

Using SPKI/SDSI for Distributed Maintenance of Attribute Release Policies in Shibboleth

Sidharth Nazareth Sean W. Smith
Department of Computer Science
6211 Sudikoff Laboratory
Dartmouth College
Hanover, NH 03755 USA

Computer Science Technical Report TR2004-485
January 2004

Abstract: The *Shibboleth* middleware from Internet2 provides a way for users at higher-education institutions to access remote electronic content in compliance with the inter-institutional license agreements that govern such access. To protect end-user privacy, Shibboleth permits users to construct attribute release policies that control what user credentials a given content provider can obtain. However, Shibboleth leaves unspecified *how* to construct these policies.

To be effective, a solution needs to accommodate the typical nature of a university: a set of decentralized fiefdoms. This need argues for a *public-key infrastructure (PKI)* approach—since public-key cryptography does not require parties to agree on a secret beforehand, and parties distributed throughout the institution are unlikely to agree on anything. However, this need also argues against the strict hierarchical structure of traditional PKI—policy in different fiefdoms will be decided differently, and originate within the fiefdom, rather than from an overall root.

This paper presents our design and prototype of a system that uses the decentralized public-key framework of SPKI/SDSI to solve this problem.

1 Introduction

In this wired age, one might think of delivery of licensed Web content as a relation between two entities: the individual requesting the content, and the server providing that content. Whether the server provides that content to that individual depends on what deal they arrange.

However, in educational settings, many aspects of this transaction occur at the level of the institution, not of the individual. The *higher education institute (HEI)* negotiates (and often pays for) access to licensed material from a given content provider; whether an individual user can access this material depends on their relation to the HEI. Furthermore, it should come as no surprise that most HEIs are neither monolithic nor particularly centralized—the university fragments into schools and departments; and typically each increasingly local unit runs things differently from its peer units.

To address the basic problem of content delivery to users at HEIs, Internet2/MACE (with support from IBM) developed the *Shibboleth* system. Working within Web infrastructure and legacy authentication systems, Shibboleth permits content providers to learn, via the requesting user's HEI, whether the user has the necessary credentials to see that content.

Educational institutions value privacy, and Shibboleth respects that by hiding the user’s identity from the content provider, and by letting users have *attribute release policies* that control what credentials get released to what content providers. However, the basic system does not address how HEIs, with their Byzantine fiefdoms, can create, maintain, and resolve these attribute release policies. For any given user, many parties may need to participate, following that user’s local organization structure; but for two different users, these structures may differ.

This paper reports our use of SPKI/SDSI to design and prototype a system—*SPADE*—that solves this problem with Shibboleth. The existence of multiple distributed parties suggests the use of PKI; decentralization and the focus on authorization suggest the use of the SPKI/SDSI PKI framework. The egalitarian nature of SPKI/SDSI allows the system to model the natural hierarchy of the HEI. This also provides a datapoint for two sparsely populated spaces: PKI that does not focus on *identity*, and PKI that does not focus on the standard *X.509* format.

Section 2 presents the the basic Shibboleth system. Section 3 presents the attribute release policy problem. Section 4 presents the SPKI/SDSI framework. Section 5 and Section 6 then present our system. Section 7 reviews related work, and Section 8 suggests some avenues for future work.

2 Shibboleth Background

At its essence, education focuses on the sharing of information. In this information age, HEIs naturally have moved towards sharing information via the electronic medium of the Web; where appropriate, institutions would like to share material to students and staff at other institutions.

A standard framework would make it easier to share information—furthering the goals of education—by freeing each pair of institutions from having to develop their own scheme. Considering the scenario of a user Alice at institution I_A requesting some content from institution I_B leads to some design requirements:

- Institution I_B needs to know whether Alice is authorized to see the information, according to the agreement between I_B and I_A .
- Institution I_A probably already has its own system of authenticating whether a user is Alice; the framework should use that.
- Institution I_B does *not* need to know Alice’s identity, but just whether she’s authorized; institution I_A would probably rather not reveal Alice’s identity to I_B .
- We can’t expect Alice to use anything more than the desktop tools already common at I_A —e.g., a Web browser, and possibly an I_A -specific authentication tool, such as *Sidecar* [28].
- Similarly, we would like the added infrastructure for I_A and I_B to be only a small delta from their current infrastructure, and to conform with standards.

Shibboleth [13] provides such a framework. Shibboleth is a federated administrated system that supports inter-institutional authentication and authorization for sharing of resources, available to users from those institutions. As a “middleware architecture,” Shibboleth seeks to accommodate the different security systems existing in organizations and college campuses today. Shibboleth promotes interoperability by using standards, such as SAML and documented methods of information exchange; it assumes that users employ standard Web browsers to access these resources. Shibboleth places a great importance on user privacy: the content provider only knows an opaque, session-specific *handle* for Alice, not her name.

Components and Terminology To continue with the above usage scenario, suppose user Alice at institution I_A (the *origin* site) requests resource R from institution I_B (the *target* site). Figure 1 shows this situation. Institution I_A has some legacy way to determine who Alice is. Institution I_B needs to decide whether to send resource R back to Alice.

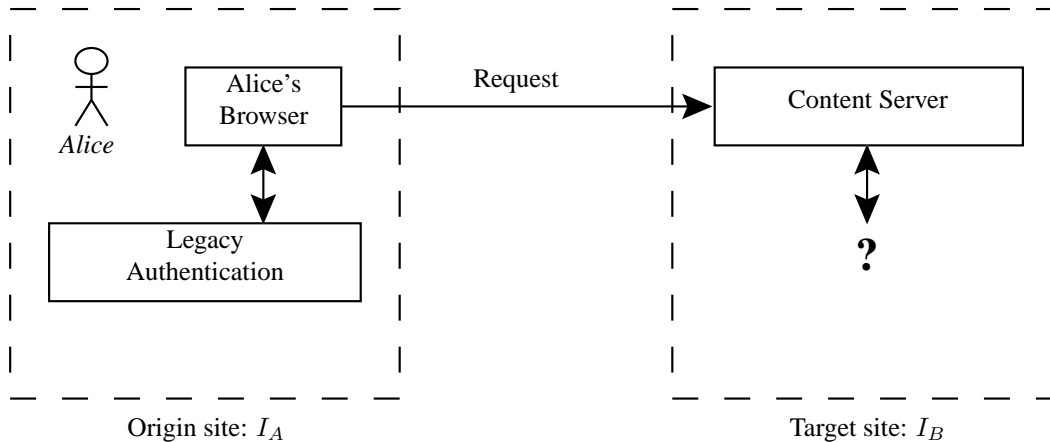


Figure 1: A basic usage scenario (without Shibboleth)

Shibboleth addresses this problem by adding additional authorization infrastructure at both sides, that exchange some additional messages. Figure 2 shows this situation.

First, Shibboleth permits the target site to learn an opaque *handle* to use when asking the origin site about that user.

- Alice’s request for R lands at the *Shibboleth Indexical Reference Establisher (SHIRE)* at I_B (message M_1 in Figure 2).
- The SHIRE is usually a Web server, and redirects the user’s request to the *Where Are You From (WAYF)* module (message M_2).
- The WAYF module—usually a part of the target site—interacts with the user and asks her where she is from. (The user usually has to select from a drop-down menu. In this way, the WAYF can also control which institutions have access to the target site.)
- The WAYF stores the mapping between the user’s origin site and the URL of the user’s *Handle Service (HS)*, an origin-side component that ensures the user has authenticated within I_A and that creates pseudonymous *handles* for users.
- Once the WAYF determines Alice’s HS, it asks the HS for her handle (message M_3).
- The HS responds by returning the handle to SHIRE at the target site (message M_4).

Shibboleth then permits the target site to ask the origin site if the user with this handle has the necessary credentials for this resource.

- In Shibboleth, these credentials are termed *attributes*; Shibboleth provides for one or more *Attribute Authorities (AA)* at the origin site. (This overlap with X.509 “attribute” terminology is unfortunate.) As part of the message M_4 , the HS also tells the SHIRE the URL of the AA to use for this user.
- At the target site, SHIRE passes the handle, request, and AA URL to the *Shibboleth Attribute Requester (SHAR)* (message M_5 in Figure 2).
- SHAR contacts the user’s AA and asks for this user’s attributes, in an *Attribute Query Message (AQM)* (message M_6 in Figure 2).
- The AA retrieves the relevant attributes and returns them to the SHAR in an *Attribute Response Message (ARM)* (message M_7).

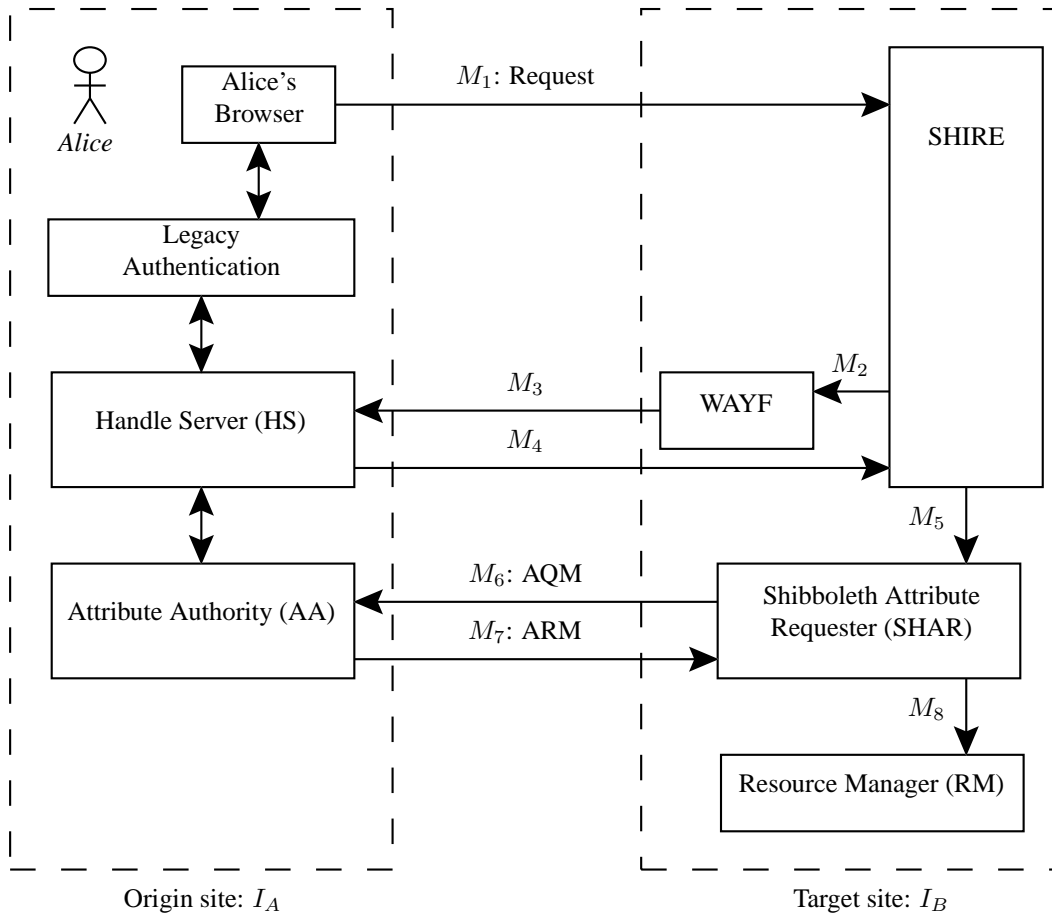


Figure 2: The basic usage scenario, with Shibboleth

- After the SHAR receives the user's attributes from the AA, it sends them to the *Resource Manager (RM)* for the resource R (message M_8 in Figure 2). The RM may have its own *Attribute Acceptance Policy (AAP)* which decides whether the user will be granted or denied access to the resource based on the attributes presented.

Shibboleth uses the *Security Assertion Markup Language (SAML)* for the AQM and ARM.

3 The Attribute Problem

Attributes User attributes are the basis for how the target site makes its authorization decisions. In Shibboleth, attributes are usually name/value pairs relevant to the user, such as:

- Role=Physics Student
- Name=Jane Smith
- Email=jane.smith@college.edu

Shibboleth also provides for *hidden attributes* relevant to the origin and target site, but not necessarily to the user. For example, Dartmouth College may have a contractual agreement with the Smithsonian Institute that allows Dartmouth users to access it; thus Dartmouth must provide additional information, such as a contract number.

```
SHAR: http://www.wisc.edu
URL: http://www.wisc.edu/*
ATTRIBUTES: Role='Student', Name='John Doe'
```

Figure 3: Example of Shibboleth Wildcarded ARP

Attribute Release Policies. Shibboleth places a large emphasis on user privacy. However, Shibboleth target sites are greedy and will try to obtain as many of the user's attributes as possible. To balance access and privacy, Shibboleth allows its users a choice in what information gets released about them and to which site. To achieve this balance, Shibboleth lets each user have an *Attribute Release Policy (ARP)*.

When the SHAR at the target site asks the AA at the origin site for attributes for the user with a specific handle, the AA retrieves that user's ARP and only releases attributes consistent with that policy. Shibboleth ARPs consist of a list of entries, each with three main fields:

1. A destination SHAR name (e.g., SHAR=http://www.mit.edu)
2. A resource URL (e.g., URL=http://www.mit.edu/ai/reconfiguring.jsp)
3. A list of attributes that can be released to this SHAR and URL (if the user has these attributes and the SHAR wants them)

The SHAR is usually the website where the URL resides and which hosts the resource.

As noted earlier, the origin site might have hidden attributes, such as a contract number, and an institution-specific ARP to specify when they should be released. Thus, the AA may serve to add additional attribute values into the Attribute Response Message (or ARM) sent to the SHAR.

Since it is not possible for a user and her institution to be able to provide ARPs for every possible target resource on the Internet, and every SHAR and URL pairing, Shibboleth permits *default* and *wildcarded* ARPs. Figure 3 shows an example of a wildcarded ARP that specifies that the values of the user's two attributes of Role and Name can be released to any site that the user accesses at the University of Wisconsin.

The Shibboleth standard requires that the AA must provide users at the origin side a means by which they can specify their Attribute Release Policies, This is usually done using a GUI such as a web browser and enables the user to control his own privacy. The downside, of course, is that a user's choice of ARP may not be proper to be able to grant him access to a target resource. Due to this problem, it is often preferable for the user to be aware of each site's attribute requirements, possibly shown on the interface.

The Problem. Shibboleth defines the basic structure and use of an ARP. However, how the AA retrieves a user's ARP is not part of the Shibboleth standard and is left open to interpretation. The user may have a single ARP or multiple ARPs. They may be dispersed throughout the organization or they may be collected in one place. How the user's ARP is retrieved, validated and enforced is left to the implementers. The Shibboleth draft [13] states:

AA implementers are free to support many different kinds of ARPs with varying semantics as long as the AA can efficiently process requests and determine the effective policy to apply...Shibboleth doesn't specify or constrain how an AA can answer these kinds of questions.

In a typical HEI, the lines of administrative control are dispersed—e.g., the user is one place; his or her department office somewhere else; the college office yet a third place. Furthermore, the structure of this distribution will vary from user to user. In a typical HEI, it is also likely that the decision procedure for attribute release will follow such administrative lines. This leaves us the questions:

- How do we build an ARP system that permits, for each user, the ARP to be easily resolved from the components each of these various parties introduces?
- How do we build an ARP system that enables each such party to easily build their local components?

Section 4 describes the PKI tool we used. Section 5 and Section 6 describe the system we built.

4 SPKI/SDSI: Lightweight Authorization PKI

To solve our distributed trust problem, the AA at the origin site needs to collect policy components created by many other parties throughout the institution. This task leads to some challenges:

- How does the AA verify the authenticity of these components produced by parties distributed throughout the institution?
- How does the AA merge these components?
- How does the AA even find these components?

The technology of public-key cryptography is well-suited to the first challenge—since parties distributed throughout the institution are unlikely to be able to agree on a secret beforehand (nor to agree on anything else, for that matter). To apply this technology to this problem, we looked at the tools available for *public-key infrastructure (PKI)*: “mechanisms for creating, distributing and using public keys” [21]. Typically, the main purpose of PKI is to specify how names and attributes should be bound to public keys. Usually PKI does this by using a *public key certificate*: a signed statement binding something to a public key. Typically, in a PKI, there are two types of certificates: an *identity certificate* usually binds a public key to a name; an *authorization certificate* (or *attribute certificate*) binds a public key to something else, and signifies an authorization or privilege that is meaningful to the signer and some set of interpreters.

4.1 X.509

Of the existing PKI tools in use today, *X.509* has emerged as the common standard. *X.509* uses a *certification authority (CA)* to sign public-key certificates for entities in the system. These CAs thus act as *trust roots* and verify that an entity (usually a user) is the rightful holder of a particular public key.

An *X.509 principal* is a user’s (allegedly) globally unique *distinguished name (DN)*. An *X.509* certificate thus binds a user’s DN to a public key to produce an identity certificate. *X.509* is based on a hierarchical global namespace; designated Certification Authorities sign and issue certificates to *X.509* users, and sometimes for each other. Trust is established by examining a chain of certificates from a trusted CA’s key to a user’s public key.

Recently, *version 3* of *X.509 (X.509v3)* has been released. *X.509v3* contains support for expressing attributes and authorization, by permitting additional *extension* fields for these items. Thus, an *X.509v3* certificate may bind a distinguished name to a public key and a set of attributes.

However, there are a number of problems with *X.509* in general, especially with regard to authorization and practical deployment.

First, by binding a user’s name to a public key and a set of attributes all in one *X.509v3* certificate, we immediately compromise the user’s anonymity: without additional countermeasures, a user cannot prove they have a given attribute without also revealing her identity (and the rest of her attributes). Moreover, while a user’s name is usually bound to her public key for long periods of time, attributes are usually bound to a user for short periods. Thus, if a user were to

want to change her attributes, we would need to revoke and reissue her identity. (This would be similar to re-issuing a university ID card each time a student added or dropped a class.)

To solve this problem, X.509 introduced a separate *attribute certificate (AC)*. Typically, this certificate binds a subject's Distinguished Name to a set of attributes and is presented to the verifier of the subject's attributes together with the subject's X.509 identity certificate. However, this approach incurs the overhead of two X.509 certificates for conveying a subject's attributes. Moreover, the principal of both the identity Certificate and the Attribute certificate is a name, but the entity that creates each certificate is different—thus the system needs to ensure that both creators meant the same thing by that name.

Another serious issue is the data format. X.509 certificates are expressed in the *Abstract Syntax Notation One (ASN.1)* standard. ASN.1 is not readable by (most) humans, and its complexity makes it difficult to program and work with.

A different complexity arises from the consideration that X.509 gives to the potential multiplicity of CAs. In order to attempt to characterize the information necessary for a relying party to make a reasonable judgment about trusting a CA's assertions, X.509 requires that a CA prepare a *Certificate Practice Statement (CPS)* that expresses the procedures and rules that the CA follows. Different CAs can be flexible in specifying their own rules, but the lack of a common mechanism results in different CAs providing a different level of service and consequently a different trust level. Moreover, a CA's CPS is specified in convoluted language with legal jargon that hardly anyone can understand (except perhaps the CA's lawyers).

4.2 SPKI/SDSI

For all these reasons, a number of new PKI standards were proposed.

In 1996, Ron Rivest and Butler Lampson at MIT designed the *Simple Distributed Security Infrastructure (SDSI)* [26] with the main goal of facilitating “the building of secure, scalable, distributed computing systems” [7]. Around the same time, Carl Ellison proposed the *Simple Public Key infrastructure (SPKI)* [5] to simplify authorization. In 1998, these two proposals merged to form SPKI/SDSI.

One of the main goals of SPKI/SDSI is to provide flexible and lightweight authorization—particularly in comparison to X.509. Moreover, to avoid the ambiguity problems in global names, SPKI/SDSI is centered on public keys—since a PKI principal always ends up needed to prove knowledge of the private key. SPKI/SDSI is egalitarian: no global hierarchy or hierarchical infrastructure necessary. Every principal is free to issue certificates to other individuals. This also leads to scalability. As noted above, SPKI/SDSI is centered on keys, and dispenses with the notion of globally unique names. A principal is free to use names, but these names are explicitly part of that principal's *local namespace*.

SPKI/SDSI works by combining a public key with a set of attributes to form an *authorization certificate* (in contrast to its counterpart, the X.509 attribute certificate that binds a distinguished name to a public key). Figure 4 shows this *authorization triangle* [12]. SPKI/SDSI authorization certificates list the issuer and subject (both identified via public keys), a *tag* field that carries the conveyed authorization, and validity dates.

One of the other main advantages of SPKI/SDSI is its ability to delegate authorization easily. This is done using a *propagate* bit in the authorization certificate. If this bit is set, then the subject of this certificate can re-delegate the permission or attributes he is given. Delegation is transitive; if Alice delegates to Bob (with permission to delegate further) and Bob delegates to Carol, then in effect Alice has delegated to Carol.

SPKI/SDSI can also create *name certificates* binding names to public key but they are not required for authorization decisions. The name is valid only for the local namespace and is used by the principal only for personal identification. SPKI/SDSI name certificates list issuer and subject (both identified via public keys), identifier (the local name), and validity dates. In SPKI/SDSI, a principal can also create *groups* by issuing multiple name certificates—one for each group member—with the same name and different subject keys. The concept of groups can also be used to create roles for role-based authorization.

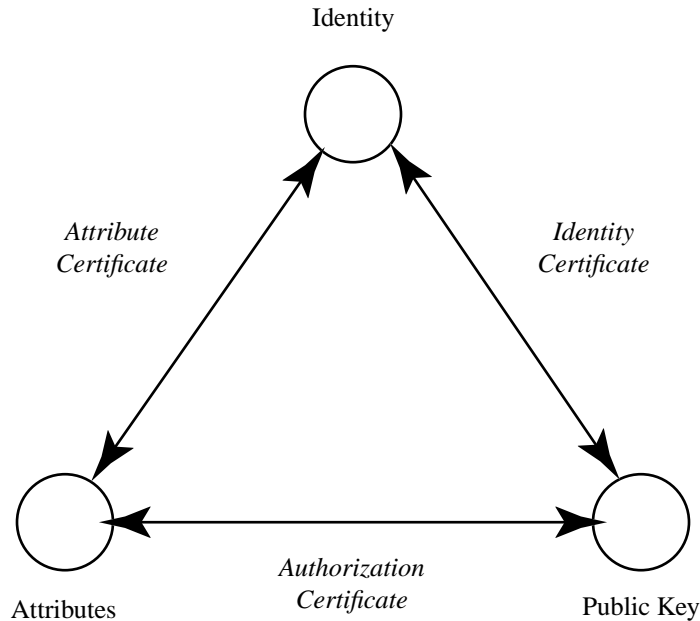


Figure 4: SPKI/SDSI authorization triangle (derived from [12]).

SPKI/SDSI can implement *threshold certificates* where authorization is a collective decision of k -of- n entities.

In SPKI/SDSI, an *authorization flow* is a chain of valid certificates—authorization only, or authorization and name—that specifies an authorization [7]. The *trust root* of an authorization flow is the first principal in the chain who started the delegation process. Unlike X.509, this principal does not need to be a distinguished CA.

SPKI/SDSI certificates use *S-expressions*: “human-readable ASCII representations” [25]. LISP-like S-expressions make it easy for humans to read a certificate and deduce its meaning, in contrast to X.509 and ASN.1. Parsing canonical S-expressions can take much less code size than parsing ASN.1 data; in one study [11], S-expression parsing code takes 8 KB, while ASN.1 parsers take 50KB to 500KB,.

As noted earlier, SPKI/SDSI uses a tag-language to express authorization information through large sets of strings or S-expressions. Standard libraries may be written to intersect certificate tags and deduce what an authorization chain authorizes. On the other hand, because of X.509’s generality, each developer is free to define an X.509v3 extension format and then define how those attributes may be intersected. Thus, each application using X.509 has to write a separate X.509 chain-processing routine.

In contrast to X.509, SPKI/SDSI emphasizes short-lived certificates, rather than *Certificate Revocation Lists (CRLs)*.

4.3 Choosing a Tool

SPKI/SDSI focuses on authorization, not authentication. In the Shibboleth model, the user is assumed to be already authenticated when we retrieve the user’s relevant ARPs. However, the focus now is on the user’s ARP and his authorization with respect to the target site resource. Shibboleth does not specify how the user constructs his ARP or how the origin site resolves it.

A SPKI/SDSI-based PKI can make use of its egalitarian design and authorization-centric approach to allow users and their HEI to specify roles in which those users can participate in, and then to follow these authorization flows to resolve the ARPs.

For all the reasons, SPKI/SDSI seemed a like good fit in this area—although no one had yet examined its use in Shibboleth. Section 5 and Section 6 describe our exploration.

5 SPADE Design

5.1 Overview

To address the attribute release problem in HEIs, we designed and prototyped *SPADE: SPKI/SDSI for Attribute Release Policies in a Distributed Environment*. SPADE logically divides an institution or organization into *domains* that correspond to physical or structural divisions, such as project groups or academic departments. Each such domain contains a group of users, as well as an administrator who distributes and manages trust for that domain.

SPADE uses SPKI/SDSI to establish and manage trust. The main concept behind SPADE is that each party in an organization uses signed SPKI/SDSI authorization certificates to specify an ARP component (in the tag field). SPADE then acts as an extension of the Shibboleth Attribute Authority (AA). When standard requests for user attributes come into the AA, they are redirected to SPADE which contacts the relevant user domain and obtains the user’s ARP component. This ARP is combined with other ARP component in the organization by obtaining a chain of certificates from the relevant domains of the organization, starting with the individual domain user and his administrator. The ARP components (the tag fields in these certificates) are intersected to derive the final ARP. By extending the AA function in this way, this work does not involve modifying any part of the Shibboleth standard.

5.2 Users and Roles

In a higher education institution, users typically act in roles—e.g., *faculty*, *assistant*, or *student*—within some organizational unit. In SPADE, we identify subjects by roles bound to their public keys, via a SPKI/SDSI name certificate created by the administrator of that user’s domain.

In typical settings, what a user does within a role depends not just on that user’s will, but also on local culture and policy. SPADE needs to reflect this flow by letting a user create their own ARP component, to control what attributes are released when the user acts in that role, but also by letting organizations establish ARP components that further constrain what users can do in a given role.

This organization-level restriction can apply before a user ARP component is created. For example, consider the above three roles, at a hypothetical college. The college may decide that, of these roles, only *faculty* and *assistant* should be allowed to release their credit card number to Shibboleth resources. This organization-level decision should constrain what a user is actually allowed to do when she creates her ARP in the first place (e.g. a user in the student role should not be able to create an ARP that allows release of credit card number).

Organization-level restrictions may need to apply dynamically, as well. For example, suppose Alice is *faculty*, and chose to allow her credit card number to be released. Two weeks later, her college’s security officer learns of credit card fraud taking place at *hacker.com*—but suspects that the college users—including Alice—are not aware of this fraud. The security officer needs to be able to modify its ARP to prevent any of its users’ credit card numbers from going to *hacker.com*. Thus, even though Alice has been allowed to specify that she would like to release her credit card number to *hacker.com*, and she has indeed specified that, the security officer’s ARP prevents the attribute from going through to the site. A week later, the fraud has been detected and taken care of and *hacker.com* has assured its users that it is safe to send personal information to the site again. The college now modifies its ARP again to allow all credit card numbers going to *hacker.com* to be released once again. In the meantime, whether Alice chooses to do so or not remains her personal choice.

This example gives only one level of organization; scenarios exist for multiple levels as well.

5.3 Deriving Policy

SPADE derives attribute release policies by permitting each relevant party to create an authorization certificate expressing its ARP component as an S-expression in the tag field, and then combining the component policies from a chain of certificates.

Admin ARP Certificates. We start by considering the input that an administrator (e.g., the “security officer” in the above scenario) needs to have in creating ARPs.

- The admin needs to express different policy for different subordinate roles.
- For each such role, the admin may wish to confine the attributes such a user may even choose to release to a given site, when creating their policy. (E.g., above, the admin originally allowed `faculty` to choose to release credit card numbers to `hacker.com`.)
- For each such role, the admin may wish to prevent certain attributes from being released to a given site—even if that role’s policy permits that. (E.g., above, the admin wanted to temporarily prevent `faculty` from releasing credit card numbers to `hacker.com`.)
- For each such role, the admin may wish to require that attributes specific to some agreement between that college and some given site be passed on, without bothering the user.

To express this information, the admin creates a SPKI/SDSI certificate whose issuer is the admin’s public key, and whose subject is this subordinate role. Within the tag field, this certificate may contain three different lists of ARP entries (e.g., tuples of a SHAR, URL, and attribute list) for that role.

- One list specifies the *releasable attributes* (`rATTRIBUTES`), which the user may choose to release when constructing his or her own policy.
- Another list specifies plain *attributes* (`ATTRIBUTES`), which constrain what that role can release right now.
- The final list specifies *hidden attributes* (`hATTRIBUTES`), which should be passed on without bothering the user.

Since this certificate expresses rights and permissions for a following certificate, its propagate bit is set.

Figure 12 (discussed in further depth later) shows an example of an admin ARP certificate.

The admin can split this specification across multiple certificates: each for the same subject, but whose ARP entries (for each type of attribute) discuss different SHAR-URL pairs. If the admin partitions the specification into a certificate for `rATTRIBUTES` and one of the other two types, we call the former a *user template* and the latter a *filter certificate*.

User ARP Certificates. An end-user wants to express what attributes she wishes to release to a given site, when she is acting in a specific role. To do this, she creates a self-signed SPKI/SDSI certificate (since she is speaking about herself), that gives a list of ARP entries (tuples of a SHAR, URL, and `rATTRIBUTE` list). The `rATTRIBUTE` field refers to the attributes that the user chooses to release.

Users typically create one ARP cert to handle all their roles. But, as with admin certs, the user can split this specification across multiple certificates whose ARP entries discuss different SHAR-URL pairs.

A user’s role is bound to her public key by using the user’s own ARP cert, and the admin’s name certificate which binds the role name to the user’s public key.

Figure 14 (discussed in further depth later) shows an example of a user ARP certificate.

Intersection. SPADE provides a tool for users to create user ARP certificates, and for the Shibboleth AA to derive an ARP for a given user in a given role. Both processes involve standard SPKI/SDSI tag intersection over an authorization flow of certificates.

When a user creates an ARP certificate for a role, SPADE lets the user select attributes from a list—but this list only consists of the intersection of the `rATTRIBUTE` lists in the relevant admin certs.

When dynamically resolving an ARP for a user and role, SPADE first intersects the `ATTRIBUTE` lists in the relevant admin certs to derive the list of attributes the admins currently feel permissible, and then intersects that result with the `rATTRIBUTE` list in the user cert. SPADE also unions the `hATTRIBUTES` lists in the relevant admin certs to derive the list of hidden attributes that admins currently wish to always be released, and unions that with the result of the above intersection.

Attribute Values. As discussed in Section 3, full Shibboleth attributes consist of a pair of *name* and *value*. Our policies discuss the attribute names only, not the values; we store the full attribute in a database at the user’s domain. Thus, we preserve confidentiality is even if outside users inspect the policies or certificates.

5.4 Domains

SPADE now has a way to derive policy along SPKI/SDSI authorization flows, but we want these flows to follow the decentralized organizational lines that arise in real educational institutions.

To do this, SPADE divides an organization into a number of logically separable *domains*, representing sets of users within some natural organizational unit—e.g., the “History Department at Dartmouth.” In our initial prototype, we assume these sets are disjoint, but the system could also work as long as each user-role pair resides in at most one domain.

SPADE also gives each domain *domain controller* that retrieves the ARP certificates forming the authorization flow for a particular user in this domain. The SPADE *head controller* maintains contact with all the domain controllers in the HEI’s SPADE infrastructure.

Domain Relationships. SPADE requires that the domains within an HEI be organized as a set of trees.

- Each domain has at most one predecessor and any number of successors.
- Within each tree, a unique *source domain* has no predecessor.
- A *leaf domain* has no successor.
- An *intermediate domain* has at least one of each.

(We could easily extend from trees to more general directed acyclic graphs.)

Domain organization follows the hierarchical structure of the organization: the predecessor domain is the logical next higher domain in the organizational hierarchy; the successor domain is the logical next lower domain. For example, for the “Arts and Science” domain at Dartmouth, the “Dartmouth” domain is its predecessor, and the “History Department” and “Computer Science Department” are among its successors.

Our ARP authorization flows follow this structure: starting with the source domain; each domain then establishes a cert that propagates ARP constraints to its successor.

Administrator Tasks. Each domain contains an *administrator* and is uniquely identified by its administrator’s public key.

The administrator is responsible for the following functions:

Establishing User Identity. The administrator obtains the public key of the user (preferably offline) and then binds it to the user’s name using a SPKI/SDSI Name certificate. (This is solely for the purpose of identifying the owner of the public key when using our SPADE GUI; is not used in authorization decisions.)

Creating Roles. The administrator creates domain roles and assigns users to them using the SPKI/SDSI mechanism for creating groups. (Thus, authorization takes place based on the user’s role, in the spirit of *Role Based Access Control—RBAC*.)

Creating Admin ARP Certs. The administrator creates SPKI/SDSI Authorization Certificates to constrain the ARPs that SPADE derives for these roles and any successor domains.

Establishing Inter-domain Trust Relationships. The administrator also manages the trust relationships between his domain and other domains at the HEI. (This structure is discussed further below.)

A domain’s administrator is responsible for correctly identifying that domain’s predecessor and successors. (In fact, our prototype uses SPKI/SDSI certificates to express these links.)

User Tasks. Each user within a domain is responsible for creating her Attribute Release Policies for different target resources in Shibboleth. This is done via a user ARP certificate, as discussed above.

5.5 Putting it All Together

When SPADE-enhanced Shibboleth receives an AQM, it starts with the user’s ARP certificate and traces back the the flow for that user. After SPADE verifies the signatures and derives the resultant ARP, it retrieves the requested attributes—if the resultant ARP permits that—and returns them to the Attribute Authority in the usual Shibboleth way, by embedding in a SAML ARM.

Figure 5 shows the entire process.

1. First, the Shibboleth Attribute Requester (SHAR) at the target site contacts the Attribute Authority (AA) at the origin site for the relevant user’s attributes. The SHAR passes the AA the user handle, and the SHAR and URL pair of the target resource that the user wishes to access.
2. The AA resolves the handle into the user name and public key (by using an HEI-issued X.509 Authentication Certificate from the user’s browser) and passes this information on to the SPADE head controller together with the SHAR and URL values for which the attribute values were requested.
3. The head controller resolves the domain of the user and contacts the user’s domain controller (DC). The head controller, in effect, hands off the processing to the user’s DC and waits for the attribute values. In this example, the user is from the History Department.
Since the user specifies which of his many roles he is acting in when he logs into the system, the DC knows what role in which the user is acting and retrieves the group cert that matches that role to that user’s public key. The user’s DC obtains the user’s ARP certificate from the domain database. It also obtains the domain administrator’s filter for the user’s role.
4. The user’s DC, after checking the administrator’s database for the URL of the the predecessor domain, contacts the predecessor’s DC and requests its administrator’s ARP certificate for this SHAR and URL. In this case, the “History Department” predecessor domain is the “Arts and Science” Domain.

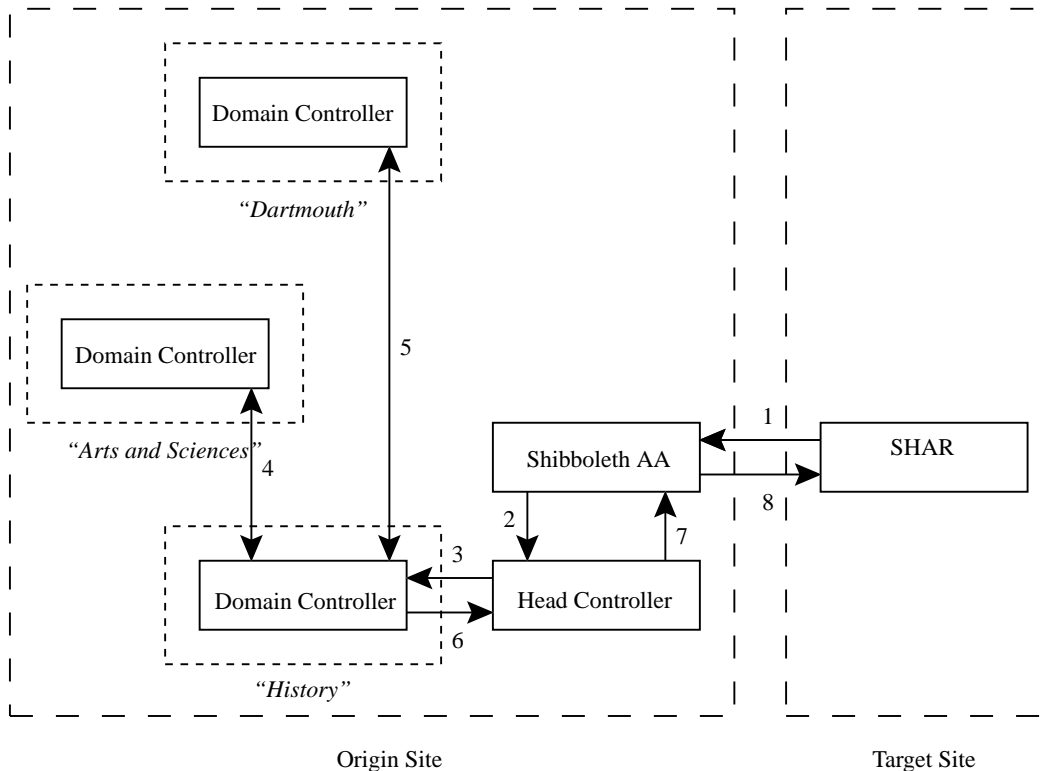


Figure 5: Example of SPADE attribute release.

5. The user's DC then obtains the "Arts and Science" predecessor domain from its database. (In this case, it's the "Dartmouth" domain.) The user's DC retrieves Dartmouth's Admin ARP for the "History Department" (and this SHAR and URL). The user's DC also stops retrieving certificates here because it knows that "Dartmouth" is this HEI's source domain.

The user's DC then intersects the user's ARP, the "History Department" ARP for that user's role, the "Arts and Science" ARP for the "History Department," and the "Dartmouth" ARP for "Arts and Sciences" to form the resultant ARP. The values of the attribute names specified in the resultant ARP are pulled from the user's database.

6. The user's DC returns the user's attribute values to the SPADE head controller.
7. The head controller in turn returns the attributes to the AA.
8. The AA, now using the Shibboleth specification, bundles the attributes into SAML and sends them off to the SHAR which decides whether or not the user gets access to the target resource.

6 SPADE Prototype

We have prototyped SPADE, and connected it to a Shibboleth test deployment at Dartmouth. As discussed above, our SPADE prototype acts as an extension of the Shibboleth Attribute Authority (AA).

We have also implemented a web-based GUI that allows the users and domain administrators to log on and manage their policies. An administrator uses the GUI to authorize new users in the system, create roles to which users may be assigned, assign those roles, and create admin ARP certs to constrain policies for users and subordinate domains. user may use the GUI to create and modify her attribute release policy.

Our prototype uses Java, HTML, and Java Servlets. The code is divided into three main parts:

SPKI/SDSI code. We started with the `sdsi` Java code developed by the Cryptography and Information Security Group at MIT [9]. This code is a GUI package designed to familiarize users with SDSI operations such as creating certificates and deriving authorization decisions from certificate chains.

Starting with this base, we made a number of modifications (adding approximately 1250 lines of code). As part of this work, we stripped away the GUI code and adding helper functions in relevant classes. We also implemented a library to intersect SPKI/SDSI certificate tags, based on the SPKI/SDSI standard [5]. This library processes normal as well as special S-expressions such as the wildcarded (`*`), (`* range`), (`* prefix`) and (`* set`) formats. We also wrote helper files to process SPADE-relevant operations quickly, such as obtaining the names of all roles in a domain by resolving its authorization certificates and name certificates.

Java Servlet code. The bulk of SPADE is our approximately 4000 lines of Java servlet code. This component plugs into the SPKI/SDSI code to create objects such as public keys, name certificates, and authorization certificates and to verify certificate chains. It handles the web-based GUI: building dynamic web pages, processing user input and interacting with the prototype databases (which are actually file system directory hierarchies).

HTML code. SPADE also contains a small amount of HTML code to call and process servlets.

In our experimental setup, the entire code package and directory hierarchy (approximately 5500 lines of code) sits on a server, where it is plugged into a Shibboleth test club implementation (Shibboleth v 0.8). The current code is available for download at <http://www.cs.dartmouth.edu/~nazareth/thesis/>

Figure 6 through Figure 14 show some examples from our prototype.

For a domain administrator, one of the first tasks might be defining the roles in that domain, and assigning users to roles. Figure 6 shows the GUI for an admin to create SPKI/SDSI groups and add users. Figure 7 shows an example group certificate indicating group measurement, for a user principal expressed as a public key; for convenience to the administrator, Figure 8 shows an example name certificate binding a local name to this public key.

Discussing an attribute release policy requires working with SHAR-URL pairings, since Shibboleth organizes ARP entries by these pairings. Figure 9 shows the GUI to create these pairings. With roles and SHAR-URL pairings established, Figure 10 then shows the GUI the administrator can use to specify the various admin ARP certificate components. The admin can include hidden attributes in the ARP certificate which he or she can choose to tell the user about. Figure 11 then shows the GUI that summarizes the policy, and lets the administrator actually create the certificate. Figure 12 shows an example of the resulting certificate.

On the user end, Figure 13 shows the GUI that lets the user construct a user ARP certificate; the user can select a role, and then, for the SHAR-URL pairings, can select the attributes (from the choice given by the admin certificates) the attributes she wishes to release. Figure 14 shows an example of the resulting certificate.

7 Related Work

Shibboleth uses attribute release policy to protect user privacy; SPADE uses SPKI/SDSI to provide a decentralized way to manage them. In this section, we quickly review some other principal work in this general area.

The *Open Profiling Standard (OPS)* [17, 18] employs a “personal profile” to let Web users specify what attributes they wish to release. Unlike Shibboleth (which it preceded), OPS allows the content provider to write a user’s profile, and requires modifications to the client browser.

The *Platform for Privacy Preferences (P3P)* project [8] provides an XML-based way for content providers (and other Web entities) to describe their data collection practices, so that users can make informed privacy judgments before

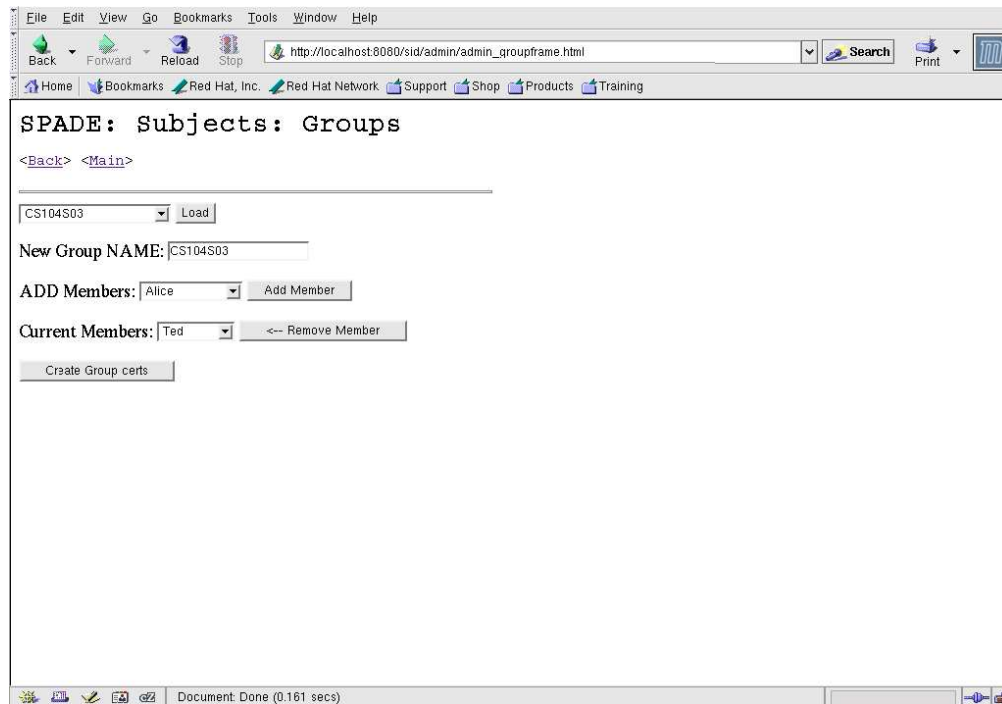


Figure 6: The SPADE GUI that let the domain administrator create and manage roles.

```
(cert
  (issuer
    (name
      (public-key
        (rsa-pkcs1-md5
          (e #010001#)
          (n
            |AMh8A5+uINnrESIh1W2Gk1Uf4kthKnWnjNyg8wXNJXPaEXax/DPTNTMDuUC
            ZRVyFGohuqr87VAR4ZFU99OkIWeryPwKsXjwyHr7Bq1K61+n0HByRNhZkBFj
            qa jpreByRw9V+fwV908WPiVkmvIgtPcMF6rCKfU3f5M2OROpJ1ZVZ| )))
      CS104S03))
  (subject
    (public-key
      (rsa-pkcs1-md5
        (e #010001#)
        (n
          |AKIZRHmb9t01jedSUwph3Jxk79OzEEuPlZmmEToqhUratcj6pp9TzL07NGWUT
          XTKdFT5591M8QU0P6t3FV/uyIO3MTZJrFcUaiqls1I9IiwFpI5KyJisURaDzsf6
          uC+luix+lHWtpI2+x6rwy6FxFuUrrWKxyAmQ0uI38JvJSMcJrd| )))
  (valid
    (not-before "2003-01-01_00:00:00")
    (not-after "2004-01-01_00:00:00")))
```

Figure 7: SPADE might use a SPKI/SDSI group certificate like this to indicate that the domain matching first public key says that the user matching the second public key can act in the CS104S03 role.

```

(cert
  (issuer
    (name
      (public-key
        (rsa-pkcs1-md5
          (e #010001#)
          (n
            |AMh8A5+uINnrESIHlW2Gk1Uf4kthKnWnjNyg8wXNJXPaEXax/DPTNTMDuUC
            ZRVyFGohuqr87VAR4ZFU99OkIWeryPwKsXjwyHr7BqlK61+n0HByRNhZkBFj
            qajpreByRw9V+fwV908WPIVkmvIgtPcMF6rCKfU3f5M2OROpJ1ZVZ|)))
        Alice))
  (subject
    (public-key
      (rsa-pkcs1-md5
        (e #010001#)
        (n
          |AKIZRHmb9t0ljedSUwph3JxK79OzEEuPlZmmEToqhUratcj6pp9TzL07NGWUT
          XTKdFT5591M8QU0P6t3FV/uyIO3MTZJrFcUaiqs1I9IiwFpI5KyJisURaDzsf6
          uC+luix+lHWTpI2+x6rwy6FxxUuRrWKxyAmQ0uI38JvJSMcJrd|))))
  (valid
    (not-before "2003-01-01_00:00:00")
    (not-after "2004-01-01_00:00:00")))

```

Figure 8: SPADE might use a SPKI/SDSI name certificate like this to indicate that the domain matching first public key says that the user matching the second public key has local name “Alice.”

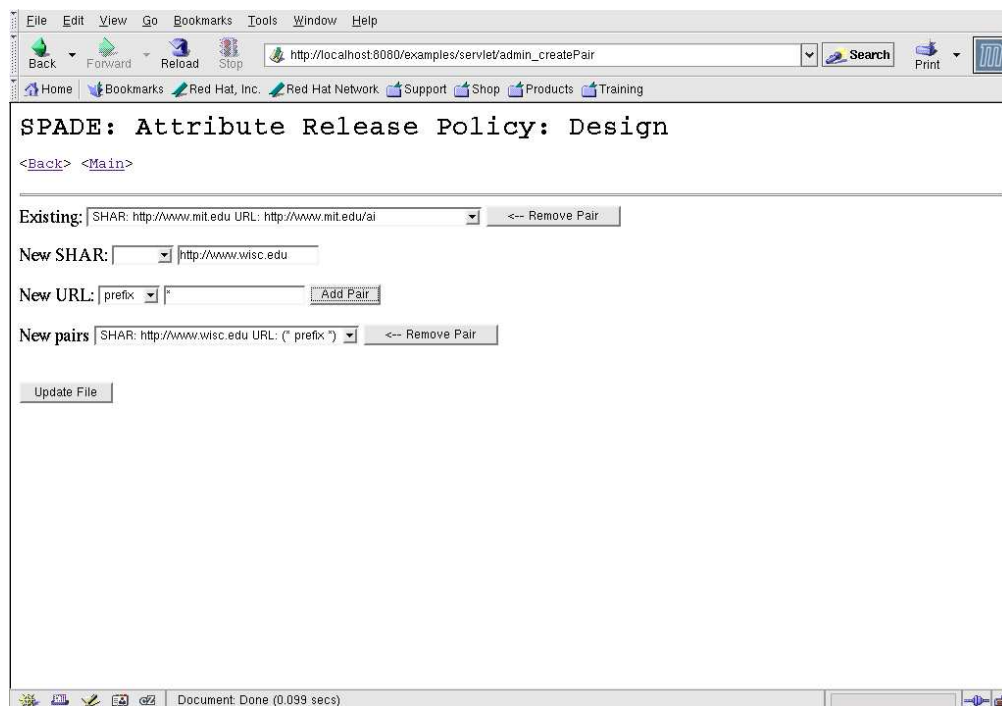


Figure 9: The SPADE GUI that enables creation of SHAR-URL pairings, to be used in ARP certificate construction.

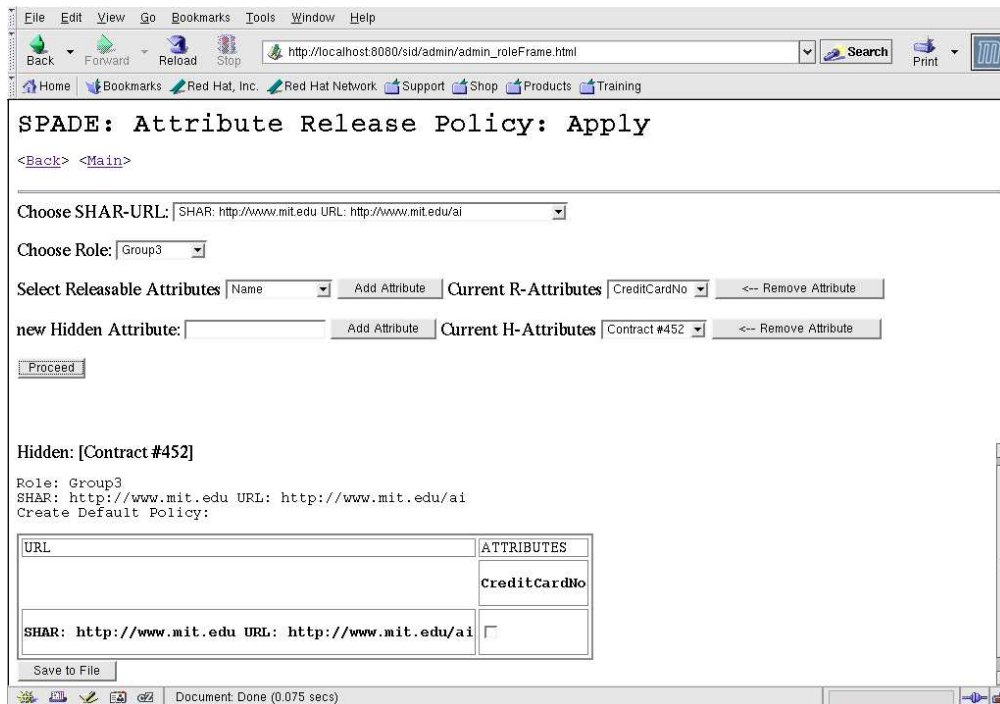


Figure 10: The SPADE GUI that lets the domain administrator construct entries for an admin ARP certificate.

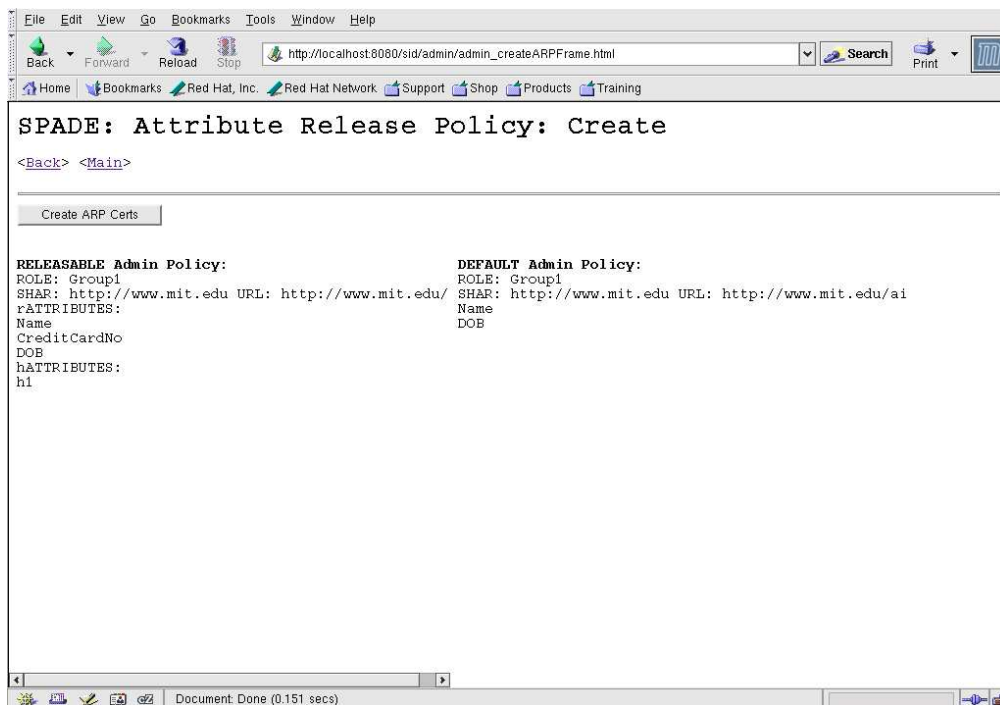


Figure 11: The SPADE GUI that summarizes the policy a domain administrator has created, and lets them go ahead and create the actual certificate.

```

(cert
  (issuer
    (public-key
      (rsa-pkcs1-md5
        (e #010001#)
        (n
          |AMh8A5+uINnrESIH1W2Gk1Uf4kthKnWnjNyg8wXNJXPaEXax/DPTNTMDuUCZR
          VyFGohuqr87VAR4ZFU99OkIWeryPwKsXjwyHr7Bq1K61+n0HByRNhZkBFjqajp
          reByRw9V+fwV908WPiVkmVigTpcMF6rCKfU3f5M2OROpJ1ZVZ|)))
    (subject
      (name
        (public-key
          (rsa-pkcs1-md5
            (e #010001#)
            (n
              |AMh8A5+uINnrESIH1W2Gk1Uf4kthKnWnjNyg8wXNJXPaEXax/DPTNTMDuUC
              ZRVyFGohuqr87VAR4ZFU99OkIWeryPwKsXjwyHr7Bq1K61+n0HByRNhZkBFj
              qajpreByRw9V+fwV908WPiVkmVigTpcMF6rCKfU3f5M2OROpJ1ZVZ|)))
          CS104S03))
      (propagate)
      (tag
        (adminARP
          (ROLE
            (CS104S03)
            (SHAR
              (http://www.dartmouth.edu)
              (URL
                (http://www.dartmouth.edu/cs)
                (rATTRIBUTES (CreditCardNo DOB Email)
                  (hATTRIBUTES (contract_number)
                    (ATTRIBUTES (DOB Email))))))))))
        (valid
          (not-before "2003-01-01_00:00:00")
          (not-after "2004-01-01_00:00:00")))

```

Figure 12: This example SPADE admin ARP certificate shows the attributes a CS104S03 user can choose to put in her ARP (rATTRIBUTES), can currently release (ATTRIBUTES), and the hidden attributes (hATTRIBUTES) that are released without bothering the user.

interacting. Users must use a browser that is aware of P3P policy; candidates include Microsoft's IE (version 6 and later) [23] as well as the *Privacy Bird* plug-in (www.privacybird.com).

The *eXtensible Access Control Markup Language (XACML)* [14] is an XML-based language for expressing a user's security policies and involves authorization for resources. XACML supports fine-grained authorization control and RBAC. However, XACML is only a language for specifying policies and how to combine them to deduce authorization. XACML does not define a policy-based infrastructure, nor does it cover policy dissemination, retrieval, enforcement and administration. Although it is not part of the Shibboleth specification, XACML could be used to specify a Shibboleth ARP. (Subsequent to our SPADE work, Lorch et al [22] have briefly explored the XACML approach as part of a large exploration of XACML-based access control, the "inherent semantic complexity" contrasts with the simpler SPADE approach.)

Like XACML, the *Security Assertion Markup Language (SAML)* [15] is another XML-based OASIS initiative. SAML provides a way to exchange information about user authentication, authorization and attributes between online sites. Unlike XACML, SAML makes no effort at privacy or policy specification. As we have discussed, SAML is used in Shibboleth to carry user attributes between the origin site and the target site.

IBM's *Enterprise Privacy Authorization Language (EPAL)* [27] is an XML-based model for "enterprise-internal privacy policies." However, unlike Shibboleth and SPADE, EPAL does not permit end subjects to manipulate policy.

The work of Blaze et al [4, 2] defined *trust management* as "a unified approach to specifying and interpreting security policies, credentials, and relationships that allows direct authorization of security-critical actions." Trust management systems handle security policies, authentication as well as authorization. An integral and central part of a Trust Management system is the compliance checker. A request, a policy and a set of credentials (usually a public key certificate) are input to compliance checker which answers "yes" "no", depending on whether the credentials and request complies with the policy. PolicyMaker [2] and KeyNote [3] are two examples of Trust Management systems. PolicyMaker expresses authorization policies and trust deference policies through assertions in a "safe" language; KeyNote was designed according to the same principles as PolicyMaker, using credentials that directly authorize based on a public key. Two additional design goals for KeyNote were standardization and aiding the ease of integration into applications.

The *Distributed Trust Management System* at the University of Maryland, Baltimore County [20] uses rights and delegations with certificates to create a trust management system. This infrastructure uses X.509 certificates and Prolog policies to enforce security. (SPADE, which uses SPKI/SDSI, possesses many of the properties of a Distributed Trust Management system such as delegation and authorization.)

IBM's *Trust Establishment Framework (TEF)* [19] maps the subject of a certificate to a role, based on the certificate and policy. Like SPADE, the IBM TEF supports decentralized trust structures "mirroring the real world." TEF uses X.509 Attribute Certificates, and uses an XML-based *trust policy language*.

Chadwick's *Privilege Management Infrastructure (PMI)* [6] looks at authorization in the same hierarchical spirit as X.509 identity: "a PMI is to authorization what a PKI is to authentication." His *PERMIS* system is the primary example of an X.509-based PMI (although PERMIS also authenticates and thus could be also be considered a Trust Management System). SPADE can be thought of as a SPKI/SDSI PMI—using SPKI/SDSI authorization certificates instead of X.509 attribute certificates. Like PERMIS, SPADE embeds the policy in the certificate. However, unlike PERMIS, we use S-expressions and the standard SPKI/SDSI tags instead of an XML-based policy.

PolicyMaker, KeyNote and PERMIS are trust management systems that include a policy model. In contrast, *Ponder* [10] and the *Simple Policy Control Protocol (SPOCP)* [16] are policy models without a trust management system. Ponder is an object-oriented language that can specify both security policies and management policies. SPOCP can be thought of as a policy engine that tests if an authorization request from a client ought to be allowed or not, given a set of policies. SPOCP uses S-expressions to specify queries and policies.

8 Conclusions and Future Work

The principal goal of SPADE is to show the viability and effectiveness of using SPKI/SDSI as a basis for a distributed trust system for specifying and conveying policies for digital libraries—particularly in institutes of higher education where separate fiefdoms make public key technology preferable to shared secrets or central servers, but where disparate local hierarchy makes traditional PKI in appropriate.

SPADE uses SPKI/SDSI certificates to delegate authority and to manage policies in an organization. SPADE allows user to define and create their own ARPs, specific to their organizational assigned role in that Domain. SPADE allows other Domains in the organization to exert their influence on the individual user's ARP and to thus base the final ARP on the intersecion of these ARPs.

As a sociologist colleague observed [1]:

SPADE is able to capture the necessary features of most of modern organization (hierarchical structure) and the benefits of hierarchy, without also capturing the costs/negative aspects of hierarchy. This is very cool

Future work includes trying the system in pilot populations, as well as extending this approach to decentralized trust management to other arenas in Shibboleth (such as attribute acceptance policies, at the content provider) and elsewhere. The issues of the "timeliness" or "freshness" of SPKI/SDSI certs must also be researched in more detail.

Acknowledgments

We are grateful to Denise Anthony, David Chadwick, Carl Ellison, John Erickson, and Ed Feustel for their helpful comments.

This research has been supported in part by the Mellon Foundation, NSF (CCR-0209144), AT&T/Internet2 and the Office for Domestic Preparedness, Department of Homeland Security (2000-DT-CX-K001). This paper does not necessarily reflect the views of the sponsors.

A preliminary report on this material appeared as the first author's master's thesis [24].

References

- [1] D. Anthony. Department of Sociology, Dartmouth College. Personal communication., 2003.
- [2] M. Blaze. The Role of Trust Management in Distributed Security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*. Springer-Verlag LNCS 1603, 1999.
- [3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System - Version 2. RFC 2704.
- [4] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.
- [5] C.Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylnen. SPKI Certificate Theory. RFC 2693.
- [6] D. Chadwick. The PERMIS X.509 Role Based Privilege Management Infrastructure. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, 2002.

- [7] D. Clark, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.
- [8] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler, and J. Reagle. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C Recommendation. <http://www.w3.org/TR/2002/REC-P3P-20020416/>, 2002.
- [9] Cryptography and Information Security Group Research Project: SDSI. <http://theory.lcs.mit.edu/~cis/sdsi.html>.
- [10] N. Dulay, E. Lupu, M. Sloman, , and N. Damianou. A Policy Deployment Model for the Ponder Language. In *Proceedings of 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.
- [11] C. Ellison. Personal correspondence.
- [12] C. Ellison. The Nature of a Useable PKI. *Computer Networks*, 31:823–830, 1999.
- [13] M. Erdos and S. Cantor. Shibboleth Architecture Draft v05. <http://middleware.internet2.edu/shibboleth/docs/draft-internet2-shibboleth-arch-v05.pdf>, May 2002.
- [14] OASIS eXtensible Access Control Markup Language (XACML). <http://www.oasis-open.org/committees/download.php/1642/>, 2003.
- [15] SAML v1.0, Assertions and Protocol. <http://www.oasis-open.org/committees/download.php/1371/>, 2002.
- [16] R. Hedberg and T. Wiberg. The SPOCP Protocol: SPOCP Project document. <ftp://ftp.su.se/pub/spocp>.
- [17] P. Hensley, M. Metral, U. Shardanand, D. Converse, and M. Myers. Implementation of OPS Over HTTP, Version 1.0. <http://www.w3.org/TR/NOTE-OPS-OverHTTP.html>.
- [18] P. Hensley, M. Metral, U. Shardanand, D. Converse, and M. Myers. Proposal for an Open Profiling Standard, Version 1.0. <http://www.w3.org/TR/NOTE-OPS-FrameWork.html>.
- [19] A. Herzberg, Y. Maas, J. Mihaeli, D. Naor, and Y. Ravid. Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.
- [20] L. Kagal, T. Finin, and Y. Peng. A Framework for Distributed Trust Management. In *Proceedings on IJCAI - 01 Workshop on Autonomy, Delegation and Control*, 2001.
- [21] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World (2nd Edition)*. Prentice Hall, 2002.
- [22] M. Lorch, S. Proctor, R. Lepro, Kafura D, and S. Shah. First Experiences Using XACML for Access Control in Distributed Systems, October 2003.
- [23] P. Madsen and C. Adams. Privacy and XML, Part 2. <http://www.xml.com/lpt/a/2002/05/01/privacy.html>.
- [24] S. Nazareth. SPADE: SPKI/SDSI for Attribute Release Policies in a Distributed Environment. Master’s thesis, Department of Computer Science, Dartmouth College, May 2003. <http://www.cs.dartmouth.edu/~pkilab/theses/sidharth.pdf>.
- [25] R. Rivest. S-expressions (draft-rivest-sexp-00.txt). <http://theory.lcs.mit.edu/~rivest/sexp.html>, May 1997.
- [26] R. Rivest and B. Lampson. SDSI - A Simple Distributed Security Infrastructure. <http://theory.lcs.mit.edu/~rivest/sdsi10.html>, April 1996.
- [27] Enterprise Privacy Authorization Language. <http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/>.
- [28] SideCar. <http://www.cit.cornell.edu/kerberos/sidecar.html>.