Authentication in Distributed Systems: Theory and Practice

BUTLER LAMPSON, MARTÍN ABADI, MICHAEL BURROWS, and EDWARD WOBBER
Digital Equipment Corporation

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 delegated authority. The theory shows 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 security mechanisms. 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, program loading, delegation, access control, and revocation.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: General — *Security and Protection*, Distributed Systems; D.4.6 [**Operating Systems**]: Security and Protection— *access controls, authentication, cryptographic controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*authentication*; E.3 [**Data**]: Data Encryption

General Terms: Security, Theory, Verification

Additional Key Words and Phrases: Certification authority, delegation, group, interprocess communication, key distribution, loading programs, path name, principal, role, secure channel, speaks for, trusted computing base

1. INTRODUCTION

Most computer security uses the access control model [16], which provides a basis for secrecy and integrity security policies. Figure 1 shows the elements of this model:

—Principals: sources for requests.

—Requests to perform operations on objects.

A preliminary version of this paper appeared in the *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles.*
Authors' address: Digital Equipment Corp., Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301. Internet address: lampson@src.dec.com.

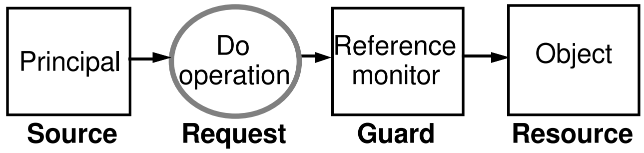

Fig. 1. The access control model.

—A reference monitor: a guard for each object that examines each request for the object and decides whether to grant it.

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

The reference monitor bases its decision on the principal making the request, the operation in the request, and an access rule that controls which principals may perform that operation on the object.[1]

To do its work the monitor needs a trustworthy way to know both the source of the request and the access rule. Obtaining the source of the request is called 'authentication'; interpreting the access rule is called 'authorization'. Thus authentication answers the question "Who said this?", and authorization answers the question "Who is trusted to access this?". Usually the access rule is attached to the object; such a rule is called an access control list or ACL. For each operation the ACL specifies a set of authorized principals, and the monitor grants a request if its principal is trusted at least as much as some principal that is authorized to do the operation in the request.

A request arrives on some channel, such as a wire from a terminal, a network connection, a pipe, a kernel call from 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, that is, it must authenticate the channel. This 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:

Autonomy: 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".

Size: The system may be much larger than a centralized one, and there may be multiple sources of authority for such tasks as registering users.

Heterogeneity: The system may have different kinds of channels that are secured in different ways. Some examples are encrypted messages, physi-

[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, so we have nothing to say about mandatory security policies that can enforce nondisclosure of secrets.

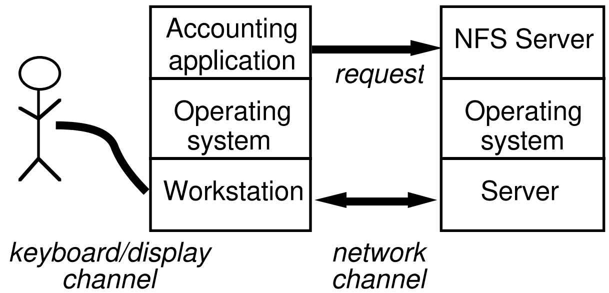

Fig. 2. A request from a complex source.

cally secure wires, and interprocess communication done by the operating system.

Fault-tolerance: Some parts of the system may be broken, off line, or otherwise inaccessible, but the system is still expected to provide as much service as possible. This is more complicated than a system which is either working or completely broken.

This paper describes both 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 how to state them precisely 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 channels based on encryption and the hardware and local operating system on each node; we assume these are trusted. 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.

We motivate our design throughout the paper with a practical example of a request that has a complex source involving several different system components. Figure 2 shows the example, in which a user logs in to a workstation and runs a protected subsystem that makes a 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 granting access.

The next section introduces the major concepts that underlie 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, a description of how our system solves the problem, and comments on the major alternatives known to us. 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 secure interprocess 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 state of our implementation.

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 the concepts precisely and treat them in detail.

If *s* is a *statement* (request, assertion, etc.) authentication answers the question "Who said *s*?" with a principal. Thus principals make statements; this is what they are for. Likewise, if *o* is an object authorization answers the question "Who is trusted to access *o*?" with a principal. 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.`

A channel is the only kind of principal that can directly make statements to a computer. There is no direct path, for example, from a person to a program; communication must be over some channel, involving keystrokes, wires, terminal ports, networks, 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	`BurrowsWS` **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` may carry UDP channels to ports `2`, `75`, and `443`, or a channel implemented by a kernel call trap from a user process may carry interprocess 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 example, an interprocess channel may carry many subchannels to various remote procedures.

Hierarchical 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 5.2 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.[2] Since only a channel can make a statement directly, a principal can make a statement only by making it on some channel that speaks for that principal.

We use 'speaks for' to formalize indirection; since any problem in computing can be solved by adding another level of indirection,[3] there are many uses of 'speaks for' in our system. 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

[2] Of course the notion of speaking for a group can have many other meanings. For instance, speaking for the U.S. Congress requires the agreement of a majority of both houses obtained according to well-defined procedures. We use only the simplest meaning in this paper: every member speaks for the group.

[3] Roger Needham attributes this observation to David Wheeler of Cambridge University.

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 and the relevant encryption mechanism and keys are included in 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 for the message and the channel on which it is transmitted are not part of the TCB. These are examples of an end-to-end argument [24], 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 should be 'fail-secure': if an untrusted component fails, the system may deny access it should have granted, but it won't 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 make the information valid. 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]. A reader who knows the authentication logic of Burrows, Abadi, and Needham [4] will find some similarities here, but its scope is narrower and its treatment of the matters within that scope correspondingly more detailed. For instance, secrecy and timeliness are fundamental there; neither appears in our theory.

To help readers who dislike formulas, we highlight the main results by boxing them. 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.

3.1 Statements

Statements are defined inductively as follows:

—There are some primitive statements (for example, "read file `foo`").[4]

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

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

—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 ″ s to mean that s is an axiom of the theory or is provable from the axioms (we mark an axiom by underlining its number) . Here are the axioms for statements:

If s is an instance of a theorem of propositional logic then ″ s. (<u>S1</u>)
For instance, ″ $s \wedge s' \supset s$.

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

″ (A **says** $s \wedge A$ **says** $(s \supset s')$) $\supset A$ **says** s'. (<u>S3</u>)
This is modus ponens for **says** instead of ″.

If ″ s then ″ A **says** s for every principal A. (<u>S4</u>)

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

[4] We want all statements to have truth values, and we give a truth value to an imperative statement like "read file `foo`" by interpreting it as "it would be a good thing to read file `foo`".

″ A **says** $(s \wedge s') \equiv (A$ **says** $s) \wedge (A$ **says** $s')$ (S5)

The intuitive meaning of ″ 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 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 another principal to infer B **says** s. For example, A can make A **says** s by sending s on a channel known to speak for A.

3.2 Principals

In our theory 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. 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:

$\boxed{″\ (A \wedge B) \textbf{ says } s \equiv (A \textbf{ says } s) \wedge (B \textbf{ says } s)}$ (P1)

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

$\boxed{″\ (A \mid B) \textbf{ says } s \equiv A \textbf{ says } B \textbf{ says } s)}$ (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:

″ $A = B \supset (A$ **says** $s \equiv B$ **says** $s)$ (P3)

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

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

″ $|$ is associative. (P5)

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

Now we can define $\Rightarrow$, the 'speaks for' relation between principals, in terms of $\wedge$ and $=$:

″ $(A \Rightarrow B) \equiv (A = A \wedge B)$ (P7)

and we get some desirable properties as theorems:

$\boxed{″\ (A \Rightarrow B) \supset ((A \textbf{ says } s) \supset (B \textbf{ says } s))}$ (P8)

This is the informal definition of 'speaks for' in Section 2.

″ $(A = B) \equiv ((A \Rightarrow B) \wedge (B \Rightarrow A))$ (P9)

Equation (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]. Or 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`. Only then can we reason from the two requests to `SRC` $\wedge$ `Manager` **says** "read from `foo`", a request that the ACL obviously grants.

For the same reason, the **as** and **for** operators defined in Sections 6 and 7 are also monotonic.

3.3 Handoff and Credentials

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

$$\forall\, (A \textbf{ says } (B \Rightarrow A)) \supset (B \Rightarrow A) \qquad \text{(P10)}$$

In other words, A has the right to allow any other principal B to speak for it. [5] 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 could 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:

$$\boxed{\forall\, ((A' \Rightarrow A) \wedge A' \textbf{ says } (B \Rightarrow A)) \supset (B \Rightarrow A)} \qquad \text{(P11)}$$

Proof: 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 basis 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:

$$\boxed{\forall\, ((A' \wedge B \Rightarrow A) \wedge (B \Rightarrow A')) \supset (B \Rightarrow A)} \qquad \text{(P12)}$$

[5] 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.

From this and (P10) we can deduce $B \Rightarrow A$ given A **says** $(A' \wedge B \Rightarrow A)$ and A' **says** $B \Rightarrow A'$. Thus A can let A' and B speak for it jointly, and A' can let B exercise this authority alone. One situation in which we might want both A and A' is when A is usually off line and therefore makes its statement with a much longer lifetime than A' does. We can think of the statement made by A' as a countersignature for A's statement. (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 conclusion, and therefore 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.

The appendix collects all the axioms of the theory so that the reader can easily survey the assumptions we are making.

4. CHANNELS AND ENCRYPTION

As we have seen, the essential property of a channel is that its statements can be taken as assumptions: formulas like C **says** s are 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 possible 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 handoff 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 only A can send messages on C.

Our treatment is informal. We give some methods of using encryption and some reasons to believe that these methods justify statements of the form "a channel implemented by DES encryption and decryption using key `#8340923` speaks for `lampson`". We do not, however, try to state precise assumptions about secrecy of keys and properties of algorithms, or to derive such facts about 'speaks for' from them. These are topics for another paper.

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 that carries this message speaks for A" won't do, because it can be subverted by copying the message to another channel. As we consider various 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; different values of X

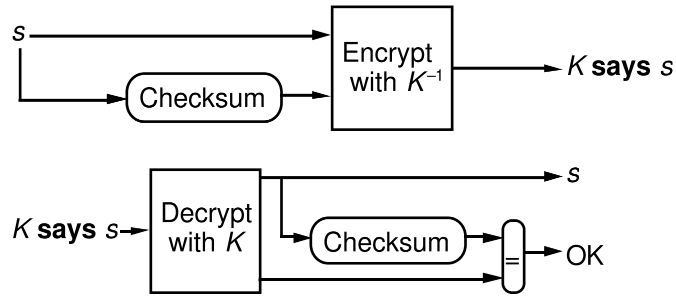

Fig. 3. Using encryption and checksums for integrity.

establish different subchannels. By (P2), C **says** X **says** s is the same thing as $C|X$ **says** s. Thus if C names the channel, $C|X$ names the subchannel. We will see many examples of this.

In what follows we concentrate on the flow of statements over secure channels and on the state that each principal must maintain. Except in Section 8, we gloss over many details that may be important for performance but are not directly relevant to security, such as the insecure messages that are sent to initiate some activity and the exact path traversed by the bits of an encrypted message.

4.1 Encryption

We are mainly interested in channels that depend on encryption 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 consists of 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 $\{x\}_{K^{-1}}$.

An encryption algorithm that is useful for computers provides a channel: for any message x, $Decrypt(K, Encrypt(K^{-1}, x)) = x$. The algorithm 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, you can't compute x.

Integrity: If you choose x but don't know K^{-1}, 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*, as shown in Figure 3. The checksum should provide enough redundancy to make it very unlikely that decrypting with K will succeed on a message not produced by encrypting with K^{-1}. Achieving this property requires some care [10, 28].

For integrity it is enough to encrypt a *digest* of the message. The digest is the result of applying a function *Digest* to the message. The *Digest* function produces a result of some fixed moderate size (typically 64 or 128 bits) no matter how large the message is. Furthermore, it is a one-way function; this means that you can't invert the function and compute a message with a given digest. Two practical digest functions are MD4 [22] and MD5[23].

An algorithm that provides integrity without 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 reading the contents of an encrypted message, and can't compromise integrity by changing it. Thus the handling of an encrypted message is not part of the TCB, since security does not depend on it. Of course the availability of an encryption channel does depend on how the network handles messages; we have nothing to say about the vulnerability of the network to disruption.

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 between groups of principals that trust each other completely, at least as far as messages on K are concerned. The most popular shared key encryption scheme is the Data Encryption Standard or DES [18]. We denote an encryption channel with the DES key K by DES (K), or simply by K when the meaning is clear; the channel 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.[6] Together, K and K^{-1} are called a key pair. The most popular public key encryption scheme is Rivest-Shamir-Adleman or RSA [21]. 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.[7] We denote an encryption channel with the RSA public key K by RSA (K), or simply by K when the meaning is clear; the channel speaks for the principal that knows K^{-1}.

Table I shows that encryption need not slow down a system unduly. It also shows that shared key encryption is about 1000-5000 times faster than public key encryption when both are carefully implemented. Hence the latter is usually used only to encrypt small messages or to set up a shared key.

4.2 Encryption Channels

With this background we can discuss how to make a practical channel from an

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

[7] To send a message with both integrity and secrecy, encrypt it both with K^{-1}_{sender} so that the receiver can decrypt with K_{sender} to verify the integrity, and with $K_{receiver}$ so that the receiver (and only the receiver) will be able to decrypt with $K^{-1}_{receiver}$ to read the message. But public key encryption is not normally used in this way.

Table I. Speeds of Cryptographic Operations [8]

	Hardware, bits/sec	Software, bits/sec/ MIPS	Notes
RSA encrypt	220 K [25]	.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

encryption algorithm. From the existence of the bits *Encrypt* (K^{-1}, s) any one who knows K 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. We often 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.

How can we name an encryption channel? One possibility is to use the key as a name, but we often 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.

The receiver needs to know what key K it should use to decrypt a message (of course it also needs to know what principal K speaks for, but that is a topic for later sections). 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 does decrypt the message correctly. If K is a shared key we can't include it with the message because K must remain secret. But we can include a *key identifier* that allows the receiver to know what the key is but doesn't disclose anything about it to others.

To describe our methods precisely we need some notation for keys and key identifiers. 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 formula K^r **says** s denotes a pair: the statement K **says** s together with a hint K^r to R about the key that R should use to check the signature. Concretely, K^r **says** s is the pair (*Encrypt* (K^{-1}, s), K^r). Thus the r doesn't affect the meaning of the

[8] 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.

statement at all; it simply helps the receiver to decrypt it. This help is sometimes essential to the functioning of a practical protocol.

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), where K_{rm} is a master key that only R knows,

—a pair (K'^{r}, *Encrypt* (K', K)), where K'^{r} is a key identifier for the 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, since it is established practice to distribute session keys encrypted by master keys [19, 26, 28].

4.3 Broadcast Encryption Channels

We conclude the general treatment of encryption channels by explaining the special role of public keys, and showing how to get the same effect using shared 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, so that certification authorities can be off line. 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. Relays use the key identifiers introduced above, and the explanation here depends on the notation defined there.

Since R simulates public key encryption, we assume that any principal A can get a channel to R. This channel is a key shared by R and A along with key identifiers for both parties. To set it up, A goes to a trusted agent,[9] which

[9] Even with public key encryption A generally needs a trusted agent to get a certificate for its

Table II. Simulating Public Key with Shared Key Encryption Using a Relay

	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 $K^a = (K_a^a, \mathit{Encrypt}(K_a, K))$, $K^b = (K_b^b, \mathit{Encrypt}(K_b, K))$.

may be off line, and talks to it on some physically secure channel such as a hard-wired keyboard and display or a dedicated RS-232 line. The agent makes up a key K and tells A both K and K^r, R's key identifier for K; to make K^r the agent must know R's master key. Finally, A constructs K^a, its own key identifier for K. Now A has K^{ar}, its two-way channel to R.

Given both K_a^r **says** s (a message encrypted by a shared key K_a together with R's key identifier for K_a) and K_b^{br} (the pair (K_b^b, K_b^r), which constitutes a two-way shared-key channel between R and some B with the shared key K_b), the relay R will make $K_b^b \mid K_a^r$ **says** s on demand. The relay thus multiplexes all the channels it has onto its channel to B, indicating the source of each message by the key identifier. 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. Thus it makes the channel K_a^r into a broadcast channel, and 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. B's name for the channel from A is $K_b^b \mid K_a^r$.

There is an added complication for authenticating a channel. 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 unless X happens to be B. 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

public key, although A's interaction with the agent need not keep any secrets; see Section 5.1.

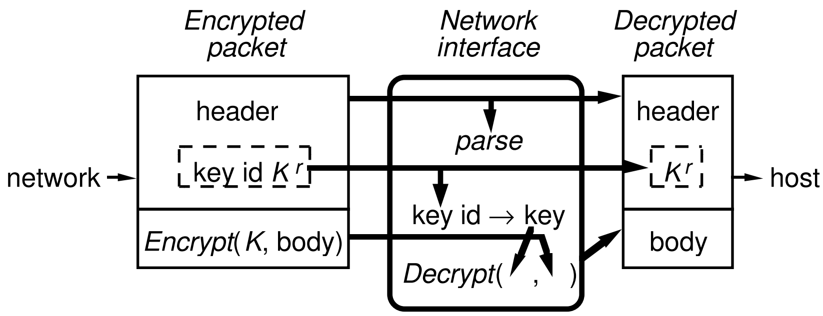

Fig. 4. Fast decryption.

II; 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 $K_b{}^b|\ K_{aa}{}^r$ **says** $K^{ab} \Rightarrow A$, just what B needs for authenticated communication with A. For two-way authentication, R needs $K_{aa}{}^r$ **says** $K_b{}^{br} \Rightarrow B$ instead of just $K_b{}^{br}$; from this it can make the symmetric certificate $K_a{}^a|\ K_{aa}{}^r$ **says** $K^{ab} \Rightarrow B$.

Table II 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 its minimal state, it is practical to make such a relay highly available as well as highly secure.

Even with this simulation, shared key encryption is not as secure as public key: if the relay is compromised then existing shared keys are compromised too. With public key encryption a certificate like K_{aa} **says** $K_a \Rightarrow A$ which authenticates the key K_a can be both issued and used without disclosing K_a^{-1} to anyone. Public key also has a potential performance advantage: there is no common on-line agent that must provide service in order for authenticated communication to be established between A and B. The simulation requires the relay to be working and not overloaded, and all other schemes that use shared keys share this property as well.

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

4.4 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

Table III. A's View of Node-to-Node Channel Setup; B's is Symmetric

	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}

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 4 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; after the node sends or receives a message we want to limit the time during which an adversary that compromises the node can read the message. We also need a way to efficiently change the individual node-to-node channel 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. The integrity checksum acts as 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, 26], which combine key exchange with authentication, but is similar to the Diffie-Hellman key exchange protocol [10].

We set up a node-to-node channel between nodes A and B in three phases; see Table III. 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 $K = Hash(J_a, J_b)$,[10] makes a key

[10] *Hash* is a commutative one-way function. We use it to prevent a chosen-plaintext attack on a master key and to keep K secret even if one of the J's is disclosed. It takes the compromise of both J_a and J_b (presumably as a result of the compromise of both K_a^{-1} and K_b^{-1}) for an adversary

identifier for K, and sends it 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.[11] Each node can make its own key identifier however it likes; for concreteness, Table III shows it being done by encrypting K with the node's master key.

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 $K \Rightarrow K_b$.[12] 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** "The key with digest $D \Rightarrow K_b$", where $D =$ *Digest* (K).

Now whenever A sees K **says** s, it can immediately conclude K_b **says** s. Thus when A receives a message on channel K, which changes whenever there is rekeying, it also receives the 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, running the protocol again makes a new DES channel that speaks for the same public key on each node. Thus the DES channel behaves like a cache entry; it can be discarded at any time and later re-established transparently.

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 `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.[13]

5. PRINCIPALS WITH NAMES

When users refer to principals they must do so by names that make sense to people, since users can't understand alternatives like unique identifiers or

to be able to compute K, even if it sees the key establishment messages.

[11] 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.

[12] 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 we ignore this complication. There are protocols in use that encode this multiplexing in strange and wonderful ways.

[13] Alternatively, the node could 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 that can serve as the name.

keys. Thus an ACL must grant access to named principals.[14] 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. But the credentials don't depend on the method. We describe them for the case we actually implement, where C is a public key.

5.1 A Single Certification Authority

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 that 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 and indexed by A. This database is usually called a name service; it 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 5 shows what B does, and the symmetric left side shows what A does to establish two-way authentication.

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

[14] Anonymous principals on ACLs are sometimes useful. For instance, a numbered bank account or highway toll account might grant its owner access by having on its ACL a public key negotiated when the account is established. But usually human review of the ACL must be possible.

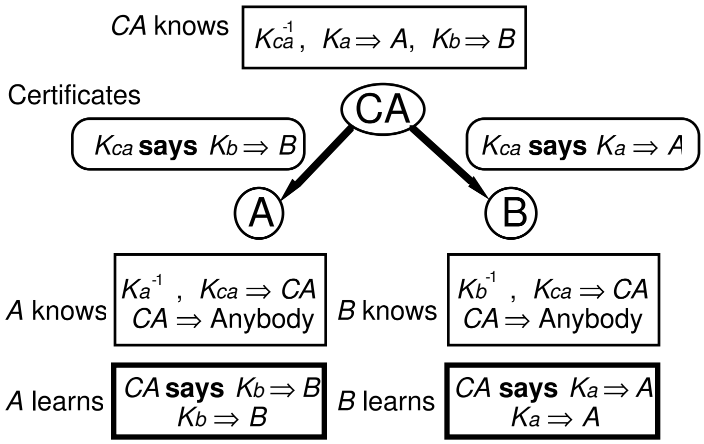

Fig. 5. Authenticating channels with a single certification authority.

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.

As we have seen, 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 we 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 countersigns this by making $O|K_a$ **says** $K_a \Rightarrow O|K_a$. From these two, $K_{ca} \Rightarrow A$, and (P12) we 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. Note that CA makes a separate certificate for each K_a it authenticates, and each such certificate makes it possible for O to convince a third party that $K_a \Rightarrow A$ only for specific values of K_a and A. Thus the TCB for granting access is just CA, because O acting on its own can't do anything, but CA speaks for A; the TCB for revocation is CA and O, since either one can prevent access from being revoked.

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 this version 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 presence of the certificate K_{ca} **says** $(O|K_a \wedge K_a) \Rightarrow A$ on the channel C_o as an encoding of the statement $C_o|K_a$ **says** $K_a \Rightarrow O|K_a$. Because $C_o \Rightarrow O$, this implies $O|K_a$ **says** $K_a \Rightarrow O|K_a$, which 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.

A straightforward alternative to an on line agent that asserts $O| K_a$ **says** $K_a \Rightarrow O| K_a$ is a 'black-list' agent or recent certificate that asserts "all of CA's certificates are valid except the ones for the following keys: K_1, K_2, ..." [5]. For obvious reasons this must be said in a single mouthful. Such revocation lists are used with Internet privacy-enhanced mail.

Changing a principal's key is easy. The principal chooses a new key pair and tells the certification authority its public key. The authority issues a new certificate and installs it in the database. If the key is being rolled over routinely rather than changed because of a suspected compromise, it may be desirable to leave the old certificate in the database for some time. Changing the authority's key is more difficult. First the authority chooses a new key pair. Then it writes a new certificate, signed by the new key, for each existing certificate, and installs the new certificates in the database. Next the new public key is distributed to all the clients; when a client gets the new key it stops trusting the old one. Finally, the old certificates can be removed from the database. During the entire period that the new key is being distributed, certificates signed by both keys must be in the database.

The formalization of Figure 5 also describes the Kerberos protocol [15, 26]. Kerberos uses shared rather than public key encryption. Although its designers didn't know about the relay simulation described in Section 4.3, the protocol can be explained as an application of that idea to public key certificates. Here are the steps; they correspond to the union of Figure 5 and Table II. First A gets from CA a certificate $K_{ca}{}^{r}$ **says** $K_a{}^{ar} \Rightarrow A$.[15] Kerberos calls CA the 'authentication server', the certificate a 'ticket granting ticket', and the relay R the 'ticket granting server'. The relay also has a channel to every principal that A might talk to; in particular R knows $K_b{}^{br} \Rightarrow B$.[16] 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',[17] 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, application programs normally use Kerberos to authenticate network connections, which the applications then rather unrealistically treat as secure channels. To do this, A makes K^{b} **says** $ci_a \Rightarrow A$, where ci_a is A's net-

---

[15] 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 today is usually derived from A's password.

[16] 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 motivated by the application for which Kerberos was originally designed, in which A is a workstation with plenty of cycles while B is a busy server. It is justified by the notion that there are only a few servers and they are friendly 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 give it to R, which can now be symmetric and stateless.

[17] 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.

work 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 analyze.

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. Because the statements in the authentication protocol are not imperative, it is not necessary to guarantee at-most-once delivery for the messages of the protocol, but it is important to ensure that statements were made recently enough. Furthermore, when the protocol is used to authenticate a channel that does carry imperative statements, it is necessary to guarantee at-most-once delivery on that channel.

The same techniques are used (or misused) for both security and communication, sometimes under different names: 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. It also uses the fact that the shared key channel between two nodes depends on two random numbers, one from each node; therefore each node knows that any message on the channel was sent since the node chose its random number. The details are not new [4], and we omit them here.

5.2 Path Names and Multiple Authorities

In a large system 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. There are many ways to do this, varying in how hard they are to manage and in which authorities a principal must trust to authenticate different parts of the name space.

If the name space is a tree, so that names are path names, it is natural to arrange the certification authorities in a corresponding tree. The lack of global trust means that a parent cannot unconditionally speak for its children; if it did, the root would speak for everyone. 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 along the shortest path from B to A, which runs up to the least common ancestor of B and A and back down to A. Figure 6 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]; it is also implemented in SPX [27].

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

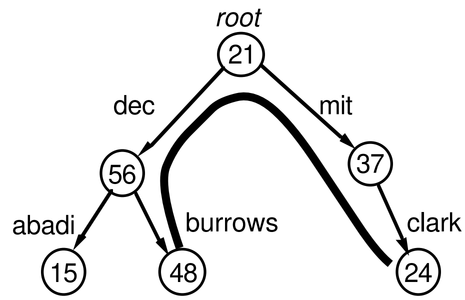

Fig. 6. Authentication with a tree of authorities

We follow the usual convention and separate the simple names by '/' symbols.[18] Informally, P **except** N is a principal that speaks for any path name that is an extension of P as long as the first name after P isn't N, and for any prefix of P as long as N isn't '..'. The purpose of **except** is to force a traversal of the authority tree to keep going outward, away from its starting point. If instead the traversal could retrace its steps, then a more distant authority would be authenticating a nearer one, contrary to our idea that trust should be as local as possible. The axioms for **except** are:

$\vdash P$ **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 by (N3), because of the **except** '..').[19]

$\vdash M \neq$ '..' $\supset (P/N$ **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 by (N2), because of the **except** N).

The quoting principals on the left side of $\Rightarrow$ prevent something asserted by P **except** M from automatically being asserted by all the longer path names. Note that usually both (N2) and (N3) apply. For instance, `/dec` **except** `burrows` speaks for `/dec/abadi` **except** '..' by (N2) and for `/` **except** `dec` by (N3).

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

[18] We follow Unix conventions and write / for the root rather than the empty string that our axioms produce.
[19] By putting several names after the **except** rather than one, we could further constrain the path names that a principal can authenticate.

believe $C_b \Rightarrow B$ **except** N. This could be anyone, but it's simplest to let B be the authenticating party. In Figure 6 this is `/dec/burrows`, so initially we believe $C_{burrows} \Rightarrow$ `/dec/burrows` **except** `nil`, and this channel is trusted to authenticate both up and down. In other words, Burrows knows his name and his public key and trusts himself.[20] 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 $C' \Rightarrow B'$. Then (N1) yields $C_{clark} \Rightarrow$ `/mit/clark`, which authenticates the channel 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[21] plus a message on some channel from an on line revocation agent; see Section 5.1. 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. Note that `clark` would not accept the bodies of *any* of the certificates on the path from `burrows`. Also, the intermediate results of this authentication differ 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 is because `burrows` and `clark` have different ideas about how much to trust `dec`.

[20] You may find it more natural to assume that Burrows knows the name and public key of his local certification authority. This corresponds to initially believing $C_{dec} \Rightarrow$ `/dec` **except** `nil`.

[21] 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. Thus one certification authority can certify names in several directories.

It's neither necessary nor desirable to include the entire path name of the principal in each child 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 child certificate is

C_{mit} | `clark` **says**
"For any path name P, if $C_{mit} \Rightarrow P$ then $C_{clark} \Rightarrow P$`/clark` **except** '..'."

In other words, the `mit` certification authority is willing to authenticate C_{clark} as speaking for `clark` relative to C_{mit} or to any name that C_{mit} might speak for; the authority takes responsibility only for names relative to itself. The corresponding assertion in a parent certificate, on the other hand, is a mistake. It would be

C_{mit} | '..' **says**
"For any path name P/N, if $C_{mit} \Rightarrow P/N$ then $C_{root} \Rightarrow P$ **except** N."

Since `mit`'s parent can change as a result of renaming higher in the tree, this certificate, which does not distinguish one parent from another, is too strong.

Our method for authenticating path names using the (N) axioms requires B to trust each certification authority on the path from B up to the least common ancestor and back down to A. If the least common ancestor is lower in the tree then B needs to trust fewer authorities. We can make it lower 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 more system management: the cross-link has to be installed, and it also has to be changed when `mit`'s key changes. Note that although the tree of authorities has become a directed acyclic graph, the least-common-ancestor rule still applies, so it's still 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 takes one RPC to each database in both pull and push models; the only difference is whether receiver or sender does the calls. If certificates from several authorities are stored in the same database, a single call can retrieve several of them. Either end can cache retrieved certificates; 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.

A principal doing a lookup might have channels from several other principals instead of the single channel C_b from itself that we described. Then it could start with the channel from the principal that is closest to the target A and thus reduce the number of intermediaries that must be trusted. This is essential if the entire name space is not connected, for instance if it is a forest with more than one root, since with only one starting point it is only possible to reach the names in one connected component of the name space. Each starting point means another built-in key, however, and maintaining these keys obviously makes it more complicated to manage the system. This is why our system doesn't 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, 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 by giving each principal a unique identifier that remains the same in spite of name changes. 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 identifier 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 maybe it is easier to handle. Our system avoids these complications by using the pull model throughout.

5.3 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 its members. Looking up a group name G yields one or more group membership certificates K_{ca} **says** $P_1 \Rightarrow G$, K_{ca} **says** $P_2 \Rightarrow G$, ..., where $K_{ca} \Rightarrow G$, just as the result of looking up an ordinary principal name P is a certificate for its channel K_{ca} **says** $C \Rightarrow P$, where $K_{ca} \Rightarrow P$. 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. If there were just one membership certificate for the whole group, it would be possible to prove nonmembership, but that approach has severe drawbacks: the certificate for a large group is large, and it must be replaced completely every time the group loses or gains a member.

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_{ca} **says** $K_g \Rightarrow G$, and to 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 one certificate and one 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. A third is that the method doesn't work for principals that don't have permanent secret keys, such as roles or programs. 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 do we want this to 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. One way for A to express the fact that it is acting in role R when it says s is for A to make A **says** R **says** s. This idea motivates us to treat a role as a kind of principal and to define A **as** R to be $A | R$, so that A **as** R **says** s is the same as A **says** 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 simple principals and the axioms:[22]

$$\text{"}\ A \ \textbf{as}\ R = A \mid R \qquad \text{for all } R \in Roles \tag{R1}$$

$$\text{"}\ A \Rightarrow A \ \textbf{as}\ R \qquad \text{for all } R \in Roles \tag{R2}$$

Acting in a certain way is much the same as executing a certain program. This suggests that 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 does this by issuing a `su` command, which expresses her intention to issue further commands that are appropriate for

[22] A third axiom allows us write clearer and more concise ACLs, and it has no apparent bad effects:

$$\text{"}\ \textbf{as} \text{ is commutative and idempotent on roles} \tag{R3}$$

Section 9 describes how the access checking algorithm uses this axiom.

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 does, 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. Now we can express the fact that digest D speaks for the program named P by writing $D \Rightarrow P$.[23] 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 $A \mid P$, 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** $P)$.

6.1 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 or in the treatment of booting below; 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. 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. In some cases 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.

[23] 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.

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. There's no need for A 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.

It is often useful to form a group of programs, for instance, `/com/dec/src/trustedSW`. A principal speaking for this name, for example, the key K_{ca} of its certification authority, can issue a certificate K_{ca} **says** $P \Rightarrow$ `/com/dec/src/trustedSW` for a trusted program 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 control is one obvious application. To certify a program as virus-free we compute its digest D and issue a membership certificate K_{ca} **says** $D \Rightarrow$ `trustedSW` (from now on we elide `/com/dec/src/`). There are two ways to use these certificates:

- When A loads a program with digest D, it assigns the identity A **as** `trustedSW` to the loaded program if $D \Rightarrow$ `trustedSW`. Every object that should be protected from an untrusted program gets an ACL of the form (`SomeNodes` **as** `trustedSW`) $\wedge$ (...). Here `SomeNodes` is a group containing all the nodes that are trusted to access the object, and the elided term gives the individuals that are trusted. Alternatively, if A sees no certificate for D it assigns the identity A **as** `unknown` to the loaded program; then the program will be able to access only objects whose ACLs explicitly grant access to `SomeNodes` **as** `unknown`.
- The node A has an ACL that controls the operation of loading a program into A, and `trustedSW` is on this ACL. Then no program will be loaded unless its digest speaks for `trustedSW`. This method is appropriate when A cannot protect itself from a running program, for example, when A is a PC running MS-DOS.

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 such as `DBServers` **as** `Ingres` $\Rightarrow$ `DBSystems`, `Mainframes` **as** `DB2` $\Rightarrow$ `DBSystems`, and so on.

6.2 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 nonvolatile 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_{ca} **says** $K_m \Rightarrow M$. Alternatively, a certification authority constructs (K_m, K_m^{-1}), makes the certificate K_{ca} **says** $K_m \Rightarrow M$, and delivers K_m^{-1} to M in some suitably secure way. It is an interesting problem to devise a practical installation procedure.

The other difference is that when M (the boot code that gets control after 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 to know 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. As with the second method of virus control, this amounts to an ACL for running on M.

—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.

The machine M also needs to know the name and public key of some principal that it can use to start the path name authentication described in Section 5; this principal can be M itself or its local certification authority. This information can be stored in M during installation, or it can be recorded in a certificate signed by K_m and supplied to M during booting along with P.

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} to express its complete trust.

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. 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: [24]

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

″ **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, given (1) and (2), B can speak for B **for** A by quoting A.[25]

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.

7.1 Login

A minor variation of the basic 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

---

[24] We introduce **for** as an independent operator and axiomatize it by (D1–2) and some other axioms that make it easier to write ACLs (see Section 9):

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

″ $(A$ **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:

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

″ $A \Rightarrow D | A$ (D6)

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

[25] 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.

protocols are discussed elsewhere [1, 26, 27]. 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 of the delegation has expired. This protects the user in case the workstation is compromised after she logs out. If 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:

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

7.2 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 introduce a level of indirection by having 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 INTERPROCESS 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

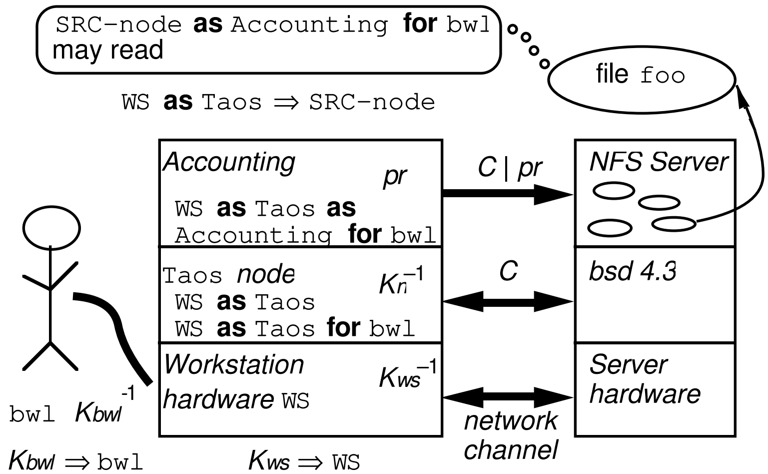

Fig. 7. Principals and keys for the workstation-server example.

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 need to speak for its client during normal operation and for itself during recovery.

Figure 7 is an expanded version of the example in Figure 1. For each component it indicates the principals that the component speaks for and the channel it can send on (usually an encryption key). Thus the `Taos` node speaks for `WS` **as** `Taos` and has the key K_n^{-1} so it can send on channel K_n. The accounting application speaks for `WS` **as** `Taos` **as** `Accounting` **for** `bwl`; it runs as process pr, which means that the node will let it send on $K_n|\ pr$ or $C|\ pr$. Consider a request from the accounting application to read file `foo`. It has the form $C|\ pr$ **says** "read `foo`"; in other words, $C|\ pr$ is the channel carrying the request. This channel speaks for K_{ws} **as** `Taos` **as** `Accounting` **for** K_{bwl}. The credentials of $C|\ pr$ are:

K_{ws} **says** $K_n \Rightarrow K_{ws}$ **as** `Taos`	From booting WS (Section 6).
K_{bwl} **says** $(K_n \wedge K_l)\vert K_{bwl} \Rightarrow K_n$ **for** K_{bwl}	From `bwl`'s login (Section 7).
K_l **says** $K_n \Rightarrow K_l$	Also from login.
$K_n\vert\ K_{bwl}$ **says** $C\vert\ pr \Rightarrow ((K_{ws}$ **as** `Taos`) **as** `Accounting`) **for** K_{bwl}	Sent on $C\vert\ K_{bwl}$.

The server gets certificates for the first three premises in the credentials. The last premise does not have a certificate. Instead, it follows directly from a message on the shared key channel C between the `Taos` node and the server, because this channel speaks for K_n as described in Section 4.

To turn these into credentials of $C|\ pr$ for `WS` **as** `Taos` **as** `Accounting` **for** `bwl`, the server must obtain the certificates that authenticate channels for the

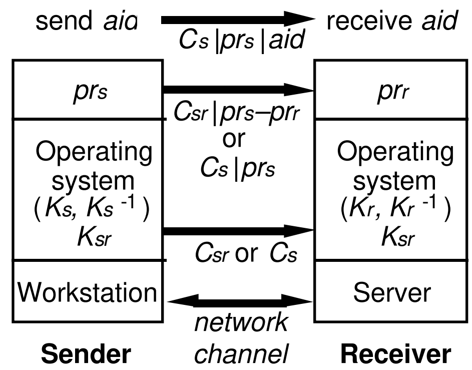

Fig. 8. Multiplexing a node-to-node channel.

names `bwl` and `WS` from the certification database as described in Section 5. Finally, to complete the access check, the server must obtain the group membership certificate `WS` **as** `Taos` ⇒ `SRC-node`. A system using the push model would substitute names for one or both of the keys K_{ws} and K_{bwl}. It would also get the name certificates for `WS` and `bwl` from the database and add them to the credentials.

The rest of this section explains in some detail how this scheme works in practice and extends it so that a single process can conveniently speak for a number of principals.

8.1 Interprocess Communication Channels

We describe the details of our authenticated interprocess communication mechanism in terms of messages from a sender to a receiver. The mechanism 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. As we saw in the example, 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 $C_{sr} | pr_s–pr_r$; see Figure 8. A shared key K_{sr} defines the node-to-node channel C_{sr} = DES (K_{sr}).

Henceforth we concentrate on the integrity of the channels,[26] so we care only that the message comes from the sender, not that it goes to the receiver. Section 4 explains how to establish DES (K_{sr}) ⇒ 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

[26] 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 sending process, where $C_s =$ RSA(K_s). Some messages don't use DES (K_{sr}) but instead 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.

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. A single process may speak for several principals, and we express this by multiplexing the channel from the process. Our strategy 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 $C_{aid} = C_s | pr_s | aid$; this is the channel $C_s | pr_s$ from the process multiplexed with *aid* as the subchannel address. 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.

This design is good because the sending process doesn't need to tell the operating system about *aid* in order to send C_{aid} **says** s to the receiver, since *aid* is just an integer. The only role of the operating system is to implement the channel $C_s | pr_s$ securely by labelling each message with the process pr_s that sends it. Thus a principal is passed as cheaply as an integer, except for 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 name or public key. To do this, it needs to know $C_{aid} \Rightarrow A$; we 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. The channel C on which the sender transmits a premise to the receiver could be either 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 in detail here. The latter case can arise in two ways. One is that shared key encryption is being used to simulate public key encryption, as described in Section 4. The other is as an optimization when there is already a shared-key channel that speaks for the public key, such as the channel DES (K_{sr}) described just above. This optimization works only for premises that the receiver does not need to forward to a third party.

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 C_s can speak for A. In our system all the operands of A are either roles or the public keys of nodes or of users; in the example of Figure 7 the keys are K_{ws} and K_{bwl}, and all the names are roles. Sections 6 and 7 explain how the sending system gets credentials for these keys as a result of booting or login. Section 5

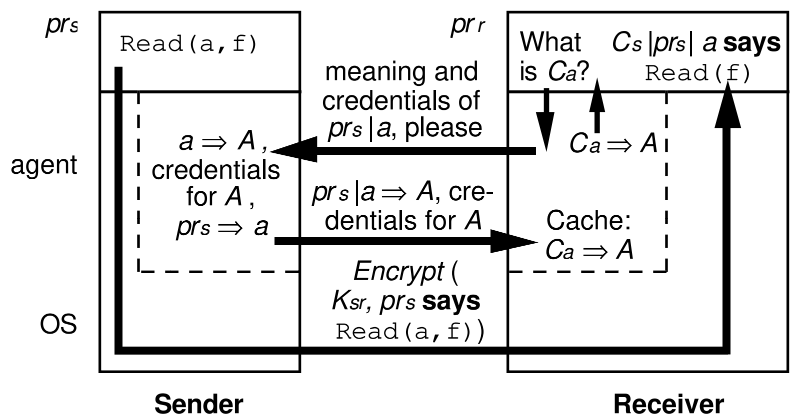

Fig. 9. Messages to the agents for authenticating a channel.

explains how the receiving system pulls credentials that authenticate these keys as speaking for named principals.

8.2 The Authentication Agent

Thus 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, checks the proof, and caches the result. In our system a component of the receiver's operating system called the *authentication agent* does this work for the receiver.

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 must be refreshed after that time. Figure 9 shows communication through the agent.

The agent has three jobs: caching credentials, supplying credentials, and handing off authorities.

Its first job, 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

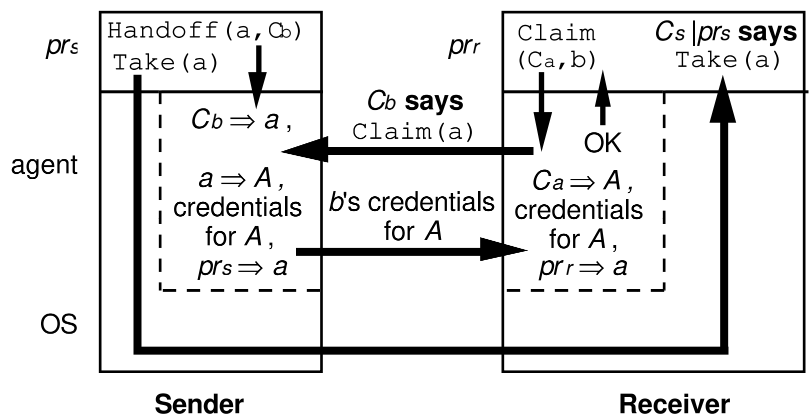

Fig. 10. Messages to the agents for handing off an authority.

discard entries whenever it likes. If the information it needs isn't in the cache, the receiver's agent asks the sender's agent 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 to 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 $C_s | pr | a$ as 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 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 authorities from one process to another, as shown in Figure 10. A sending process can hand off the authority a to another principal b by calling `Handoff(a, Cb)`. 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. This means that the claiming can be automatic. Suppose that process pr passes on an authority a, the recipient asks for a's credentials, and pr hasn't claimed a. In this case pr's agent can claim a automatically as long as 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, for instance one that corresponds to a delegation as described in Section 7. The login procedure of Section 7.1 is an example of this: the user hands off authority for "machine **for** user" rather than her entire authority.

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 it calls `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 *result* 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** *result* $\Rightarrow B$ **for** A. 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 the receiver 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.

Table IV. The State of an Agent

Key	(K_n, K_n^{-1}), the public key pair of this node.
Principal cache	A table mapping a channel $C_a = C_s \mid pr_s \mid a$ to A. An entry means the agent has seen credentials proving $C_a \Rightarrow A$; the entry also has a lifetime.
Authorities	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 V. Programming Interface from a Process to its Local 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`	$a \mid r$ **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** $b \mid a \Rightarrow b$ **for** a
`Accept(a: C, b: A): A`	Retrieve a **says** $b \mid a \Rightarrow b$ **for** a; $b \mid a$ **says** $b \mid a \Rightarrow b$ **for** $a \wedge$ *result* $\Rightarrow b \mid a$
`CheckAccess(acl: ACL, b: C, op: Operation) : Boolean`	Does *acl* grant b the right to do *op*?

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

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 IV summarizes the state of the agent. Table V 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 message it sends. This gets rid of most authority arguments, but it is less flexible and less explicit than the basic scheme in which each message specifies its authorities.
- —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 to store group memberships. Here 'trusted' means that it speaks for these groups. This method can avoid a lot of public key encryption.
- —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.

8.3 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,` C_a`, read)`, which returns true or false. Section 9 explains how this works.

8.4 Pragmatics

The performance of our interprocess authentication mechanism 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. Later 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, so that the operating system wouldn't be involved and the process identifiers wouldn't be needed. We 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, this is a lot of 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 trusted computing base.

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, as described in Section 1.

We take an ACL to be a set of principals, each with some rights to the ACL's object. [27] 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. So it's not surprising that we have to restrict the form of ACLs to get an algorithm that is complete (that is, always finds a proof if there is one) and also runs reasonably fast.

There are many ways to do this. Our choice is described by the following

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

grammar for the principal in an ACL entry or a request: [28]

principal	::=	forList | principal $\wedge$ 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, for example, 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 be either 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

[28] We can relax this syntax somewhat. Since **for** and **as** distribute over $\wedge$ by (D2) and (P6), we can push any nested $\wedge$ operators outward. Since (A **for** B) **as** $C = A$ **for** (B **as** C) by (D4), we can push any **as** operators inward into the second operands of **for**.

—looking up a name only if its tag appears in the request or if it is specially marked to be looked up unconditionally (for instance, the name of a group that is 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 a request with missing tags is denied. 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.

What about denying access to a specific principal? This 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 if it is in the group.

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.

Instead, we can 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.

9.1 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, all listed in the appendix. 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;
- —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.

The theory also explains a number of new methods used in our system for:

- —treating certificates and online communication with authorities as logically equivalent mechanisms;
- —revoking secure long-lived certificates rapidly by requiring them to be countersigned with refreshable short-lived ones;
- —loading programs and booting machines securely;
- —delegating authority in a way that combines and limits the power of both parties;
- —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.

The system is 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. Node-to-node channel setup, process-to-process authentication, roles, delegation, and secure loading are all working, and our implementation is the default authentication system for the 50 researchers at SRC.

Work is in progress on software support of network-controller based DES encryption. Although our current implementation does not allow either composite or hierarchical principal names in ACLs, we expect to experiment with these in the future. A forthcoming paper will describe the engineering details and the performance of the implementation.

APPENDIX

Here is a list of all the axioms in the paper, and therefore of all the assumptions made by our theory.

Statements

If s is an instance of a theorem of propositional logic then ″ s. (S1)

If ″ s and ″ $s \supset s'$ then ″ s'. (S2)

″ $(A$ **says** $s \wedge A$ **says** $(s \supset s')) \supset A$ **says** s'. (S3)

If ″ s then ″ A **says** s for every principal A. (S4)

Principals

″ $(A \wedge B)$ **says** $s \equiv (A$ **says** $s) \wedge (B$ **says** $s)$ (P1)

″ $(A \mid B)$ **says** $s \equiv A$ **says** B **says** s (P2)

″ $A = B \supset (A$ **says** $s \equiv B$ **says** $s)$ (P3)

″ $\mid$ is associative. (P5)

″ $\mid$ distributes over $\wedge$ in both arguments. (P6)

″ $(A \Rightarrow B) \equiv (A = A \wedge B)$ (P7)

″ $(A$ **says** $(B \Rightarrow A)) \supset (B \Rightarrow A)$ (P10)

Path names

″ P **except** $M \Rightarrow P$ (N1)

″ $M \neq N \supset (P$ **except** $M) \mid N \Rightarrow P/N$ **except** '..' (N2)

″ $M \neq$ '..' $\supset (P/N$ **except** $M) \mid$ '..' $\Rightarrow P$ **except** N (N3)

Roles

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

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

″ **as** is commutative and idempotent on roles (R3)

Delegation

″ B **for** $A = B \mid A \wedge D \mid A$ (D5)

″ $A \Rightarrow D \mid A$ (D6)

ACKNOWLEDGEMENTS

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

REFERENCES

1. ABADI, M., BURROWS, M., KAUFMAN, C., AND LAMPSON, B. Authentication and delegation with smart-cards. In *Theoretical Aspects of Computer Software*, LNCS 526, Springer, 1991, pp. 326-345. Also Res. Rep. 67, Systems Research Center, Digital Equipment Corp., Palo Alto, Calif., Oct. 1990. To appear in *Science of Computer Programming*.
2. ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. A calculus for access control in distributed systems. In *Advances in Cryptology— Crypto '91*, LNCS 576, Springer, 1992, pp.

1-23. Also Res. Rep. 70, Systems Research Center, Digital Equipment Corp., Palo Alto, Calif., March 1991. To appear in *ACM Trans. Program. Lang. Syst.*

3. BIRRELL, A., LAMPSON, B., NEEDHAM, R., AND SCHROEDER, M. Global authentication without global trust. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, Calif., May 1986), pp. 223-230.
4. BURROWS, M., ABADI, M., AND NEEDHAM, R. A logic of authentication. *ACM Trans. Comput. Syst. 8*, 1 (Feb. 1990), 18-36. An expanded version appeared in *Proc. Royal Society* A *426*, 1871 (Dec. 1989), 233-271 and as Res. Rep. 39, Systems Research Center, Digital Equipment Corp., Palo Alto, Calif., Feb. 1989.
5. CCITT. *Information Processing Systems — Open Systems Interconnection — The Directory Authentication Framework*. CCITT 1988 Recommendation X.509. Also ISO/IEC 9594-8:1989.
6. COMBA, P. Exponentiation cryptosystems on the IBM PC. *IBM Syst. J. 28*, 4 (Jul. 1990), 526-538.
7. DAVIS, D. AND SWICK, R. Network security via private-key certificates. *ACM Oper. Syst. Rev. 24*, 4 (Oct. 1990), 64-67.
8. DENNING, D. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236-243.
9. DEPARTMENT OF DEFENSE. *Trusted Computer System Evaluation Criteria*. DOD 5200.28-STD, 1985.
10. DIFFIE, W. AND HELLMAN, M. New directions in cryptography. *IEEE Trans. Inf. Theor. IT-22*, 6 (Nov. 1976), 644-654.
11. EBERLE, H. AND THACKER, C. A 1 Gbit/second GaAs DES chip. In *Proceedings of the IEEE 1992 Custom Integrated Circuit Conference* (Boston, Mass., May 1992), pp. 19.7.1-19.7.4.
12. GASSER, M., GOLDSTEIN, A., KAUFMAN, C., AND LAMPSON, B. The Digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference* (Baltimore, Md., Oct. 1989), pp. 305-319.
13. GASSER, M., AND MCDERMOTT, E. An architecture for practical delegation in a distributed system. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, Calif., May 1990), pp. 20-30.
14. HERBISON, B. Low cost outboard cryptographic support for SILS and SP 4. In *Proceedings of the 13th National Computer Security Conference* (Baltimore, Md., Oct. 1990), pp. 286-295.
15. KOHL, J., NEUMAN, C., AND STEINER, J. The Kerberos network authentication service. Version 5, draft 3, Project Athena, MIT, Cambridge, Mass., Oct. 1990.
16. LAMPSON, B. Protection. *ACM Oper. Syst. Rev. 8*, 1 (Jan. 1974), 18-24.
17. LINN, J. Practical authentication for distributed systems. *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, Calif., May 1990), pp. 31-40.
18. NATIONAL BUREAU OF STANDARDS. *Data Encryption Standard*. FIPS Pub. 46, Jan. 1977.
19. NEEDHAM, R. AND SCHROEDER, M. Using encryption for authentication in large networks of computers. *Commun. ACM 21*, 12 (Dec. 1978), 993-999.
20. NEUMAN, C. Proxy-based authorization and accounting for distributed systems. Tech. Rep. 91-02-01, University of Washington, Seattle, Wash., March 1991.
21. RIVEST, R., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb. 1978), 120-126.
22. RIVEST, R. The MD4 message digest algorithm. In *Advances in Cryptology—Crypto '90*, Springer, 1991, pp. 303-311.
23. RIVEST, R. AND DUSSE, S. *The MD5 Message-Digest Algorithm*. Internet Draft [MD5-A]: draft-rsadsi-rivest-md5-01.txt, July 1991.
24. SALTZER, J., REED, D., AND CLARK, D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (Nov. 1984), 277-288.
25. SHAND, M., BERTIN, P., AND VUILLEMIN, J. Resource tradeoffs in fast long integer multiplication. In *2nd ACM Symposium on Parallel Algorithms and Architectures* (Crete, July 1990).
26. STEINER, J., NEUMAN, C., AND SCHILLER, J. Kerberos: An authentication service for open network systems. In *Proceedings of the Usenix Winter Conference* (Berkeley, Calif., Feb. 1988), pp. 191-202.

27. TARDO, J. AND ALAGAPPAN, K. SPX: Global authentication using public key certificates. *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, Calif., May 1991), pp. 232-244.
28. VOYDOCK, V. AND KENT, S. Security mechanisms in high-level network protocols. *ACM Comput. Surv. 15*, 2 (Jun. 1983), 135-171.

Received December 1991; revised June 1992; accepted August 1992