End-to-end authorization

Jon Howell*
Consystant Design Technologies

David Kotz
Department of Computer Science
Dartmouth College

Abstract

Many boundaries impede the flow of authorization information, forcing applications that span those boundaries into hop-by-hop approaches to authorization. We present a unified approach to authorization. Our approach allows applications that span administrative, network, abstraction, and protocol boundaries to understand the end-to-end authority that justifies any given request. The resulting distributed systems are more secure and easier to audit.

We describe boundaries that can interfere with end-to-end authorization, and outline our unified approach. We describe the system we built and the applications we adapted to use our unified authorization system, and measure its costs. We conclude that our system is a practical approach to the desirable goal of end-to-end authorization.

1 Introduction

As systems grow more complex, they are often grown by affixing one system to another using some form of gateway to bridge boundaries between the systems. The boundaries can take several forms; we discuss four in this paper.

When we assemble systems in this way, frequently the authorization information available at the client system cannot be translated to the terms of authorization at the server system. As a result, the gateway often ends up making access-control decisions on behalf of the server system, and the server system is ignorant of any authorization information beyond a blind trust in the gateway. Our end-to-end authorization system remedies this situation.

2 Goals

Saltzer et al. describe a general principle for computer engineering: implement end-to-end semantics to achieve correctness, and only implement hop-by-hop semantics to boost the performance of the end-to-end implementation [19]. Voydock and Kent argue for end-to-end security measures when the hops are between network routers [24]. The same principle applies to authorization semantics when the hops are between gateways that span administrative boundaries, network scales, levels of abstraction, or protocol boundaries. End-to-end authorization makes systems more secure by reducing the number of programs that make access-control decisions, by giving those programs that do control access more thorough information, and by providing more useful audit trails. In this section, we illustrate four kinds of boundaries in distributed systems that impede the flow of authorization information from one end of a system to another. We discuss how, by giving clients and servers the ability to form and verify proofs, our unified system can support end-to-end authorization through the gateways that span these boundaries.

2.1 Spanning administrative domains

Administrative boundaries frequently interfere with end-to-end authorization. The conventional approach to authorization involves authenticating the client to a local, administratively-defined user identity, then authorizing that user according to an access-control list (ACL) for the resource. When resources are to be shared across administrative boundaries, this scheme fails because the server has no local knowledge of the recipient's identity.

Typical solutions to this problem involve authenticating the remote user in the local domain, either by having the local administrator create a new account, or by the resource owner sharing her password. Another approach is to install a gateway that accesses the resource with the local user's privilege but on behalf of the remote user. With the gateway the owner achieves her goal of sharing, but obscures the identity and authority of the actual client from the service that supplies the underlying resource.

Another way a user might share resources across administrative boundaries is by *delegating*[1] her authority with *restriction*. In the example, Alice may

*Supported by a research grant from the USENIX Association. This work represents part of Jon's Ph.D work at Dartmouth College. Jon can be reached at `jonh@alum.dartmouth.org`.

[1] We call *delegation* what Abadi et al. call *handoff*.

authorize Bob to perform some restricted set of actions on certain resources. Authority information flows across the administrative boundary: the delegation provides the resource server with sufficient information to reason about the client regardless of her membership in the local administrative domain. Indeed, the authorization mechanism has *no* inherent notion of administrative domain.

2.2 Spanning network scales

A second boundary that interferes with end-to-end authorization is network scale. Network scale affects an application's choice of hop-by-hop authorization protocol. For example, a strong encryption protocol is appropriate when crossing a wide-area network. Inside a firewall where routers are locally administered, some installations may base authority decisions on IP source addresses. On a local machine, we can often trust the OS kernel to correctly identify the participants in an interprocess communication.

Our unified approach separates policy from mechanism, creating two benefits. First, applications reason about policy using a toolkit with a narrow interface. The toolkit can transparently support multiple access mechanisms, and simply enable those that policy allows. Second, when an application does not support a desired mechanism, we can build a gateway that forwards requests from another mechanism while still passing end-to-end authorization information in a form the server understands and verifies. Ultimately, the high-level security analysis of a program is independent of mechanism, and reflects end-to-end trust relationships.

2.3 Spanning levels of abstraction

Another use for gateway programs is to introduce another level of abstraction over that provided by a lower-level resource server. A file system takes disk blocks and makes files; a calendar takes relational database records and makes events; a source-code repository takes files and makes configuration branches. Typically, an abstracting gateway controls the lower-level resource completely and exclusively, so that the gateway makes all access-control decisions. With end-to-end authorization, one can instead allow multiple mutually untrusting gateways to share a single lower-level resource.

For example, a system administrator might control the disk-block allocator. To grant Alice access to a specific file X, the sysadmin may allow Alice to speak for the file system regarding X, and allow the conjunction of Alice and the file system quoting Alice to speak for the disk blocks. In this configuration, the file system cannot access the lower-level disk block resource without Alice's agreement (due to the conjunction), and Alice cannot meddle with arbitrary disk blocks without the file system agreeing that the requests are appropriate. The system helps us adhere to the principal of least privilege by encoding partial trust in the user and in the file system program. Furthermore, auditing any request for disk blocks provides end-to-end information indicating the involvement of both Alice and the file system program.

2.4 Spanning protocols

Commonly a gateway is installed between two systems simply to translate requests from one wire protocol to another. Like any gateway, these gateways often impede the flow of authorization information from client to server.

In our system, authorization information is encoded in a data structure that has both robust and efficient wire transfer encodings [18]. Thus the unified system is easily adapted for transfer over a variety of existing protocols. In this paper, we describe its implementation over HTTP and over Java Remote Method Invocation (RMI). Adapting more protocols, such as NFS and SMTP, to support unified authorization will result in wider applicability of end-to-end authorization.

The four boundaries described above turn up in real systems that accrete from smaller subsystems. Gateway software installed at each boundary maps requests from clients on one side of the boundary to requests for services on the other side. The system described herein allows us, at each boundary, to preserve the flow of authorization information alongside the flow of requests. By allowing gateways to defer authorization decisions to the final resource server when appropriate, and ensuring that resource servers have a full explanation for the authority of the requests they service, we provide applications with end-to-end authorization.

3 Unified authorization

Above, we motivate the use of a unified system to support end-to-end authorization, and allude to some of its features. In this section, we give an overview of the system we built, part of a project called Snowflake that facilitates naming and sharing across administrative boundaries.

The main idea behind our end-to-end authorization is a compact logic of authority. The logic is founded in a possible-worlds semantics that provides intuition and guidance about possible extensions. Due to its length, the detailed semantics appears in a companion paper [11].

Logical assumptions represent statements that a principal believes based on some verification (outside the logic), such as the result of a digital signature verification. Principals combine assumptions and logical theorems to produce inherently auditable proofs of authority. Such proofs are not bearer capabilities but simply verifiable facts: while they prove that a given principal has authority, knowledge of the proof by an adversary does not bestow authority on the adversary. The primary form of statement is $B \stackrel{T}{\Rightarrow} A$, read "Bob speaks for Alice regarding the statements in set T." The statement means that Alice agrees with Bob about any statement in T that Bob might make; the *speaks for* captures delegation, and the *regarding* captures restriction.

The logic stems from the Logic of Authentication due to Abadi, Burrows, Lampson, Plotkin, and Wobber [1, 13, 25]; as in their logic, ours can encode conjunction (multiple parties exercising joint authority) and quoting (one party claiming to speak on behalf of another). The logic is backed by a semantics that not only provides unambiguous meaning for every logical statement, but tells us how the system may and may not be safely extended.

The formalism suggests a natural implementation language that fits nicely with the Simple Public Key Infrastructure (SPKI) [9]. Our system generalizes SPKI by allowing other forms of principal, so that the same framework can be used for authorization on a single host using a trusted kernel, authorization within an administrative domain using a secret-key protocol, or authorization in the wide area using a public key protocol. We extended the SPKI framework rather than create our own to simplify potential interoperation with SPKI, to exploit SPKI's unambiguous S-expression representation, and to build on existing implementations of SPKI in C and Java.

We present the implementation in three sections: the infrastructure of the system, the channels of communication we have supported, and some applications that exploit the authorization model. The applications culminate in a configuration that bridges each of the four boundaries described above.

Principals, statements and proofs are the language of our system. Section 4 describes each, and discusses our implementation. It also describes the Prover, a tool used by clients to generate proofs. Requests to be authorized are delivered over various kinds of channels, from fast local channels connected by a trusted kernel, to cryptographically-protected network connections. We discuss our implementation of authorization over channels in Section 5. In Section 6, we describe the applications and services we have built that participate in and interoperate using the unified authorization system. We measure and analyze the costs of our approach in Section 7. Section 8 discusses related work, and we summarize in Section 9.

4 Infrastructure

The basic elements of the system are statements and principals. A statement is any assertion, such as "it would be good to read file X," or "Bob speaks for Alice," or "Charlie says Alice speaks for Charlie." A principal is any entity that can make a statement. Examples include the binary representation of a statement itself (that says only what it says), a cryptographic key (that says any message signed by the key), a secure channel (that says any message emanating from the channel), a program (that says its output), and a terminal (that says whatever the user types on it).

A proof of authority, like a proof of a mathematical theorem, is simply a collection of statements that together convince the reader of the veracity of the conclusion statement. Of course, in an authorization system, a proof is read by a program, not by a mathematician.

4.1 Statements

Snowflake's implementation of sharing begins with the Java implementation of SPKI by Morcos [14]. It is a useful starting point because not only do we wish to preserve features of SPKI, but SPKI includes a precise and easily extensible specification of the representation of various abstractions. Furthermore, starting with a SPKI implementation offers an easier path to SPKI interoperability.

The restriction imposed on a delegation is specified using *authorization tags* from SPKI. Authorization tags concisely represent infinitely refinable sets, which makes them an attractive format for user-definable restrictions. We replaced Morcos' minimal implementation of authorization tags with a complete one that performs arbitrary intersection operations [12, Chapter 6].Our semantics paper explains how SPKI's revocation mechanisms (lists and one-time revalidations) can be expressed as statements in our logic [11].

4.2 Principals

SPKI makes a distinction between principals and "subjects," entities that can speak for others but can utter no statements directly, such as threshold (conjunct) principals. Our formalism does not make that distinction. It also supports new compound principals, such as the quoting principal of Lampson et al. Therefore, we extended Morcos' Principal class to support SPKI threshold (conjunction) principals and Lampson's quoting principals. When a service reads a request from a communications channel, it associates the request with an appropriate principal object that represents the channel; this principal is the one that "says" the request. Because the channel itself is a principal, it may claim to quote some other principal; that assertion is noted by associating the channel with a `Quoting` principal object. The object's `quoter` field is the channel itself, and its `quotee` field is the (possibly compound) principal the channel claims to quote.

4.3 Proofs

We implemented a `Proof` class that represents a structured proof consisting of axioms and theorems of the logic and basic facts (delegations by principals). An instance of `Proof` describes the statement that it proves and can verify itself upon request. While `Proof` objects may be received from untrusted parties, their methods are loaded from a local code base, so that the results of verification are trustworthy. Servers receive from clients instances of the `Proof` class that show the client's authority to request service. Conversely, a server may send a `Proof` to a client to establish its authenticity, that is, to prove its authority to identify itself by some name or to provide some service the client expects.

Proofs can be transmitted as SPKI-style S-expressions or directly transferred between JVMs using Java serialization. No precision is lost in the latter case, since the basic internal structure of every proof component is a Java object corresponding to an S-expression.

SPKI's *sequence* objects also represent proofs of authority. SPKI sequences are poorly defined, but they are linear programs apparently intended to run on a simple verifier implemented as a stack machine. When certificates and opcodes are presented to the machine in the correct order, the machine arrives at the desired conclusion [8].

Transmitting proofs in a structured form rather than as SPKI sequences is attractive for three reasons. First, the structured proofs clearly exhibit their own meaning; to quote Abadi and Needham, "every message should say what it means" [2]. Second, the structured proof components map one-to-one to implementation objects that verify each component. The SPKI sequence verifier, in contrast, requires an external mapping to show that the state machine corresponds to correct application of the formal logic. Third, it is simple to extract lemmas (subproofs) from structured proofs, allowing the prover to digest proofs into reusable components (Section 4.4).

The logic encodes expiration times as part of the restriction of a delegation, so that each proof need be verified only once. The step of matching a request to a proof automatically disregards expired conclusions, since a current request cannot match a conclusion with a restriction that it was valid only in the past. Figure 1 illustrates a proof. Since the structure of the proof is preserved, if the topmost statement should expire (perhaps because it depends on the short-lived statement $H_D \Rightarrow K_S$), the still-useful proof of $K_S \Rightarrow K_C \cdot N$ may be extracted and reused in future proofs.

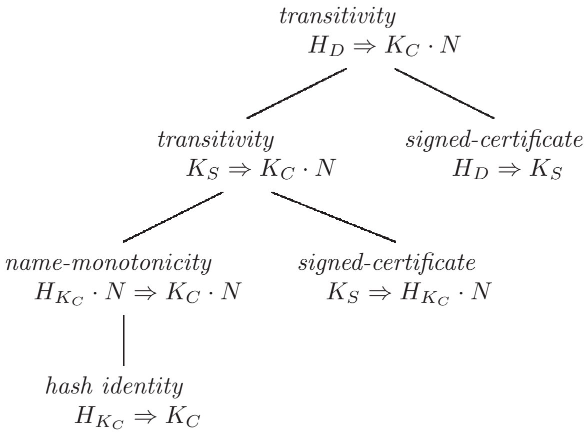

Figure 1: *A structured proof. This proof shows that document D is the object client C associates with the name N. H_{K_C} is a hash of the client's key K_C, H_D a hash of the document, and K_S the server's key.*

4.4 The prover

A `Prover` object helps Snowflake applications collect and create proofs. It has three tasks: it collects delegations, caches proofs, and constructs new delegations.

A user's application collects delegations from other users. Gateways collect delegations directly from client applications. Both sorts of applications use a `Prover` to maintain their collected delegations in a graph where nodes represent principals and edges represent a proof of authority from one principal to the next (see Figure 2). The `Prover` traverses the graph breadth first to find proofs of delegation required by the application. For example, if the `Prover` must prove that a channel K_{CH}

speaks for a server S, it works backwards from the node S to find the proof that $A \stackrel{V \cap X}{\Rightarrow} S$. A is *final*, meaning that the `Prover` can make statements as A; therefore, `Prover` simply issues a delegation $K_{CH} \Rightarrow A$ to complete the proof.

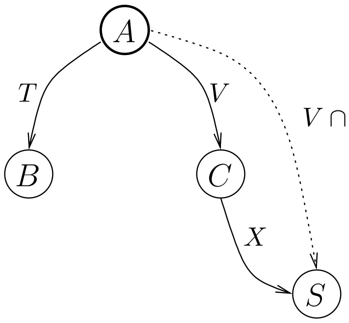

Figure 2: *A look inside Alice's `Prover`. Each node represents a principal, and each edge a proof. For example, the edge from A to B represents the proof consisting of the single delegation $A \stackrel{T}{\Rightarrow} B$. The node A is distinguished because it is* final*: it represents a principal that the `Prover` can cause to say things.*

When the `Prover` receives a delegation that is actually a proof involving several steps, the `Prover` "digests" the proof into its component parts for storage in the graph. Whenever it receives or computes a derived proof composed of smaller components, the `Prover` adds a shortcut edge (dotted line in Figure 2) to the graph to represent the proof. These shortcuts form a cache that eliminates most deep traversals of the graph.

When an application controls one or more principals (e.g., by holding the corresponding private key or capability), its `Prover` can store a closure (an object that knows the private key or how to exercise the capability) in its graph to represent the controlled principal. When desired, the `Prover` can not only find existing proofs, but complete new proofs by finding an existing chain of delegations from the controlled principal to the required issuer, then using the closure to delegate to the required subject restricted authority over the controlled principal.

Our simple Prover is incomplete, but it is suitable for most authorization tasks applications face. Abadi et al. note that solutions to the general access-control problem in the presence of both conjunction and quoting require exponential time [1, p.726]. Elien gives a polynomial-time algorithm for discovering proofs in a graph with only SPKI certificates (no quoting principals) [7]. In the common case, we expect applications to collect authorization information in the course of resolving names, so that proofs are built incrementally with graph traversals of constant depth.

5 Channels

With the infrastructure above in place, applications and services have the tools they need to generate, propagate, and analyze authority from the source of a request to its final resource server. The authorization information must be propagated from one program to the next through channels.

When a client makes a request of a server, the server needs some mechanism to ensure that the client really uttered the request. We implemented three such mechanisms: a secure network channel, a local channel vouched for by a trusted authority in the same (virtual) machine, and a signed request. We describe each and discuss how they are represented as principals in our unified system.

5.1 Secure channels

To implement a secure channel, we built a Java implementation of the `ssh` protocol that can interoperate with the Unix `sshd` service [26]. Then we built Java `ServerSocket` and `Socket` classes based on `ssh` that provide a secure connection. Either end of the connection can query its socket to discover the public key associated with the opposite end.[2]

We plugged our `ssh` sockets into RMI using socket factories. Ssh ensures that the channel is secure between some pair of public keys. To make that guarantee useful, we embody the channel as a principal. Consider the channel in Figure 3. To establish the channel, the server (principal P_S) uses public key K_1 and the client (P_C) key K_2 in the key exchange, and together they establish secret key K_{CH} as the symmetric session key.

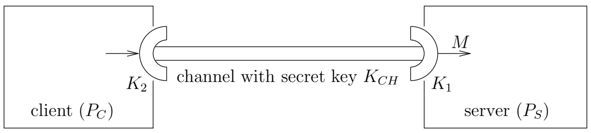

Figure 3: *Treating a channel as a principal*

Suppose a message M emerges from the channel at the server. In the language of the formalism,

[2]Why did we build an `ssh` implementation? Some have suggested that we use SSL over RMI, which is apparently now fairly practical. When we began this work, however, RMI did not have easily pluggable socket factories, and even once it did, the only open-source SSL implementation we could find did not operate well under RMI.

the ssh implementation promises that $M \Rightarrow K_{CH}$. The initial key exchange convinced the server that $K_{CH} \Rightarrow K_2$, and the client may explicitly establish that $K_2 \Rightarrow P_C$. Because $M \Rightarrow K_{CH} \Rightarrow K_2 \Rightarrow P_C$, the server concludes that $M \Rightarrow P_C$, that is, the message says what the client is thinking.

5.1.1 How channels work

Figure 4 illustrates our RMI/`ssh` channel in action. Initially, the server creates an instance of an RMI remote object ⓐ, defines the key K_S that controls it, and associates the object with an `SSHContext` that manages any incoming messages for the object ⓑ. The `SSHContext` is associated with the RMI listener socket ⓒ that will receive incoming requests for the object, and defines the public key (K_1) that will participate in `ssh` session establishment.

The client retrieves a stub ⓓ for the remote object from a name service it trusts. To exercise its authority on the object, the client first establishes its identity in thread scope. In a `try ... finally` block, it establishes its own `SSHContext` ⓔ and a `Prover` ⓕ that holds its private key K_C. Any method called in the run-time scope of the `try` block will inherit the established authority, but the authority will be canceled when control exits the block.

Then the client invokes a method m on the remote stub. The remote stub has been mechanically rewritten to wrap its remote invocations with calls to the `invoker` helper method ⓖ. The invoker method makes the usual RMI remote call through the remote reference ⓗ, and the reference creates an `ssh` socket ⓘ using the `SSHSocketFactory` specified in the stub. The `ssh` channel is established ⓙ, and each context learns the public key associated with the opposite end (K_1, K_2). The method call passes through the channel to the skeleton object on the server ⓚ, which forwards the call to the implementation object.

The programmer has prepended to each remote method implementation a call to the no-argument method `checkAuth()` ⓛ. This routine discovers from the local `SSHContext` the key K_2 associated with the channel that the request arrived on, and concludes K_2 **says** `m`. The server object was associated at creation with the key K_S, however, and `checkAuth()` does not know that K_2 speaks for K_S, so it throws an `SfNeedAuthorizationException`.

RMI passes the exception back through the channel, where the client's invoker method catches it. The invoker inspects the exception to discover the issuer K_S it must speak for and the minimum restriction set regarding which it must speak for that issuer.[3] The invoker queries the Prover ⓕ for a proof of the required authority; since the prover controls the client's private key K_C, it can construct a statement to delegate authority from K_C to K_2. The exception carries a reference to a special remote `proofRecipient` object; the invoker calls a method on it to pass ⓜ the proof to the server. The `proofRecipient` object ⓝ stores the proof at the server, and returns to the client.

The invoker again sends the original invocation m through the remote reference, and the request travels the same path to `checkAuth` on the server. This time, the proof that $K_2 \overset{T}{\Rightarrow} K_S$ (via K_C) is available, `checkAuth()` returns without exception, and the remote object's implementation method runs to completion. Future calls encounter no exception as long as the proof at the server remains valid, and are only slowed by the layer of encryption protecting the integrity of the `ssh` channel.

The client programmer need only establish the client's authority at the top of a code block; inside that scope, the `Prover` and the invoker together handle the nitty-gritty of proof generation and authorization. In the idiom we adopt, the server programmer defines the object server key K_S and the mapping from method invocation to restriction set (T) for a server object, then prefixes each Remote method with calls to a generic `checkAuth()` that uses those definitions. We chose this approach because it would be simple to automate the injection of `checkAuth()` calls to insure that no Remote interface is left unprotected.

5.2 Local channels

Setting up a secure network channel is an expensive operation because it involves public-key operations to exchange keys. If a server trusts its host machine enough to run its software, it may as well trust the host to identify parties connected to local IPC channels. Within our Java environment, we treat the JVM and a few system classes as the trusted host, and bypass encryption when connecting to a server in the same JVM.

[3] In this example, the minimum restriction set $T = \{m\}$ contains the singleton request (method invocation) made by the invoker. When some more-sophisticated mapping is involved, where the server's minimum restriction set may reveal sensitive structure of the service, the server may reveal the set only incrementally. For example, its first challenge may tell the client how to prove authority to learn the "real" restriction set. The situation is analogous to `ls -l foo/bar` in Unix: it reveals the authority required by a client to access a resource `bar`, but only after the client has shown its authority to learn that information by logging in with a UID that has permission to read the directory `foo`.

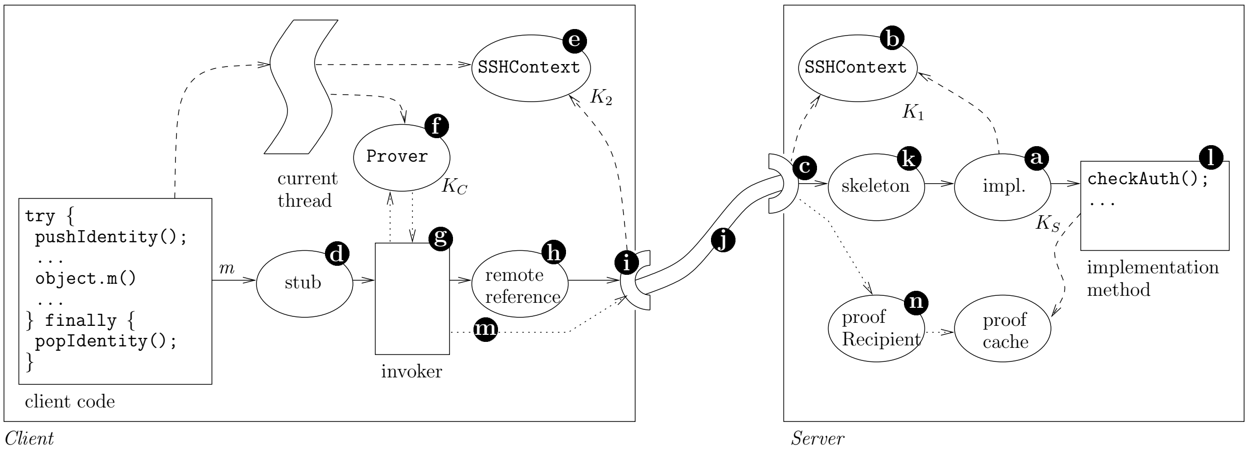

Figure 4: *How our* `ssh` *RMI channel is integrated with Snowflake's authorization service. Dashed arrows represent object references. Solid arrows represent the critical remote call path, and dotted arrows represent the longer path taken when the server requires fresh proof of the client's authority.*

In the local case, the `ssh` channel is replaced with a Java "IPC" pipe implemented without any operating system IPC services, and the public keys corresponding to the channel endpoints (K_1 and K_2) are swapped directly. Because it was involved in constructing the key pairs and the keys are stored in immutable objects, the trusted system class knows whether a client holds the private key corresponding to a given public key. Hence when a client is colocated in the same JVM with the server, there is no encryption or system-call overhead associated with the channel, only RMI serialization costs.

5.3 Signed requests

Not all applications can assume that our `ssh`-enhanced version of RMI is available as an RPC mechanism. Indeed, the most visible RPC mechanism on the Internet is HTTP. To facilitate applications that use HTTP, we created a Snowflake version of the HTTP authorization protocol.

HTTP defines a simple, extensible challenge-response authorization mechanism [10]. The client sends an HTTP request to the server. The server replies with a "401 Unauthorized" response, including a `WWW-Authenticate` header describing the method and other parameters of the required authorization. The client resends its request, this time including an `Authorization` header. If the `Authorization` satisfies the server's challenge, the server honors the request and replies with the return value of the operation. Otherwise, the server returns a "403 Forbidden" response to indicate the authorization failure.

HTTP defines two standard authorization methods. In Basic Authentication, the client's `Authorization` header includes a password in cleartext. In Digest Authentication, the server's `WWW-Authenticate` challenge includes a nonce, and the client's `Authorization` header consists of a secure hash of the nonce and the user's password. Both methods *authenticate* the client as the holder of a secret password, and leave *authorization* to an ACL at the server.

In our new method, called Snowflake Authorization, the parameters embedded in the server's `WWW-Authenticate` challenge are the issuer that the client needs to speak for and the minimum restriction set that the delegation must allow. The `Authorization` header in the client's second request simply includes a Snowflake proof that the request speaks for the required issuer regarding the specified restriction set. The subject of the proof is a hash of the request, less the `Authorization` header. Figure 5 shows an example.

5.3.1 Signed request optimization

The signed request protocol described above is rather slow, since it incurs a public-key signature for every request. We implemented a more efficient protocol that amortizes the public-key operation by having the server send an encrypted, secret message authentication code (MAC) to the client. The client then authorizes messages by sending a hash of ⟨ message, MAC ⟩. The protocol is represented in the end-to-end authorization chain by representing the MAC as a principal.

SSL channels offer an alternative approach to amortizing the initial public-key operation, with dif-

```
HTTP/1.0 401 UNAUTHORIZED
Content-Type:  text/html
MIME-Version:  1.0
Server:  MortBay-Jetty-2.3.3
Date:  Sat, 08 Apr 2000 15:18:47 GMT
WWW-Authenticate:  SnowflakeProof
  Authorize-Client
Sf-ServiceIssuer:  (hash md5
  |ehtQYd4EpQXOa/ON6Smesg==|)
Sf-MinimumTag:  (tag
  (web (method GET)
    (service |Sm9uJ3MgUHJvdGVjdGVpY2U=|)
    (resourcePath "")))
Connection:  close
```

Figure 5: *An HTTP authorization challenge message from a Snowflake server. It indicates the method, the required resource issuer, and the minimum restriction of a delegation that must be proven.*

ferent security and performance trade-offs.

5.3.2 Authorization vs. authentication

The SPKI group argues that authorizing a request without authentication as an intermediate step reduces indirection and hence removes opportunities for attack [9]. When authentication is desired, one can use the logic to demand it. For example, one may delegate a resource to "authentication server's Alice," requiring Alice to authenticate herself to the server to invoke her authority over the resource. Alternatively, one can resolve the secure bindings that map keys to names after the fact to discover whose authority was invoked. How meaningful an authentication is depends on one's philosophy about delegation control [11].

5.3.3 Server authorization

Often a client also wants to verify that it is communicating with the "right" server. The notion of "right" can be as simple as the server speaking for the client's idea of a well-known name like `www.dartmouth.edu`, but in general the real question is still one of authorization: Does this server have the right to claim authority about Dartmouth's course list? Does that server have authority to receive my e-mail?

We addressed a limited version of this problem with a second HTTP extension that enables a server to show the authenticity of a document using the authorization system. The server includes with document headers a proof that the hash of the document speaks for the server. The client completes the proof chain and determines whether the authentication is satisfactory.

5.3.4 Server implementation

We implement the server side of the signed-requests protocol as an abstract Java Servlet `ProtectedServlet` [15]. Concrete implementations extend `ProtectedServlet` with a method that maps a request to an issuer that controls the requested resource and to the minimum restriction set required to authorize the request. The concrete class also supplies the service implementation that maps a request to a response. When each request arrives, the `ProtectedServlet` ensures that appropriate authorization has been supplied, and if not, constructs and returns the "401 Unauthorized" response to the client.

Notice that the server identifies only a single principal that controls the resource, not an ACL. An ACL is a specific group of users authorized to access a resource; in our system, the client is responsible to know and exploit its group memberships as represented in delegations [11].

5.3.5 Client implementation

We realize our client as an HTTP proxy that enhances a browser with Snowflake authorization and server document-authentication services. Like any proxy, it forwards each HTTP request from the browser to a server. When a reply is "401 Unauthorized" and requires Snowflake authorization, the proxy uses its `Prover` to find a suitable proof, rewrites the request with an `Authorization` header, and retries the request.

The proxy provides an HTML user interface to its services at a virtual URL `http://security.localhost/`. Through this interface, the user can create a new private key pair, import principal identities and delegations, and delegate his authority to others. To delegate his authority, the user views a history of recently-visited pages, clicks the "delegate" link next to the page he wishes to share, and selects the recipient from a list of principals. The proxy generates an HTML snippet for the user to deliver to the recipient. A link inside the snippet names the destination page and carries both the delegation from the user as well as the proof the user needed to access the page. When the recipient follows the link, his own proxy imports the authorization information and redirects his browser to the named page.

6 Applications

We built three applications to demonstrate the Snowflake architecture for sharing.

6.1 Protected web server

The first application is simply a protected web file server that uses Snowflake's sharing architecture. One user establishes control over the file server by specifying the hash of his public key when starting up the server; he may delegate to others permission to read subtrees or individual files from the server using the mechanisms described above.

6.2 Protected database

The second application attaches Snowflake security to a relational email database. The original database server accepts insert, update, and select requests as RMI invocations on a Remote `Database` object, and returns the results of the query as serialized objects from the database. Adapting the application to Snowflake required only minimal changes. We modified the database instance constructor to use a `SshSocketFactory` so that all connections to the object use our `ssh` secure channels. Then, we prepended each implementation of a method in the remote interface with a call to the `checkAuth()` method. The database clients required only a modification to their initialization code to install an `SSHContext` and a `Prover`.

6.3 Quoting protocol gateway

The third application is a protocol gateway that provides an HTML over HTTP front-end to the email database. A database can be configured to allow certain principals access to certain data records. In the course of serving multiple users, the gateway can simultaneously access both Alice and Bob's email records. It is important that the gateway not misuse its authority and accidentally allow Bob to read Alice's email. The gateway programmer could try to prevent this mistake by checking access-control restrictions itself, but this approach duplicates the access control checks in the database, and increases the opportunity for error.

A better approach is to use quoting. The gateway's authority to access Alice's email in the database depends on the gateway intentionally quoting Alice in its requests. Therefore, as long as the gateway correctly quotes its clients in its requests on the database server, the correct access-control decision is made by the server.

A transaction begins when the client (C) sends an unauthorized request (R) to the gateway (G). The gateway queries the client for the identity the client wishes to use, and a delegation that the gateway speaks for the client to perform the task. The gateway attempts to access the database server (S), but the RMI authorization fails because the gateway has no authority. The gateway sees an exception that indicates the required issuer S and restriction set (T). The gateway generates a "401 Unauthorized" Snowflake Authorization HTTP response, and in that response indicates it needs a proof that $G|? \overset{T}{\Rightarrow} S$. By $G|?$ the gateway means it needs a proof of authority that the gateway quoting the client speaks for the database. The client knows to substitute its identity for the "pseudo-principal" ?; this shortcut saves a round-trip from the gateway to the client to discover the client's identity.

The client proxy now knows it needs to delegate its authority over the server to the principal "gateway quoting client," $G|C$. The client proxy generates the proof and submits it to the gateway along with a signed copy of its original request (showing $R \Rightarrow C$). The gateway digests the new proof and forwards the request to the database server. This time, the automatic RMI authorization protocol of Section 5.1.1 finds the proof in the gateway's `Prover`, and the database fulfills the request. The gateway builds an HTML interface from the database results for presentation to the user. Subsequent requests are accepted without so much fanfare, since the database server holds the appropriate proof of delegation.

The quoting gateway is a motivating application because it spans each of the four boundaries discussed in Section 2. Our gateway operates identically whether the client and the server are in the same administrative domain or different ones. It can be colocated with the server, in which case its RMI transactions automatically avoid encryption overhead by using the local channels of Section 5.2. The gateway constructs a view of an e-mail message from several rows and tables of a relational database, and so introduces a level of abstraction above the server resource. Finally, the gateway spans protocols by connecting an HTTP-speaking web browser with an RMI-speaking database server. Despite each of these boundaries, the gateway preserves the entire chain of authority that connects the client to the final server, enabling the server to make a fully-informed access-control decision.

6.3.1 Correctness and trust

The client trusts the gateway not to abuse the client's authority, and for some applications, the client may even trust the gateway to tell it how much authority the gateway needs to do its job. To establish that trust, the a client might first challenge the gateway to authenticate itself. If a gateway has received delegated authority from multiple clients (Alice and Bob), it must ensure that when it fulfills Bob's request it does not accidentally invoke Alice's authority. Where a conventional gateway would actually make access-control decisions to determine what Bob is allowed to do, our gateway only need be careful to correctly quote each client. It is therefore easier to verify that a quoting gateway is correct with respect to authorization.

In our system the notion of TCB is parameterized by the resource being protected. For example, the client software and hardware are part of the TCB for any resources the client is authorized to manipulate; when the client delegates a subset of those resources to the gateway, the gateway software and hardware become part of the TCB for that subset of resources. Although quoting helps us write the gateway application with greater confidence in its correctness, we cannot escape the fact that a compromised gateway still compromises the resources delegated to the gateway. Because the gateway is involved in the transfer of authority, authorization is not end-to-end in the pure sense of abstracting away intermediate steps. It is end-to-end, however, in the sense that authorization information now passes all the way from client to server, and the proof of authority verified by the server even includes evidence of the gateway principal's involvement.

7 Measurement

To better understand the costs of the Snowflake authorization model, and how they compare to costs of related systems, we timed the performance of our Snowflake-enhanced RMI implementation and our Snowflake-enhanced HTTP implementation. For comparison, we also timed standard RMI and standard HTTP servers with and without SSL support.

7.1 Experimental method

The values reported in this section are the parameters of linear regressions. In *setup cost and bandwidth* experiments, we vary the file length to separate copy cost from connection setup. In *setup and per-request* experiments, we vary the number of connections made after some slow setup operation to determine the amortizable part of the cost.

We made the measurements on 270 MHz Sun Ultra 5 hosts with 128 MB RAM, connected by a shared 10 Mbps Ethernet segment. The hosts run Solaris 2.7, Apache 1.3.12, OpenSSL 0.9.5, a locally-compiled Java JDK 1.2.2 with `green threads`, PureTLS 0.9b1, and Cryptix 3.1.1. We used 1024-bit RSA keys.

We ran each experiment ten times, discarding the first iteration so that caches are warm except where we intentionally measure setup costs. On each run, we repeated an operation 10 to 1000 times, enough to amortize measurement overhead, and noted the total wall-clock time. When the nine runs had coefficient of variation greater than 0.1, we re-ran the experiment. We report values to two significant figures. The figures show values for single-machine experiments, where computation time, the dominant source of overhead, cannot hide under network latency. The raw data, complete tables of computed parameters, standard deviations and R^2 fitness coefficients are available [12, Chapter 12]. We computed 95% confidence intervals on the linear-regression parameters and found them vanishingly small.

7.2 RMI authorization with Snowflake

In this section, we quantify our implementation of Snowflake authorization over Java remote method invocation as described in Section 5.1. Figure 6 summarizes the overhead our prototype adds to RMI. The test operation is a Remote object that returns the contents of a file. Most of the overhead present in Snowflake is due to layering RMI over the `ssh` protocol. The extra work is is the server's `checkAuth()` call, which retrieves the caller's public key, finds a cached proof for that subject, and sees that the proof has already been verified. The data-copy cost is unchanged compared to the `ssh` case.

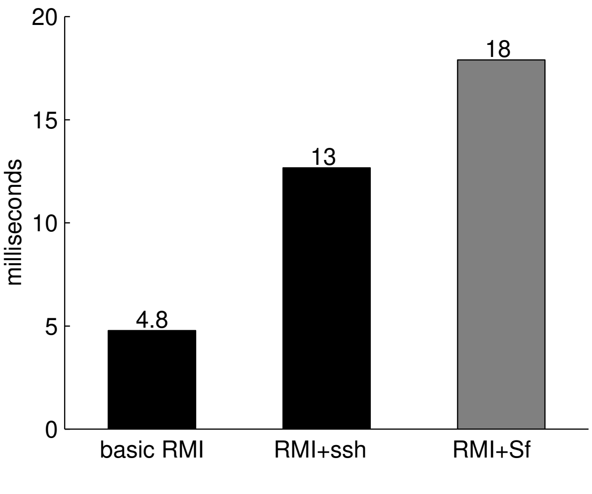

Figure 6: *The cost of introducing Snowflake authorization to RMI. A basic RMI call costs 4.8 ms. Securing the channel with* `ssh` *introduces significant overhead. Mapping the request into Snowflake and verifying the client's authority adds another 5 ms.*

It costs 470 ms to establish a new Snowflake-authorized RMI connection, reflecting the public-key operation the client performs to delegate its authority to the channel. When the client caches the delegation but we make the server forget its copy after each use, we learn that the server spends 190 ms parsing and verifying the proof from the client.

7.3 HTTP authorization with Snowflake

In this section, we quantify our implementation of Snowflake authorization over the HTTP protocol as described in Section 5.3. As shown in Figure 7, the overhead of Java client and server code introduces a five-fold slowdown over an optimized C implementation of HTTP. Most of the rest of Snowflake's slowdown we have accounted for in the slow libraries described in Section 7.4.3.

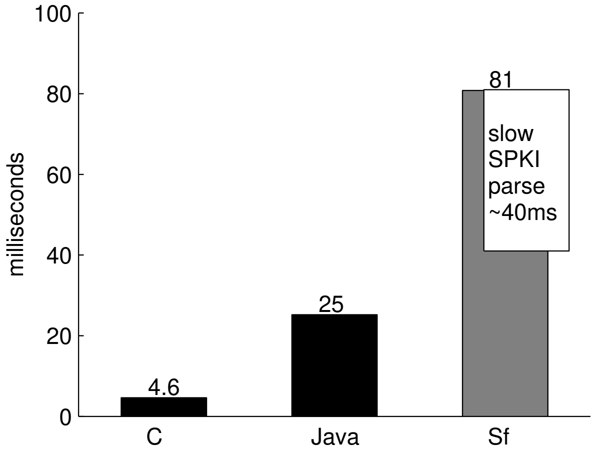

Figure 7: *The cost of introducing Snowflake authorization to HTTP. A trivial C client accessing an Apache server takes 4.6 ms. Replacing the client and server with convenient but inefficient Java packages brings the baseline for HTTP to 25 ms. Most of Snowflake's overhead reflects the use of inefficient SPKI libraries, shown as an inset box.*

The black bars in Figure 8 show our measurements of a Java SSL implementation, and the gray and white bars show the costs of the Snowflake authorization and document authentication protocols described in Section 5.3. Notice that when public-key encryption operations are involved, both protocols require hundreds of milliseconds. When caching connection information (Snofwflake MAC protocol and identical requests versus a SSL request), they require tens of milliseconds. Snowflake's cached requests are a factor of two slower than SSL requests, due in part to differences in the protocol, and in part to the slow libraries discussed in Section 7.4.3.

Minimum cost of HTTP GET (C client and server)	5	5
Java+Jetty overhead for HTTP	20	20
Java SSL overhead	22	
S-expression parsing		~20
SPKI object unmarshalling		~20
Other Snowflake overhead (proof verification, SPKI object marshalling)		17
MAC costs (serialization, MD5 hash)		28
Total	47	110

Table 1: *Breakdown of time spent in MAC authorization protocol. Units are milliseconds.*

7.4 Observations

We hypothesize that the Snowflake authorization model is not prohibitively expensive. In fact, because it can subsume many hop-by-hop authorization models, it allows applications and users to make performance–security tradeoffs freely by selecting alternate hop-by-hop authorization protocols and plugging them into the same authorization framework.

Do our measurements support our hypothesis? Unfortunately, since our implementation is unoptimized and built on top of slow libraries, the numbers do not support our hypothesis unequivocally. By comparing them with baseline experiments, however, we believe we can make a strong case for the hypothesis. In the next two sections, we examine the two parts of our hypothesis. In Section 7.4.3, we argue that an optimized Snowflake promises to be competitive with existing hop-by-hop protocols.

7.4.1 Comparable operations

Snowflake-enhanced protocols are not inherently more expensive than other protocols with similar guarantees. The measurements displayed in Figure 8 indicate that Snowflake performs similar encryption steps as SSL. SSL spends about 400 ms starting up, as does Snowflake. SSL can complete a request over an established channel in about 50 ms. With our MAC optimization, a Snowflake request takes about 110 ms (see Table 1).

Both SSL and Snowflake engage in similar operations. SSL verifies message authenticity with symmetric-key decryption and a CRC; Snowflake does the same with an MD5 hash. Regardless of protocol, the server parses and processes the request and returns the reply. The SSL protocol checksums and encrypts the reply; Snowflake securely hashes

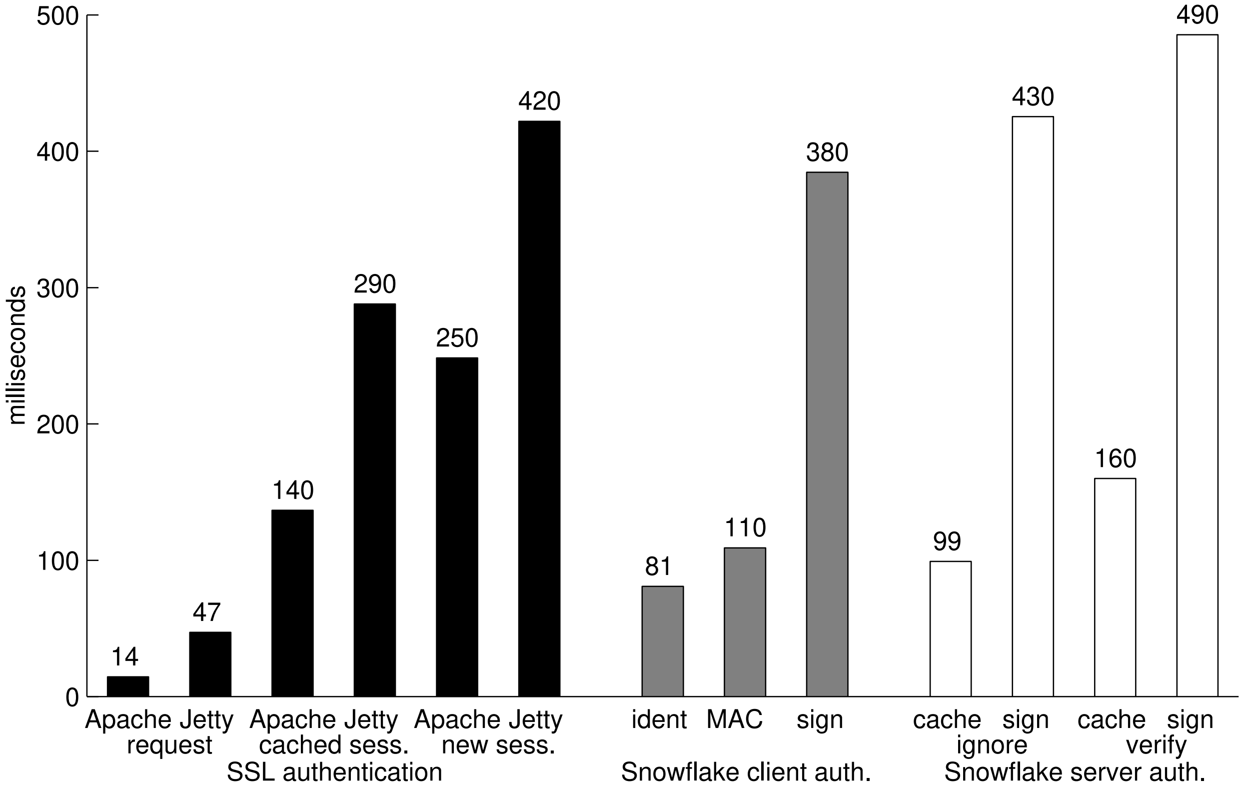

Figure 8: *This graph displays the costs of standard SSL authentication (black bars) versus Snowflake client authorization (gray bars) and server document authentication (white bars).*

the reply document. In both cases, the client uses a corresponding operation to verify the reply. Because the expensive cryptographic operations are comparable, one expects optimized implementations to perform comparably.

The additional sources of overhead in Snowflake are time spent walking the proof graph and memory consumed maintaining cached proofs. Our experiments do not explore that space in depth, but as we hint in Section 5.3.5, proofs are usually constructed incrementally while walking the name graph, an operation driven by the client user or application.

7.4.2 The performance–security tradeoff

By comparing our authorized-request protocol to SSL we somewhat compare apples and oranges, for the protocols make different performance–security tradeoffs. For example, our protocol does not verify the authenticity of the server's reply header; since SSL provides integrity for the entire channel, a Snowflake–SSL protocol could as easily show the authenticity of all messages from the server.

In fact, part of the purpose of our system is to enable such tradeoffs. With Snowflake, one is free to choose an established hop-by-hop protocol or to develop a new one. By stating in our logic the authorization promises the protocol makes, one can integrate the protocol into Snowflake's end-to-end authorization model. Conceivably, new protocols can be dynamically integrated into existing Snowflake-aware applications; in other cases, a protocol-translating gateway can introduce the new protocol to the distributed system without hiding authorization information from the underlying application.

7.4.3 Slow libraries

Our formal measurements and informal tests indicate that a large fraction of Snowflake's cost is needless overhead. Our baseline HTTP measurements indicate that using Java and the convenient Jetty web server incurs substantial overhead (250%). Furthermore, our SSL measurements indicate that the Java encryption library Cryptix imposes a substantial bandwidth overhead.

What surprised us most was the overhead of the

SPKI implementation on which we built Snowflake's objects. In informal tests, parsing a 2 KB S-expression from a string takes around 20 ms, and converting the resulting tree into typed Java objects takes another 20 ms. There is no reason a well-implemented library should spend milliseconds parsing short strings in a simple language; and 40+ ms delays such as these explain much of the difference between Snowflake's warm-connection performance and that of simple HTTP transactions (See Figure 7).

8 Related work

Our work is built primarily on the Logic of Authentication due to Abadi, Burrows, Lampson, Plotkin, and Wobber [1, 13, 25]. The Logic of Authentication introduced the notion of conjunct and quoting principals, and their applicability for modeling practical mechanisms such as channels and multiplexed gateways. We have preserved the generality and formality of the Logic of Authentication while introducing the crucial feature of restricted delegation. The structure of our implementation is similar to that of Taos, but we generally shift the burden of proof to the client so that the collection of access-control information happens in the course of name resolution as described in Section 4.4.

Sollins describes the restricted delegation problem as "cascaded authentication," and proposes as a solution a restricted delegation mechanism called *passports* [21] that provides for authorization of servers. Varadharajan et al. propose a more general mechanism that incorporates both symmetric and asymmetric encryption [23]. Neuman's *proxies* are tokens that express restricted delegation [17]. The PolicyMaker system has a notion of delegations with restrictions specified by arbitrary code [5]. As we mention in Section 3, SPKI has a notion of restricted delegation close to the one we use. Because the only principals in SPKI are public keys, it has high overhead for authorization on a single machine [9].

Sollins' passports, Neuman's proxies, PolicyMaker, and SPKI certificates are mechanisms with only informally-described semantics, and hence have no obvious and safe route to generalization. As we discuss in the companion paper [11], our formal semantics not only provides intuition for restricted delegation and end-to-end authorization, but it can advise us about the safety of possible extensions. Furthermore, it guides us in building a system with a minimal verification engine.

Appel and Felten's higher-order predicate logic is similarly inspired and applicable to SPKI [3]. Because our logic is a first-order propositional modal logic, we can employ a conventional modal-logic semantics [11]. Our logic is also simpler; we factor implementation details out of the logic and leave only the structure of authorization. For example, concepts such as "digital signature" do not appear in our proof rules; instead, we integrate them by mapping a key to a logical principal, and asserting that a digital signature check validates the logical statement K **says** x.

Several single-machine operating systems have been built on the notion of restricted delegation; these are often called *capability-based* systems. Capabilities in KeyKOS, Eros, and Mach are unforgeable because the kernel manages them. A process delegates its authorization by asking the kernel to pass a capability, possibly with restriction, to another process [6, 20, 4]. Amoeba capabilities, in contrast, are secret random numbers, and may be transmitted as raw data [22, 16]. Amoeba must assume that a cluster is a secure network; we consider such a cluster a single administrative domain. Snowflake end-to-end authorization could integrate either sort of capability implementation as a fast, local authorization mechanism.

9 Summary and future work

We make a case for end-to-end authorization. Our proposal is based on a formal logic that models restricted delegations and hence models several existing hop-by-hop protocols. We describe the infrastructure of Snowflake, our implementation, including two hop-by-hop protocols and applications that exploit its end-to-end nature. Our end-to-end approach lets us connect systems with gateways that preserve authorization information, and by integrating multiple hop-by-hop mechanisms, it gives us freedom to easily trade off performance and security.

We would like to cross our work on end-to-end authorization with work on models of secrecy and information flow, to work toward an end-to-end model that can capture notions of who should *know* what. In such an architecture we imagine a gateway that operates with only partial access to the information it translates, passing from server to client encrypted content that it need not view to accomplish its task.

Acknowledgements

Thanks to Hany Farid for providing statistical intuition. We especially thank our reviewers and our shepherd Butler Lampson for all of their helpful comments and guidance.

References

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

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.

[3] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, pages 52–62, Singapore, Nov. 1999. ACM Press.

[4] D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–31, 1992.

[5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[6] A. Bomberger, A. Frantz, W. Frantz, A. Hardy, N. Hardy, C. Landau, and J. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.

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

[8] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. Simple public key certificate. Internet draft `draft-ietf-spki-cert-structure-05.txt` (expired), Mar. 1998.

[9] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI certificate theory, Oct. 1999. Internet RFC 2693.

[10] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication, June 1999. Internet RFC 2617.

[11] J. Howell and D. Kotz. A Formal Semantics for SPKI. In *Proceedings of the 6th European Symposium on Research in Computer Security*, Toulouse, France, Oct. 2000.

[12] J. R. Howell. *Naming and sharing resources across administrative boundaries*. PhD thesis, Department of Computer Science, Dartmouth College, 2000.

[13] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[14] A. Morcos. A Java implementation of Simple Distributed Security Infrastructure. Master's thesis, Massachusetts Institute of Technology, May 1998.

[15] K. Moss. *Java Servlets*. Computing McGraw-Hill, July 1998.

[16] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[17] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 283–291, May 1993.

[18] R. Rivest. S-Expressions. Internet draft `draft-rivest-sexp-00.txt` (expired), May 1997.

[19] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[20] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. *ACM Operating Systems Review*, 33(5):170–185, Dec. 1999.

[21] K. R. Sollins. Cascaded authentication. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 156–163, 1988.

[22] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, Dec. 1990.

[23] V. Varadharajan, P. Allen, and S. Black. An analysis of the proxy problem in distributed systems. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 255–275, 1991.

[24] V. Voydock and S. Kent. Security mechanisms in high-level network protocols. *Computing Surveys*, 15(2):135–171, 1983.

[25] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb. 1994.

[26] T. Ylonen. The SSH (secure shell) remote login protocol. Internet draft `draft-ylonen-ssh-protocol-00.txt` (expired), May 1996.