt with no proof	723
failed proof attempt	184
fetch file fact + failed proof attempt	216
fetch key fact + successful proof attempt	497
fetch URL attempt (empty server cache)	592
failed proof attempt	184
fetch file fact + successful proof attempt	295
fetch URL attempt (server cached module)	330
total	3219

Figure 5. Worst-case performance.

protocol stage	*ms*
fetch URL attempt with no proof	180
failed proof attempt	184
fetch file fact + successful proof attempt	295
fetch URL attempt (server cached module)	330
total	989

Figure 6. Typical performance.

As one might expect, the performance of our system varies greatly depending on how much information

protocol stage	*ms*
construct proof from cached facts	270
fetch URL attempt (server cached module)	330
total	600

Figure 7. Fully-cached performance.

protocol stage	*ms*
fetch URL attempt (already authorized)	175
total	175

Figure 8. Performance with valid session ID.

is cached by the proxy and by the server. The relevant metric is the amount of time it takes to fetch a protected web page. We evaluated our system using the example of Alice trying to access `midterm.html` (see figures 5–8; for comparison, figure 9 shows the length of time to fetch a page that is not protected; the actual example from which we obtained performance data did not include facts about time).

The slowest scenario, detailed in figure 5, is when all the caches are empty and the first attempt to fetch the protected page incurs initialization overhead on the server (this is why the first attempt to fetch the URL takes so long even though a proof isn't included). In this case, it takes 3.2 seconds for the proxy to fetch the necessary facts, construct a proof, and fetch the desired page.

A more typical situation is that a user attempts to access a protected page on a previously visited site (figure 6). In this case, the user is already likely to have proven to the server that she is allowed access to the server and the directory, and must prove only that she is also allowed to access the requested page. In this case she probably needs to fetch only a single (file or goal) fact, and the whole process takes 1 second. Speculative proving would likely eliminate the overhead of an attempted fetch of a protected page without a proof, saving about .2 seconds. If the client already knows the file fact (figure 7), that length of the access is cut to about .6 seconds.

When a user wants to access a page that she has already accessed and the session identifier used during the previous, successful attempt is still valid, access is granted based on just the possession of the identifier – this takes about 175 milliseconds.

protocol stage	*ms*
fetch URL attempt (page not protected)	50
total	50

Figure 9. Access control turned off.

Alice's proof might have to be more complicated than in our example; it could, for example, contain a chain of delegations. For each link of the chain Alice would first have to discover that she couldn't construct the proof, then she would have to fetch the relevant fact and attempt to construct the proof again – which in our system would currently take about .6 seconds.

The performance results show that, even when all the facts are assembled, generating proofs is slow (at least 200 ms) and grows slower as the user learns more facts. While this is a fundamental bottleneck, the performance of our prover is over an order of magnitude slower than it need be. If this were a production-strength implementation, we would likely have implemented the theorem prover in Java. The capabilities of Twelf are far greater than what we need and impose a severe performance penalty; a custom-made theorem prover that had only the required functionality would be more lightweight. This also impacts the proof-checking performance; a specialized checker [21] would be much faster.

5 Conclusion

In this paper we describe an authorization system for web browsers and web servers that solves the problem of interoperability across administrative or trust boundaries by allowing the use of arbitrarily complex security policies. Our system 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 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 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.

Our performance results demonstrate that it is possible to reduce the inherent overhead to a level where a system like ours is efficient enough for real use. 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 investigating the use of oblivious transfer and other protocols for fetching proof components without revealing unnecessary information and further refining 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 for 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 existing (non-higher-order) security logics in a way that would enable authentication frameworks based on different logics to interact and share resources. Finally, we intend to significantly improve the performance of our system, in particular by using a specialized prover and proof checker.

6 Acknowledgments

The authors would like to thank Andrew W. Appel for his advice and the anonymous reviewers for their helpful comments.

7 Availability

More information about our system and proof-carrying authorization, including a downloadable version of our prototype implementation, is available at `http://www.cs.princeton.edu/sip/projects/pca`.

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] M. Erdos and S. Cantor. Shibboleth architecture draft v04. http://middleware.internet2.edu/shibboleth/docs/, Nov. 2001.

[10] 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.

[11] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, Aug. 2001.

[12] 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.

[13] 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.

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

[15] O. Kornievskaia, P. Honeyman, B. Doster, and K. Coffman. Kerberized credential translation: A solution to web access control. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, Aug. 2001.

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

[17] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33-38, Sept. 1994.

[18] 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.

[19] V. Samar. Single sign-on using cookies for web applications. In *Proceedings of the 8th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 158–163, Palo Alto, CA, 1999.

[20] A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. *Submitted to 14th conference on Computer Aided Verification*, 2002.

[21] A. Stump and D. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In *18th International Conference on Automated Deduction*, 2002.

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.

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

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

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

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

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

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

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

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

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

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

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