Authentication in Distributed Systems: Theory and Practice

Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber

Systems Research Center
Digital Equipment Corporation
130 Lytton Ave.
Palo Alto, CA 94301

Abstract

We describe a theory of authentication and a system that implements it. Our theory is based on the notion of principal and a "speaks for" relation between principals. A simple principal either has a name or is a communication channel; a compound principal can express an adopted role or delegation of authority. The theory explains how to reason about a principal's authority by deducing the other principals that it can speak for; authenticating a channel is one important application. We use the theory to explain many existing and proposed mechanisms for security. In particular, we describe the system we have built. It passes principals efficiently as arguments or results of remote procedure calls, and it handles public and shared key encryption, name lookup in a large name space, groups of principals, loading programs, delegation, access control, and revocation.

1. Introduction

Most computer security is based on the access control model [16], which provides a foundation for secrecy and integrity security policies.[1] The elements of this model are:

Objects, resources such as files, devices, or processes.

Requests to perform operations on objects.

Sources for requests, which are called *principals*.

A reference monitor that examines each request and decides whether to grant it.

The reference monitor bases its decision on the object, the principal making the request, the operation in the request, and a rule that says what principals may perform that operation.

To do its work the monitor needs a trustworthy way to know the access control rule and the source of the request. Usually the access control rule is attached to the object; such a rule is called an access control list or ACL. For each operation it specifies a set of authorized principals, and the monitor grants a request if its principal is trusted at least as much as one of the principals in the set for the requested operation.

The immediate source of the request is some *channel*, for instance, a wire from a terminal, a network connection, a pipe, a kernel call made by a user process, or the successful decryption of an encrypted message. The monitor must deduce the principal responsible for the request from the channel it arrives on. This is called authenticating the channel. It is easy in a centralized system because the operating system implements all the channels and knows the principal responsible for each process. In a distributed system several things make it harder:

The path to the object from the principal ultimately responsible for the request may be long and may involve several machines that are not equally trusted. We might want the authentication to take account of this, say by reporting the principal as "Abadi working through a remote machine" rather than simply "Abadi".

The system may be much larger, and there may be multiple sources of authority for such tasks as registering users.

The system may have different kinds of channels that are secured in different ways. Some examples are encrypted messages, physically secure wires, and inter-process communication done by the operating system.

Some parts of the system may be broken, off line, or otherwise inaccessible.

This paper describes a theory of authentication in distributed systems and a practical system based on the theory. It also uses the theory to explain several other security mechanisms, both existing and proposed. What is the theory good for? In any security system there are assumptions about authority and trust. The theory tells you precisely how to state them and what the rules are for working out their consequences. Once you have done this, you can look at the assumptions, rules, and consequences and decide whether you like them. If so, you have a clear record of how you got to where you are. If not, you can figure out what went wrong and change it.

We use the theory to analyze the security of everything in our system except the low-level details of encryption and the hardware and local operating system on each node. Of course we made many design choices for reasons of performance or scaling that are outside the scope of the theory; its job is to help us work out the implications for security.

The example in figure 1 motivates the design. A user logs in to a workstation and runs a protected subsystem that makes a

[1] The access control model is less useful for availability, which is not considered in this paper. Information flow [8] is an alternative model which is also not considered.

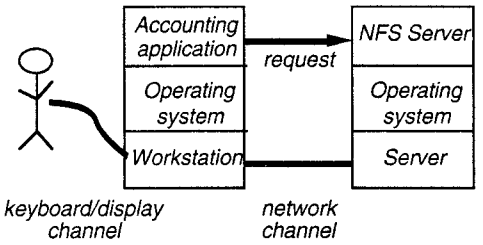

Figure 1: An example

request to an object implemented by a server on a different machine. The server must decide whether to grant the request. We can distinguish the user, two machines, two operating systems, two subsystems, and two channels, one between the user and the workstation and one between the workstation and the server machine. We shall see how to take account of all these components in deciding whether to grant access.

The next section introduces the major concepts behind this work and gives a number of informal examples. In section 3 we explain the theory that is the basis of our system. Each of the later sections takes up one of the problems of distributed system security, presenting a general approach to the problem, a theoretical analysis, the specific details of the solution in our system, and comments on the major alternatives that we know of. Sections 4 and 5 describe two essential building blocks: secure channels and names for principals. Section 6 deals with roles and program loading, and section 7 with delegation. Section 8 treats the mechanics of efficient authenticated inter-process communication, and section 9 sketches how access control uses authentication. A conclusion summarizes the new methods introduced in the paper, the new explanations of old methods, and the current state of our system.

2. Concepts

Both the theory and the system get their power by abstracting from many special cases to a few basic concepts: principal, statement, and channel; trusted computing base; and caching. This section introduces these concepts informally and gives a number of examples to bring out the generality of the ideas. Later sections define them precisely and treat them in detail.

If *s* is a *statement* (request, assertion, etc.), the answer to the question "Who said *s*?" is a principal. Thus principals make statements; this is what they are for. We describe some different kinds of principals and then explain how they make statements.

Principals are either simple or compound. The simple ones in turn are named principals or channels. The most basic named principals have no structure that we care to analyze:

People	Lampson, Abadi
Machines	VaxSN12648, 4thFloorPrinter
Roles	Manager, Secretary, NFS-Server

Other principals with names stand for sets of principals:

Services	SRC-NFS, X-server
Groups	SRC, DEC-Employees

Channels are principals that can say things directly:

Wires or I/O ports	Terminal 14
Encrypted channels	DES encryption with key #574897
Network addresses	IP address 16.4.0.32

Channels are special because in most cases there is no direct way for a computer to know that a principal made a statement. There is no direct path, for example, from a person to a program; communication must involve keystrokes, wires, I/O ports, etc. Of course some of these channels, such as the IP address, are not very secure.

There are also compound principals, built up out of other principals by operators with suggestive names (whose exact meaning we explain later):

Principals in roles	Abadi **as** Manager
Delegations	MikesWS **for** Burrows
Conjunctions	Lampson $\wedge$ Wobber

How do we know that a principal has made a statement? Our theory cannot answer this question for a channel; we simply take such facts as assumptions, though we discuss the basis for accepting them in section 4. However, from statements made by channels and facts about the "speaks for" relation described below, we can use our theory to deduce that a person, a machine, a delegation, or some other kind of principal made a statement.

Different kinds of channels make statements in different ways. A channel's statement may arrive on a wire from a terminal to serial port 14 of a computer. It may be obtained by successfully decrypting with DES key #574897, or by verifying a digital signature on a file stored two weeks ago. It may be delivered by a network with a certain source address, or as the result of a kernel call to the local operating system. Most of these channels are real-time, but some are not.

Often several channels are produced by *multiplexing* a single one. For instance, a network channel to the node with IP address 16.4.0.32 carries UDP channels to ports 2, 75, and 443, or a channel implemented by a kernel call trap from a user process carries inter-process communication channels to several other processes. Different kinds of multiplexing have much in common, and we handle them all uniformly. The subchannels are no more trustworthy than the main channel. Multiplexing can be repeated indefinitely; for instance, an inter-process channel can carry many subchannels to different remote procedures.

Path names are closely connected to multiplexed channels: a single name like `/com/dec/src` can give rise to many others (`/com/dec/src/burrows`, `/com/dec/src/abadi`, ...). Section 5 explores this connection.

There is a fundamental relation between principals that we call the "speaks for" relation: *A* speaks for *B* if the fact that principal *A* says something means we can believe that principal *B* says the same thing. Thus the channel from a terminal speaks for the user at that terminal, and we may want to say that each member of a group speaks for the group. Since only a channel can make a statement directly, a principal can only make a statement by making it on some channel that speaks for that principal.

Any problem in computing can be solved by adding another level of indirection[2], and there are many examples of this in our system. We use "speaks for" to formalize indirection. Often one principal has several others that speak for it: a person or machine and its encryption keys or names (which can change), a single long-term key and many short-term ones, the authority of a job position and the various people that may hold it at different times, an organization or other group of people and its changing membership. The same idea lets a short name stand for a long one; this pays if it's used often.

Another important concept is the "trusted computing base" or TCB [9], a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security. Gathering information to justify an access control decision may require searching databases and communicating with far-flung servers. Once the information is gathered, however, a very simple algorithm can check that it does justify granting access. With the right organization only the checking algorithm need be part of the TCB. Similarly, we can fetch a digitally signed message from an untrusted place without any loss of confidence that the signer actually sent it originally; thus the storage and the transmission channel for the message are not part of the TCB. These are examples of an end-to-end argument [23], which is closely related to the idea of a TCB.

It's not quite true that components outside the TCB can fail without affecting security. Rather, the system is "fail-secure": if an untrusted component fails the system may deny access it should have granted, but it will not grant access it should have denied. Our system uses this idea when it invalidates caches, stores digitally signed certificates in untrusted places, or interprets an ACL that denies access to specific principals.

Finally, we use caching to make frequent operations fast. A cache usually needs a way of removing entries that become invalid. For example, when caching the fact that key #574897 speaks for Burrows we must know what to do if the key is compromised. We might remember every cache that may hold this information and notify them all when we discover the compromise. This means extra work whenever a cache entry is made, and it fails if we can't talk to the cache.

The alternative, which we adopt, is to limit the lifetime of the cache entry and refresh it from the source when it's used after it has expired, or perhaps when it's about to expire. This approach requires a tradeoff between the frequency (and therefore the cost) of refreshing and the time it takes for cached information to expire.

Like any revocation method, refreshing requires the source to be available. Unfortunately, it's very hard to make a source of information that is both highly secure and highly available. This conflict can be resolved by using two sources in conjunction. One is highly secure and uses a long lifetime, the other is highly available and uses a short lifetime; both must agree to validate the information. If the available source is compromised, the worst effect is to delay revocation.

A cache can discard an entry at any time because a miss can always be handled by reloading the cache from the original source. This means that we don't have to worry about deadlocks caused by a shortage of cache entries or about tying up too much memory with entries that are not in active use.

3. Theory

Our theory deals with principals and statements; all principals can do is to say things, and statements are the things they say. Here we present the essentials of the theory, leaving a fuller description to another paper [2]. To help readers who don't like formulas, we highlight the main results by enclosing them in boxes. These readers do need to learn the meanings of two symbols: $A \Rightarrow B$ (A speaks for B) and $A|B$ (A quoting B); both are explained below.

Statements are defined inductively as follows:

There are some primitive statements (e.g., "read file `foo`").

If s and s' are statements, then so are $s \wedge s'$ (s and s'), $s \supset s'$ (s implies s'), and $s \equiv s'$ (s is equivalent to s').

If A is a principal and s is a statement, then so is A **says** s.

If A and B are principals, then $A \Rightarrow B$ (A speaks for B) is a statement.

Throughout the paper we write statements in a form intended to make their meaning clear. When processed by a program or transmitted on a channel they are encoded to save space or make it easier to manipulate them. It has been customary to write them in a style closer to the encoded form than the meaningful one. For example, a Needham-Schroeder authentication ticket [19] is usually written $\{K_{ab}, A\}_{K_{bs}}$. We write K_{bs} **says** $K_{ab} \Rightarrow A$ instead, viewing this as the abstract syntax of the statement and the various encodings as different concrete syntaxes. The choice of encoding does not affect the meaning as long as it can be parsed unambiguously.

We write $\vdash s$ to mean that s is an axiom of the theory (marked by underlining its number) or is provable from the axioms. Here are the axioms for statements:

If s is an instance of a theorem of propositional logic then $\vdash s$. (S1)
For instance, $\vdash s \wedge s' \supset s$.

If $\vdash s$ and $\vdash s \supset s'$ then $\vdash s'$. (S2)
This is modus ponens, the basic rule for reasoning from premises to conclusions.

$\vdash (A \textbf{ says } s \wedge A \textbf{ says } (s \supset s')) \supset A \textbf{ says } s'$. (S3)
This is modus ponens for **says** instead of $\vdash$.

If $\vdash s$ then $\vdash A$ **says** s for every principal A. (S4)

It follows from (S1)-(S4) that **says** distributes over $\wedge$:

$\vdash A \textbf{ says } (s \wedge s') \equiv (A \textbf{ says } s) \wedge (A \textbf{ says } s')$ (S5)

The intuitive meaning of $\vdash A$ **says** s is not quite that A has uttered the statement s, since in fact A may not be present and may never have seen s. Rather it means that A is responsible for s, or that we can proceed as though A has uttered s.

Informally, we write that A *makes* the statement B **says** s when we mean that A does something to make it possible for

[2] Roger Needham attributes this observation to David Wheeler of the Cambridge Computer Laboratory.

another principal to infer B **says** s. For example, A can make A **says** s by uttering s on a channel known to speak for A.

There is a set of principals; we gave many examples in section 2. The symbols A and B denote arbitrary principals, and usually C denotes a channel. In our theory there are two basic operators on principals, $\wedge$ (and) and | (quoting). The set of principals is closed under these operators. We can grasp their meaning from the axioms that relate them to statements:

$$\vdash (A \wedge B)\ \textbf{says}\ s \equiv (A\ \textbf{says}\ s) \wedge (B\ \textbf{says}\ s) \quad \text{(P1)}$$

$(A \wedge B)$ says something if both A and B say it.

$$\vdash (A \mid B)\ \textbf{says}\ s \equiv A\ \textbf{says}\ B\ \textbf{says}\ s \quad \text{(P2)}$$

$A \mid B$ says something if A quotes B as saying it. This does not mean B actually said it: A could be mistaken or lying.

We also have equality between principals, with the usual axioms such as reflexivity. Naturally, equal principals say the same things:

$$\vdash A = B \supset (A\ \textbf{says}\ s \equiv B\ \textbf{says}\ s) \quad \text{(P3)}$$

The $\wedge$ and | operators satisfy certain equations:

$\vdash \wedge$ is associative, commutative, and idempotent. (P4)

$\vdash$ | is associative. (P5)

$\vdash$ | distributes over $\wedge$ in both arguments. (P6)

Now $\Rightarrow$, the "speaks for" relation between principals, can be defined in terms of $\wedge$ and $=$:

$$\vdash (A \Rightarrow B) \equiv (A = A \wedge B) \quad \text{(P7)}$$

and we get some desirable properties as theorems:

$$\vdash (A \Rightarrow B) \supset (\,(A\ \textbf{says}\ s) \supset (B\ \textbf{says}\ s)\,) \quad \text{(P8)}$$

This is the informal definition of "speaks for" in section 2.

$$\vdash (A = B) \equiv (\,(A \Rightarrow B) \wedge (B \Rightarrow A)\,) \quad \text{(P9)}$$

(P7) is a strong definition of "speaks for". It's possible to have a weaker, 'qualified' version in which (P8) holds only for certain statements s. For instance, we could have "speaks for reads" which applies only to statements that request reading from a file, or "speaks for file `foo`" which applies only to statements about file `foo`. Neuman discusses various applications of this idea [20]. Alternatively, we can use roles (see section 6) to compensate for the strength of $\Rightarrow$, for instance by saying $A \Rightarrow (B$ **as** reader) instead of $A \Rightarrow B$.

The operators $\wedge$ and $\Rightarrow$ satisfy the usual laws of the propositional calculus. In particular, $\wedge$ is *monotonic* with respect to $\Rightarrow$. This means that if $A \Rightarrow B$ then $A \wedge C \Rightarrow B \wedge C$. It is also easy to show that | is monotonic in both arguments and that $\Rightarrow$ is transitive. These properties are critical because $C \Rightarrow A$ is what authenticates that a channel C speaks for a principal A or that C is a member of the group A. If we have requests K_{abadi} **says** "read from `foo`" and $K_{burrows}$ **says** "read from `foo`", and file `foo` has the ACL SRC $\wedge$ Manager, we must get from $K_{abadi} \Rightarrow$ Abadi $\Rightarrow$ SRC and $K_{burrows} \Rightarrow$ Burrows $\Rightarrow$ Manager to $K_{abadi} \wedge K_{burrows} \Rightarrow$ SRC $\wedge$ Manager. This lets us reason from the two requests to SRC $\wedge$ Manager **says** "read from `foo`", and the ACL obviously grants this. For the same reason, the **as** and **for** operators defined in sections 6 and 7 are also monotonic.

The following *handoff* axiom makes it possible for a principal to introduce new facts about $\Rightarrow$:

$$\vdash (\,A\ \textbf{says}\ (B \Rightarrow A)\,) \supset (B \Rightarrow A) \quad \text{(P10)}$$

In other words, A has the right to allow any other principal B to speak for it.[3] There is a simple rule for applying (P10): when you see A **says** s you can conclude s if it has the form $B \Rightarrow A$. The same A must do the saying and appear on the right of the $\Rightarrow$, but B can be any principal.

What is the intuitive justification for (P10)? Since A can make A **says** $(B \Rightarrow A)$ whenever it likes, (P10) gives A the power to make us conclude that A **says** s whenever B **says** s. But B can just ask A to say s directly, which has the same effect provided A is competent and accessible.

From (P10) we can derive a theorem asserting that it is enough for the principal doing the saying to speak for the one on the right of the $\Rightarrow$, rather than being the same:

$$\vdash (A' \Rightarrow A) \wedge A'\ \textbf{says}\ (B \Rightarrow A) \supset (B \Rightarrow A) \quad \text{(P11)}$$

It holds because the premise implies A **says** $B \Rightarrow A$ by (P8), and this implies the conclusion by (P10). This theorem, called the handoff rule, is the foundation of our methods for authentication. When we use it we say that A' hands off A to B.

A final theorem deals with the exercise of joint authority:

$$\vdash (B \wedge B' \Rightarrow A) \wedge (B \Rightarrow B') \supset (B \Rightarrow A) \quad \text{(P12)}$$

From this and the handoff axiom we can deduce $B \Rightarrow A$ given A **says** $(B \wedge B' \Rightarrow A)$ and B' **says** $B \Rightarrow B'$. Thus A can let B and B' speak for it jointly, and B' can let B exercise this authority alone. One situation in which we might want both B and B' is when B' is inaccessible most of the time and therefore makes its statement with a much longer lifetime than B's. (P12) is the basis for revoking authentication certificates (section 5) and ending a login session (section 7).

The last two theorems illustrate how we can prove $B \Rightarrow A$ from our axioms together with some premises of the form A' **says** $(B' \Rightarrow A')$. Such a proof together with the premises is called B's *credentials* for A. Each premise has a *lifetime*, and the lifetime of the credentials is the lifetime of the shortest-lived premise. We could add lifetimes to our formalism by introducing a statement form s **until** t and modifying (S2)-(S3) to apply the smallest t in the premises to the conclusion, but here we content ourselves with an informal treatment.

4. Channels and encryption

As we have seen, the essential property of a channel is that its statements can be taken as assumptions: C **says** s is the raw material from which everything else must be derived. On the other hand, the channel by itself doesn't usually mean much — seeing a message from terminal port 14 or key #574897 isn't very interesting unless we can deduce something about who must have sent it. If we know the possible senders on C, we say that C has integrity. Similarly, if we know the possi-

[3] In this paper we take (P10) as an axiom for simplicity. However, it is preferable to assume only some instances of (P10)—the general axiom is too powerful, for example when A represents a group. If the conclusion uses a qualified form of $\Rightarrow$ it may be more acceptable

ble receivers we say that C has secrecy, though we have little to say about secrecy in this paper.

Knowing the possible senders on C means finding a meaningful A such that $C \Rightarrow A$; we call this authenticating the channel. Why should we believe that $C \Rightarrow A$? Only because A, or someone who speaks for A, tells us so. Then the hand-off rule (P11) lets us conclude $C \Rightarrow A$. In the next section we study the most common way of authenticating C. Here we investigate why A might trust C enough to make A **says** $C \Rightarrow A$, or in other words, why A should believe that A is the only possible source of messages on C.

The first thing to notice is that for A to assert $C \Rightarrow A$ it must be able to name C. A circumlocution like "the channel carrying this message speaks for A" won't do, because it can be subverted just by copying it to another channel. As we consider various kinds of channels, we discuss how to name them.

A sender on a channel C can always make C **says** X **says** s, where X is any identifier. We take this as the definition of multiplexing; various values of X establish a number of subchannels. By (P2) C **says** X **says** s is the same thing as $C|X$ **says** s. Thus if C is a name for the channel, $C|X$ is a name for the subchannel. We will see many examples of this.

Encryption channels

We are mainly interested in channels that depend on cryptography for their security; as we shall see, they add less to the TCB than any others. We begin by summarizing the essential facts about such channels. An encryption channel is two functions *Encrypt* and *Decrypt* and two keys K and K^{-1}. By convention we normally use K to receive (decrypt) and K^{-1} to send (encrypt). Another common notation for $Encrypt(K^{-1}, x)$ is $\{s\}_{K^{-1}}$.

An encryption algorithm that is useful for computers provides a channel: $Decrypt(K, Encrypt(K^{-1}, x)) = x$ for any message x. It keeps the keys secret: if you know only x and $Encrypt(K^{-1}, x)$ you can't compute K or K^{-1}, and likewise for *Decrypt*. Of course "can't compute" really means that the computation is too hard to be feasible.

In addition, the algorithm should provide one or both of:

Secrecy: If you know $Encrypt(K^{-1}, x)$ but not K, then you can't compute x.

Integrity: If you know x but not K^{-1}, then you can't compute a y such that $Decrypt(K, y) = x$.

The usual way to get both properties at once is to add a suitable checksum to the cleartext and check it in *Decrypt* [27].

For integrity it is enough to encrypt a *digest* of the message. A digest is the result of a one-way function; this means that you can't invert the function and compute a message with a given digest. One practical digest function is MD4 [22].An algorithm that provides integrity but not necessarily secrecy is said to implement digital signatures.

The secrecy or integrity of an encryption channel does not depend on how the encrypted messages are handled, since by assumption an adversary can't compromise secrecy by knowing the encrypted message or integrity by changing it. Thus the handling of the encrypted message is not part of the TCB, since security does not depend on it.

	Hardware, bits/sec	Software, bits/sec/MIPS	Notes
RSA encrypt	220 K [24]	.5 K [6]	500 bit modulus
RSA decrypt	—	32 K [6]	Exponent=3
MD4	—	1300 K [22]	
DES	1.2 G [11]	400 K [6]	Software uses a 64 KB table per key

Table 1: Speeds of cryptographic operations[5]

There are two kinds of encryption, shared key and public key.

In shared key encryption $K = K^{-1}$. Since anyone who can receive can also send under K, this is only useful for pairwise communication. The most popular shared key encryption scheme is the Data Encryption Standard or DES [18]. We denote an encryption channel with DES key K by $DES(K)$; it speaks for the set of principals that know K.

In public key encryption $K \neq K^{-1}$, and in fact you can't compute one from the other. Usually K is made public and K^{-1} kept private, so that the holder of K^{-1} can broadcast messages with integrity; of course they won't be secret.[4] Together K and K^{-1} are called a key pair. The most popular public key encryption scheme is Rivest-Shamir-Adleman or RSA [20]. In this scheme $(K^{-1})^{-1} = K$, so anyone can send a secret message to the holder of K^{-1} by encrypting it with K. We denote an encryption channel with RSA public key K by $RSA(K)$; it speaks for the principal that knows K^{-1}.

Table 1 shows that encryption need not slow down a system unduly. It also shows that shared key encryption can be about 1000-5000 times faster than public key. Hence public key is usually used only to encrypt small messages or to set up a shared key.

With this background we can discuss how to make a practical channel from an encryption algorithm. We denote an encryption channel simply by K when the meaning is obvious. From the existence of the bits $Encrypt(K^{-1}, s)$ anyone can infer K **says** s, so we tend to identify the bits and the statement; of course for the purposes of reasoning we use only the latter. Often we call such a statement a *certificate*, because it is simply a sequence of bits that can be stored away and brought out when needed like a paper certificate. We say that K *signs* the certificate.

A certificate can name an encryption channel by its key, but we sometimes want a name that need not be kept secret. This is straightforward for a public-key channel, since the key is not secret. For a shared key channel we can use a digest of the key. It's possible that the receiver doesn't actually know the key, but instead uses a sealed and tamper-proof encryption box to encrypt or decrypt messages. In this case the box can generate the digest on demand, or it can be computed by encrypting a known text (such as 0) with the key.

---

[4] Sometimes K^{-1} is called the decryption key, but we prefer to associate encryption with sending and to use the simpler expression K for the public key.

[5] Many variables affect performance; consult the references for details, or believe these numbers only within a factor of two. The software numbers come from data in the references and assumed speeds of .5 MIPS for an 8 Mhz Intel 286 and 9 MIPS for a 20 MHz Sparc.

The receiver needs to know what key K it should use to decrypt a message. If K is a public key we can send it along with the encrypted message; all the receiver has to do is check that K actually decrypts the message correctly. If K is a shared key we can't include it with the message because K has to remain secret. We can, however, include a *key identifier* that allows the receiver to know what the key is but doesn't disclose anything about it to others.

We need some notation for keys. Subscripts and primes on K denote different keys; the choice of subscript may be suggestive, but it has no formal meaning. A superscripted key does have a meaning: it denotes a key identifier for that key, and the superscripts indicate who can extract the key from the identifier. Thus K^r denotes R's key identifier for K, and if K^a and K^b are key identifiers for the two parties to the shared key K, then K^{ab} denotes the pair (K^a, K^b). The statement K^r **says** s denotes the pair $(K^r, Encrypt(K^{-1}, s))$.

A key identifier K^r for a receiver R might be any one of:

an index into a table of keys that R maintains,

$Encrypt(K_{rm}, K)$, with K_{rm} a master key only R knows,

a pair $(K'^r, Encrypt(K', K))$, where K'^r is a key identifier for another key K'.

In the second case R can extract the key from the identifier without any state except its master key K_{rm}, and in the third case without any state except what it needs for K'^r. An encrypted key may be weaker cryptographically than a table index, but we believe that it is safe to use it as a key identifier.

We conclude the general treatment of encryption channels by explaining the special role of public keys. A public key channel is a broadcast channel: you can send a message without knowing who will receive it. As a result:

You can generate a message before *anyone* knows who will receive it. In particular, an authority can make a single certificate asserting, for instance, that RSA$(K_a) \Rightarrow A$. This can be stored in any convenient place (secure or not), and anyone can receive it later, even if the authority is then off-line.

If you receive a message and forward it to someone else, he has the same assurance of its source that you have.

By contrast, a shared key message must be directed to its receiver when it is generated. This tends to mean that it must be sent and received in real time, because it's too hard to predict in advance who the receiver will be. An important exception is a message sent to yourself, such as the key identifier encrypted with a master key that we described just above.

For these reasons our system uses public key encryption for authentication. It can still work, however, even if all public key algorithms turn out to be insecure or too slow, because shared key can simulate public key using a *relay*. This is a trusted agent R that can translate any message m encrypted with a key that R knows. If you have a channel to R, you can ask R to translate m, and it will decrypt m and return the result to you. Since R simulates public key encryption, we assume that anyone can get a channel to R. Relays use the key identifiers introduced above, and the explanation here depends on the notation defined there.

	Public key	**Shared key with relay**
To send s, principal A	encrypts with K_a^{-1} to make K_a **says** s.	encrypts with K_a^{ar} to make K_a^r **says** s.
To receive s, principal B	gets K_a **says** s and decrypts it with K_a.	gets K_a^r **says** s, sends it and K_b^{br} to R, gets back $K_b^b \mid K_a^r$ **says** s, and decrypts it with K_b^b.
A certificate authenticating A to B is	K_{ca} **says** $K_a \Rightarrow A$.	K_{ca}^r **says** $K_b^b \mid K_a^r \Rightarrow A$.
To relay a certificate K_{ca}^r **says** $K_a^{ar} \Rightarrow A$ to K_b^{br}, R	is not needed.	invents a key K and makes $K_b^b \mid K_{ca}^r$ **says** $K^{ab} \Rightarrow A$ where $K^{ab} = (K^a, K^b)$ and $K^a = (K_a^a, Encrypt(K_a, K))$, $K^b = (K_b^b, Encrypt(K_b, K))$.

Table 2: Simulating public key with shared key encryption

Given both K_a^r **says** s (a message encrypted by a key K_a together with R's key identifier for K_a) and K_b^{br} (a two-way channel to some B), the relay R will make $K_b^b \mid K_a^r$ **says** s. The relay thus multiplexes all the channels it has onto its channel to B, using the key identifier to indicate the source of each message. Note that the relay is not vouching for the source A of s but only for the key K_a that was used to encrypt s. In other words, it is simply identifying the source by labelling s with K_a^r and telling anyone who is interested the content of s. This is just what public key encryption can do. Like public key encryption, the relay provides no secrecy; of course it could be made fancier, but we don't need that for authentication. From B's point of view the channel $K_b^b \mid K_a^r$ is the source of the message.

With public keys a certificate like K_{ca} **says** $K_a \Rightarrow A$ authenticates the key K_a. In the simulation this becomes K_{ca}^r **says** $K_a^x \Rightarrow A$ for some X, and relaying this to B is not useful because B cannot extract K_a from K_a^x. But given K_{ca}^r **says** $K_a^{ar} \Rightarrow A$ and K_b^{br} as before, R can invent a new key K and *splice* the channels K_a^{ar} and K_b^{br} to make a two-way channel $K^{ab} = (K^a, K^b)$ between A and B. Here K^a and K^b are defined in the lower right corner of table 2; they are the third kind of key identifier mentioned earlier. Observe that A can decrypt K^a to get hold of K, and likewise for B and K^b. Now R can translate the original message into $K_b^b \mid K_{ca}^r$ **says** $K^{ab} \Rightarrow A$, just what B needs for authenticated communication with A. For two-way authentication R needs K_{ca}^r **says** $K_b^{br} \Rightarrow B$ instead of K_b^{br}; from this it can symmetrically make $K_a^a \mid K_{ca}^r$ **says** $K^{ab} \Rightarrow B$.

Table 2 summarizes the construction, which uses an essentially stateless relay to give shared key encryption the properties of public key encryption. The only state the relay needs is its master key; the client supplies the channels K_a^r and K_b^{br}. Because of their minimal state, it is practical to make such relays highly available as well as highly secure.

Davis and Swick give a more detailed account of the scheme from a somewhat different point of view [7].

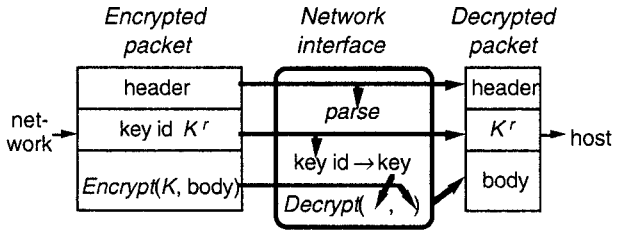

Figure 2: Fast decryption

Node-to-node secure channels

A *node* is a machine running an operating system, connected to other machines by wires that are not physically secure. Our system uses shared key encryption to implement secure channels between the nodes of the distributed system and then multiplexes these channels to obtain all the other channels it needs. Since the operating system in each node must be trusted anyway, using encryption at a finer grain than this (for instance, between processes) can't reduce the size of the TCB. Here we explain how our system establishes the node-to-node shared keys; of course, many other methods could be used.

We have a network interface that can parse an incoming packet to find the key identifier for the channel, map the identifier to a DES key, and decrypt the packet on the fly as it moves from the wire into memory [14]. This makes it practical to secure all the communication in a distributed system, since encryption does not reduce the bandwidth or much increase the latency. Our key identifier is the channel key encrypted by a master key that only the receiving node knows. Figure 2 shows how it works.

We need to be able to change the master key, because this is the only way a node can lose the ability to decrypt old messages; we want to limit the length of time after the node sends or receives a message during which compromising the node allows an adversary to read the message. We also need a way to efficiently change the individual node-to-node keys, for two reasons. One is cryptographic: a key should encrypt only a limited amount of traffic. The other is to protect higher-level protocols that reuse sequence numbers and connection identifiers. Many existing protocols do this, relying on assumptions about maximum packet lifetimes. If an adversary can replay messages these assumptions fail, but changing the key allows us to enforce them. In effect the integrity checksum becomes an extension of the sequence number.

However, changes in the master or channel keys should not force us to reauthenticate a node-to-node channel or anything multiplexed on it, because this can be quite expensive (see section 8). Furthermore, we separate setting up the channel from authenticating it, since these operations are done at very different levels in the communication protocol stack; setup is done between the network and transport layers, authentication in the session layer or above. In this respect our system differs from the Needham-Schroeder protocol and its descendants [15, 19, 25], which combine key exchange with authentication, but is similar to the Diffie-Hellman protocol for key exchange [10].

	A knows before	*B* to *A*	*A* knows after
Phase 1	K_a, K_a^{-1}, K_{am}	K_b	K_b
Phase 2	J_a	$Encrypt(K_a, J_b)$	J_b
Phase 3	$K = Hash(J_a, J_b)$, $K^a = Encrypt(K_{am}, K)$	K^b	K^{ab}

Table 3: *A*'s view of node-to-node channel setup; *B*'s is symmetric

We set up a node-to-node channel between nodes *A* and *B* in three phases; see table 3. In the first phase each node sends its public RSA key to the other node. It knows the corresponding private key, having made its key pair when it was booted (see section 6). In phase two each node chooses a random DES key, encrypts it with the other node's public key, and sends the result to the other node which decrypts with its own private key. For example, *B* chooses J_b and sends $Encrypt(K_a, J_b)$ to *A*, which decrypts with K_a^{-1} to recover J_b. In the third phase each node computes $K = Hash(J_a, J_b)$ using the same commutative one-way hash function, encrypts *K* with its own master key to make a key identifier, and sends that to the other node. Now each node has K^{ab} (the key identifiers of *A* and *B* for the shared key *K*); this is just what they need to communicate.[6]

A believes that only someone who can decrypt $Encrypt(K_b, J_a)$ could share its knowledge of *K*. In other words, *A* believes that $K \Rightarrow K_b$.[7] This means that *A* takes $K \Rightarrow K_b$ as an assumption of the theory; we can't prove it because it depends both on the secrecy of RSA encryption and on prudent behavior by *A* and *B*, who must keep the *J*'s and *K* secret. We have used the secrecy of an RSA channel to avoid the need for the certificate K_b **says** $Digest(K) \Rightarrow K_b$.

Now whenever *A* sees *K* **says** *s* it can immediately conclude K_b **says** *s*. This means that when *A* receives a message on channel *K*, which changes whenever there is rekeying, it receives the same message on channel K_b, which does not change as long as *B* is not rebooted. Of course *B* is in a symmetric state. Finally, if either node forgets *K*, executing the protocol again makes a new DES channel that corresponds to the same public key on each node. Thus the DES channel, like a cache entry, can be flushed and re-established without any external effect.

The only property of the key pair (K_a, K_a^{-1}) that channel setup cares about is that K_a^{-1} is *A*'s secret. Indeed, channel setup can make up the key pair. But K_a is not useful without credentials. The node *A* has a node key K_n and its credentials $K_n \Rightarrow A'$ for some more meaningful principal *A'*, for instance

[6] We use *Hash* to prevent a chosen-plaintext attack on a master key and to keep *K* secret even if one of the *J*'s is disclosed. The third phase can compute lots of keys, for instance *K*, *K*+1, ..., and exchange lots of key identifiers. Switching from one of these keys to another may be useless cryptographically, but it is quite adequate for allowing connection identifiers to be reused.

[7] Actually *K* speaks for *A* or K_b, since *A* also knows and uses *K*. To deal with this we multiplex the encryption channel to make *K*|*A* and *K*|*B* (a single bit can encode *A* or *B* in this case), and *A* never makes *K*|*B* **says** *s*. Then *A* knows that $K|B \Rightarrow K_b$. To reduce clutter in the formulas we ignore this complication. There are protocols in use that encode this multiplexing in strange and wonderful ways.

VaxSN5437 **as** VMS5.4 (see section 6). If K_a comes out of the blue, the node has to sign another certificate K_n **says** $K_a \Rightarrow K_n$ to complete K_a's credentials, and everyone authenticating the node has to check this added certificate. That is why in our system the node tells channel setup to use (K_n, K_n^{-1}) as its key pair, rather than allowing it to choose a key pair.[8]

5. Principals with names

When users refer to principals they must do so by name, since users can't understand alternatives like unique identifiers or keys. Thus an ACL must grant access to named principals. But a request arrives on a channel, and it is granted only if the channel speaks for one of the principals on the ACL. In this section we study how to find a channel C that speaks for the named principal A.

There are two general methods, push and pull. Both produce the same credentials for A, a set of certificates and a proof that they establish $C \Rightarrow A$, but the two methods collect the certificates differently.

Push: The sender on the channel collects A's credentials and presents them when it needs to authenticate the channel to the receiver.

Pull: The receiver looks up A in some database to get credentials for A when it needs to authenticate the sender; we call this *name lookup*.

Our system uses the pull method, like DSSA [12] and unlike most other authentication protocols. However, the credentials don't depend on the method. We describe them for the case where C is a public key, since this is what we implement.

The basic idea is that there is a *certification authority* that speaks for A and so is trusted when it says that C speaks for A, because of the handoff rule (P11). In the simplest system

there is only one such authority CA,

everyone trusts CA to speak for every named principal, and

everyone knows CA's public key K_{ca}, that is, $K_{ca} \Rightarrow CA$.

So everyone can deduce $K_{ca} \Rightarrow A$ for every named A. At first this may seem too strong, but trusting CA to authenticate channels from A means that CA can speak for A, because it can authenticate as coming from A some channel CA controls.

For each A that it speaks for, CA issues a certificate of the form K_{ca} **says** $K_a \Rightarrow A$ in which A is a name. The certificates are stored in a database indexed by A, usually called a name service; the database is not part of the TCB because the certificates are digitally signed by K_{ca}. To get A's credentials you go to the database, look up A, get the certificate K_{ca} **says** $K_a \Rightarrow A$, verify that it is signed by the K_{ca} that you believe speaks for CA, and use the handoff rule to conclude $K_a \Rightarrow A$, just what you wanted to know. The right side of figure 3 shows what B does, the symmetric left side what A does to establish two-way authentication.

[8] An alternative is for the node to directly authenticate the shared key K by making K_n **says** $K \Rightarrow K_n$. This prevents channel setup from changing K on its own, which is a significant loss of functionality. Authentication can't be done without a name for the channel, so the interface to channel setup must either accept or return some key naming the channel.

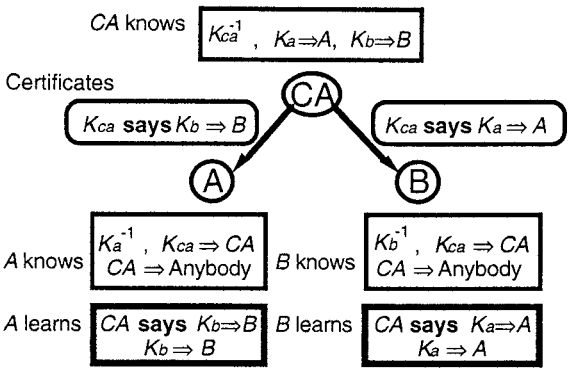

Figure 3: Authenticating channels with one certification authority

The figure shows only the logical flow of secure messages. An actual implementation has extra insecure messages, and the bits of the secure ones may travel by circuitous paths. To push, the sender A calls the database to get K_{ca} **says** $K_a \Rightarrow A$ and sends it along with a message signed by K_a. To pull, the receiver B calls the database to get the same certificate when B gets a message that claims to be from A or finds A on an ACL. The Needham-Schroeder protocol [19] combines push and pull. When A wants to talk to B it gets two certificates from CA, the familiar K_{ca} **says** $K_a \Rightarrow A$ which it pushes along to B, and K_{ca} **says** $K_b \Rightarrow B$ for A's channel from B.

With public key certificates it's not necessary to talk to CA directly; it suffices to talk to a database that stores CA's certificates. Thus CA itself can be normally off-line, and hence much easier to make highly secure. Certificates from an off-line CA, however, must have fairly long lifetimes. For rapid revocation you add an on-line agent O and use the joint authority rule (P12). CA makes a weaker certificate K_{ca} **says** $(O|K_a \wedge K_a) \Rightarrow A$, and O makes $O|K_a$ **says** $K_a \Rightarrow O|K_a$. From these two, $K_{ca} \Rightarrow A$, and (P12) you again get $K_a \Rightarrow A$, but now the lifetime is the minimum of those on CA's certificate and O's certificate. Since O is on-line, its certificate can time out quickly and be refreshed often. The TCB for granting access is just CA; that for revocation is CA and O.

Our system uses the pull method throughout; we discuss the implications in sections 8 and 9. Hence we can use a cheap version of the joint authority scheme for revocation in which a certificate from CA is believed only if it comes from the server O that stores the database of certificates. To authenticate A we first authenticate a channel C_o from O. Then we interpret the certificate K_{ca} **says** $(O|K_a \wedge K_a) \Rightarrow A$ returned on C_o as $O|K_a$ **says** $K_a \Rightarrow O|K_a$. This is the same statement as before, so we get the same conclusion. Note that O doesn't sign a public-key certificate for A, but we must authenticate the channel from O, presumably using the basic method. Or replace O by K_o everywhere. Either way, we can't revoke O's authority quickly; it's not turtles all the way down.

The same formalization also describes the Kerberos protocol [15, 25]. Kerberos uses shared rather than public key encryption. Although it wasn't designed this way, the protocol simulates public key certificates with shared keys using the relay construction of section 4. Here are the steps; they correspond to the union of figure 3 and table 2. First A gets from CA a certificate $K_{ca}{}^r$ **says** $K_a{}^{ar} \Rightarrow A$. Kerberos calls CA the

"authentication server", the certificate a "ticket granting ticket", and the relay R the "ticket granting server".[9] The relay also has a channel to every principal that A might talk to; in particular R knows $K_b{}^{br} \Rightarrow B$.[10] To authenticate a channel from A to B, A sends the certificate to R, which splices $K_a{}^{ar}$ and $K_b{}^{br}$ to turn it into $K_b{}^b$ **says** $K^{ab} \Rightarrow A$. This is called a "ticket",[11] and A sends it on to B, which believes $K_b \Rightarrow$ Anybody because K_b is B's channel to CA. As a bonus, R also sends A a certificate for B: $K_a{}^a$ **says** $K^{ab} \Rightarrow B$.

In practice, Kerberos is normally used to authenticate network connections, which are then rather unrealistically treated as secure channels. To accomplish this, A makes K^b **says** $ci_a \Rightarrow A$, where ci_a is A's network address and connection identifier; this is called an "authenticator". A sends both the ticket and the authenticator to B, which can then deduce $ci_a \Rightarrow A$ in the usual way. The ticket has a fairly long lifetime so that A doesn't have to talk to R very often; the authenticator has a very short lifetime in case the connection is closed and ci_a then reused for another connection not controlled by A. Kerberos has other features that we lack space to describe.

Our channel authentication protocol is a communication protocol and must address all the issues that such protocols must address. In particular, it must deal with duplicate messages; in security jargon, it must prevent replays or establish timeliness. The same techniques are used (or misused) in both worlds: timestamps, unique identifiers or nonces, and sequence numbers. Our system uses timestamps to limit the lifetimes of certificates and hence relies on loosely synchronized clocks; the details are old [4] and we omit them here.

Path names

In a system of any size there can't be just one certification authority—it's administratively impractical, and there may not be anyone who is trusted by everybody in the system. The authority to speak for names must be decentralized. The natural way to do this is to use path names and arrange the certification authorities in a corresponding tree. The lack of global trust means that a parent cannot unconditionally speak for its children. Instead when you want to authenticate a channel from $A = /A_1/A_2/.../A_n$ you start from an authority that you believe has the name $B = /B_1/B_2/.../B_m$ and traverse the authority tree from B up to the least common ancestor of B and A and back down to A. Figure 4 shows the path from `/dec/burrows` to `/mit/clark`; the numbers stand for public keys. The basic idea is described in Birrell et al. [3] and is also implemented in SPX [26].

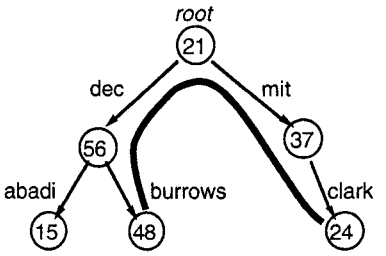

Figure 4: Authentication with a tree of authorities

We can formalize this with a new kind of compound principal, written P **except** M, and some axioms that define its meaning. Here P is any path name and M or N any simple name, a component of a path name.

$\vdash P$ **except** $M \Rightarrow P$ (N1)
So P **except** M is stronger than P; other axioms say how.

$\vdash M \neq N \supset (P$ **except** $M) \mid N \Rightarrow P/N$ **except** '..' (N2)
P **except** M can speak for any path name P/N just by quoting N, as long as N isn't M. This lets us go down the tree (but not back up, because of the **except** '..').[12]

$\vdash M \neq$ '..' $\supset (P/N$ **except** $M) \mid$ '..' $\Rightarrow P$ **except** N (N3)
P/N **except** M can speak for the shorter path name P just by quoting '..', as long as M isn't '..'. This lets us go up the tree (but not back down the same path, because of the **except** N).

We use the quoting principals on the left side of $\Rightarrow$ to make sure that something asserted by P **except** M isn't automatically taken to be asserted by all the longer path names.

Now we can describe the credentials that establish $C \Rightarrow A$ in our system. Suppose A is `/mit/clark`. To use the (N) rules we must start with a channel from some principal B that can authenticate path names; that is, we need to believe $C_b \Rightarrow B$ **except** N. This could be anyone, but it's simplest to let B be the authenticating party. In figure 4 this is `/dec/burrows`, so initially we believe $C_{burrows} \Rightarrow$ `/dec/burrows` **except** `nil` (this channel is trusted to authenticate both up and down). In other words, Burrows knows his name and his public key. Then each principal on the path from B to A must provide a certificate for the next one. Thus we need

$C_{burrows}$	| '..'	**says** C_{dec}	$\Rightarrow$ `/dec` **except**	`burrows`
C_{dec}	| '..'	**says** C_{root}	$\Rightarrow$ `/` **except**	`dec`
C_{root}	| `mit`	**says** C_{mit}	$\Rightarrow$ `/mit` **except**	'..'
C_{mit}	| `clark`	**says** C_{clark}	$\Rightarrow$ `/mit/clark` **except**	'..'

The certificates quoting '..' can be thought of as 'parent' certificates pointing upward in the tree, those quoting `mit` and `clark` as 'child' certificates pointing downward. They are similar to the certificates specified by CCITT X.509 [5].

From this and the assumption $C_{burrows} \Rightarrow$ `/dec/burrows` **except** `nil`, we deduce in turn the body of each certificate, because for each A' **says** $C' \Rightarrow B'$ we have $A' \Rightarrow B'$ by reasoning from the initial belief and the (N2-3) rules, and thus we can apply (P11) to get $C' \Rightarrow B'$. Hence we derive $C_{clark} \Rightarrow$ `/mit/clark` by (N1), so we have authenticated the chan-

[9] K_a is a login session key. CA invents K_a and tells A about it (that is, generates $K_a{}^a$) by encrypting it with A's permanent key, which in current implementations is derived from A's password.

[10] The Kerberos relay is asymmetric between A and B, since it knows $K_b{}^{br} \Rightarrow B$ but gets its channel to A out of A's certificate from CA. This is justified by the notion that A is a workstation while B is a server that is friendlier with R, but it's unfortunate because asymmetry is bad and because R has to have some state for each B. There is an option (called ENC-TKT-IN-SKEY in [15]) for A to get K_{ca} says $K_b{}^{br} \Rightarrow B$ from B and provide it to R, which now becomes symmetric and stateless.

[11] The ticket lacks the "$K_{ca}{}^r$ **says**" that a true relay would include because in Kerberos R handles only statements from CA and therefore doesn't need to identify the source of the statement.

[12] By putting several names after the **except** rather than one, we could further constrain the path names that a principal can authenticate.

nel C_{clark} from `/mit/clark`. In the most secure implementation each line represents a certificate signed by the public key of an off-line certifier[13] plus a message on some channel from the on-line agent that can revoke the certificate; we described this scheme earlier. But any kind of channel will do.

If we start with a different assumption, we may not accept the bodies of all these certificates. Thus if `/mit/clark` is authenticating `/dec/abadi`, we start with $C_{clark} \Rightarrow$ `/mit/clark` **except** `nil` and believe the bodies of the certificates

C_{clark}	| '..'	**says** C_{mit}	$\Rightarrow$ `/mit` **except** `clark`
C_{mit}	| '..'	**says** C_{root}	$\Rightarrow$ `/` **except** `mit`
C_{root}	| `dec`	**says** C_{dec}	$\Rightarrow$ `/dec` **except** '..'
C_{dec}	| `abadi`	**says** C_{abadi}	$\Rightarrow$ `/dec/abadi` **except** '..'

Since this path is the reverse of the one we traversed before except for the last step, each principal that supplies a parent certificate on one path supplies a child certificate on the other. Observe that `clark` would not accept the bodies of *any* of the certificates on the path from `burrows`. Furthermore, the intermediate results of this authentication are different from those we saw before. For example, when B was `/dec/burrows` we got $C_{dec} \Rightarrow$ `/dec` **except** `burrows`, but if B is `/mit/clark` we get $C_{dec} \Rightarrow$ `/dec` **except** '..'. From either we can deduce $C_{dec} \Rightarrow$ `/dec`, but C_{dec}'s authority to authenticate other path names is different. This reflects the fact that `burrows` and `clark` have different ideas about how much to trust `dec`.

It's neither necessary nor desirable to include the entire path name of the principal in each certificate. It's unnecessary because everything except the last component is the same as the name of the certifying authority, and it's undesirable because we don't want the certificates to change when names change higher in the tree. So the actual form of a certificate is C_{mit} | `clark` **says** $C_{clark} \Rightarrow$ "my name"`/clark` **except** '..'.

This method requires trusting each certification authority on the path from B up to the least common ancestor and back down to A. To trust fewer authorities we can lower the least common ancestor in the tree by adding a cross-link named `mit` from node 56 to node 37: C_{dec} **says** $C_{mit} \Rightarrow$ `/dec/mit` **except** '..'. Now `/dec/mit/clark` names A, and node 21 is no longer involved in the authentication. The price is that the cross-link has to be installed and changed when `mit`'s key changes. Note that the least-common-ancestor rule still applies, so it's easy to explain who is being trusted.

The implementation obtains all these certificates by talking in turn to the databases that store certificates from the various authorities. This requires one RPC to each database in both pull and push models; the only difference is whether receiver or sender does the calls. Certificates from several authorities might be stored in the same database, in which case several can be retrieved with a single call. Once retrieved, certificates can be cached; this is especially important for those from the higher reaches of the name space. The cache hit rate may differ between push and pull, depending on traffic patterns.

[13] A single certifier with a single key K can act for several principals by multiplexing its channel, that is, by assigning a distinct identifier id_P to each such principal P and using $K|id_P$ as C_P.

A principal doing a lookup might have channels to several other principals instead of the single one C_b to itself that we described. Then it could start with the channel to the principal that is closest to the target A and reduce the number of intermediaries that must be trusted. This is essential if the entire name space has more than one root, but it obviously complicates managing the system. For this reason our system does not use such sets of initially trusted principals.

When we use path names, the names of principals are more likely to change, because they change when the directory tree is reorganized. This is a familiar phenomenon in file systems, where it is dealt with by adding either extra links or symbolic links to the renamed objects (usually directories) that allow old names to keep working. Our system works the same way; a link is a certificate asserting that some channel $C \Rightarrow P$, and a symbolic link is a certificate asserting $P' \Rightarrow P$. This makes pulling more attractive, however, because pushing requires the sender to guess which name the receiver is using for the principal so that the sender can provide the right certificates.

We can push without guessing if we add a level of indirection in the form of a unique identifier for the principal. Instead of $C \Rightarrow P$ we have $C \Rightarrow id$ and $id \Rightarrow P$. The sender pushes $C \Rightarrow id$ and the receiver pulls $id \Rightarrow P$. In general the receiver can't just use id, on an ACL for example, because it has to have a name so that people can understand the ACL. Of course it can cache $id \Rightarrow P$; this corresponds to storing both the name and the id on the ACL. There is one tricky point about this method: id can't simply be an integer, because there would be no way of knowing who can speak for it and therefore no way to establish $C \Rightarrow id$. Instead, it must have the form *A/integer* for some other principal A, and we need a rule $A \Rightarrow$ *A/integer* so that A can speak for id. Now the problem has been lifted from arbitrary names like P to authorities like A, and perhaps it's easier to handle. Our system avoids these complications by using the pull model throughout.

Groups

A group is a principal that has no public key or other channel of its own. Instead, other principals speak for the group; they are the group members. The result of looking up a group name G is one or more group membership certificates $P_1 \Rightarrow G$, $P_2 \Rightarrow G$, ..., just as the result of looking up an ordinary principal name P is one or more certificates $C \Rightarrow P$ for its channels. A symbolic link can be viewed as a special case of a group. This representation makes it impossible to prove that P is not a member of G.

A quite different way to express group membership when the channels are public keys is to give G a key K_g and a corresponding certificate $K_g \Rightarrow G$, and store $Encrypt(K_p, K_g^{-1})$ for each member P in G's database entry. This means that each member will be able to get K_g^{-1} and therefore to speak for the group, while no other principals can do so.

The advantage is that to speak for G, P simply makes K_g **says** s, and to verify this a third party only needs $K_g \Rightarrow G$. In the other scheme P makes K_p **says** s, and a third party needs both $K_p \Rightarrow P$ and $P \Rightarrow G$. So a certificate and a level of indirection are saved. One drawback is that to remove anyone from the group requires choosing a new K_g and encrypting it

with each remaining member's K_p. Another is that P must explicitly assert its membership in every group G needed to satisfy the ACL, either by signing s with every K_g or by handing off from every K_g to the channel that carries s. Our system doesn't use this method.

6. Roles and programs

A principal often wants to limit its authority, in order to express the fact that it is acting according to a certain set of rules. For instance, a user may want to distinguish among playing an untrusted game program, doing normal work, and acting as system administrator. A node authorized to run several programs may want to distinguish running NFS from running an X server. To express such intentions we introduce the notion of *roles*.

If A is a principal and R is a role, we write A **as** R for A acting in role R. What does this mean? Since a role is a way for a principal to limit its authority, A **as** R should be a weaker principal than A in some sense, because a principal should always be free to limit its own authority. We define A **as** R to be $A|R$. This means that A **as** R **says** s is the same as A **says** R **says** s. Since A can make A **says** s for any s, it can certainly make A **as** R **says** s. Because | is monotonic, **as** is also.

We capture the fact that A **as** R is weaker than A by assuming that A speaks for A **as** R. Because adopting a role implies behaving appropriately for that role, A must be careful that what it says on its own is appropriate for any role it may adopt. Note that we are not assuming $A \Rightarrow A|B$ in general, but only when B is a role. Formally, we introduce a subset *Roles* of the principals and the axioms:

$\vdash A$ **as** $R = A \mid R$ for all $R \in$ *Roles* (R1)

$\vdash A \Rightarrow A$ **as** R for all $R \in$ *Roles* (R2)

$\vdash$ **as** is commutative and idempotent on roles (R3)

The last axiom makes it possible to write clearer and more concise ACLs; section 9 describes how the access checking algorithm uses it.

Acting in a certain way is much the same as executing a certain program. In this sense we can equate a role with a program. Here by a program we mean something that obeys a specification—several different program texts may obey the same specification and hence be the same program in this sense. How can a principal know it is obeying a program?

If the principal is a person, it can just decide to do so; in this case we can't give any formal rule for when the principal should be willing to assume the role. Consider the example of a user acting as system manager for her workstation. Traditionally (in Unix) she assumes this role by issuing the `su` command; this expresses her intention to issue further commands that are appropriate for the manager. In our system she assumes the role "user **as** manager". There is much more to be said about roles for users, enough to fill another paper.

If a machine is going to run the program, however, we can be more precise. One possibility that is instructive, though not at all practical, is to use the program text or image I as the role. So the node N can make N **as** I **says** s for a statement s made by a process running the program image I. But of course I is too big. A more practical method compresses I to a digest D small enough that it can be used directly as the role (see section 4). Such a digest distinguishes one program from another as well as the entire program text, so N can make N **as** D **says** s instead of N **as** I **says** s.

Digests are to roles in general much as encryption keys are to principals in general: they are unintelligible to people, and the same program specification may apply to several program texts (perhaps successive versions) and hence to several digests. In general we want the role to have a name, and we say that the digest speaks for the role. This consideration and the encoding of **as** by | both motivate us to treat roles as principals; they are a special kind of principal because a role never says anything on its own. Now we can express the fact that digest D speaks for the program named P by writing $D \Rightarrow P$.[14] There are two ways to use this fact. The receiver of A **as** D **says** s can use $D \Rightarrow P$ to conclude that A **as** P **says** s because **as** is monotonic. Alternatively, A can use $D \Rightarrow P$ to justify making A **as** P **says** s whenever program D asserts s.

So far we have been discussing how a principal can decide what role to assume. The principal must also be able to convince others. Since we are encoding A **as** P as $A|P$, however, this is easy. To make A **as** P **says** s, A just makes A **says** P **says** s as we saw earlier, and to hand off A **as** P to some other channel C it makes A **as** P **says** $(C \Rightarrow A$ **as** $P)$.

Loading programs

With these ideas we can explain exactly how to load a program securely. Suppose A is doing the loading. Usually A will be a node, that is, a machine running an operating system. Some principal B tells A to load program P; no special authority is needed for this except the authority to consume some of A's resources. In response, A makes a separate process pr to run the program, looks up P in the file system, copies the resulting program image into pr, and starts it up.

If A trusts the file system to speak for P, it hands off to pr the right to speak for A **as** P, using the mechanisms described in section 8; this is much like running a Unix `setuid` program. Now pr is a protected subsystem; it has an independent existence and authority consistent with the program it is running. B might hand off to pr some of the principals it can speak for. For instance, if B is a shell it might hand off its right to speak for the user that is logged in to that shell. Because pr can speak for A **as** P, it can issue requests to an object with A **as** P on its ACL, and the requests will be granted. Such an ACL entry should exist only if the owner of the object trusts A to run P.

If A doesn't trust the file system, it computes the digest D of the program text and looks up the name P to get credentials for $D \Rightarrow P$. Having checked these credentials it proceeds as before. A doesn't need to record the credentials, since no one else needs to see them; if you trust A to run P, you have to trust A not to lie to you when it says it is running P.

[14] Connoisseurs of program specification will find this formula familiar—it looks like the implication relation between an implementation and its specification. This is certainly not an accident.

It is often useful to form a group of programs, for instance, `/com/dec/src/trustedSW`. A principal speaking for this name can issue certificates $P \Rightarrow$ `/com/dec/src/trustedSW` for trusted programs P. If A **as** `/com/dec/src/trustedSW` appears on an ACL, any program P with such a certificate will get access when it runs on A because **as** is monotonic. Note that it's explicit in the name that `/com/dec/src` is certifying this particular set of trusted software. Virus prevention is one obvious application.

There can also be groups of nodes. An ACL might contain DBServers **as** Ingres; then if $A \Rightarrow$ DBServers (A is a member of the group DBServers), A **as** Ingres gets access because **as** is monotonic. If we extend these ideas, DBSystems can be a principal that stands for a group of systems, with membership certificates DBServers **as** Ingres $\Rightarrow$ DBSystems, Mainframes **as** DB2 $\Rightarrow$ DBSystems, etc.

Booting

Booting a machine is very much like loading a program. The result is a node that can speak for M **as** P, if M is the machine and P the name or digest of the program image that is booted. There are two interesting differences.

One is that the machine is the base case for authenticating a system, and it authenticates its messages by knowing a private key K_m^{-1} which is stored in non-volatile memory. Making and authenticating this key is part of the process of installing M, that is, putting it into service when it arrives. In this process M constructs a public key pair (K_m, K_m^{-1}) and outputs the public key K_m. Then someone who can speak for the name M, presumably an administrator, makes a certificate $K_m \Rightarrow M$. It is an interesting problem to devise a practical installation procedure.

The other difference is that when M (the boot code that gets control when the machine is reset) gives control to the program P that it boots (normally the operating system), M is handing over all the hardware resources of the machine, for instance any directly connected disks. This has three effects:

Since M is no longer around, it can't multiplex messages from the node on its own channels. Instead, M invents a new public key pair (K_n, K_n^{-1}) at boot time, gives K_n^{-1} to P, and makes a certificate K_m **says** $K_n \Rightarrow M$ **as** P. The key K_n is the node key described in section 4.

M needs some assurance that P can be trusted with M's hardware resources. It's enough for M to know the digests of trustworthy programs, or the public key that is trusted to sign certificates for these digests.

If we want to distinguish M itself from any of the programs it is willing to boot, then M needs a way to protect K_m^{-1} from these programs. This requires hardware that makes K_m^{-1} readable when the machine is reset, but can be told to hide it until the next reset. Otherwise one operating system that M loads could impersonate any other such system, and if any of them is compromised then M is compromised too.

You might think that all this is too much to put into a boot ROM. Fortunately, it's enough if the boot ROM can compute the digest function and knows one digest (set at installation time) that it trusts completely. Then it can just load the program P_{boot} with that digest, and P_{boot} can act as part of M. In this case, of course, M gives K_m^{-1} to P_{boot}.

7. Delegation

We have seen how a principal can hand off all of its authority to another, and how a principal can limit its authority using roles. We now consider a combination of these two methods that allows one principal to *delegate* some of its authority to another one. For example, a user on a workstation may wish to delegate to a compute server, much as she might `rlogin` to it in vanilla Unix. The server can then access files on her behalf as long as their ACLs allow this access. Or a user may delegate to a database system, which combines its authority with the delegation to access the files that store the database.

The intuitive idea of delegation is imprecise, but our formal treatment gives it a precise meaning; we discuss other possible meanings elsewhere [2]. We express delegation with one more operator on principals, B **for** A.[15] Intuitively this principal is B acting on behalf of A, who has delegated to B the right to do so. The basic axioms of **for** are:

$\vdash A \wedge B|A \Rightarrow B$ **for** A. (D1)

$\vdash$ **for** is monotonic and distributes over $\wedge$. (D2)

To establish a delegation, A first delegates to B by making

A **says** $B|A \Rightarrow B$ **for** A. (1)

We use $B|A$ so that B won't speak for B **for** A by mistake. Then B accepts the delegation by making

$B|A$ **says** $B|A \Rightarrow B$ **for** A. (2)

To put it another way, **for** equals delegation (1) plus quoting (2). We need this explicit action by B because when B **for** A says something, the intended meaning is that *both* A and B contribute, and hence both must consent. Now we can deduce

$(A \wedge B|A)$ **says** $B|A \Rightarrow B$ **for** A using (P1), (1), (2);

$B|A \Rightarrow B$ **for** A using (D1) and (P11).

In other words, B can speak for B **for** A just by quoting A.[16]

We use timeouts to revoke delegations. A gives (1) a fairly short lifetime, say 30 minutes, and B must ask A to refresh it whenever it's about to expire.

---

[15] We introduce **for** as an independent operator and axiomatize it by (D1-2) and some other axioms that make it easier to write ACLs:

$\vdash A$ **for** $(B$ **for** $C) \Rightarrow (A$ **for** $B)$ **for** C (half of associativity); (D3)

$\vdash (A$ **for** $B)$ **as** $R = A$ **for** $(B$ **as** $R)$. (D4)

However, **for** can be defined in terms of $\wedge$ and $|$ and a principal D whose purpose is to quote A whenever B does so. You can think of D as a "delegation server": A tells D that A is delegating to B, and then whenever $B|A$ **says** s, $D|A$ **says** s also. Now B **for** A is just short for $B|A \wedge D|A$. We don't want to implement D (if we did, it might be compromised). So A has to be able to do D's job; in other words, $A \Rightarrow D|A$. Formally, we add the axioms:

$\vdash B$ **for** $A = B|A \wedge D|A$ (D5)

$\vdash A \Rightarrow D|A$ (D6)

Now (D1)-(D4) become theorems. So do some other statements of more debatable merit. Our other paper goes into more detail [2].

[16] Using D, A can delegate to B by making A **says** $B|A \Rightarrow D|A$. When B wants to speak for B **for** A it can quote A and appeal to the joint authority rule (P12). This is simpler but less explicit.

Login

A similar scheme handles delegation from the user U to the workstation W on which she logs in. The one difference arises from the assumption that the user's key K_u is available only while she is logging in. This seems reasonable, since getting access to the user's key will require her to type her password or insert her smart card and type a PIN; the details of login protocols are discussed elsewhere [1, 25, 26]. Hence the user's delegation to the workstation at login must have a rather long lifetime, so that it doesn't need to be refreshed very often. We therefore use the joint authority rule (P12) to make this delegation require a countersignature by a temporary public key K_l. This key is made at login time and called the login session key. When the user logs out, the workstation forgets K_l^{-1} so that it can no longer refresh any credentials that depend on the login delegation, and hence can no longer act for the user after the 30 minute lifetime has expired. This protects the user in case the workstation is compromised after she logs out. If there is a threat that the workstation might be compromised within 30 minutes after a logout, then it should also discard its master key and node key at logout.

The credentials for login start with a long-term delegation from the user to $K_w \wedge K_l$ (here K_w is the workstation's node key), using K_u for A and K_w for the second B in (1):

$$K_u \textbf{ says } (K_w \wedge K_l)|K_u \Rightarrow K_w \textbf{ for } K_u.$$

K_w accepts the delegation in the usual way, so we know that

$$(K_w \wedge K_l)|K_u \Rightarrow K_w \textbf{ for } K_u,$$

and because | distributes over $\wedge$ we get

$$K_w|K_u \wedge K_l|K_u \Rightarrow K_w \textbf{ for } K_u.$$

Next K_l signs a short-term certificate

$$K_l \textbf{ says } K_w \Rightarrow K_l.$$

This lets us conclude that $K_w|K_u \Rightarrow K_l|K_u$ by the handoff rule and the monotonicity of |. Now we can apply (P12) and reach the usual conclusion for delegation, but with a short lifetime:

$$K_w|K_u \Rightarrow K_w \textbf{ for } K_u.$$

Long-running computations

What about delegation to a process that needs to keep running after the user has logged out, such as a batch job? We would still like some control over the duration of the delegated authority, and some way to revoke it on demand. The basic idea is to have a single highly available agent for the user that replaces the login workstation and refreshes the credentials for long-running jobs. The user can explicitly tell this agent which credentials should be refreshed. We have not worked out the details of this scheme; it is a tricky exercise in balancing the demands of convenience, availability, and security. Disconnected operation raises similar issues.

8. Authenticating inter-process communication

We have established the foundation for our authentication system: the theory of principals, encrypted secure channels, name lookup to find the channels or other principals that speak for a named principal, and compound principals for roles and delegation. This section explains the mechanics of authenticating messages from one process to another. In other words, we study how one process can make another accept a statement A **says** s. A single process must be able to speak for several A's; thus, a database server may speak for its client during normal operation and for itself during recovery.

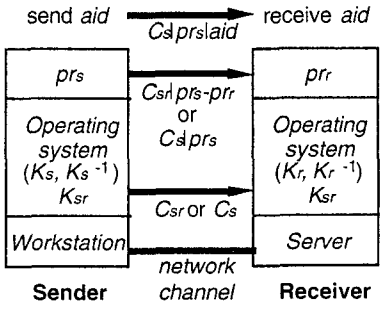

Figure 5. Multiplexing a node-to-node channel

We describe the mechanism in terms of messages from a sender to a receiver. It allows a message to be interpreted as one or more statements A **says** s. Our system implements remote procedure call, so it has call and return messages. For a call, statements are made by the caller (the client) and interpreted by the called procedure (the server); for a return, the reverse is true.

Most messages use a channel between a sending process on the sending node and a receiving process on the receiving node. This channel is made by multiplexing a channel C_{sr} between the two nodes, using the two process identifiers pr_s and pr_r as the multiplexing address, so it is $C_{sr}|pr_s\text{-}pr_r$; see figure 5. A shared key K_{sr} defines the node-to-node channel $C_{sr} = \text{DES}(K_{sr})$.

Henceforth we concentrate on the integrity of the channels, so we care only that the message comes from the sender, not that it goes to the receiver. Section 4 explains how to establish $\text{DES}(K_{sr}) \Rightarrow \text{RSA}(K_s)$, where K_s is the sending node's public key. So we can say that the message goes over $C_s|pr_s$ from the sending process, where $C_s = \text{RSA}(K_s)$. Some messages are certificates encrypted with K_s because they must be passed on to a third party that doesn't know K_{sr}; we indicate this informally by writing K_s **says** s instead of C_s **says** s.

It is obvious that we also get secrecy, as a byproduct of using shared keys. We could show this by the dual of the arguments we make for integrity, paying attention to the receiver rather than the sender.

The sender wants to communicate one or more statements A **says** s to the receiver, where A is some principal that the sender can speak for. Our strategy for doing this is to encode A as a number called an *authentication identifier* or *aid*, and to pass the *aid* as an ordinary integer. By convention, the receiver interprets a call like `Read(aid, file, ...)` as one or more statements C_{aid} **says** s, where $C_{aid} = C_s|pr_s|aid$. The receiving node supplies $C_s|pr_s$ to the receiver on demand. Recall that C_s is obtained directly from the key used to decrypt the message and pr_s is supplied by the sending node. The *aid* is supplied by the sending process. An *aid* is chosen from a large enough space that it is never reused during the

lifetime of the sending node (until the node is rebooted and its C_s changes); this ensures that a channel C_{aid} is never reused.

The motivation for this design is that the sending process doesn't need to involve the operating system in order to send C_{aid} **says** s to the receiver, because *aid* is just an integer. The only role of the operating system is to implement the channel $C_s|pr_s$ securely by labelling the message with the process pr_s that sends it. Thus a principal is passed as cheaply as an integer. There is also a one-time cost that we now consider.

The receiver doesn't actually care much about C_{aid}; it wants to interpret the message as A **says** s for some more meaningful principal A such as a user's public key. To do this it needs to know $C_{aid} \Rightarrow A$; we will call A the *meaning* of C_{aid}. There are two parts to this: finding out what A is, and getting a proof that $C_{aid} \Rightarrow A$ (that is, credentials for A). The receiver gets A and the credentials from the sender. Recall that the credentials consist of some premises C **says** $A' \Rightarrow B'$ plus the reasoning that derives $C_{aid} \Rightarrow A$ from the premises and the axioms. For the sender to transmit a premise to the receiver, either the sender must speak for C, or C must be a channel to the receiver. If C is a channel, it could be a public key channel, or a shared key channel with the receiver as one party, as in the Needham-Schroeder protocol [19]. We treat the former case here; section 4 explains how to use shared keys to simulate public keys.

The meaning A of C_{aid} is an expression whose operands are names or channels; in either case the credentials must prove that the sending system can speak for A. In our system all the operands of A are either roles or the public keys of nodes or of users. We saw in sections 6 and 7 how the sending system gets credentials for these keys as a result of booting or login. In figure 1, suppose the request has a as its *aid* and the user, workstation node, and workstation machine are U, W, and M. Then C_a is the principal making the request, and its meaning is $((K_m$ **as** OS) **as** AccountingApplication) **for** K_u. The credentials are:

K_m **says** $K_w \Rightarrow K_m$ **as** OS	From booting M.
K_u **says** $(K_w \wedge K_l)\|K_u \Rightarrow K_w$ **for** K_u	From U's login.
K_l **says** $K_w \Rightarrow K_l$	Also from login.
$K_w\|K_u$ **says** $C_a \Rightarrow ((K_m$ **as** OS) **as** AccountingApplication) **for** K_u	

The server gets certificates for the first three premises in the credentials. The last premise does not have a certificate but follows directly from a message on the shared key channel between W and the server, because this channel speaks for K_w. A system using the push model would replace some or all of the keys in the meaning with names and add additional certificates of the form K_{ca} **says** $K_p \Rightarrow P$ to the credentials.

The authentication agent

As we have seen, the credentials are a collection of certificates and statements from the sender, together with the connective tissue that assembles them into a proof of $C_{aid} \Rightarrow A$. The receiver gets them from the sender and caches them. In our system a component of the receiver's operating system called the

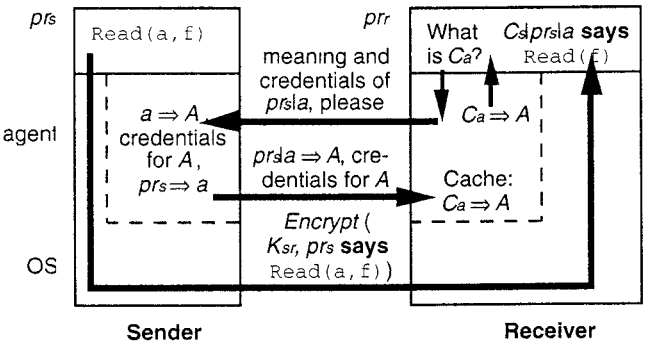

Figure 6. Messages to the agents for authenticating a channel

authentication agent does this work for the receiver. We can describe the agent's work under three headings.

The receiving process:

gets a message containing *aid*, interpreted as *aid* **says** s;

learns from its operating system that the message came on channel $C_s|pr_s$ (this is exactly like learning the source address of a message), so it believes $C_s|pr_s$ **says** *aid* **says** s which is the same as $C_s|pr_s|aid$ **says** s;

calls on its local agent to learn the principal A that $C_{aid} = C_s|pr_s|aid$ speaks for, so it believes A **says** s;

and perhaps caches the fact that $C_{aid} \Rightarrow A$ to avoid calling the agent again if it gets another message from C_{aid}.

The process doesn't need to see the credentials, since it trusts its agent to check them just as it trusts its operating system for virtual memory and the other necessities of life. The process does need to know their lifetime, since the information $C_{aid} \Rightarrow A$ that it may want to cache may be invalid after that time. Figure 6 shows communication through the agent.

The first job of the agent, acting for the receiver, is to maintain a cache of $C_{aid} \Rightarrow A$ facts and lifetimes like the cache maintained by its client processes. The agent answers queries out of the cache if it can. Because this is a cache, the agent can discard entries whenever it likes. If the information it needs isn't in the cache, it asks its partner on the sending node for the meaning and credentials of C_{aid}, checks the credentials it gets back, and caches the meaning.

The agent's second job, acting now for the sender, is to respond to these requests. To do this it keeps track of

the meaning A of each *aid* a that it is responsible for (note that a is local to the node, not a channel),

the certificates that it needs to make a's credentials, that is, to prove $C_s|a \Rightarrow A$, and

the processes that are allowed to speak for a (that is, the processes pr such that the agent believes $pr|a \Rightarrow a$ and hence is willing to authenticate $C_s|pr|a$).

An *authority* is an *aid* that a process speaks for. For a process to have an authority, its agent must have credentials to prove that some channel controlled by the agent speaks for the authority's meaning, and the agent must agree that the process speaks for the authority. Each process pr starts out with one authority, which it obtains by virtue of a user login or of the program P running in pr. In the latter case, for example, the

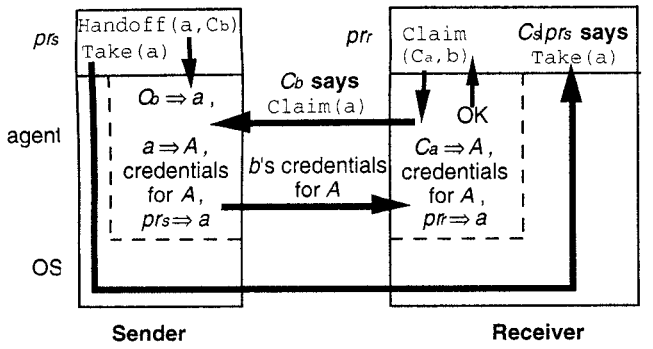

Figure 7: Messages to the agents for handing off an authority

node N loading P makes a new authority a, tells pr what it is, and records $a \Rightarrow N$ **as** P and $pr|a \Rightarrow a$.

The process can get its initial authority by calling `Self()`. If it has authorities a and b, it can get the authorities $a \wedge b$ by calling `And(a, b)` and a **as** r by calling `As(a, r)`. It can give up a by calling `Discard(a)`. What the agent knows about an authority is original information, unlike the cached facts $C_{aid} \Rightarrow A$. Hence the agent must keep it until all the processes that speak for the authority discard it or disappear.

The agent's third job is to hand off authority from one process to another. A sending process can hand off the authority a to another principal b by calling `Handoff(a, Cb)`; see figure 7. This is a statement to its local agent: a **says** $C_b \Rightarrow a$, where $C_b = C_r|pr_r|b$. The agent believes it because of (P10). The process can then pass a to the receiving process by sending it a message in the usual way, say by calling `Take(a)`. If pr_r has the authority b, it can obtain the authority a by calling `Claim(Ca,b)`. This causes the receiving agent to call the sending agent requesting its credentials for the meaning A of a (proof that $K_s|a \Rightarrow A$) plus the certificate $K_s|a$ **says** $K_r|a \Rightarrow A$. These are credentials that allow the receiving agent to speak for A. The certificate lets $K_r|a$ speak for A rather than a because the receiver needs to be able to prove its right to speak for the meaningful principal A, not the authentication identifier a. The certificate is directly signed by K_s (the sender's public key), rather than simply sent on DES(K_{sr}) (the shared key channel between sender and receiver) because the receiver needs something that it can pass on to a third party.

Claiming an authority has no effect unless you use it by passing it on to another process. So the claiming can be automatic: if a process pr passes on an authority a, the recipient asks for a's credentials, and pr hasn't claimed a, then pr's agent can claim it automatically if pr has the authority for b.

When is it appropriate to hand off an authority a? Doing this allows the recipient to speak for a as freely as you can, so you should do it only if you trust the recipient with a as much as you trust yourself. If you don't, you should hand off only a weaker authority like the delegation described in section 7.

Our system has two procedures for dealing with delegation, one for each of the certificates (1) and (2) in section 7. A process calls `For(a, Cb)` to delegate the meaning A of a to the meaning B of the principal C_b; this corresponds to making (1), which in this context is a **says** $C_b|a \Rightarrow B$ **for** A. Before calling `For` the process normally checks C_b against some ACL that expresses the principals to which it is willing to delegate.

Now the process can pass a to a receiver that has an authority b corresponding to C_b, and the receiver calls `Accept(a, b)` to obtain an authority that speaks for B **for** A. This call corresponds to claiming B **for** A, making (2), which in this context is $C_b|a$ **says** $b|a \Rightarrow B$ **for** A, and making $b|a$ **says** $r \Rightarrow B$ **for** A, where r is the result of the `Accept`. The sending agent supplies a certificate signed by its public key, $K_s|a$ **says** $K_r|a \Rightarrow B$ **for** A, along with a's credentials that prove $K_s|a \Rightarrow A$, just as in an ordinary handoff. The receiving agent can construct credentials for B **for** A based on the credentials it has for B, the claimed certificate and credentials, and the reasoning in section 7. So it can prove to others its right to speak for B **for** A.

You might feel that it's clumsy to require explicit action at both ends. After all, the ordinary handoff can be claimed automatically. But the two cases are not the same: in accepting the **for** and using the resulting authority, the receiver adds the weight of authority b to the authority from the sender. It should not do this accidentally.

What about revocation? The sending agent signs a handoff (or delegation) certificate that expires fairly soon, typically in about 30 minutes. This means that the handoff must be refreshed every 30 minutes by asking the sender for credentials again. If the sender's credentials in turn depend on a handoff from some other sender, the refresh will work its way up the chain of senders and back down. To keep the cost linear in the depth of handoff, we check all the certificates in a set of credentials whenever any one expires, and refresh those that are about to expire. This tends to synchronize the lifetimes.

Table 4 summarizes the state of the agent. Table 5 summarizes the interface from a process to its local agent.

There are many possible variations on the basic scheme described above. Here are some interesting ones:

Each thread can have an authority that is passed automatically in every call that the thread makes. This gets rid of most authority arguments, but is less flexible and explicit.

In the basic scheme authentication is symmetric between call and return; this means that each call can return the principal responsible for the result or hand off an authority. Often, however, the caller wants to authenticate the channel from the server only once. It can do this when it establishes the RPC binding if this operation returns an *aid* for the server's authority. This is called 'mutual authentication'.

Instead of passing certificates for all the premises of the credentials, the sending agent can pass the name of a place to find the certificates. This is especially interesting if that place is a trusted on-line server which can authenticate a channel from itself, because that server can then just assert the premise rather than signing a certificate for it. For example, in a system with centralized management there might be a trusted database server that stores group memberships. Here 'trusted' means that it speaks for these groups. This method can avoid a lot of public key encryption.

Key	(K_n, K_n^{-1}), the public key pair of this node.
Principal cache	A table mapping a channel $C_a = C_s\|pr_s\|a$ to A. An entry means the agent has seen credentials proving $C_a \Rightarrow A$; the entry also has a lifetime.
Author-ities	A table mapping an *aid* a to A, the principal that a speaks for. Credentials to prove this agent can speak for A. A set of local processes that can speak for a. A set of C_b that can speak for a.

Table 4: The state of an agent

Procedure	Meaning
`Self() : A`	
`Discard(a: A)`	
`And(a:A, b:A): A`	$a \wedge b$ **says** *result* $\Rightarrow a \wedge b$
`As(a:A, r:Role): A`	$a\|r$ **says** *result* $\Rightarrow a$ **as** r
`Handoff(a:A, b:C)`	a **says** $b \Rightarrow a$
`Claim(a:C, b:A): A`	Retrieve a **says** $b \Rightarrow a$; b **says** *result* $\Rightarrow a$
`For(a:A, b:C)`	a **says** $b\|a \Rightarrow b$ **for** a
`Accept(a:C,b:A): A`	Retrieve a **says** $b\|a \Rightarrow b$ **for** a ; $b\|a$ **says** $b\|a \Rightarrow b$ **for** a $\wedge$ *result* $\Rightarrow b\|a$

`CheckAccess(`
`acl: ACL, b: C,`
`op: Operation ):`
	Does *acl* grant b the right to do *op*?
	`Boolean`

Types: A for authority, represented as *aid*.
C for channel principal, which is $C_{aid} = C_s|pr_s|aid$.

Table 5: Programming interface from a process to its local agent

It's possible to send the credentials with the first use of a; this saves a round trip. However, recognizing the first use of a may be difficult. The callback mechanism is still needed for refreshing the credentials.

Granting access

Even a seemingly endless chain of remote calls will eventually result in an attempt to actually access an object. For instance, a call `Read(file f, authority a)` will be interpreted by the receiver as C_a **says** "read file f". The receiver obtains the ACL for f and wants to know whether C_a speaks for a principal that can have read access. To find this out the receiver calls `CheckAccess(f's acl, Ca, read)`, which returns true or false. Section 9 explains briefly how this works.

Pragmatics

The performance of our scheme depends on the cache hit rates and the cost of loading the caches. Each time a receiving node sees C_a for the first time, there is a miss in its cache and a fairly expensive call to the sender for the meaning and credentials. This call takes one RPC time (2.5 ms on our 2 MIPS processors) plus the time to check any certificates the receiver hasn't seen before (15 ms per certificate with 512 bit RSA keys). Each time a receiving process sees C_a for the first time, there is one operating system call time and a fast lookup in the agent's cache. Subsequently the process finds C_a in its own cache, which it can access in a few dozen instructions.

When lifetimes expire it's as though the cache was flushed. We typically use 30 minute lifetimes, so we pay less than 0.001% to refresh one certificate. If a node has 50 C_a's in constant use with two different certificates each, this is 0.1%. With the faster processors coming it will soon be much less.

The authentication agent could be local to a receiving process. Then the operating system wouldn't be involved and the process identifiers wouldn't be needed. We chose to put the agent in the operating system for a number of reasons:

When acting for a sender, the agent has to respond to asynchronous calls from receivers. Although the sending process could export the agent interface, we thought this would be too much machinery to have in every process.

An agent in the operating system can optimize the common case of authentication between two processes on the same node. This is especially important for handing off an authority a from a parent to a child process, which is very common in Unix. All the agent has to do is check that the parent speaks for a and add the child to the set of processes that speak for a. This can be implemented almost exactly like the standard Unix mechanism for handing off a file descriptor from a parent to a child.

The agent must deal with encryption keys, and cryptographic religion says that key handling should be localized as much as possible. Of course we could have put just this service in the operating system, at some cost in complexity.

Process-to-process encryption channels mean many more keys to establish and keep track of.

The operating system must be trusted anyway, so we are not missing a chance to reduce the size of the TCB.

9. Access control

Finally we have reached our goal: deciding whether to grant a request to access an object. We follow the conventional model of controlling access by means of an access control list or ACL which is attached to the object; see section 1.

We take an ACL to be a set of principals, each with some rights to the ACL's object.[17] The ACL grants a request A **says** s if A speaks for B and B is a principal on the ACL that has all the rights the request needs. So the reference monitor needs an algorithm that will generate a proof of $A \Rightarrow B$ (then it grants access), or determine that no such proof exists (then it denies access). This is harder than the task of constructing the credentials for a request, because there we are building up a principal one step at a time and building the proof at the same time. And it is much harder than checking credentials, because theorem proving is much harder than proof checking.

[17] A capability for an object can be viewed as a principal that is automatically on the ACL.

So it's not surprising that we have to restrict the form of ACLs to get an algorithm that is complete and runs reasonably fast.

There are doubtless many ways to do this. The one we have chosen is described by the following syntax for the principal in an ACL entry or a request:[18]

principal	::=	forList | principal ∧ forList
forList	::=	asList | forList **for** asList
asList	::=	properPrincipal | asList **as** role
role	::=	pathName
properPrincipal	::=	pathName | channel

The roles and the properPrincipals must be disjoint. In addition to A and a set of B's we also have as input a set of premises $P \Rightarrow Q$, where P and Q are properPrincipals or roles. The premises arise from group membership certificates or from path name lookup; they are just like the premises in credentials. Now there is an efficient algorithm to test $A \Rightarrow B$:

Each forList in B must have one in A that speaks for it.

One forList speaks for another if they have the same length and each asList in the first forList speaks for the corresponding asList in the second forList.

A **as** R_1 **as** ... **as** $R_n \Rightarrow B$ **as** R_1' **as** ... **as** R_m' if $A \Rightarrow B$ and for each R_j there is an R_k' such that $R_j \Rightarrow R_k'$.

One role or properPrincipal A speaks for another B if there is a chain of premises $A = P_0 \Rightarrow ... \Rightarrow P_n = B$.

Another paper discusses algorithms for access checking in more detail [2]. Our theory of authentication is compatible with other theories of access control, such as one in which the order of delegation hops (operands of **for**) is less important.

The inputs to the algorithm are the ACL, the requesting principal, and the premises. We know how to get the ACL (attached to the object) and the principal (section 8). Recall that because we use the pull model, the requesting principal is an expression in which every operand is either a role or a public key that is expected to speak for some named principal; section 8 gives an example. What about the premises? As we have seen, they can either be pushed by the sender or pulled from a database by the receiver. Our system pulls all the premises needed to authenticate a channel from a name, by looking up the name as described in section 5.

If there are many principals on the ACL or many members of a group, it will take too long to look up all their names. We deal with this by

attaching an integer hint called a *tag* to every named principal on an ACL or in a group membership certificate,

sending with the credentials a tag for each principal involved in the request, and

looking up only names whose tags appear in the request or which are specially marked to be looked up unconditionally (for instance, names of groups that are local to the receiver).

The tags don't have to be unique, just different enough to make it unlikely that two distinct named principals have the same tag. For instance, if the chance of this is less than .001 we will seldom do any extra lookups in a set of 500 names.

Note that if the tags are wrong, the effect is to deny access. Hence a request must claim membership in all groups that aren't looked up unconditionally, by including their tags. In particular, it must claim any large groups; they are too expensive to look up unconditionally. This is a small step toward the push model, in which a request must claim all the names that it speaks for and present the proof of its claims as well.

Reliably denying access to a principal is tricky in our system for two reasons:

Principals can have more than one name or key.

Certificates are stored insecurely, so we can't securely determine that a principal is *not* in a group because we can't count on finding the membership certificate.

The natural form of denial for us is an ACL modifier which means that the access checker should disbelieve a certificate for any principal that satisfies some property. For example, we can disbelieve certificates for a principal with a given name, or one with a given key, or one whose name starts with 'A', or one with a given tag (in which case the tags should be unique or we will sometimes deny access improperly). The idea behind this approach is that the system should be fail-secure: in case of doubt it should deny access. This means that it views positive premises like $A \Rightarrow B$ skeptically, negative ones like "deny Jim access" trustingly.

Of course we can also represent the entire membership of a group securely, either by entrusting it to a secure on-line server or by using a single certificate that lists all the members. But these methods sacrifice availability or performance, so it is best to use them only when the extra information is really needed.

Auditing

Our theory yields a formal proof for every access control decision. The premises in the proof are statements made on channels or assumptions made by the reference monitor (for instance the premise that starts off a name lookup). Every step in the proof is justified by one of a small number of rules. The proof can be written into the audit trail, and it gives a complete account of what access was granted and why. The theory thus provides a formal basis for auditing. Furthermore, we can treat intermediate results of the form $A \Rightarrow B$ as lemmas to be proved once and then referenced in other proofs. Thus the audit trail can use storage efficiently.

10. Conclusion

We have presented a theory that explains many known methods for authentication in distributed systems:

the secure flow of information in the Needham-Schroeder and Kerberos protocols;

authentication in a hierarchical name space;

[18] We can relax this syntax somewhat. Since **for** and **as** distribute over ∧ we can push any nested ∧ operators outward. Since (A **for** B) **as** C = A **for** (B **as** C) we can push any **as** operators inward into the second operands of **for**.

many variations in the paths along which bits are transmitted: from certification authority to sender to receiver, from certification authority directly to receiver, etc.;

lifetimes and refreshing for revoking grants of authority;

unique identifiers as partial substitutes for principal names.

It also explains a number of new methods used in our system:

secure loading of programs and booting of machines;

delegating authority in a way that combines and limits the power of both parties;

treating uniformly both certificates and on-line communication with authorities;

passing RPC arguments or results that are principals as efficiently as passing integers (after an initial startup cost) and refreshing their authority automatically;

taking account of roles and delegations in granting access;

countersigning a secure long-lived certificate with a refreshable short-lived one for rapid revocation.

The system is currently being implemented. The basic structure of agents, authentication identifiers, authorities, and ACLs is in place. Our operating system and distributed file system are both clients of our authentication and access control. This means that our ACLs appear on files, processes, and other operating system objects, not just on new objects like name service entries. Roles, node-to-node channel setup, process-to-process authentication, and delegation have all been demonstrated, and our implementation will soon be released as the default authentication system for the 50 researchers at SRC.

Work is in progress on secure loading and software support of network-controller based DES encryption. Although our current implementation does not make use of either composite or hierarchical principal names in ACLs, we expect to do future experiments in these areas.

Acknowledgements

Many of the ideas discussed here were developed as part of the Digital Distributed System Security Architecture [12, 13, 17] or greatly influenced by it. Morrie Gasser, Andy Goldstein, and Charlie Kaufman participated in that work. We benefited from discussions with Andrew Birrell and from comments by Morrie Gasser, Maurice Herlihy, Cynthia Hibbard, John Kohl, Tim Mann, Roger Needham, Greg Nelson, Fred Schneider, and Mike Schroeder.

References

1. M. Abadi, M. Burrows, C. Kaufman, and B Lampson. Authentication and delegation with smart-cards. To appear in *Theoretical Aspects of Computer Software*, Springer, 1991. Also research report 67, Systems Research Center, Digital Equipment Corp., Palo Alto, Oct. 1990.
2. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. To appear in *Proc. Crypto '91*, Springer, 1992. Also research report 70, Systems Research Center, Digital Equipment Corp., Palo Alto, March 1991.
3. A. Birrell, B. Lampson, R. Needham, and M. Schroeder. Global authentication without global trust. *Proc. IEEE Symposium on Security and Privacy*, Oakland, 1986, 223-230.
4. M Burrows, M. Abadi, and R. Needham. A logic of authentication *ACM Trans. Computer Systems* **8**, 1, Feb. 1990, 18-36.
5. CCITT. Information processing systems — Open systems interconnection — The directory authentication framework. CCITT 1988 Recommendation X 509. Also ISO/IEC 9594-8:1989.
6. P. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal* **28**, 4, 1990, 526-538
7. D. Davis and R. Swick. Network security via private-key certificates. *ACM Operating Systems Review* **24**, 4, Oct. 1990, 64-67.
8. D. Denning. A lattice model of secure information flow. *Comm. ACM* **19**, 5, May 1976, 236-243.
9. Department of Defense. *Trusted Computer System Evaluation Criteria*. DOD 5200.28-STD, 1985.
10. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Information Theory* **IT-22**, 6, Nov. 1976, 644-654.
11. H. Eberle, Systems Research Center, Digital Equipment Corp., Palo Alto. Private communication.
12. M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital distributed system security architecture. *Proc. 12th National Computer Security Conference*, NIST/NCSC, Baltimore, 1989, 305-319.
13. M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. *Proc. IEEE Symposium on Security and Privacy*, Oakland, 1990, 20-30.
14. B. Herbison. Low cost outboard cryptographic support for SILS and SP4. *Proc. 13th National Computer Security Conference*, NIST/NCSC, Baltimore, 1990, 286-295.
15. J. Kohl, C. Neuman, and J. Steiner. The Kerberos network authentication service. Version 5, draft 3, Project Athena, MIT, Oct. 1990.
16. B. Lampson. Protection. *ACM Operating Systems Review* **8**, 1, Jan. 1974, 18-24.
17. J. Linn. Practical authentication for distributed systems. *Proc. IEEE Symposium on Security and Privacy*, Oakland, 1990, 31-40.
18. National Bureau of Standards. *Data Encryption Standard*. FIPS Pub. 46, Jan. 1977.
19. R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Comm. ACM* **21**, 12, Dec. 1978, 993-999.
20. C. Neuman. Proxy-based authorization and accounting for distributed systems. Technical report 91-02-01, University of Washington, Seattle, March 1991.
21. R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM* **21**, 2, Feb. 1978, 120-126.
22. R. Rivest. The MD4 message digest algorithm TM 434, Laboratory for Computer Science, MIT, Oct. 1990.
23. J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Trans. Computer Systems* **2**, 4, Nov. 1984, 277-288.
24. M. Shand, P. Bertin, and J. Vuillemin. Resource tradeoffs in fast long integer multiplication. *2nd ACM Symposium on Parallel Algorithms and Architectures*, Crete, July 1990.
25. J. Steiner, C. Neuman, and J Schiller. Kerberos: An authentication service for open network systems. *Proc. Usenix Winter Conference*, Usenix Association, Berkeley, CA, Feb. 1988, 191-202.
26. J. Tardo and K. Alagappan. SPX. Global authentication using public key certificates. *Proc. 14th National Computer Security Conference*, NIST/NCSC, Baltimore, 1991.
27. V. Voydock and S. Kent Security mechanisms in high-level network protocols. *ACM Computing Surveys* **15**, 2, June 1983, 135-171.