Security

Butler Lampson

TECS Week 2005

January 2005

Outline

Introduction: what is security?

Principals, the “speaks for” relation, and chains of responsibility

Secure channels and encryption

Names and groups

Authenticating systems

Authorization

Implementation

REAL-WORLD SECURITY

It's about value, locks, and punishment.

- –Locks good enough that bad guys don't break in very often.
- –Police and courts good enough that bad guys that do break in get caught and punished often enough.
- –Less interference with daily life than value of loss.

Security is expensive—buy only what you need.

- –People *do* behave this way
- –We don't *tell* them this—a big mistake
- –Perfect security is the worst enemy of real security

Elements of Security

Policy: *Specifying* security
What is it supposed to do?

Mechanism: *Implementing* security
How does it do it?

Assurance: *Correctness* of security
Does it really work?

Abstract Goals for Security

Secrecy	controlling who gets to read information
Integrity	controlling how information changes or resources are used
Availability	providing prompt access to information and resources
Accountability	knowing who has had access to information or resources

Dangers

Dangers

Vandalism or sabotage that	
–damages information	*integrity*
–disrupts service	*availability*
Theft of money	*integrity*
Theft of information	*secrecy*
Loss of privacy	*secrecy*

Vulnerabilities

Vulnerabilities

- Bad (buggy or hostile) **programs**
- Bad (careless or hostile) **people** giving instructions to good programs
- Bad guys corrupting or eavesdropping on **communications**

Threats

- Adversaries that can and want to exploit vulnerabilities

Defensive strategies

Coarse: **Isolate**—Keep everybody out

- –Disconnect

Medium: **Exclude**—Keep the bad guys out

- –Code signing, firewalls

Fine: **Restrict**—Let the bad guys in, but keep them from doing damage

- –Hardest to implement
- –Sandboxing, access control

Recover—Undo the damage. Helps with integrity.

- –Backup systems, restore points

Punish—Catch the bad guys and prosecute them

- –Auditing, police

Assurance

Trusted Computing Base (TCB)

- Everything that security depends on
- Must get it right
- Keep it small and simple

Elements of TCB

- Hardware
- Software
- Configuration

Defense in depth

Assurance: Defense in Depth

Network, with a firewall

Operating system, with sandboxing

- –Basic OS (such as NT)
- –Higher-level OS (such as Java)

Application that checks authorization directly

All need authentication

TCB Examples

Policy: Only outgoing Web access

TCB: firewall allowing outgoing port 80 TCP connections, but no other traffic

Hardware, software, and configuration

Policy: Unix users can read system directories, and read and write their home directories

TCB: hardware, Unix kernel, any program that can write a system directory (including any that runs as superuser).

Also `/etc/passwd`, permissions on all directories.

TCB: Configuration

Done again for each system, unlike HW or SW

- –New chance for mistakes each time

Done by amateurs, not experts

- –No learning from experience
- –Little training

Needs to be very simple

- –At the price of flexibility, fine granularity

Making Configuration Simple

Users—keep it simple

- At most three levels: self, friends, others
  Three places to put objects
- Everything else done automatically with policies

Administrators—keep it simple

- Work by defining policies. Examples:
  Each user has a private home folder
  Each user in one workgroup with a private folder
  System folders contain vendor-approved releases
  All executable programs signed by a trusted party

Today's systems don't support this very well

Assurance: Configuration Control

It's 2 am. Do you know what software is running on your machine?

Secure configuration ⇒ some apps don't run

- –Hence must be optional: "Secure my system"
- –Usually used only in an emergency

Affects the entire configuration

- –Software: apps, drivers, macros
- –Access control: shares, authentication

Also need configuration audit

Why We Don't Have "Real" Security

A. People don't buy it

- –Danger is small, so it's OK to buy features instead.
- –Security is expensive.
  Configuring security is a lot of work.
  Secure systems do less because they're older.
- –Security is a pain.
  It stops you from doing things.
  Users have to authenticate themselves.

B. Systems are complicated, so they have bugs.

- –Especially the configuration

“Principles” for Security

Security is not formal

Security is not free

Security is fractal

Abstraction can’t keep secrets

–“Covert channels” leak them

It’s all about lattices

ELEMENTS OF SECURITY

Policy: *Specifying* security
What is it supposed to do?

Mechanism: *Implementing* security
How does it do it?

Assurance: *Correctness* of security
Does it really work?

Specify: Policies and Models

Policy — specifies the whole system informally.

Secrecy	Who can read information?
Integrity	Who can change things, and how?
Availability	How prompt is the service?

Model—specifies just the computer system, but does so precisely.

Access control model	guards control access to resources.
Information flow model	classify information, prevent disclosure.

Implement: Mechanisms and Assurance

Mechanisms — tools for implementation.

Authentication	Who said it?
Authorization	Who is trusted?
Auditing	What happened?

Trusted computing base.

Keep it small and simple.

Validate each component carefully.

Information flow model (Mandatory security)

A lattice of labels for data:

- `unclassified` $<$ `secret` $<$ `top secret`;
- `public` $<$ `personal` $<$ `medical` $<$ `financial`

$\text{label}(f(x)) = \max(\text{label}(f), \text{label}(x))$

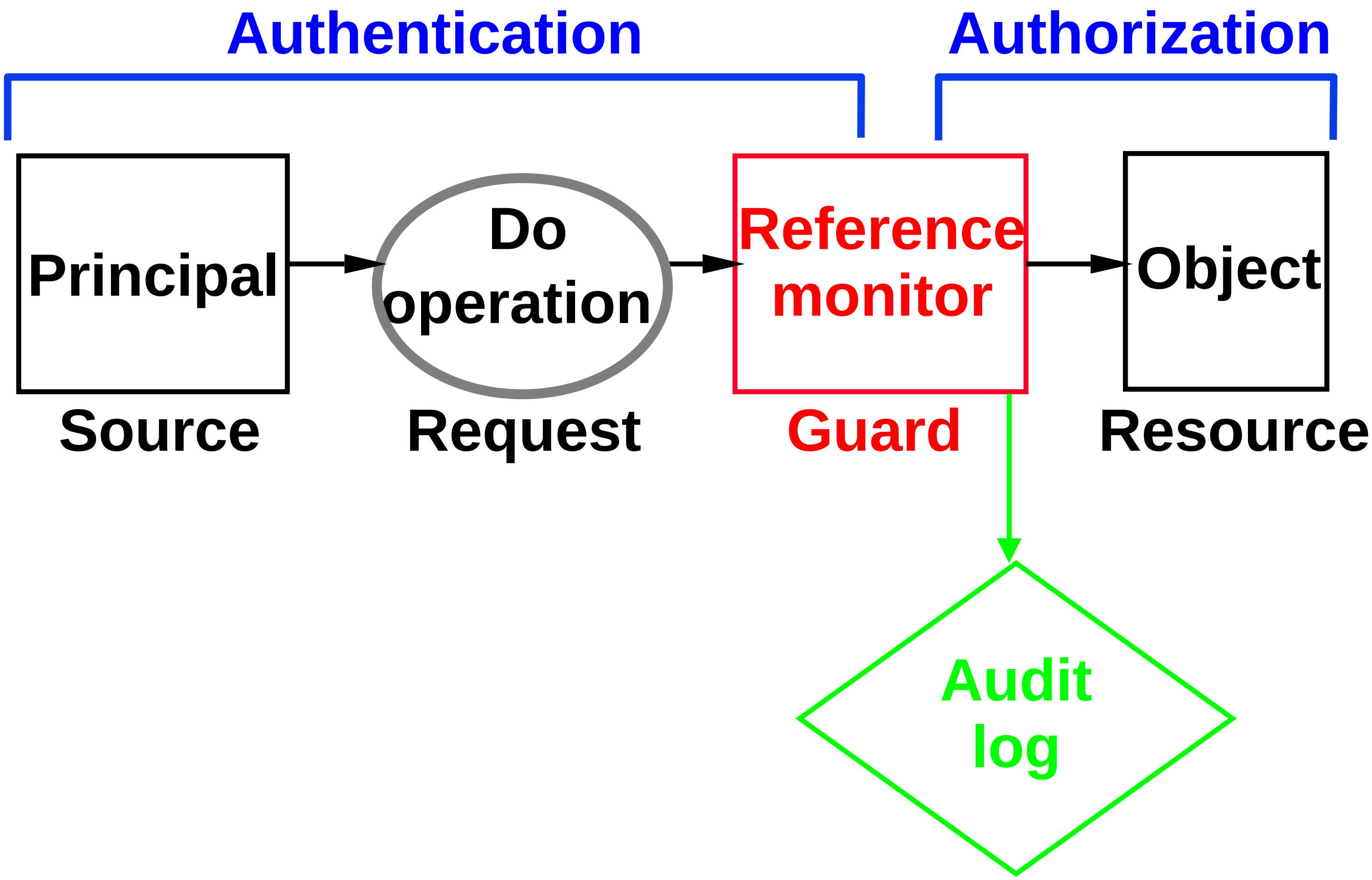
Labels can keep track of data properties:

- how secret *Secrecy*
- how trustworthy *Integrity*

When you use (release or act on) the data, user needs a ≥ clearance

Access Control Model

Guards control access to valued resources.

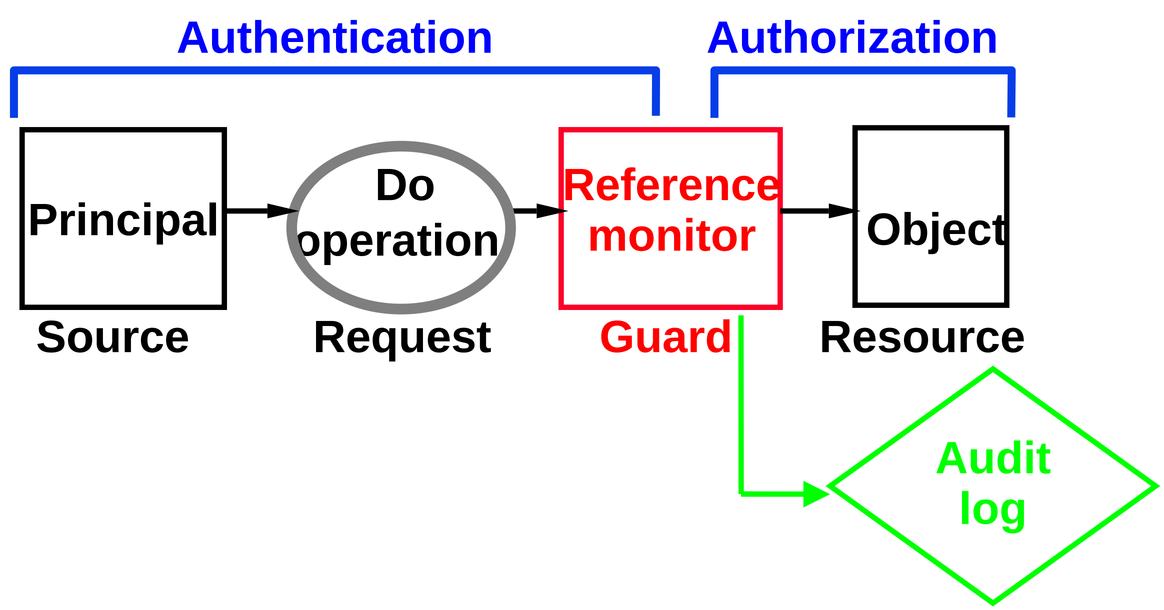


Access Control

Guards control access to valued resources.

Structure the system as —

Objects	entities with state.
Principals	can request operations on objects.
Operations	how subjects read or change objects.



Access Control Rules

Rules control the operations allowed
for each principal and object.

Principal may do	*Operation* on	*Object*
`Taylor`	Read	File “`Raises`”
`Lampson`	Send “`Hello`”	Terminal `23`
Process `1274`	Rewind	Tape unit `7`
`Schwarzkopf`	Fire three shots	`Bow gun`
`Jones`	Pay invoice `432`	Account `Q34`

Mechanisms—The Gold Standard

Authenticating principals

- –Mainly people, but also channels, servers, programs (encryption makes channels, so key is a principal)

Authorizing access

- –Usually for *groups*, principals that have some property, such as “Microsoft employee” or “type-safe” or “safe for scripting”

Auditing

Assurance

- –Trusted computing base

Standard Operating System Security

Assume secure channel from user (without proof)

Authenticate user by local password

- –Assign local user and group SIDs

Access control by ACLs: lists of SIDs and permissions

- –Reference monitor is the OS, or any RPC target

Domains: same, but authenticate by RPC to controller

Web servers: same, but *simplified*

- –Establish secure channel with SSL
- –Authenticate user by local password (or certificate)
- –ACL on right to enter, or on user's private state

NT Domain Security

Just like OS except for authentication

OS does RPC to domain for authentication

- –Secure channel to domain
- –Just do RPC(user, password) to get user's SIDs

Domain may do RPC to foreign domain

- –Pairwise trust and pairwise secure channels
- –SIDs include domain ID, so a domain can only authenticate its own SIDs

Web Security Today

Server: Simplified from single OS

- –Establish secure channel with SSL
- –Authenticate user by local password (or certificate)
- –ACL on right to enter, or on user's private state

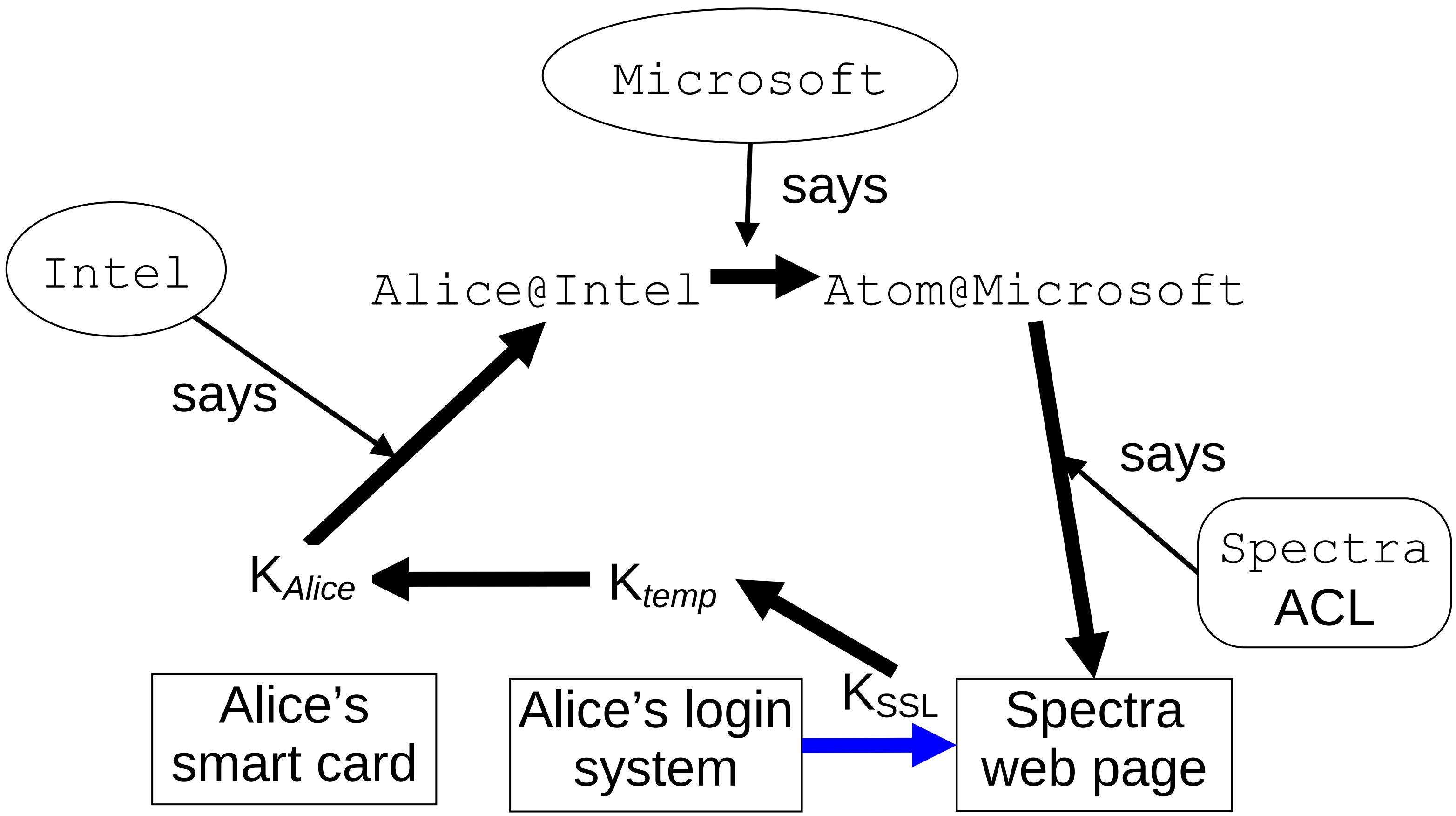
Browser (client): Basic authentication

- –Of server by DNS lookup, or by SSL + certificate
- –Of programs by supplier's signature
  - Good programs run as user
  - Bad ones rejected or totally sandboxed

END-TO-END EXAMPLE

`Alice` is at `Intel`, working on `Atom`, a joint Intel-Microsoft project

Alice connects to `Spectra`, Atom's web page, with SSL



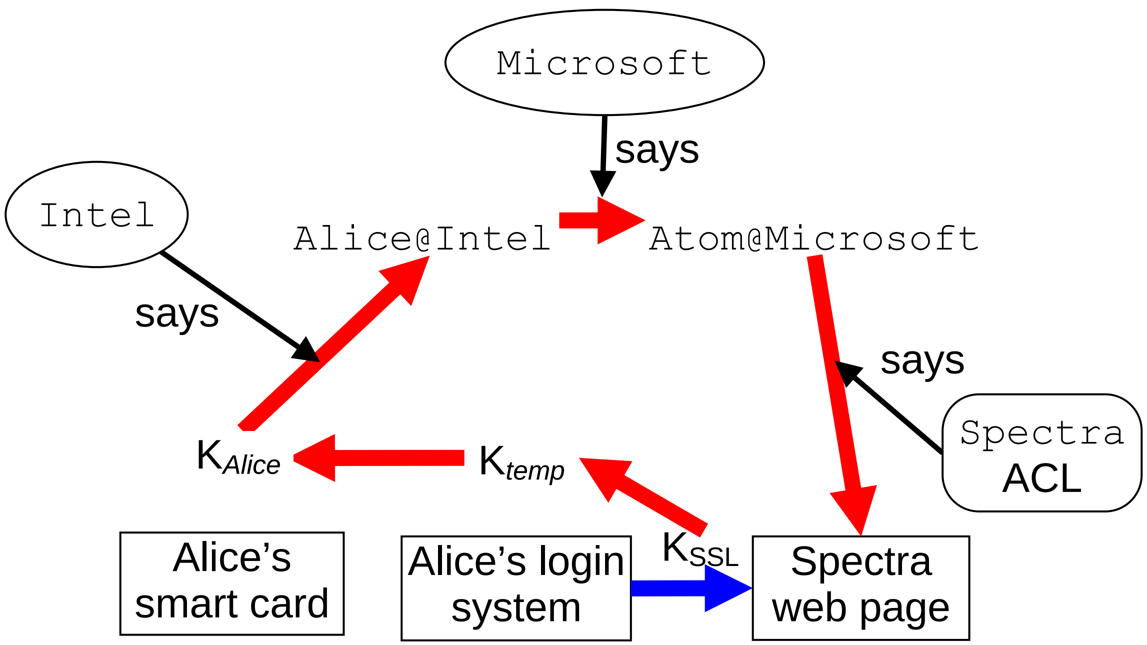
Chain of responsibility

`Alice` at `Intel`, working on `Atom`, connects to `Spectra`, Atom's web page, with SSL

Chain of responsibility:

$$K_{SSL} \Rightarrow K_{temp} \Rightarrow K_{Alice}$$

$\Rightarrow$ `Alice@Intel` $\Rightarrow$ `Atom@Microsoft` $\Rightarrow$ `Spectra`



Principals

Authentication: Who sent a message?

Authorization: Who is trusted?

Principal — abstraction of "who":

People	`Lampson, Taylor`
Machines	`VaxSN12648, Jumbo`
Services	`SRC-NFS, X-server`
Groups	`SRC, DEC-Employees`
Roles	`Taylor` **as** `Manager`
Joint authority	`Taylor` **and** `Lampson`
Weakening	`Taylor` **or** `UntrustedProgram`
Channels	`Key #7438`

Theory of Principals

Principal says statement P **says** s

Lampson **says** "`read /SRC/Lampson/foo`"

SRC-CA **says** "Lampson's key is `#7438`"

Axioms

If A **says** s and A **says** (s implies s') then A **says** s'

If $A = B$ then (A **says** s) = (B **says** s)

The "Speaks for" Relation $\Rightarrow$

Principal *A* speaks for *B* about *T* $A \Rightarrow_{T} B$

If *A* says something in set *T*, *B* does too:

Thus, *A* is stronger than *B*, or responsible for *B*, about *T*

Precisely: $(A\ \textbf{says}\ s) \wedge (s \in T)$ implies $(B\ \textbf{says}\ s)$

These are the links in the chain of responsibility

Examples

`Alice`	$\Rightarrow$ `Atom`	*group of people*
Key `#7438`	$\Rightarrow$ `Alice`	*key for Alice*

Delegating Authority

How do we establish a link in the chain: a fact $Q \Rightarrow R$

The "verifier" of the link must see evidence, of the form

"P **says** $Q \Rightarrow R$"

There are three questions about this evidence

- How do we *know* that P says the delegation?
- Why do we *trust* P for this delegation?
- Why is P *willing* to say it?

How Do We *Know P* says *X*?

If *P* is	then
a key	*P* signs *X* cryptographically
some other channel	message *X* arrives on channel *P*
the verifier itself	*X* is an entry in a local database

These are the only ways that the verifier can *directly* know who said something: receive it on a secure channel or store it locally

Otherwise we need $C \Rightarrow P$, where *C* is one of these cases

–Get this by recursion

Why Do We *Trust* The Delegation?

We trust *A* to delegate its own authority.

Delegation rule: If P **says** $Q \Rightarrow R$ then $Q \Rightarrow R$

Reasonable if *P* is competent and accessible.

Why Is *P Willing* To Delegate To *Q*?

Some facts are installed manually

- $K_{Intel} \Rightarrow$ Intel, when Intel and Microsoft establish a direct relationship
- The ACL entry `Lampson` $\Rightarrow$ `usr/Lampson`

Others follow from the properties of some algorithm

- If Diffie-Hellman yields K_{DH}, then I can say
  "$K_{DH} \Rightarrow$ me, provided
  - You are the other end of the K_{DH} run
  - You don't disclose K_{DH} to anyone else
  - You don't use K_{DH} to send anything yourself."

In practice I simply sign $K_{DH} \Rightarrow K_{me}$

Why Is *P Willing* To Delegate To *Q*?

Others follow from the properties of some algorithm

- If server *S* starts process *P* from and sets up a channel *C* from *P*, it can say $C \Rightarrow$ `SQLv71`

  Of course, only someone who believes $S \Rightarrow$ `SQLv71` will believe this

  To be conservative, *S* might compute a strong hash H_{SQLv71} of `SQLv71.exe` and require

  `Microsoft` **says** "$H_{SQLv71} \Rightarrow$ `SQLv71`"

  before authenticating *C*

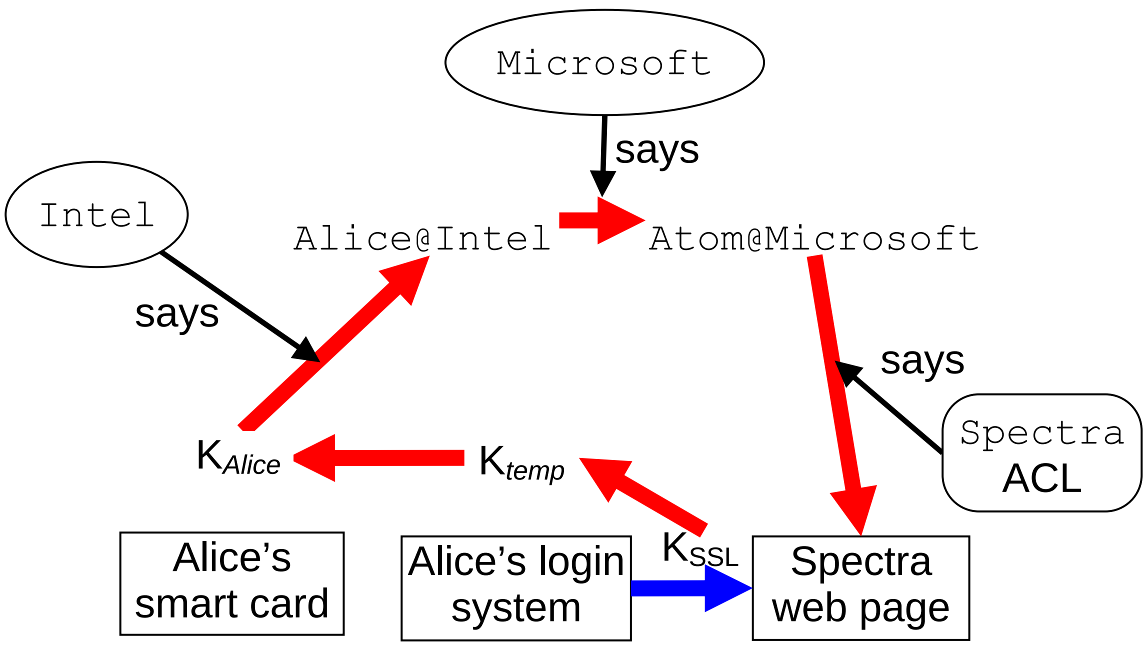
Chain of responsibility

Alice at Intel, working on Atom, connects to Spectra, Atom's web page, with SSL

Chain of responsibility:

$K_{SSL} \Rightarrow K_{temp} \Rightarrow K_{Alice}$

$\Rightarrow$ Alice@Intel $\Rightarrow$ Atom@Microsoft $\Rightarrow$ Spectra



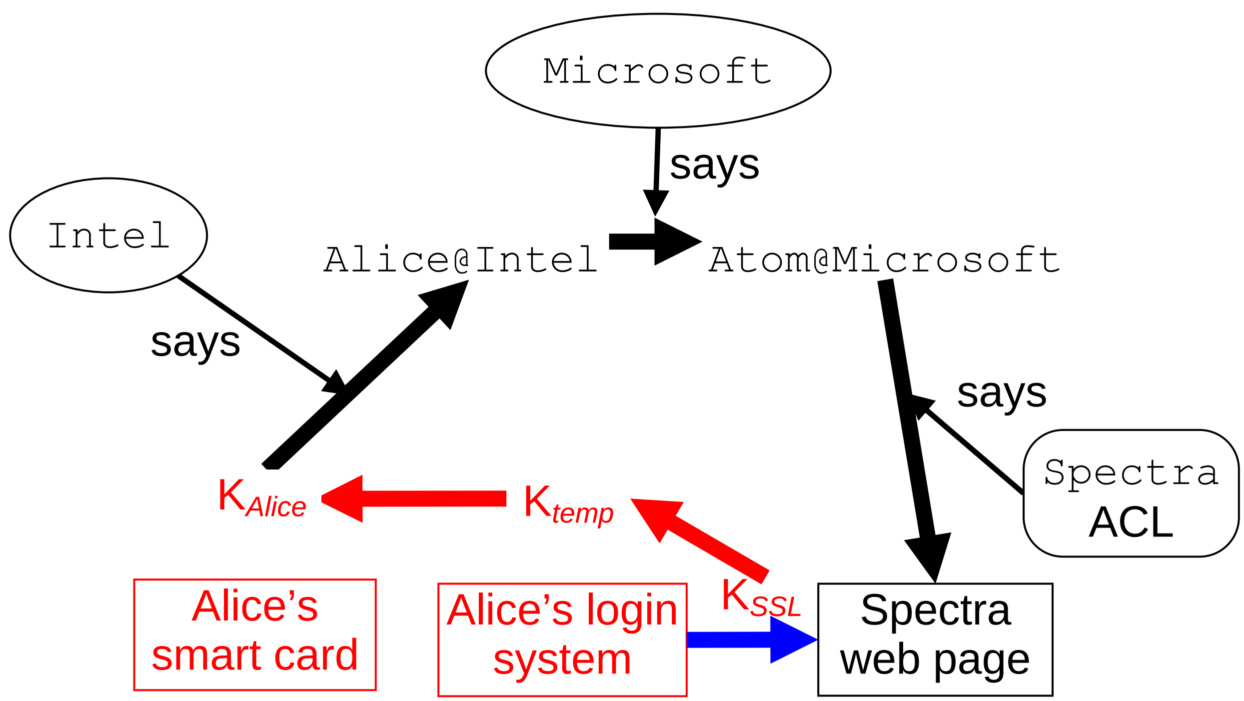
Authenticating Channels

Chain of responsibility:

$K_{SSL} \Rightarrow K_{temp} \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow \ldots$

K_{temp} **says** ↑ K_{Alice} **says** ↑

(SSL setup) (via smart card)



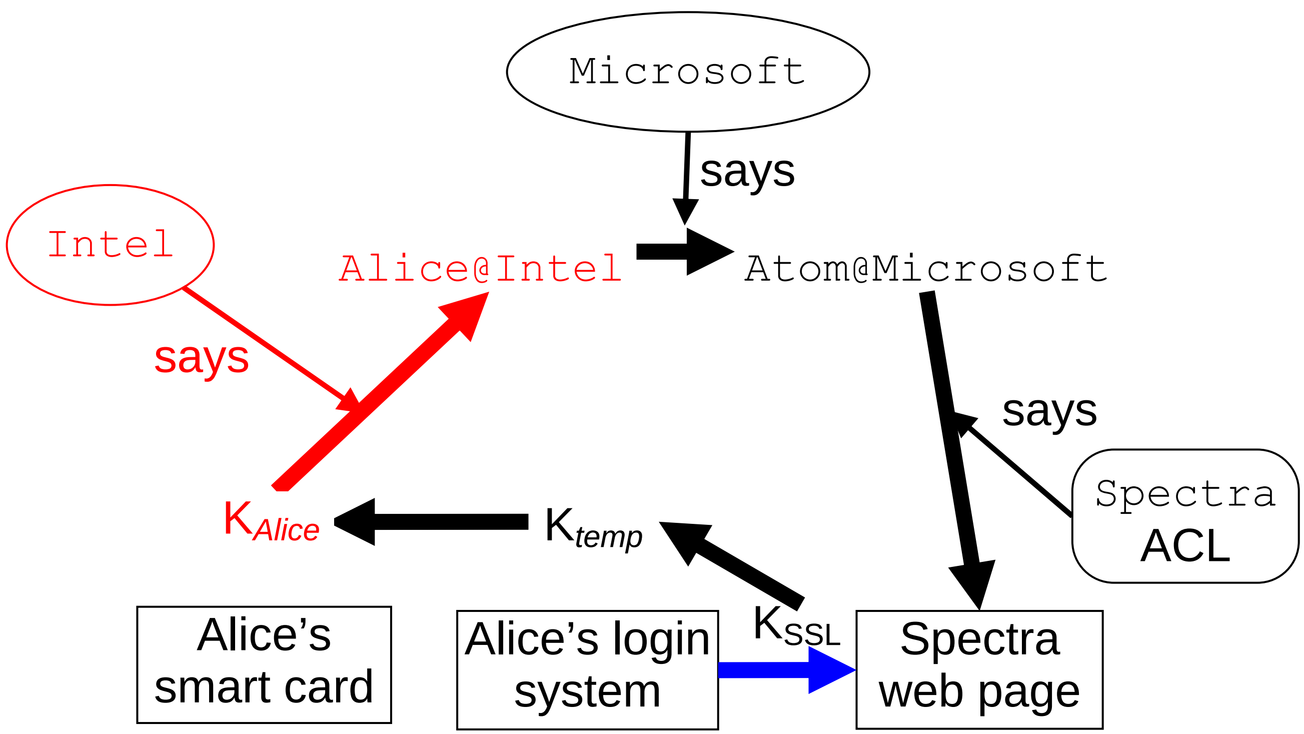
Authenticating Names: SDSI

A name is in a name space, defined by a principal *P*

- *P* is like a directory. The root principals are keys.

Rule: *P* speaks for *any* name in its name space

$K_{Intel} \Rightarrow$ `Intel` $\Rightarrow$ `Intel/Alice` (= `Alice@Intel`)

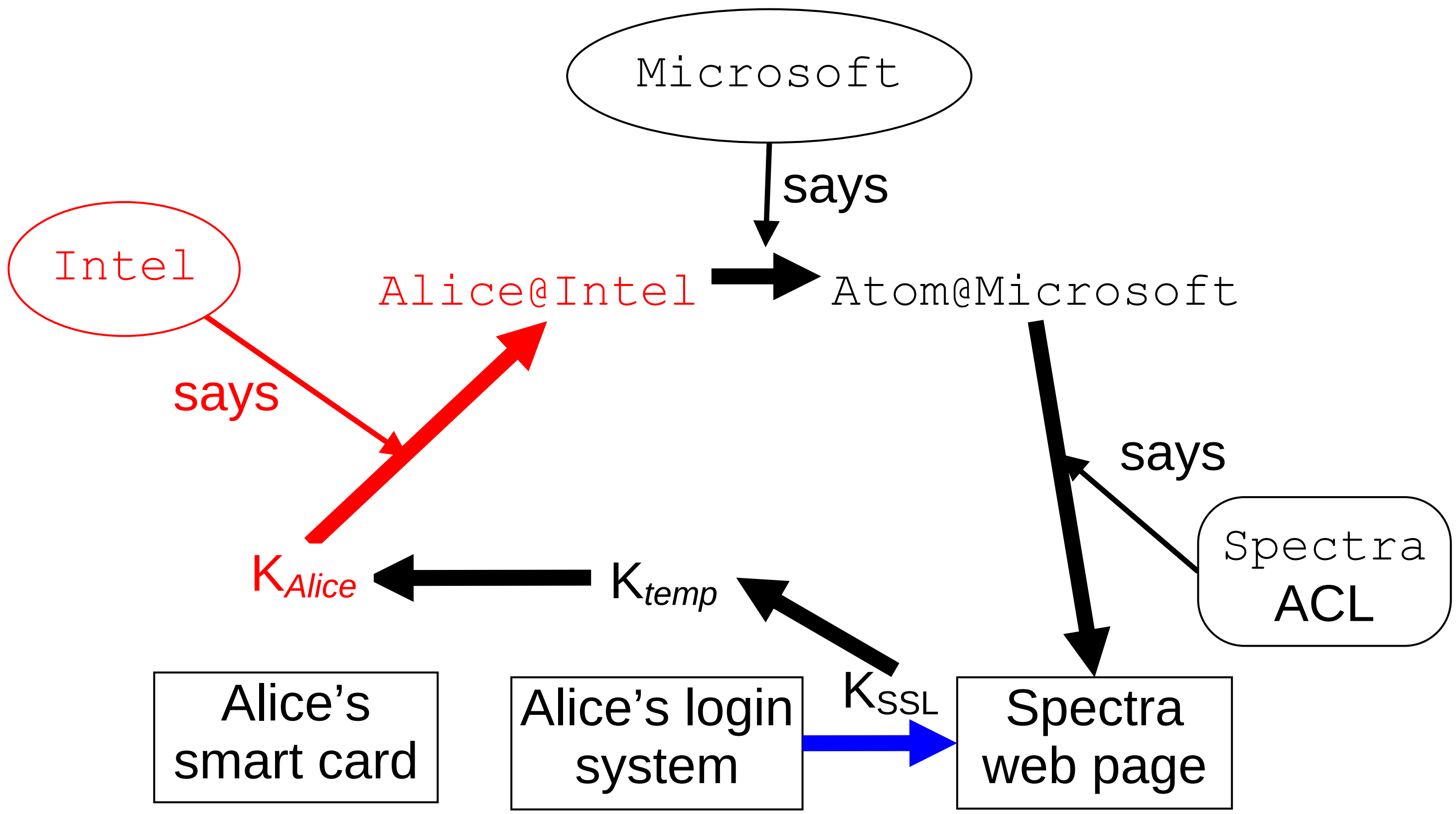


Authenticating Names

$K_{Intel} \Rightarrow$ `Intel` $\Rightarrow$ `Intel/Alice` (= `Alice@Intel`)

$K_{temp} \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow$ …

K_{Intel} **says**



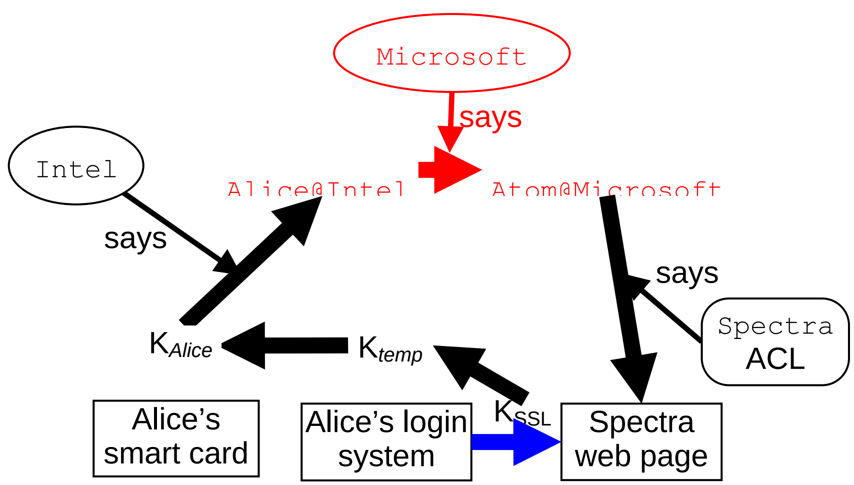
Authenticating Groups

A group is a principal; its members speak for it

- `Alice@Intel` $\Rightarrow$ `Atom@Microsoft`
- `Bob@Microsoft` $\Rightarrow$ `Atom@Microsoft`
- …

Evidence for groups: Just like names and keys.

$K_{Microsoft} \Rightarrow$ `Microsoft` $\Rightarrow$ `Microsoft/Atom` ($=$ `Atom@Microsoft`)

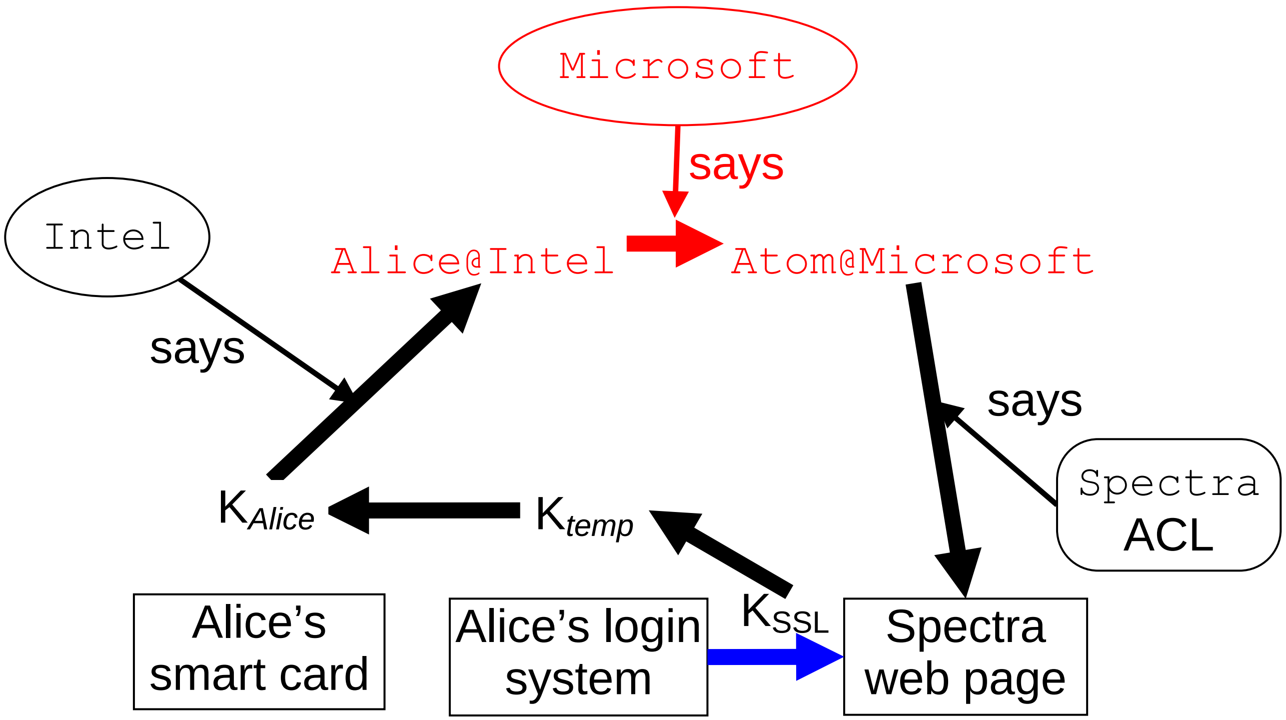


Authenticating Groups

$K_{Microsoft} \Rightarrow$ `Microsoft` $\Rightarrow$ `Atom@Microsoft`

$\ldots \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow$ `Atom@Microsoft` $\Rightarrow \ldots$

$K_{Microsoft}$ **says**



Authorization with ACLs

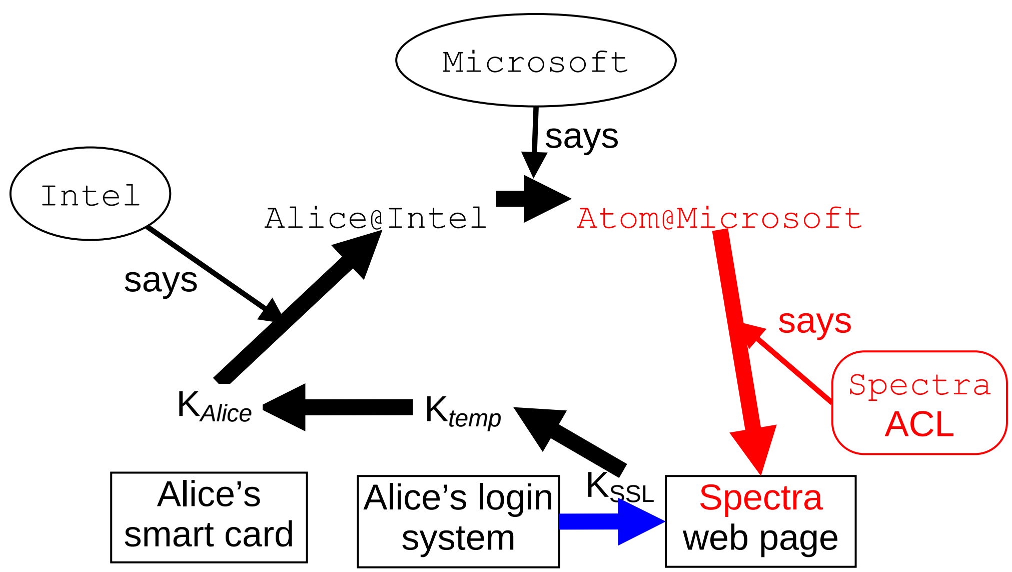
View a resource object *O* as a principal

P on *O*'s ACL means *P* can speak for *O*

- Permissions limit the set of things *P* can say for *O*

If `Spectra`'s ACL **says** `Atom` **can** `r/w`, that means

`Spectra` **says** `Atom@Microsoft` $\Rightarrow_{r/w}$ `Spectra`

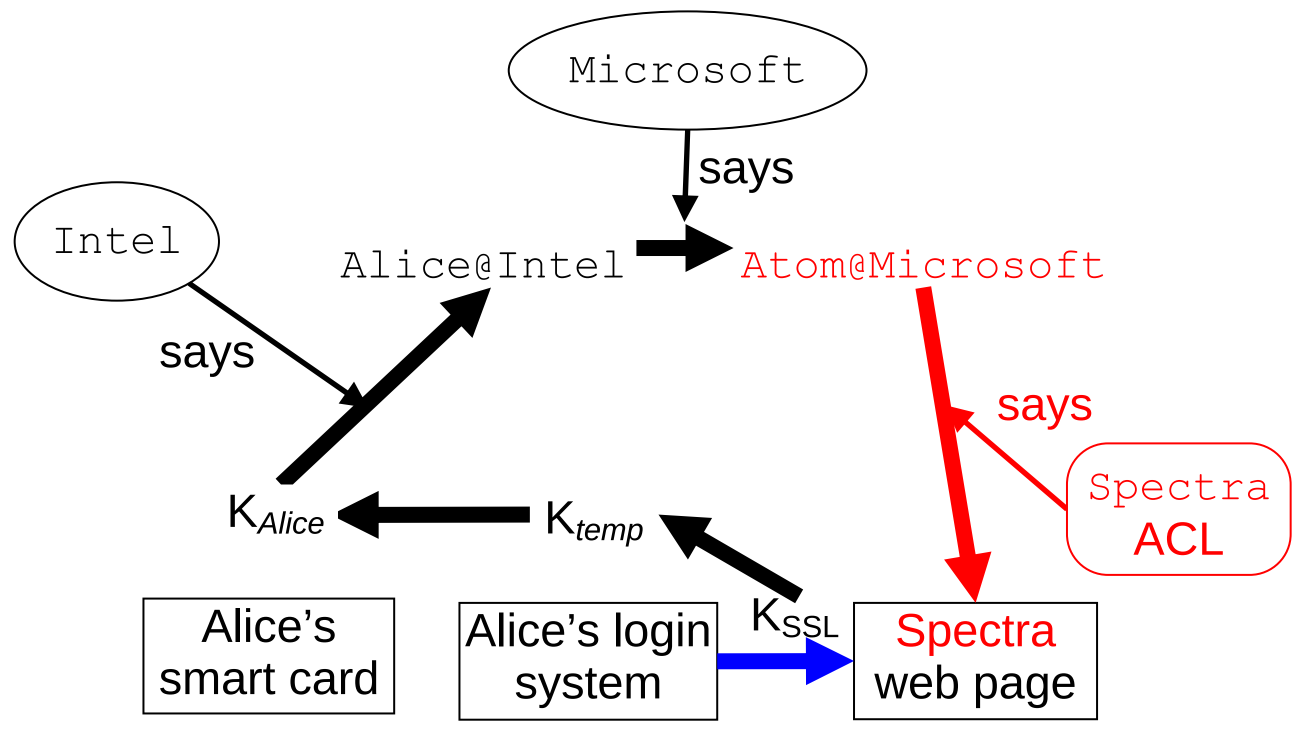


Authorization with ACLs

`Spectra`'s ACL **says** `Atom` **can** `r/w`

…⇒ `Alice@Intel` ⇒ `Atom@Microsoft` $\Rightarrow_{r/w}$ `Spectra`

`Spectra` **says**



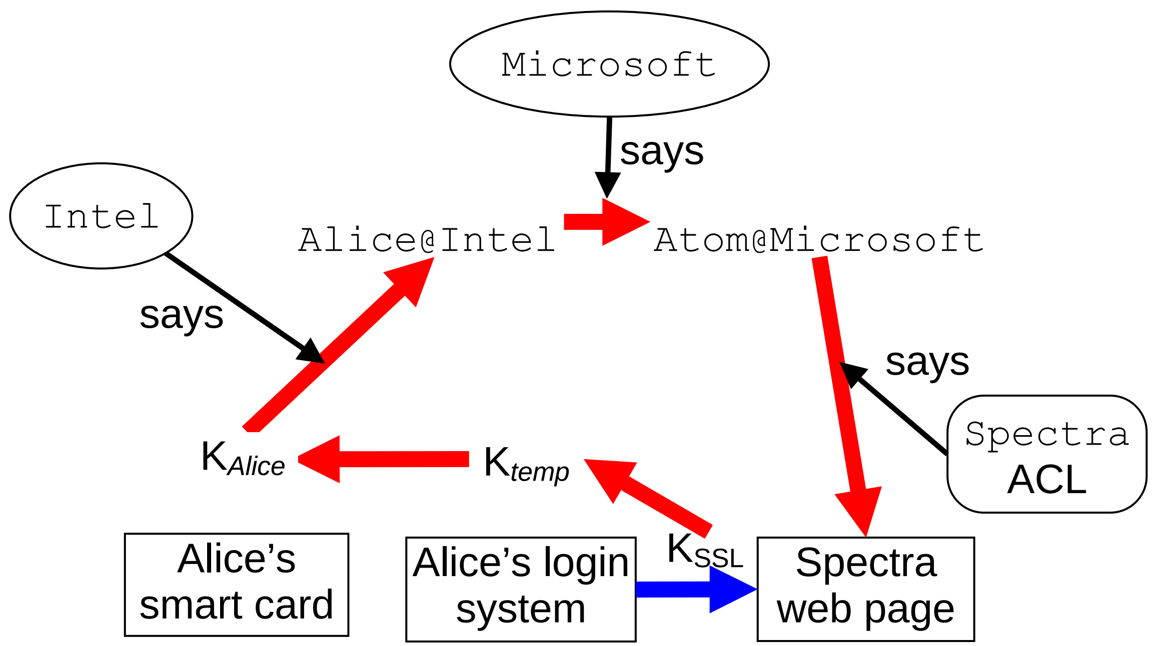
End-to-End Example: Summary

Request on SSL channel: K_{SSL} **says** "read `Spectra`"

Chain of responsibility:

$K_{SSL} \Rightarrow K_{temp} \Rightarrow K_{Alice}$

$\Rightarrow$ `Alice@Intel` $\Rightarrow$ `Atom@Microsoft` $\Rightarrow$ `Spectra`



Compatibility with Local OS?

(1) Put network principals on OS ACLs

(2) Let network principal speak for local one

– `Alice@Intel` $\Rightarrow$ `Alice@microsoft`

– Use network authentication

  replacing local or domain authentication

– Users and ACLs stay the same

(3) Assign SIDs to network principals

– Do this automatically

– Use network authentication as before

Summaries

The chain of responsibility can be long

K_{temp} **says** $K_{SSL} \Rightarrow K_{temp}$

K_{Alice} **says** $K_{temp} \Rightarrow K_{Alice}$

K_{Intel} **says** $K_{Alice} \Rightarrow$ `Alice@Intel`

$K_{Microsoft}$ **says** `Alice@Intel` $\Rightarrow$ `Atom@Microsoft`

`Spectra` **says** `Atom@Microsoft` $\Rightarrow_{\mathrm{r/w}}$ `Spectra`

Can replace a long chain with one summary certificate

`Spectra` **says** $K_{SSL} \Rightarrow_{\mathrm{r/w}}$ `Spectra`

Need a principal who speaks for the end of the chain

This is often called a capability

Lattice of Principals

A **and** *B* max, least upper bound

$(A \textbf{ and } B) \textbf{ says } s \equiv (A \textbf{ says } s) \textbf{ and } (B \textbf{ says } s)$

A **or** *B* min, greatest lower bound

$(A \textbf{ or } B) \textbf{ says } s \equiv (A \textbf{ says } s) \textbf{ or } (B \textbf{ says } s)$

Now $A \Rightarrow B \equiv (A = A \textbf{ and } B) \equiv (B = A \textbf{ or } B)$

Thus $\Rightarrow$ is the lattice's partial order

Could we interpret this as sets? Not easily: **and** is not intersection

Facts about Principals

$A = B$ is equivalent to ($A \Rightarrow B$) **and** ($B \Rightarrow A$)

$\Rightarrow$ is transitive

and, **or** are associative, commutative, and idempotent

and, **or** are monotonic:

If $A' \Rightarrow A$ then (A' **and** B) $\Rightarrow$ (A **and** B)

(A' **or** B) $\Rightarrow$ (A **or** B)

Important because a principal may be stronger than needed

Lattices: Information Flow to Principals

A lattice of labels:

- `unclassified` $<$ `secret` $<$ `top secret`;
- `public` $<$ `personal` $<$ `medical` $<$ `financial`

Use the same labels as principals, and let $\Rightarrow$ represent clearance

- `lampson` $\Rightarrow$ `secret`

Or, use names rooted in principals as labels

- `lampson/personal`, `lampson/medical`

Then the principal can declassify

SECURE CHANNELS

A secure channel:

- says things directly — C **says** s
- has known possible receivers — secrecy
  possible senders — integrity
- if P is the only possible sender, then $C \Rightarrow P$

Examples

Within a node: operating system (pipes, etc.)
Between nodes:

Secure wire	difficult to implement
Network	fantasy for most networks
Encryption	practical

Names for Channels

A channel needs a name to be authenticated properly

- K_{Alice} **says** $K_{temp} \Rightarrow K_{Alice}$

It's not OK to have

- K_{Alice} **says** "this channel $\Rightarrow K_{Alice}$"

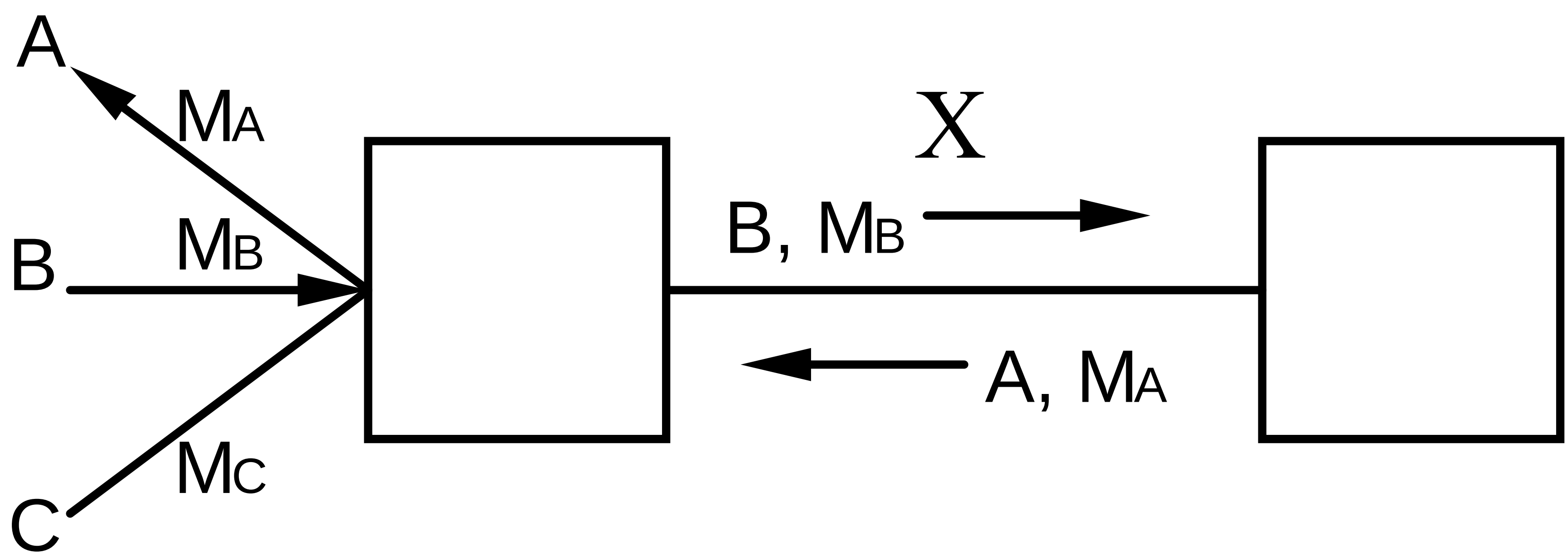
unless you trust the receiver not to send this on another channel!

- Thus it is OK to authenticate yourself by sending a password to amazon.com on an SSL channel already authenticated (by a Verisign certificate) as going to Amazon.

Multiplexing a Channel

Connect *n* channels *A*, *B*, ... to one channel *X* to make *n* new sub-channels *X*|*A*, *X*|*B*, ... Each subchannel has its own address on *X*

The multiplexer must be trusted



Quoting

$A \mid B$ **A quoting B**

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

Axioms

| is associative

| distributes over **and**, **or**

$A \Rightarrow_{* \Rightarrow A|B} A \mid B$

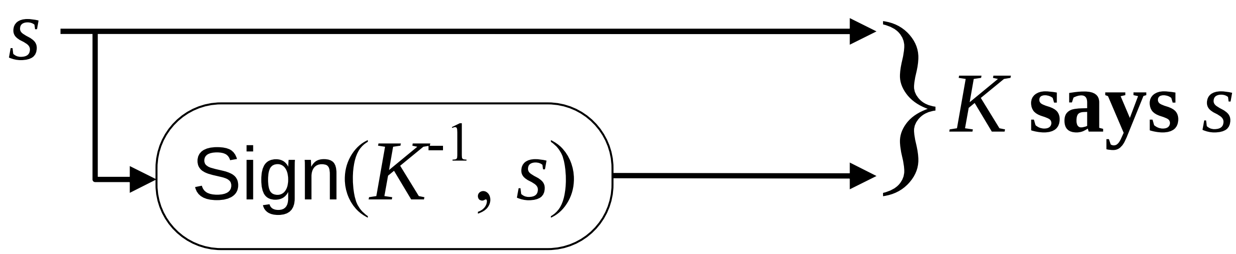
Multiplexing a Channel: Examples

Multiplexer	*Main channel*	*Subchannels*	*Address*
OS	node–node	process–process	port or process ID
Network routing	node–network	node–node	node address

Signed Secure Channels

The channel is defined by the key: If only A knows K^{-1}, then $K \Rightarrow A$ (Actually, if only A *uses* K^{-1}, then $K \Rightarrow A$)

K **says** s is a message which K can verify



The bits of "K **says** s" can travel on any path

Abstract Cryptography: Sign/Verify

Verify(K, M, sig) = true iff sig = Sign(K', M) and $K' = K^{-1}$

– Is sig K's signature on M?

Concretely, with RSA public key:

– Sign(K^{-1}, M) = RSAencrypt(K^{-1}, SHA1(M))

– Verify(K, M, sig) = (SHA1(M) = RSAdecrypt(K, sig))

Concretely, with AES shared key:

– Sign(K, M) = SHA1(K, SHA1($K \| M$))

– Verify(K, M, sig) = (SHA1(K, SHA1($K \| M$)) = sig)

Concrete crypto is for experts only!

Abstract Cryptography: Seal/Unseal

Unseal(K^{-1}, Seal(K, M)) = M, and without K^{-1} you can't learn anything about M from Seal(K, M)

Concretely, with RSA public key:

- Seal(K, M) = RSAencrypt(K^{-1}, $IV \| M$)
- Unseal(K, M_{sealed}) = RSAdecrypt(K, M_{sealed}).M

Concretely, with AES shared key:

- Seal(K, M) = AESencrypt(K, $IV \| M$)
- Unseal(K, M_{sealed}) = AESdecrypt(K, M_{sealed}).M

Concrete crypto is for experts only!

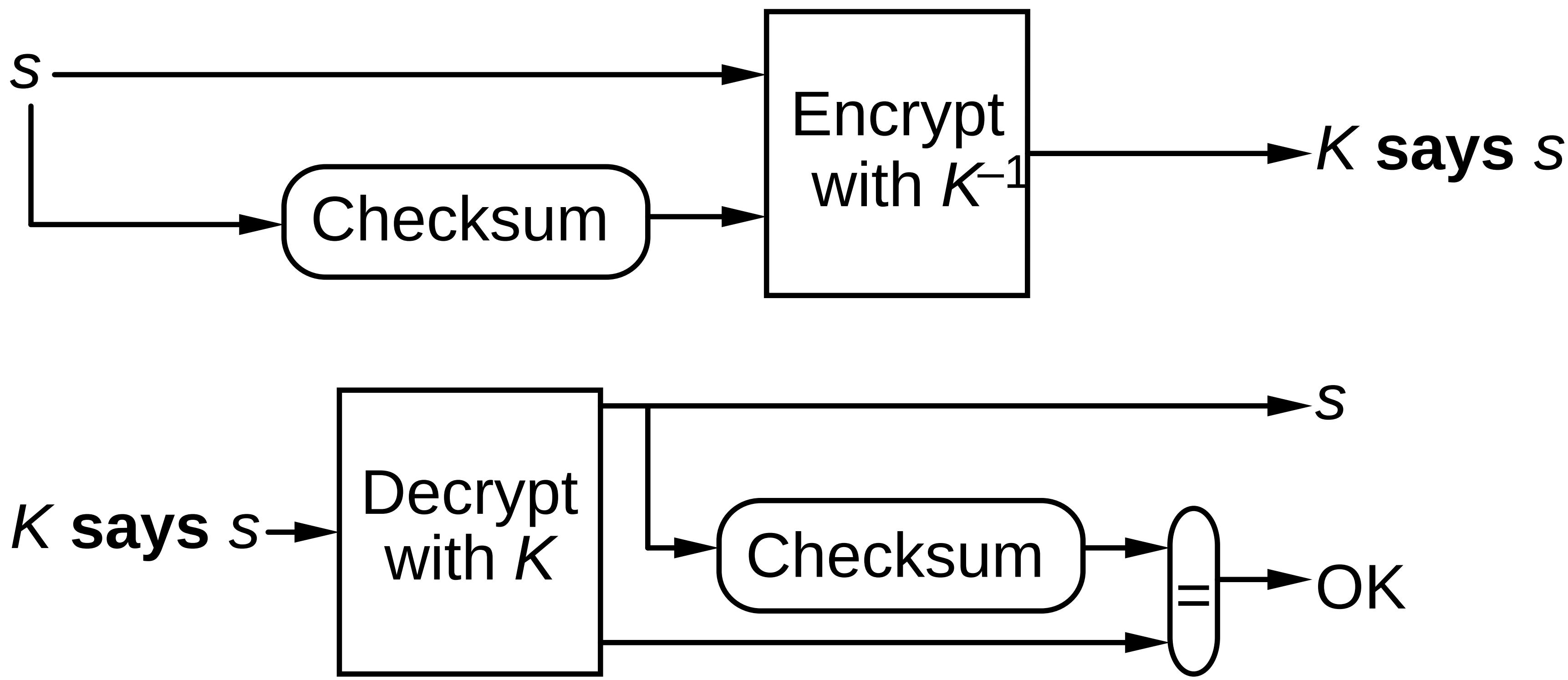
Sign and Seal

Normally when sealing must sign as well!

– Seal(K_{seal}^{-1}, M || Sign(K_{sign}^{-1}, M))

Often Sign is replaced with a checksum ???

Concrete crypto is for experts only!



Public Key vs. Shared Key

Public key: $K \neq K^{-1}$

- Broadcast
- Slow
- Non-repudiable (only one possible sender)
- Used for certificates

Key $\Rightarrow$ name: K_{Intel} **says** $K_{Alice} \Rightarrow$ `Alice@Intel`

Temp key $\Rightarrow$ key: K_{temp} **says** $K_{SSL} \Rightarrow K_{temp}$

K_{Alice} **says** $K_{temp} \Rightarrow K_{Alice}$

Shared key: $K = K^{-1}$

- Point to point
- Fast

Can simulate public key with trusted on-line server

How Fast is Encryption?

			Use	Notes
RSA encrypt	5	ms (25 KB/s)	*sign*	1000 bit modulus
RSA decrypt	0.2	ms (625 KB/s)	*verify*	Exponent=17
SHA-1	70	MBytes/s	*sign*	HMAC
AES	50	MBytes/s	*seal*	256 bit key

On 2 GHz Pentium, Microsoft Visual C++. Data from Wei Dai at www.cryptopp.com

Might be 2x faster with careful optimization

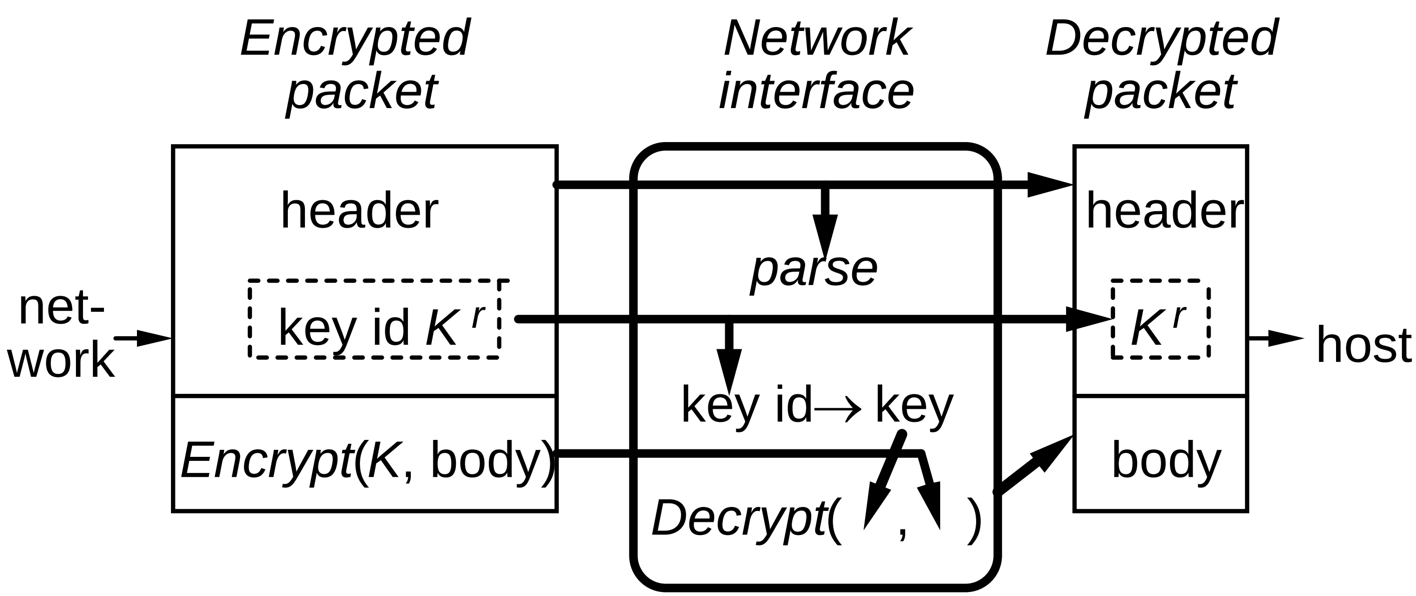
Fast Encryption in Practice

Want to run at network speed.

How? Put encryption into the data path.

Network interface parses the packet to find a *key identifier* and maps it to a key for decryption

Parsing depends on network protocol (e.g., TCP/IP)



Messages on Encrypted Channels

If *K* **says** *s*, we say that *s* is *signed* by *K*

Sometimes we call "*K* **says** *s*" a *certificate*

The channel isn't real-time: *K* **says** *s* is just bits

***K* says *s* can be viewed as**

- An event: *s* transmitted on channel *K*
- A pile of bits which makes sense if you know the decryption key
- A logical formula

Messages vs. Meaning

Standard notation for Seal(K_{seal}^{-1}, M || Sign(K_{sign}^{-1}, M)) is $\{M\}_K$. This does not give the meaning

Must *parse* message bits to get the meaning

- Need *unambiguous* language for *all* K's messages
- In practice, this implies version numbers

Meaning could be a logical formula, or English

- A, B, $\{K\}_{K_{CA}}$ means C **says** (to A) "K is a key". C says nothing about A and B. This is useless
- $\{A, B, K\}_{K_{CA}}$ means C **says** "K is a key for A to talk to B". C says nothing about when K is valid
- $\{A, B, K, T\}_{K_{CA}}$ means C **says** "K is a key for A to talk to B first issued at time T"

Replay

Encryption doesn't stop replay of messages.

Receiver must discard duplicates.

This means each message must be unique.

Usually done with sequence numbers.

Receiver must remember last sequence number while the key is valid.

Transport protocols solve the same problem.

Timeliness

Must especially protect authentication against replay

If C **says** $K_A \Rightarrow A$ to B and Eve records this, she can get B to believe in K_A just by replaying C's message.

Now she can replay A's commands to B.

If she *ever* learns K_A, even much later, she can also impersonate A.

To avoid this, B needs a way to know that C's message is not old.

Sequence numbers impractical—too much long-term state.

Timestamps and Nonces

Timestamps

With synchronized clocks, *C* just adds the time *T*, saying to *B*

K_C **says** $K_A \Rightarrow A$ **at** T

Nonces

Otherwise, *B* tells *C* a *nonce* N_B which is new, and *C* sends to *B*

K_C **says** $K_A \Rightarrow A$ **after** N_B

NAMES FOR PRINCIPALS

Authorization is to named principals. Users have to read these to check them.

`Lampson` may read file `report`

Root names must be defined locally

$K_{Intel} \Rightarrow$ `Intel`

From a root you can build a path name

`Intel/Alice` (= `Alice@Intel`)

With a suitable root principals can have global names.

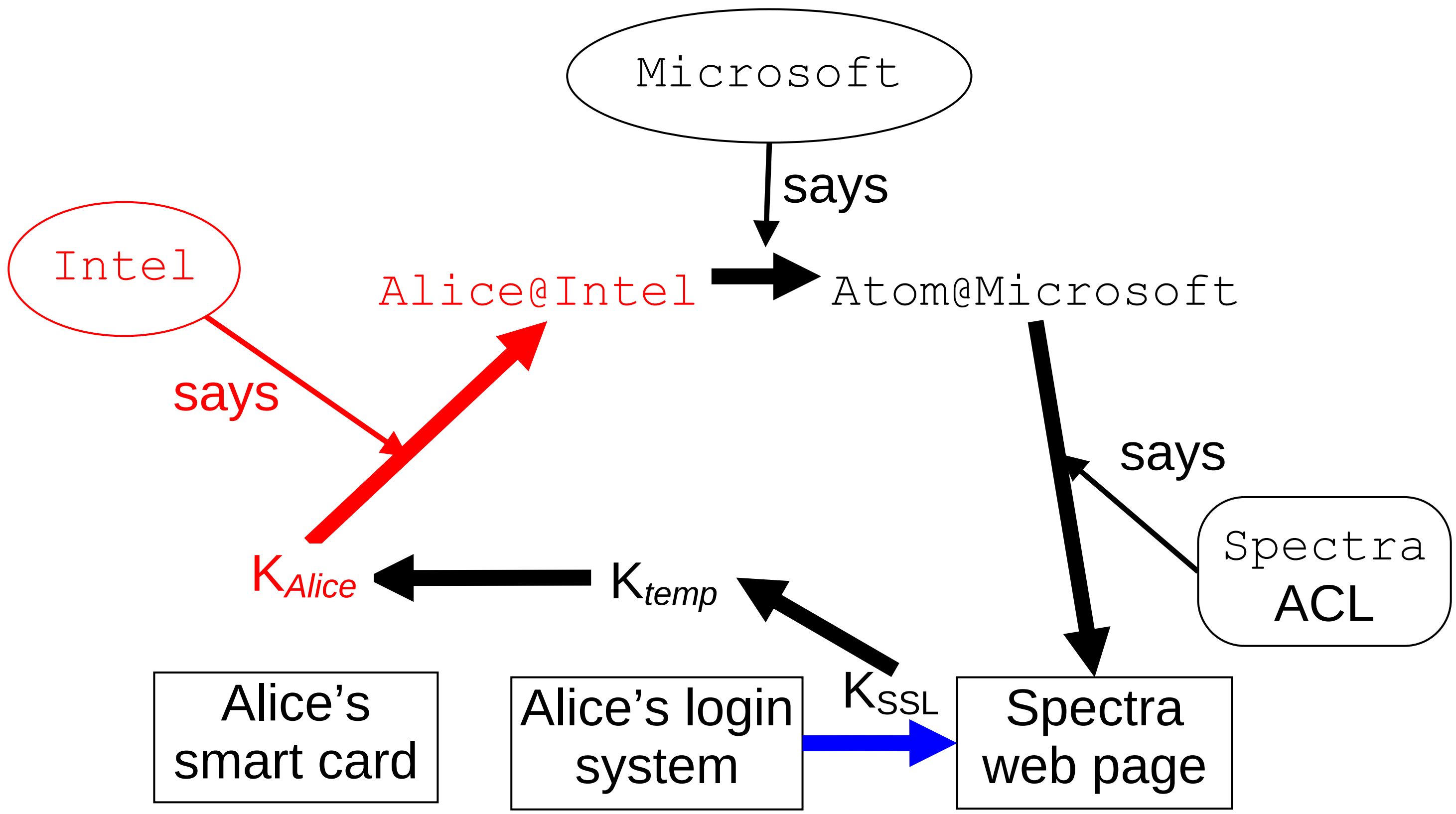
`/DEC/SRC/Lampson` may read file `/DEC/SRC/udir/Lampson/report`

Authenticating Names

$K_{Intel} \Rightarrow$ `Intel` $\Rightarrow$ `Intel/Alice` (= `Alice@Intel`)

$K_{temp} \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow \ldots$

K_{Intel} **says**



Authenticating a Channel

Authentication — who can send on a channel.

$C \Rightarrow P$; C is the channel, P the sender.

Initialization — some such facts are built in**:** $K_{ca} \Rightarrow CA$.

To get new ones, must trust some principal, a *certification authority***.**

Simplest: trust *CA* to authenticate any name:

$CA \Rightarrow$ Anybody

Then *CA* can authenticate channels:

K_{ca} **says** $K_{ws} \Rightarrow$ `WS`

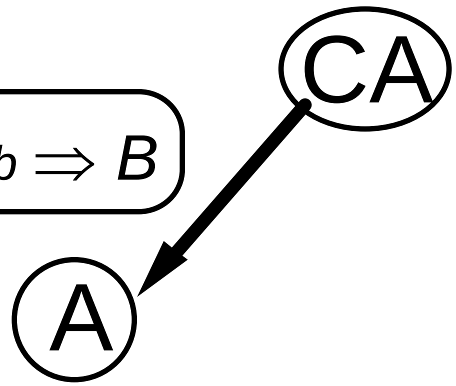
K_{ca} **says** $K_{bwl} \Rightarrow$ `bwl`

One-Way Authentication

CA knows: K_{ca}^{-1} , $K_b \Rightarrow B$

Certificates

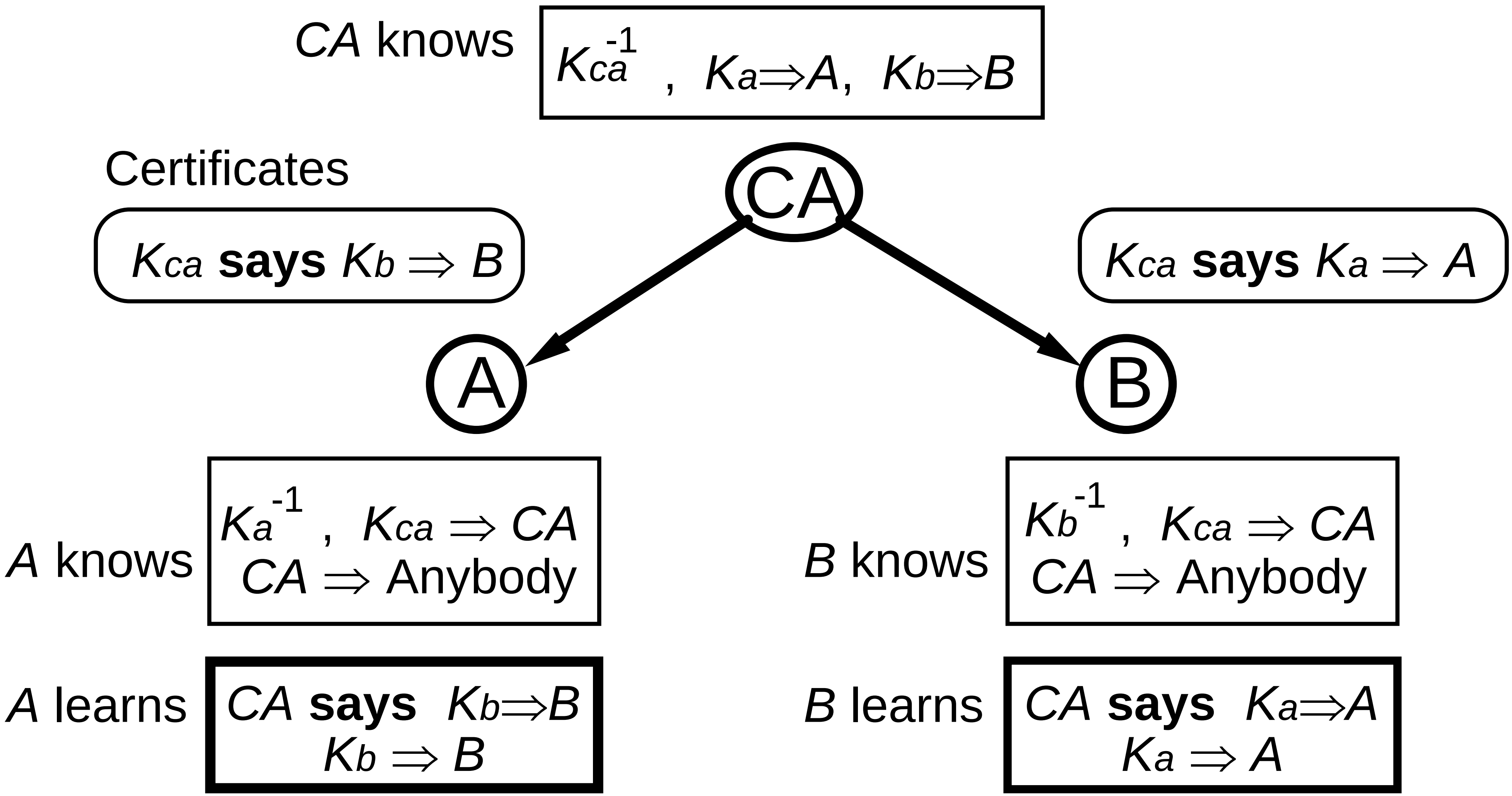
K_{ca} **says** $K_b \Rightarrow B$



A knows: K_a^{-1} , $K_{ca} \Rightarrow CA$
$CA \Rightarrow$ Anybody

A learns: CA **says** $K_b \Rightarrow B$
$K_b \Rightarrow B$

Mutual Authentication



This also works with shared keys, as in Kerberos.

Who Is The CA

"Built In"

CA's in browsers

- –There are lots
- –Because of politics
- –Look at `Tools / Internet options / Content / Publishers / Trusted root certification authorities`

This is a configuration problem

Revocation

Revoke a certificate by making the receiver think it's invalid.

To do this fast, the source of certificates must be online.

This loses a major advantage of public keys, and reduces security.

Solution: countersigning —

An offline CA_{assert}, highly secure.

An online CA_{revoke}, highly timely.

Both must sign for the certificate to be believed, i.e.,

CA_{assert} **and** $CA_{revoke} \Rightarrow$ Anybody

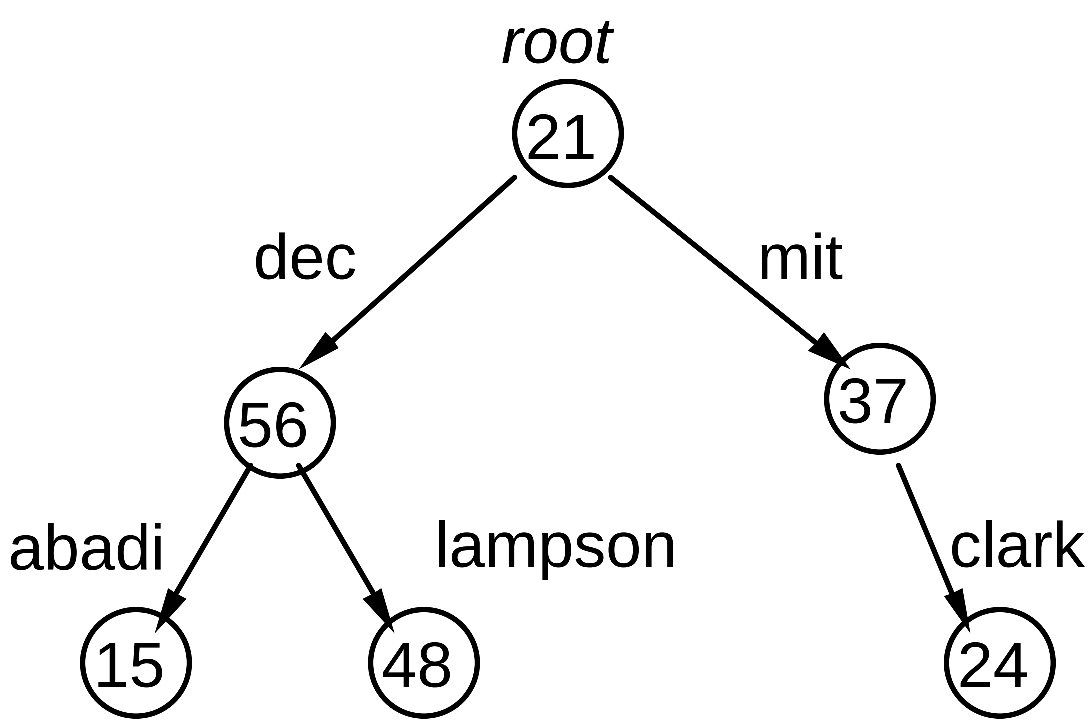
Large-Scale Authentication

A large system can't have *CA* ⇒ Anybody.

Instead, must have many *CA*'s, one for each part.

One natural way is based on a naming hierarchy:

A tree of directories with principals as the leaves



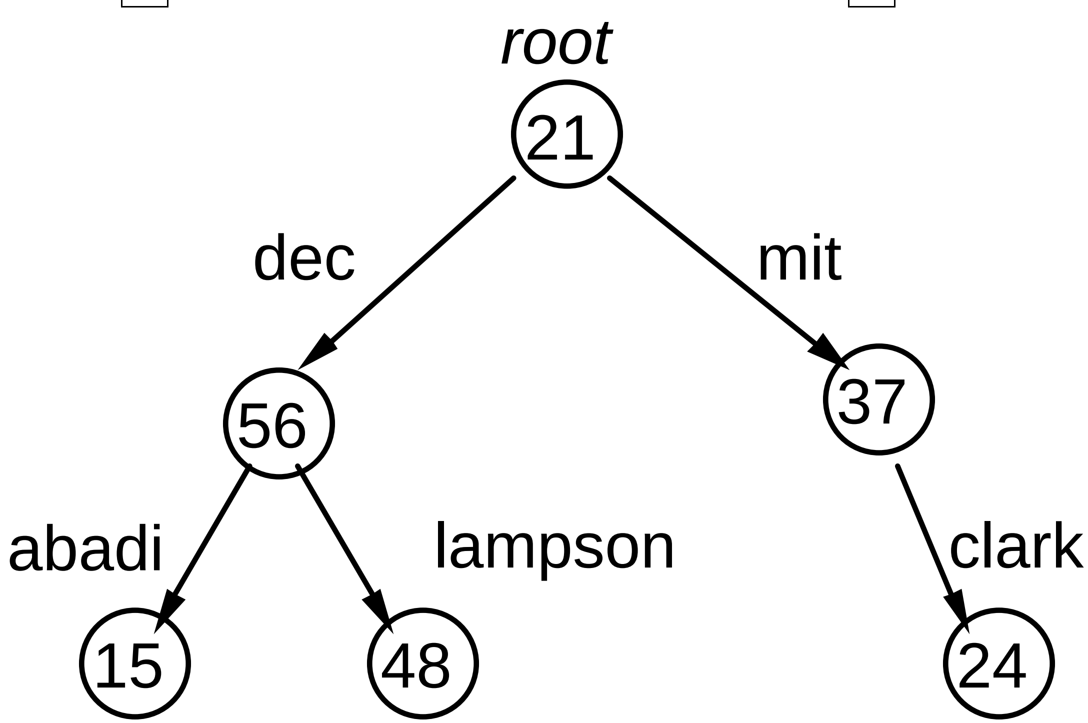
Large-Scale Authentication: Example

Keep trust as local as possible:

Authenticating *A* to *B* needs trust only up to least common ancestor

`dec` **for** `/dec/lampson` → `/dec/abadi`

`root` **for** `/dec/lampson` → `/mit/clark`



Rules for Path Names

New operator **except**:

Informally, P **except** M can speak for P / N as long as $N \neq M$

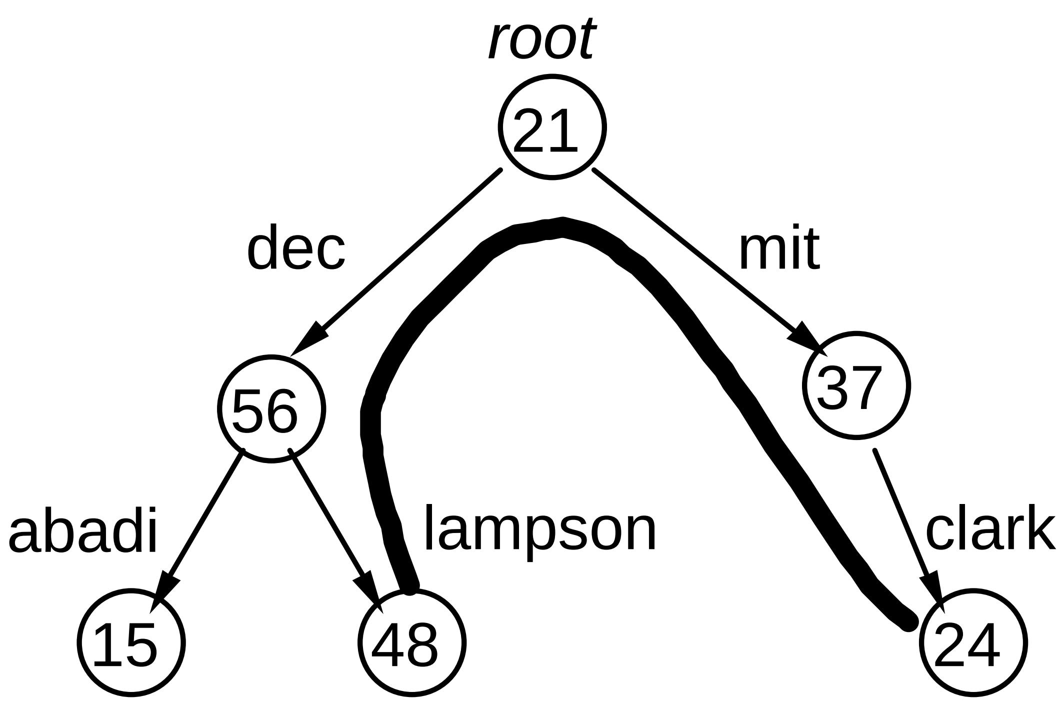
Axioms

P **except** M	$\Rightarrow P$	
(P **except** M) | N	$\Rightarrow P$ / N **except** '..' if $N \neq M$	*child*
(P / N **except** M) | '..'	$\Rightarrow P$ **except** N if $N \neq$ '..'	*parent*

Effect: Authentication can traverse the tree outward from the starting point, but can never retrace its steps

Rules for Path Names: Example

Start with	$C_{lampson}$	$\Rightarrow$ `/dec/lampson` **except** `nil`		*known*
$C_{lampson}$	**says** C_{dec}	$\Rightarrow$ `/dec` **except** `lampson`		*parent*
C_{dec}	**says** C_{root}	$\Rightarrow$ **/** **except** `dec`		*parent*
C_{root}	**says** C_{mit}	$\Rightarrow$ `/mit` **except** "`..`"		*child*
C_{mit}	**says** C_{clark}	$\Rightarrow$ `/mit/clark` **except** "`..`"		*child*

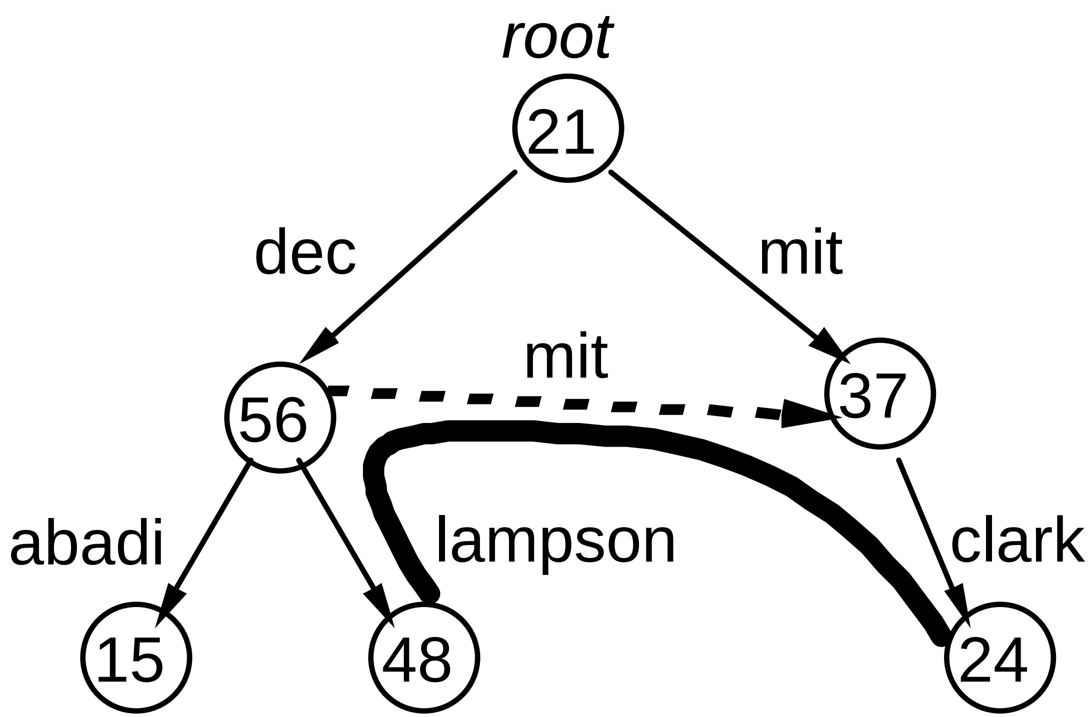


Trusting Fewer Authorities: Cross-Links

For less trust, add links to the tree

Now `lampson` trusts only `dec` for

`/dec/lampson` → `/dec/mit/clark`



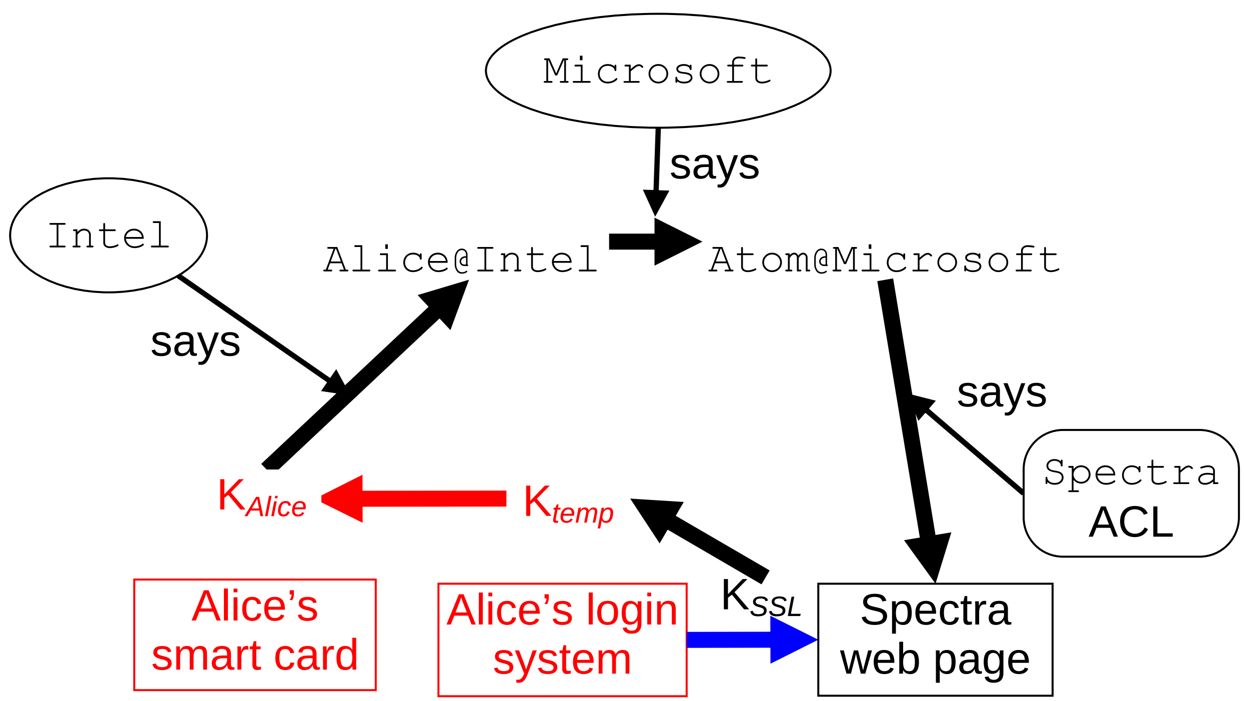
Login

Chain of responsibility:

$K_{SSL} \Rightarrow K_{temp} \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow \ldots$

K_{temp} **says** ↑ K_{Alice} **says** ↑

(SSL setup) (via smart card)



Authenticating Users

Goals

Hide the secret that authenticates the user

Authenticate without disclosing it

Let a node N speak for the user: $N \Rightarrow \texttt{Alice}$

Method

$K_{Alice} \Rightarrow \texttt{Alice}$

K_{Alice} **says** $N \Rightarrow \texttt{Alice}$

${K_{Alice}}^{-1}$ is the user's secret

It can be stored encrypted by her *password*, or better, held inside a *smart card*.

Identifying Nodes for Login Delegation

Usually a workstation has no permanent identity

- –Not true for servers
- –Workstation might have a “meets ITG policy” identity

Need a temporary principal for Alice to delegate to at login

Generate login session key K_{temp}

User Credentials

CA generates:

- user key: $K_{Alice}{}^{-1}$
- child certificate: K_{CA} **says** $K_{Alice} \Rightarrow$ `Alice`

Certificate is public

Where to keep $K_{Alice}{}^{-1}$?

- Smart card
- Encrypted by password
- On a server

Server-mediated Login

Workstation talks to login server

Server confining user's presence

- –Password
- –One-time password
- –Time-varying password
- –Smart card
- –Biometrics

Two-factor Authentication

Problems with passwords

Advantages of physical "tokens"

What if token is stolen?

Combine token and something tied to user

- –Password / PIN
- –Biometrics

Problem with passwords: exhaustive search

Problems with biometrics: not secret, can't change

Login with Node Identity

Check K_{ca} **says** $K_{Alice} \Rightarrow$ `Alice`

Generate K_{temp}^{-1}, a login session key.

Delegate to session key K_{temp} **and** node key K_n

K_{Alice} **says** $(K_{temp}$ **and** $K_n) \Rightarrow K_{Alice}$

Then the session key countersigns with a short timeout, say 30 minutes:

K_{temp} **says** $K_n \Rightarrow K_{temp}$

OS discards K_{temp}^{-1} at logout, and the delegation expires within 30 minutes.

GROUPS and Group Credentials

Defining groups: A group is a principal; its members speak for it

`Alice@Intel` $\Rightarrow$ `Atom@Microsoft`

`Bob@Microsoft` $\Rightarrow$ `Atom@Microsoft`

. . .

Proving group membership: Use certificates

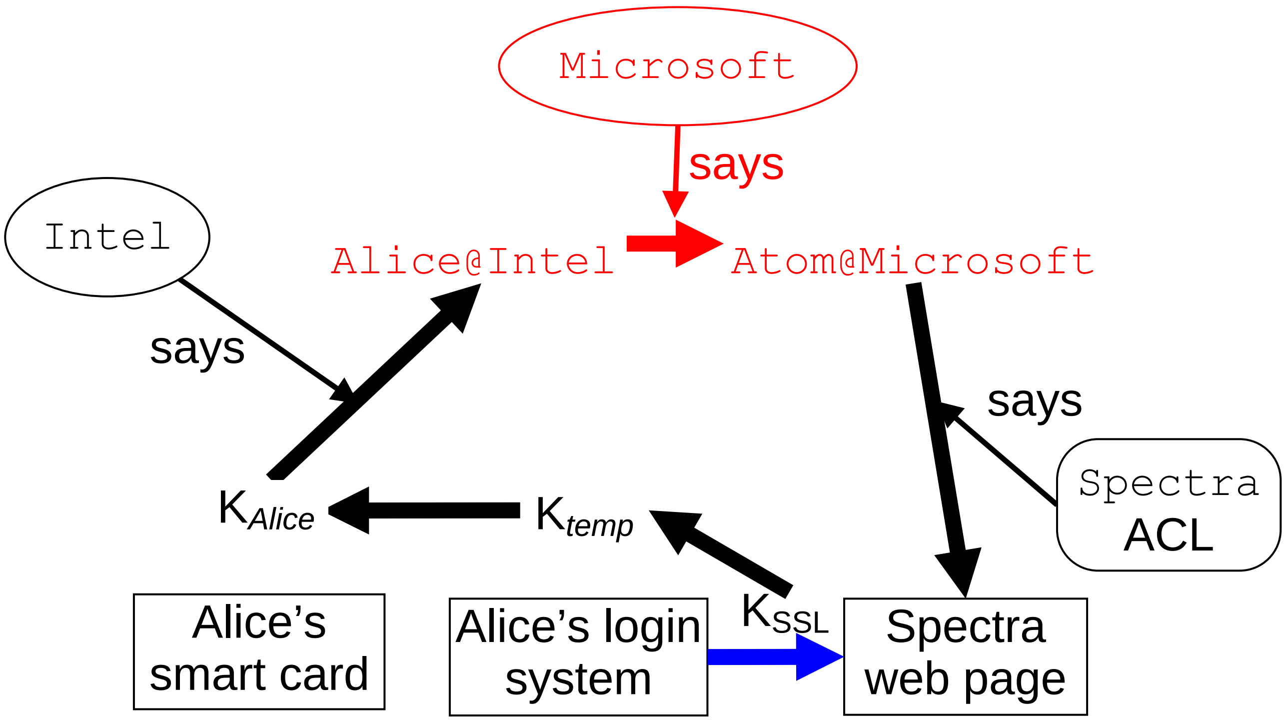
$K_{Microsoft}$ **says** `Alice@Intel` $\Rightarrow$ `Atom@Microsoft`

Authenticating Groups

$K_{Microsoft} \Rightarrow$ `Microsoft` $\Rightarrow$ `Atom@Microsoft`

$\ldots \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow$ `Atom@Microsoft` $\Rightarrow \ldots$

$K_{Microsoft}$ **says**



What Is A Group

Set of principals

– `Alice@Intel` ⇒ `Atom@Microsoft`

Principals with some property

– Resident over 21 years old

– Type-checked program

Can think of the group (or property) as an *attribute* of each principal that is a member

Certifying Properties

Need a trusted authority: $CA \Rightarrow$ `typesafe`

- Actually K_{MS} **says** $CA \Rightarrow K_{MS}$ / `typesafe`

Usually done manually

Can also be done by a program P

- A compiler
- A class loader
- A more general proof checker

Logic is the same: $P \Rightarrow$ `typesafe`

- Someone must authorize the program:
- K_{MS} **says** $P \Rightarrow K_{MS}$ / `typesafe`

Groups As Parameters

An application may have some “built-in” groups

Example: In an enterprise app, each division has

- groups: manager, employees, finance, marketing
- folders: budget, advertising plans, ...

Thus, the steel division is an instance of this, with

- steelMgr, steelEmps, steelFinance, steelMarketing
- folders: steelBudget, steelAdplans, ...

P and *Q*: Separation of Duty

Often we want two authorities for something.

A **and** *B* **says** *s* = (*A* **says** *s*) ∧ (*B* **says** *s*)

We use a compound principal with **and** to express this:

`Lampson` **and** `Taylor`		two users
`Lampson` **and** `Ingres`		user running an application
CA_{assert} **and** CA_{revoke}		online and offline CAs

P or *Q*: Weakening

Sometimes want to weaken a principal

A **or** *B* **says** *s* = (*A* **says** *s*) ∨ (*B* **says** *s*)

- $A \vee B$ **says** "read `f` " needs both $A \Rightarrow_R$ `f` and $B \Rightarrow_R$ `f`
- Example: Java rule—callee ⇒ caller ∨ callee-code
- Example: NT restricted tokens—if process *P* is running `untrusted-code` for `blampson` then
  P ⇒ `blampson` ∨ `untrusted-code`

P as *R*: Roles

To *limit* its authority, a principal can assume a role.

People assume roles: `Lampson` **as** `Professor`

Machines assume roles as nodes by running OS programs: `Vax#1724` **as** BSD4.3a4 = `Jumbo`

Nodes assume roles as servers by running services:
`Jumbo` **as** `SRC-NFS`

Metaphor: a role is a program

Encoding: A **as** $R \equiv A \mid R$ if R is a role

Axioms: $A \Rightarrow_{* \Rightarrow A|R} A$ **as** R if R is a role

B for A: Melding

B for A: B acting on behalf of A

`Workstation22` **for** `Lampson`

`Ingres` **for** `Lampson`

Axiom: $(A \mid B)$ **and** $(B \mid A) \Rightarrow B$ **for** A

To delegate —

A offers: $A \mid B$ **says** $B \mid A \Rightarrow B$ **for** A

B accepts: $B \mid A$ **says** $B \mid A \Rightarrow B$ **for** A

Together: $(A \mid B$ **and** $B \mid A)$ **says** $B \mid A \Rightarrow B$ **for** A

Final delegation: $B \mid A \Rightarrow B$ **for** A

Using a Meld

Suppose the ACL for file `foo` says

`SRC-WS` **for** `Lampson` may read "`foo`"

If we know `WS22` ⇒ `SRC-WS`
then `WS22` **for** `Lampson` may read "`foo`"

Meld Example: Login Credentials

Get K_{bwl}^{-1} from $Encrypt$(PW, K_{bwl}^{-1}) with user's password

Check K_{ca} **says** $K_{bwl} \Rightarrow$ `bwl`

Offer meld to node key K_n:

$K_{bwl} \mid K_n$ **says** $K_n \Rightarrow (K_{ws}$ **as** `Taos`$)$ **for** K_{bwl}

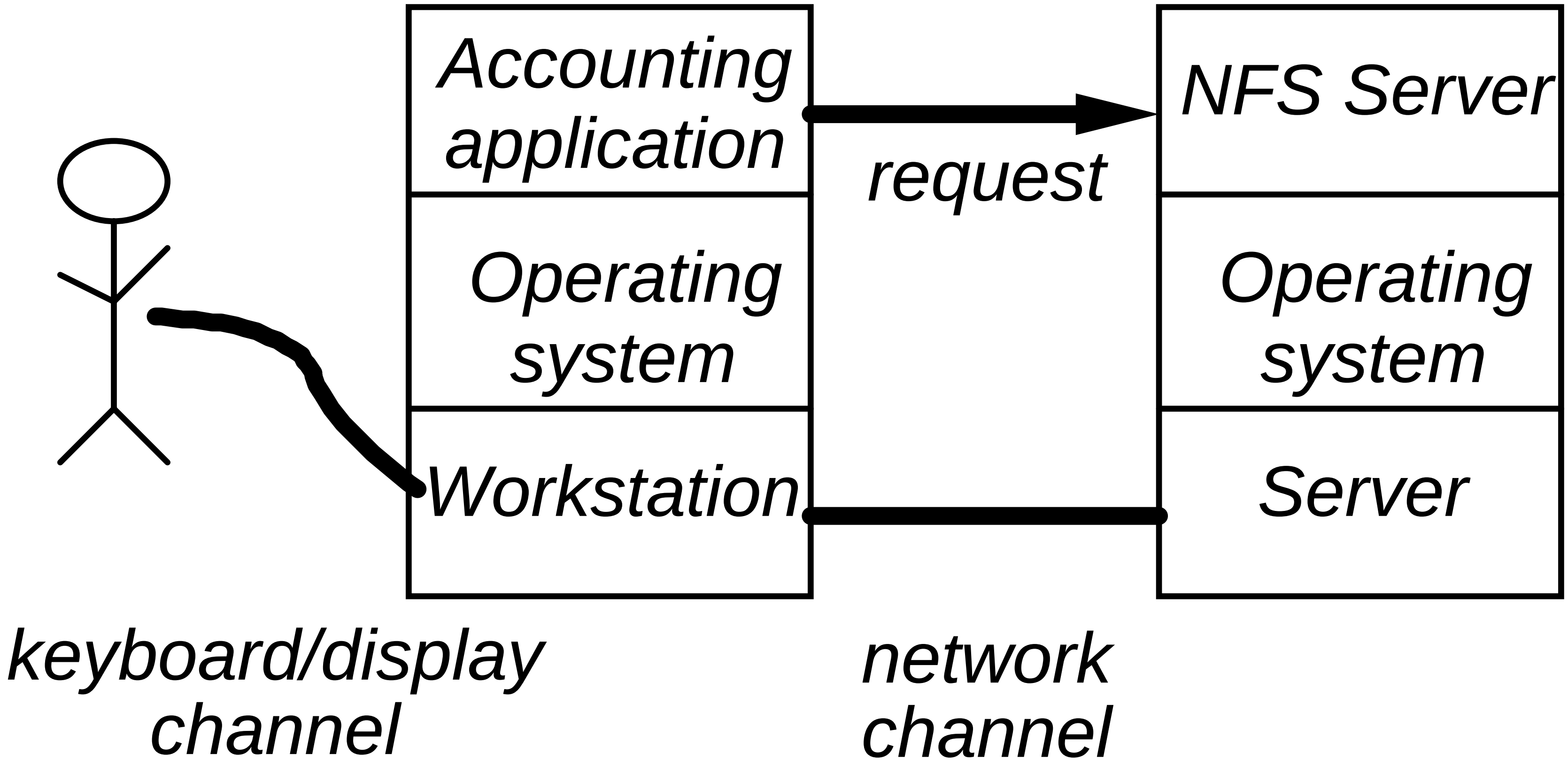
Node accepts meld (given $K_n \Rightarrow K_{ws}$ **as** `Taos`):

$K_n \mid K_{bwl}$ **says** $K_n \Rightarrow (K_{ws}$ **as** `Taos`$)$ **for** K_{bwl}

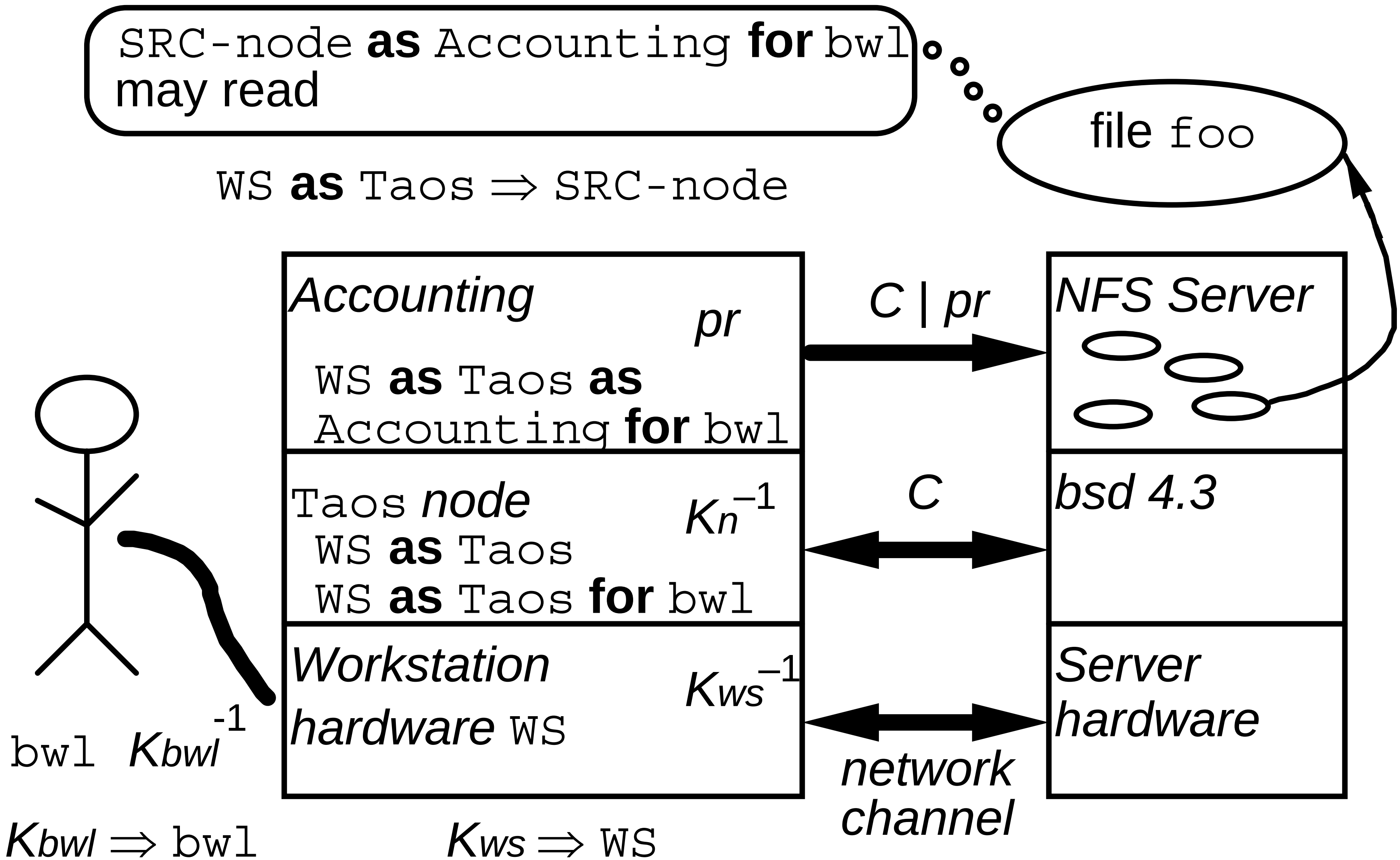
And from the **for** axiom & handoff

$K_n \Rightarrow (K_{ws}$ **as** Taos$)$ **for** K_{bwl}

An Example


Accounting application
Operating system
Workstation
request
NFS Server
Operating system
Server
keyboard/display channel
network channel

Example: Details



AUTHENTICATING SYSTEMS: Loading

A digest X can authenticate a **program** SQL:

– $K_{Microsoft}$ **says** "If image I has digest X then I is `SQL`"

formally $X \Rightarrow K_{Microsoft}$ / `SQL`

– This is just like $K_{Alice} \Rightarrow$ `Alice@Intel`

But a program isn't a principal: it can't say things

To become a principal, a program must be *loaded* into a *host H*

– Booting is a special case of loading

$X \Rightarrow$ `SQL` makes H

– want to run I if H likes SQL

– willing to assert that SQL is running

Authenticating Systems: Roles

A loaded program depends on the *host* it runs on.

- We write *H* **as** `SQL` for `SQL` running on *H*
- *H* **as** `SQL` **says** *s* = *H* **says** `SQL` **says** *s*

H can't *prove* that it's running `SQL`

But *H* can be *trusted* to run `SQL`

- K_{TCS} **says** *H* **as** `SQL` $\Rightarrow K_{TCS}$ / `SQL`

This lets *H* convince others that it's running `SQL`

- *H* **says** $C \Rightarrow K_{TCS}$ / `SQL`

Node Credentials

Machine has some things accessible at boot time.

A secret K_{ws}^{-1} A trusted *CA* key K_{ca}

Boot code does this:

Reads K_{ws}^{-1} and then makes it unreadable.

Reads boot image and computes digest X_{taos}.

Checks K_{ca} **says** $X_{taos} \Rightarrow$ `Taos`.

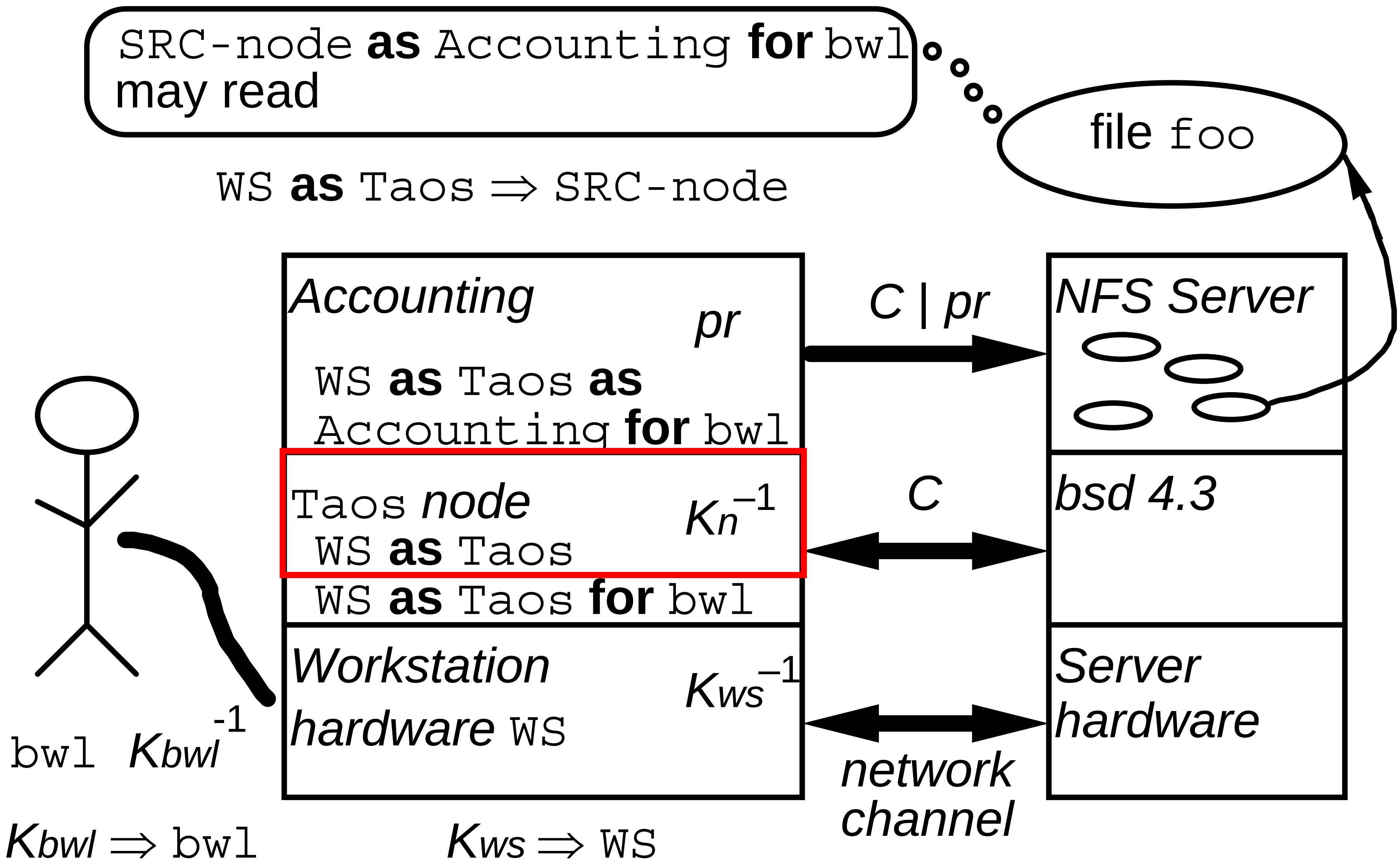
Generates K_n^{-1}, the node key.

Signs credentials K_{ws} **says** $K_n \Rightarrow K_{ws}$ **as** `Taos`

Gives image K_n^{-1} , K_{ca} , credentials, but not K_{ws}^{-1}.

Other systems are similar: K_{ws} **as** `Taos` **as** `Accounting`

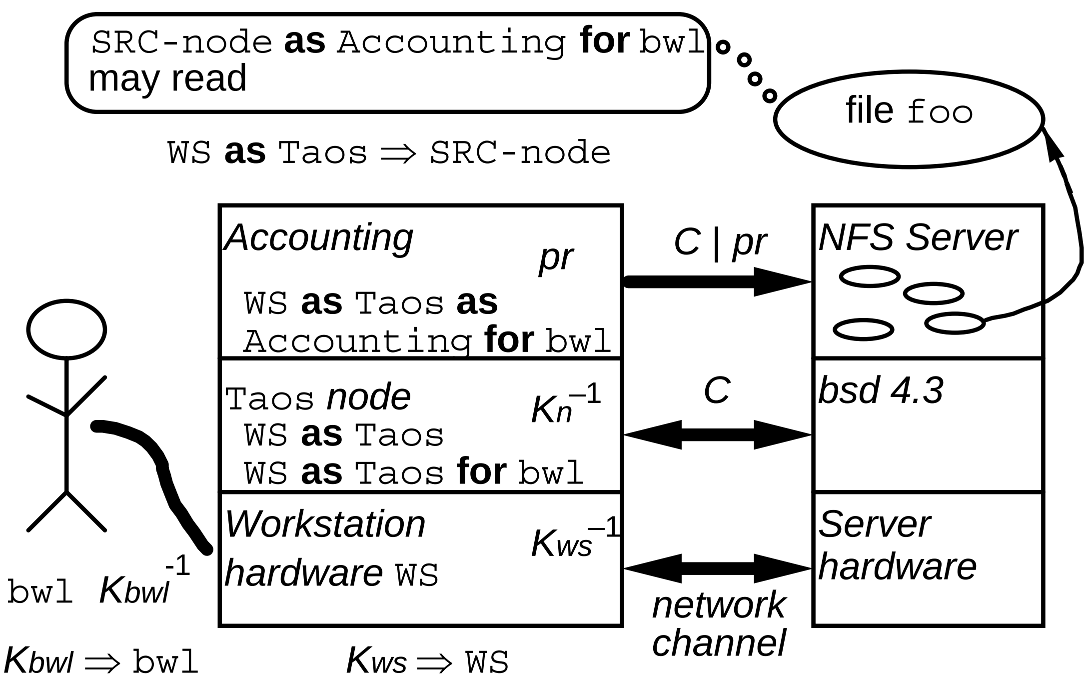
Node Credentials: Example



Example: Server's Access Control

K_{ws} **says** $K_n \Rightarrow K_{ws}$ **as** Taos	node	*credentials*
K_{bwl} **says** $K_n \Rightarrow$ (K_{ws} **as** Taos) **for** K_{bwl}	login session	
K_n says $C \Rightarrow K_n$	channel	
C says $C \mid pr \Rightarrow$ (K_{ws} **as** Taos **as** Accounting) **for** K_{bwl}	process	

$C \mid pr$ says "read file foo" *request*



Sealed Storage: Load and Unseal

Instead of authenticating a new key for a loaded system,

– K_{ws} **says** $K_n \Rightarrow K_{ws}$ **as** `Taos`

Unseal an existing key

– SK = Seal(K_{WSseal}^{-1}, < ACL: Taos, Stuff: $K_{TaosOnWS}^{-1}$>)

– Save(ACL: Taos, Stuff: $K_{TaosOnWS}^{-1}$>) returns SK

– Open(SK) returns $K_{TaosOnWS}^{-1}$ if caller $\Rightarrow$ Taos

Assurance: NGSCB (Palladium)

A cheap, convenient, "physically" separate machine

A high-assurance OS stack (we hope)

A systematic notion of program identity

- Identity = digest of (code image + parameters)
  Can abstract this: *KMS* **says** digest $\Rightarrow K_{MS}$ / `SQL`
- Host certifies the running program's identity:
  H **says** $K \Rightarrow H$ **as** *P*
- Host grants the program access to sealed data
  H seals (data, ACL) with its own secret key
  H will unseal for *P* if *P* is on the ACL

NGSCB Hardware

Protected memory for separate VMs

Unique key for hardware

Random number generator

Hardware attests to loaded software

Hardware seals and unseals storage

Secure channels to keyboard, display

NGSCB Issues

Privacy: Hardware key must be certified by manufacturer

- Use K_{ws} to get one or more certified, anonymous keys from a trusted third party
- Use zero-knowledge proof that you know a mfg-certified key

Upgrade: v7of SQL needs access to v6 secrets

- v6 signs "v7 $\Rightarrow$ v6"
- or, both $\Rightarrow$ SQL

Threat model: Other software

- Won't withstand hardware attacks

NGSCB Applications

Keep keys secure

Network logon

Authenticating server

Authorizing transactions

Digital signing

Digital rights management

Need app TCB: factor app into

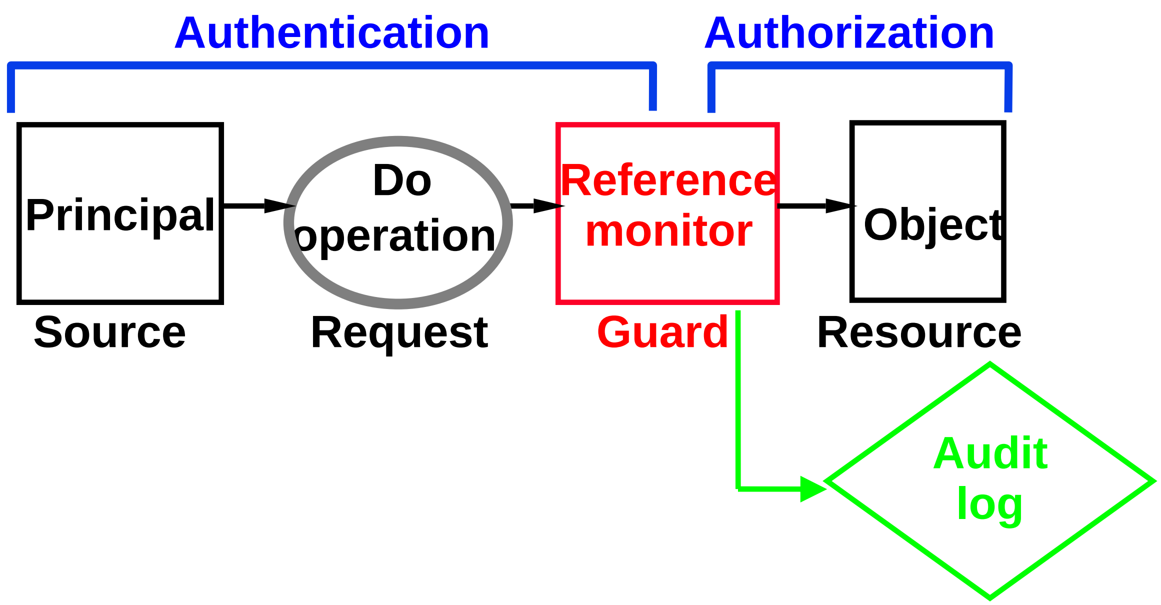
- –a complicated , secure part that runs on Windows
- –a simple, secure part that runs on NGSCB

AUTHORIZATION in Access Control

Guards control access to valued resources.

Structure the system as —

Objects	entities with state.
Principals	can request operations on objects.
Operations	how subjects read or change objects.



Authorization Rules

Rules control the operations allowed
for each principal and object.

Principal may do	*Operation* on	*Object*
`Taylor`	Read	File "`Raises`"
`Lampson`	Send "`Hello`"	Terminal `23`
Process `1274`	Rewind	Tape unit `7`
`Schwarzkopf`	Fire three shots	`Bow gun`
`Jones`	Pay invoice `432`	Account `Q34`

Access Matrix

	File `Raises`	Account `Q34`	Tape unit `7`
`Lampson`	read	deposit	
Process `1274`	read/write		r/w/rewind
Finance dept		deposit/ withdraw	

Representing the Access Matrix

	O1	**O2**	**O3**
P1	T11	T12	
P2	T21		T23
P3		T32	

Capability

ACL

Prefer ACLs for long-tem authorization

- –Usually need to audit who can access a resource

Capabilities are fine as a short-term cache

- –OS file descriptors for open files

Authorization with ACLs

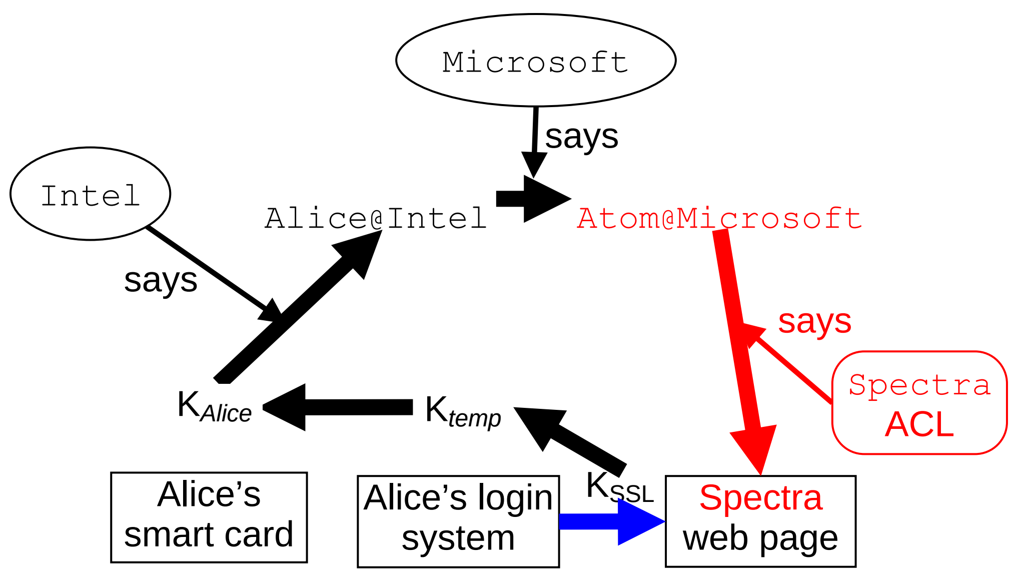
View a resource object *O* as a principal

P on *O*'s ACL means *P* can speak for *O*

- –Permissions limit the set of things *P* can say for *O*

If `Spectra`'s ACL **says** `Atom` can `r/w`, that means

`Spectra` **says** `Atom@Microsoft` $\Rightarrow_{r/w}$ `Spectra`



Access Control Lists (ACLs)

Object *O*'s ACL says: principal *P* may access *O*.

`Lampson` may read and write *O*

(`Jumbo` **for** `SRC`) may append to *O*

ACLs need named principals so people can read them.

Checking access:

Given	a request	*Q* **says** `read` *O*
	an ACL	*P* may `read/write` *O*
Check that		*Q* speaks for *P* $Q \Rightarrow P$
	rights suffice	`read/write` ≥ `read`

Permissions

Principal *A* speaks for *B* about *T* $A \Rightarrow_T B$

If *A* says something in set *T*, *B* does too:

Thus, *A* is stronger than *B*, or responsible for *B*, about *T*

– Precisely: (*A* **says** *s*) $\wedge$ ($s \in T$) implies (*B* **says** *s*)

Permissions represent sets of statements

– *P* may `read/write` *O* = $P \Rightarrow_{r/w} O$

Traditionally they appear only in ACLs, not in delegations, which are unrestricted

T can specify some objects and some of their methods

Expressing sets of statements.

SDSI / SPKI uses “tags” to define sets of statements

A tag is a regular expression, that is, a set of strings

The object interprets a string as a set of statements

- `Read(*.doc)` = reads of files named `*.doc`
- `< 5000` = purchase orders less than $5000

Also can express unions and intersections of sets

- `Read(*.doc)` and `< 5000`

Expressive *T* allows bigger objects: a single permission for all `.doc` files

Transitivity: Intersecting Sets

If $A \Rightarrow_T B$ and $B \Rightarrow_U C$ then $A \Rightarrow_{T \cap U} C$

Why?

$A \Rightarrow_T B \equiv (A$ **says** $s) \wedge (s \in T)$ implies $(B$ **says** $s)$

$B \Rightarrow_U C \equiv (B$ **says** $s) \wedge (s \in U)$ implies $(C$ **says** $s)$

How to implement set intersection ?

–Might be able to simplify the expression

–Always can test s against both T and U

Pragmatics

Authorization must be

- –set up
- –later checked for correctness
- –changed as life goes on

This works best when the authorization data is small and simple

But, want to authorize the "least privilege" needed to get the job done

Conflict. Who wins?

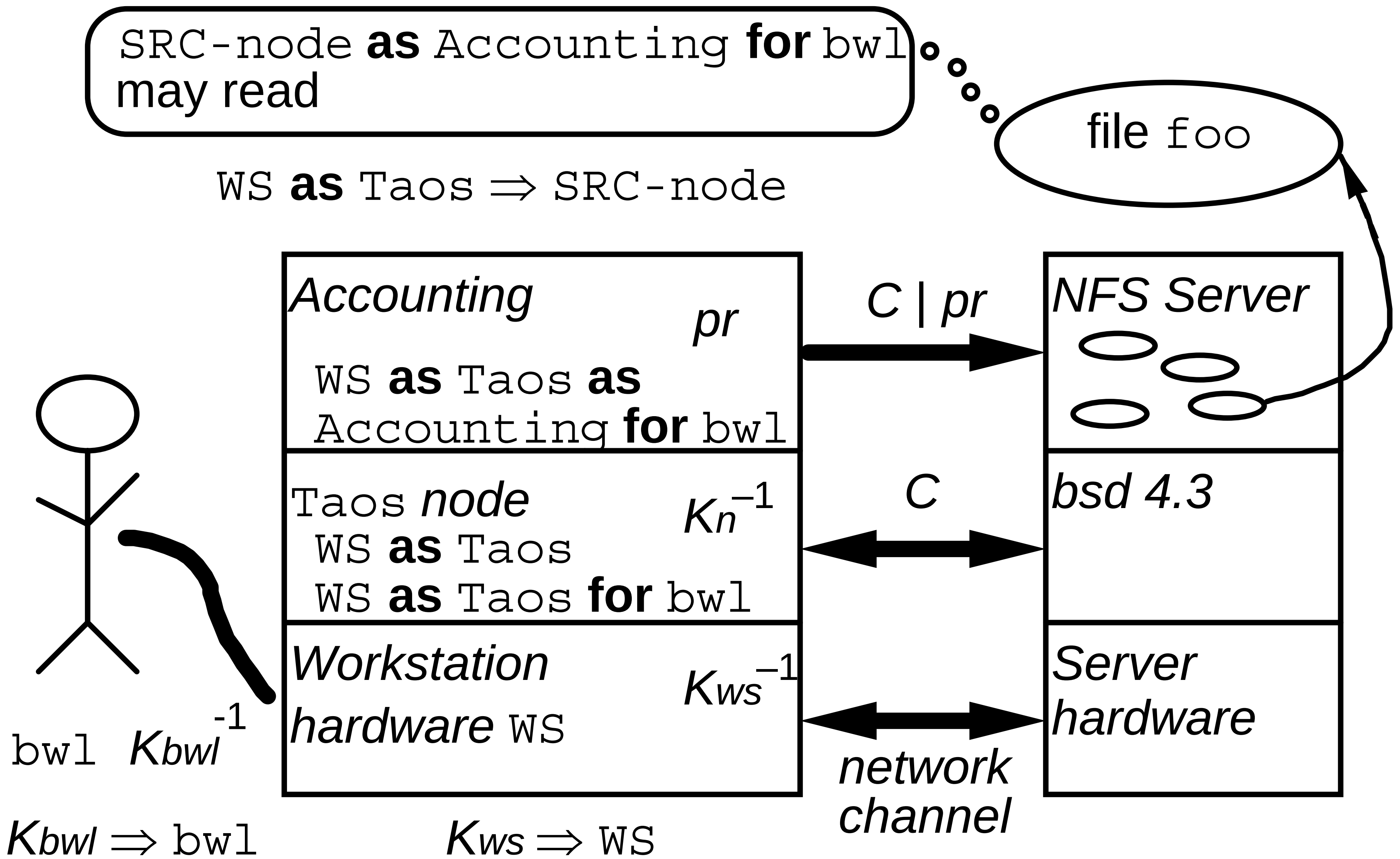
Keeping Authorization Simple

ACLs on large sets of resources

- Big subtrees of the file system
- Large sets of web sites

Usually for *groups*, principals that have some property, such as "Microsoft employee" or "type-safe" or "safe for scripting"

IMPLEMENTATION



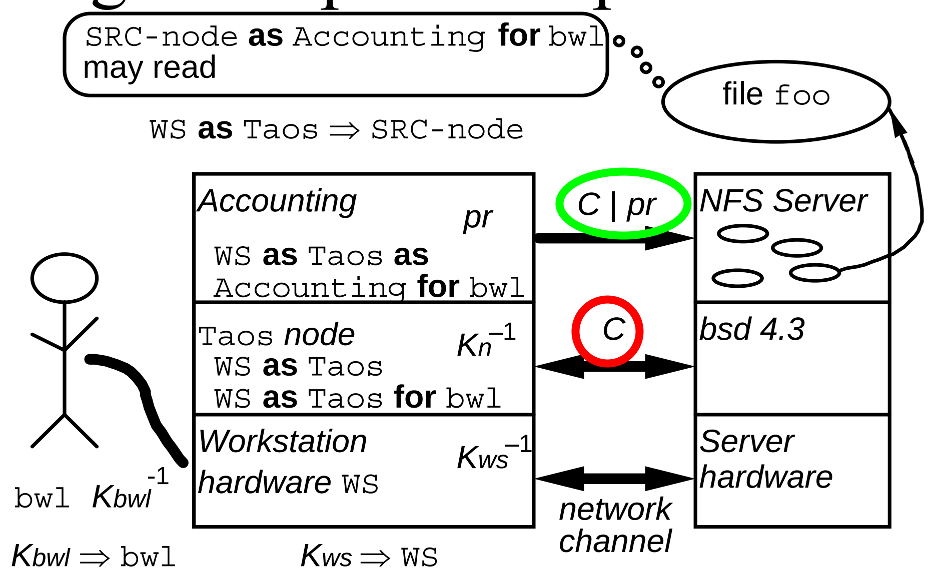
Process Credentials

Make a node-to-node channel $C = \text{DES}(K_{sr})$ using shared key encryption.

Establishing K_{sr} yields $C \Rightarrow K_n$.

The OS multiplexes this single channel among processes.

The OS issues credentials for the subchannels $C \mid pr$.
More multiplexing lets a process speak for several principals.



API for Authentication

Prin represents principals, with a subtype Auth for that a process can speak for

AID is an Auth identifier, a byte string

Authenticating messages

```
GetChan(dest:Address): Chan;
GetAID(p:Auth): AID;
Send(dest:Chan; m:Msg);
Receive(): (Chan, Msg);
GetPrin(c:Chan; aid:AID): Prin;
```

RPC marshals an Auth parameter and unmarshals an aid automatically, thus hiding all these procedures

API for Authentication (2)

Authorization

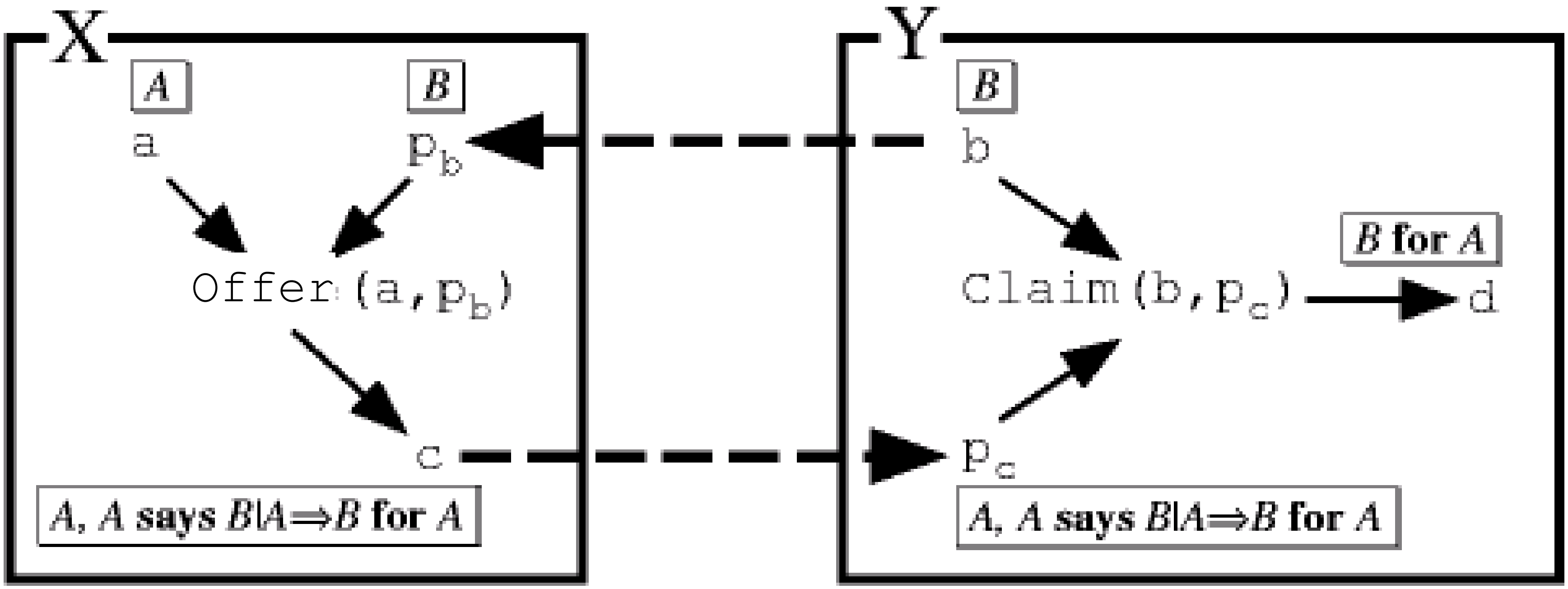
Check(acl:ACL; p:Prin): BOOL

Managing principals

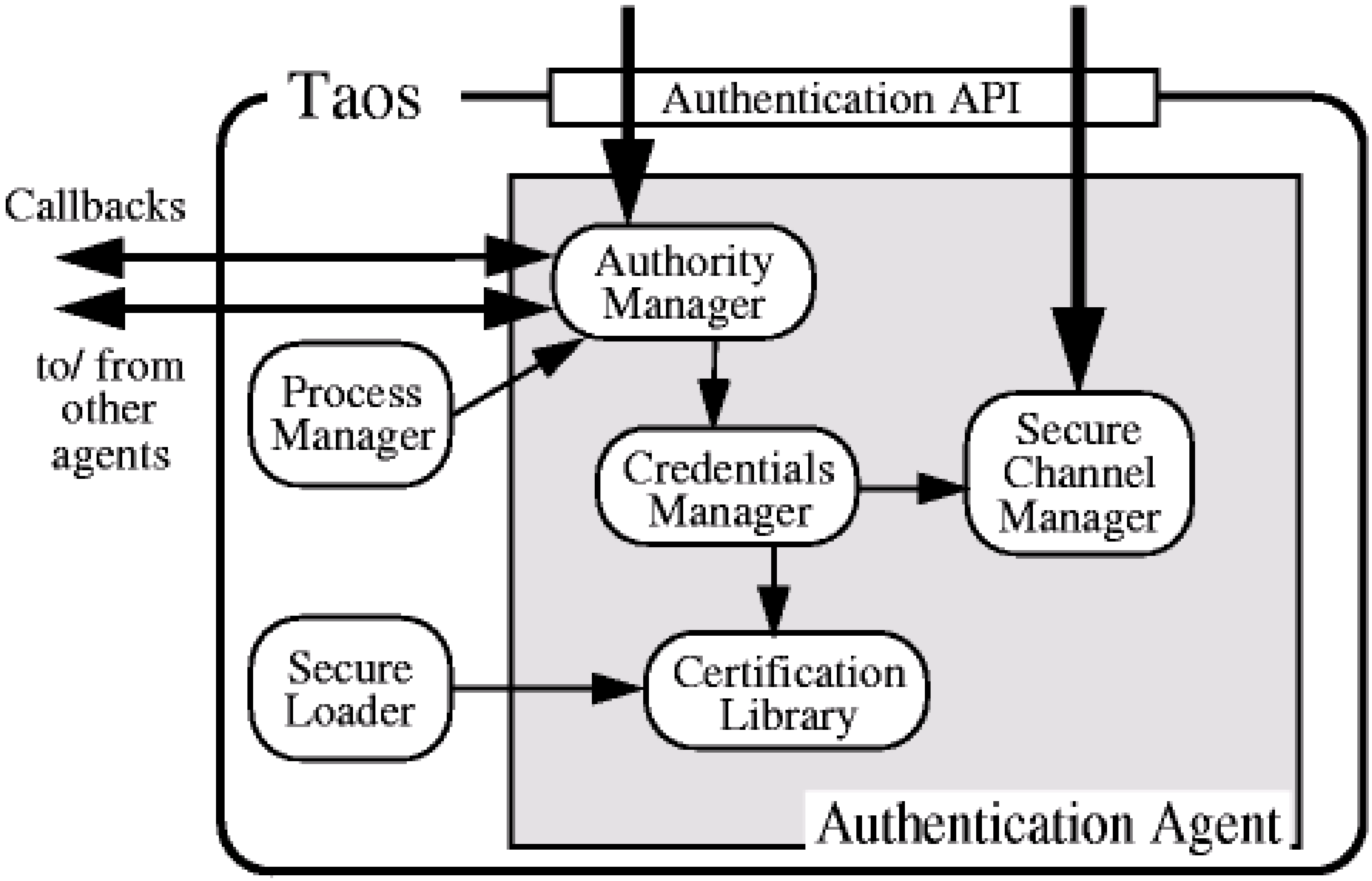
Inheritance(): ARRAY OF Auth;
Login (name, password: TEXT): Auth;
AdoptRole(a:Auth; role:TEXT): Auth;
Offer (a:Auth; b:Prin): Auth;
Claim(b:Auth; meld:Prin): Auth;
Discard(a:Auth; all:BOOL);

API for Melding

Offer (a:Auth; b:Prin): Auth;
Claim(b:Auth; meld :Prin): Auth;



Implementation Internals



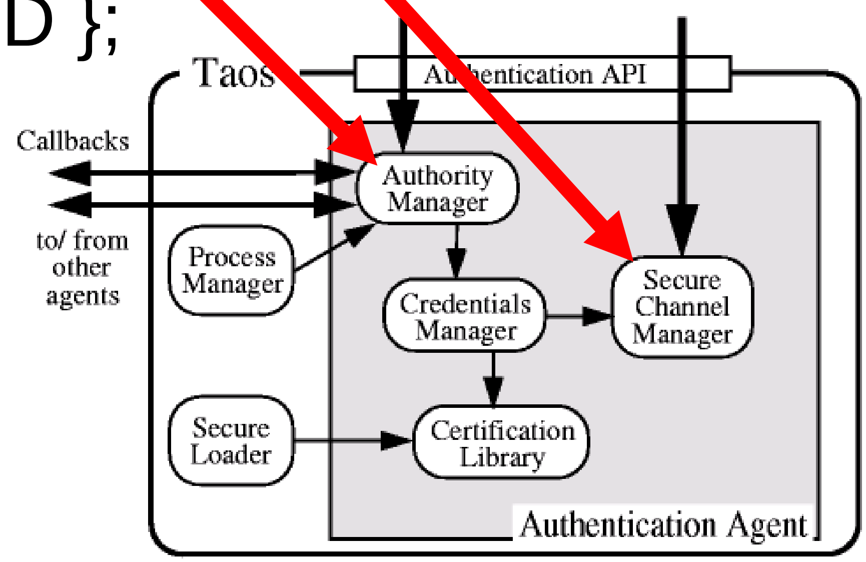
Secure Channel, Authority Managers

The **secure channel manager** creates process-to-process secure channels.

TYPE ChanID = { nk:KeyDigest; pr:INT; addr:Address };
GetChanID(ch:Chan): ChanID;
PTagFromChan(c:ChanID): PTag;

The **authority manager** associates Auths with processes and handles authentication requests.

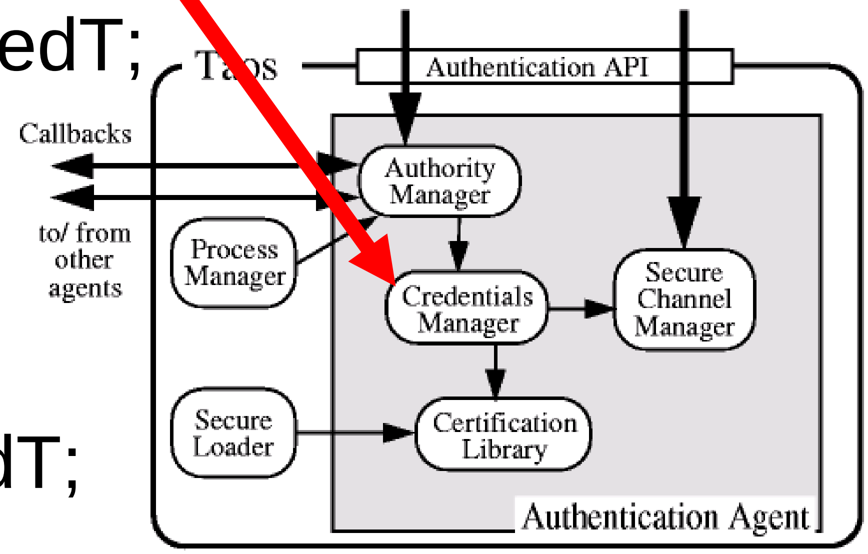
TYPE PrinID = { ch:ChanID; aid:AID };
Delegate(a:Auth; ptag:PTag);
PurgePTag(ptag: PTag);



Credentials Manager

Maintains credentials for local processes and validates certificates from other nodes.

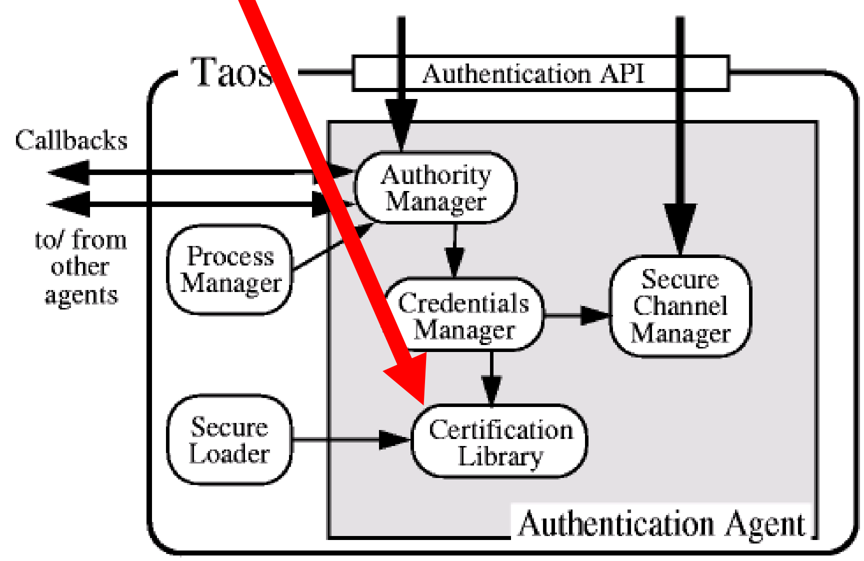
```
TYPE Cred = TEXT, CredT = ...;
New(name, password: TEXT): CredT;
AdoptRole(t:CredT; role: TEXT): CredT;
Sign(t:CredT; p:PrinID): Cred;
Validate(cr:Cred; p:PrinID): TEXT;
Extract(cr:Cred): Cred;
SignMeld(t:CredT; cr:Cred): Cred;
ClaimMeld(t:CredT; cr:Cred): CredT;
```



Certification Library

Establishes a trusted mapping between principal names and keys, and between groups and their members.

CheckKey(name:TEXT; k:Key): BOOL;
IsMember(name, group: TEXT): BOOL;
CheckImage(d:Digest; prog, cert: TEXT);



Interfaces to Authentication

There are two styles:

Implicit in communication

Authenticate at connection establishment; a client can find out the principal that the connection speaks for.

Authenticate as part of a remote procedure call; the procedure can find the principal the caller speaks for.

Explicit

Pass the sending principal explicitly in every message.

More flexible: can pass more than one principal.

Either way abstracts authentication protocol details.

The interface just tell you the authenticated principal.

Implementing Authentication: Push vs. Pull

Two ways for receiver *B* to authenticate sender *A*:

Push credentials: sender to receiver (Windows SIDs):

A sends *B* credentials of channel *C*: proof that $C \Rightarrow A$.

Pull credentials: receiver from sender (ACLs, Taos):

A just sends to *B* on *C*. *B* calls back to *A* to get credentials. *B* may *cache* them

Variations

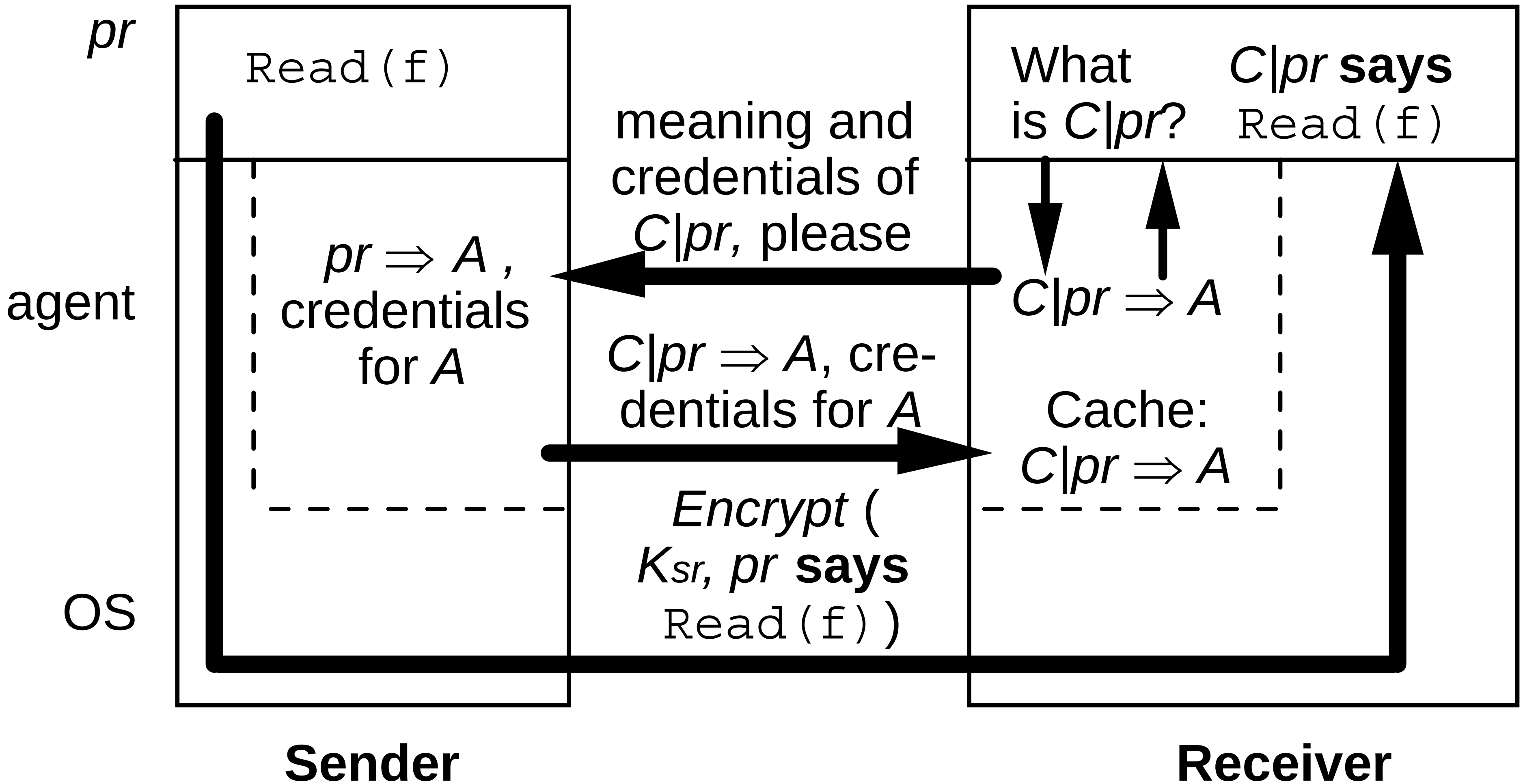
A pushes part of the credentials, and *B* pulls the rest.

B gets part of the credentials from *A*, stores part himself, and gets part from network services.

Pull Authentication: Example

Process *pr* sends on *C* | *pr*; OS multiplexes *C*.

Receiver's *auth agent* asks for *C* | *pr* credentials.



Abbreviations

Extend pull to names:

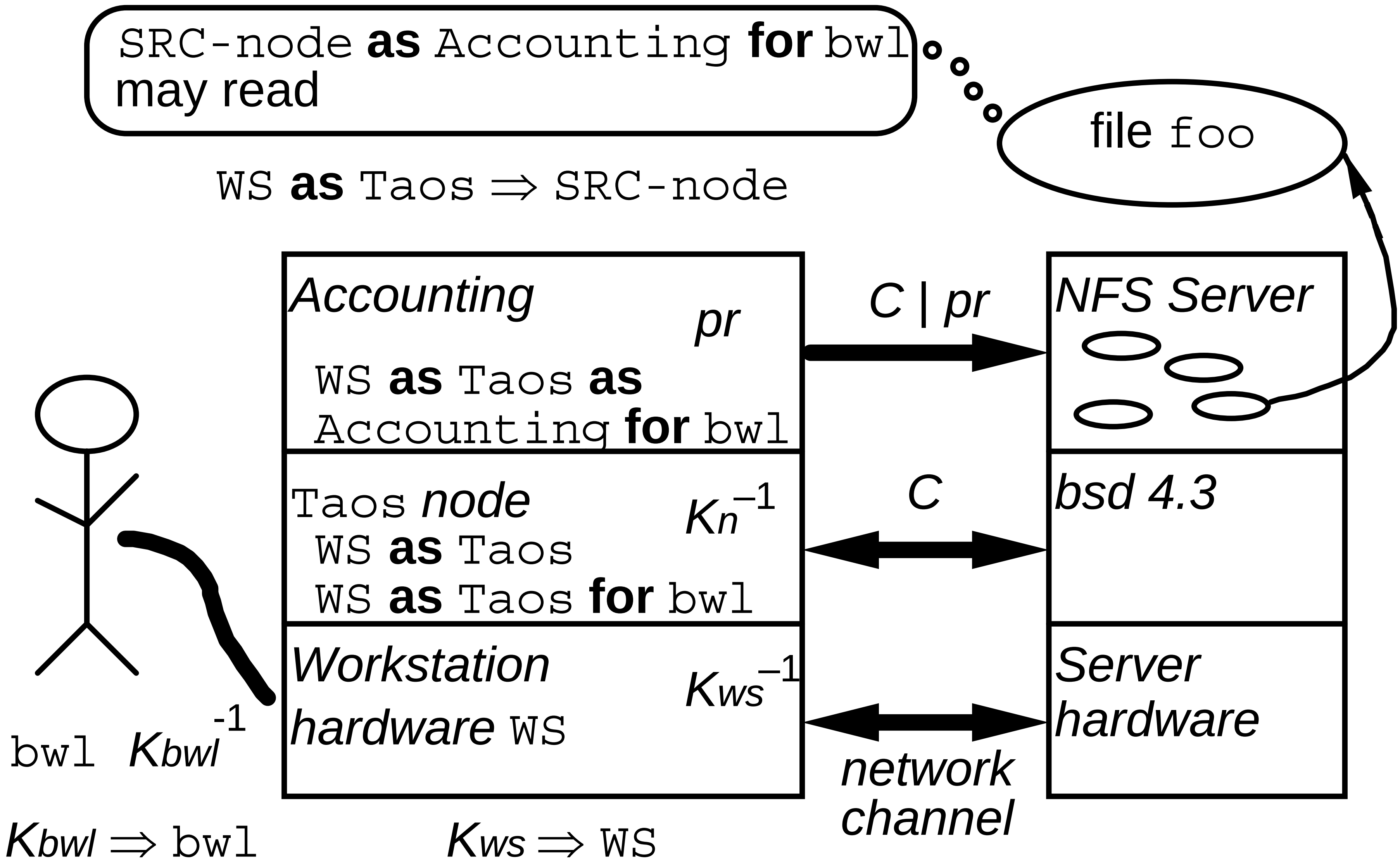
- Sender has some long names for principals
- Choose a short (integer, byte string) abbreviation for each name
  - AID is an example
- Send the short name; if receiver doesn't know its definition, it calls back to pull it over

Short names must not be reused

Receiver can discard its short name cache anytime

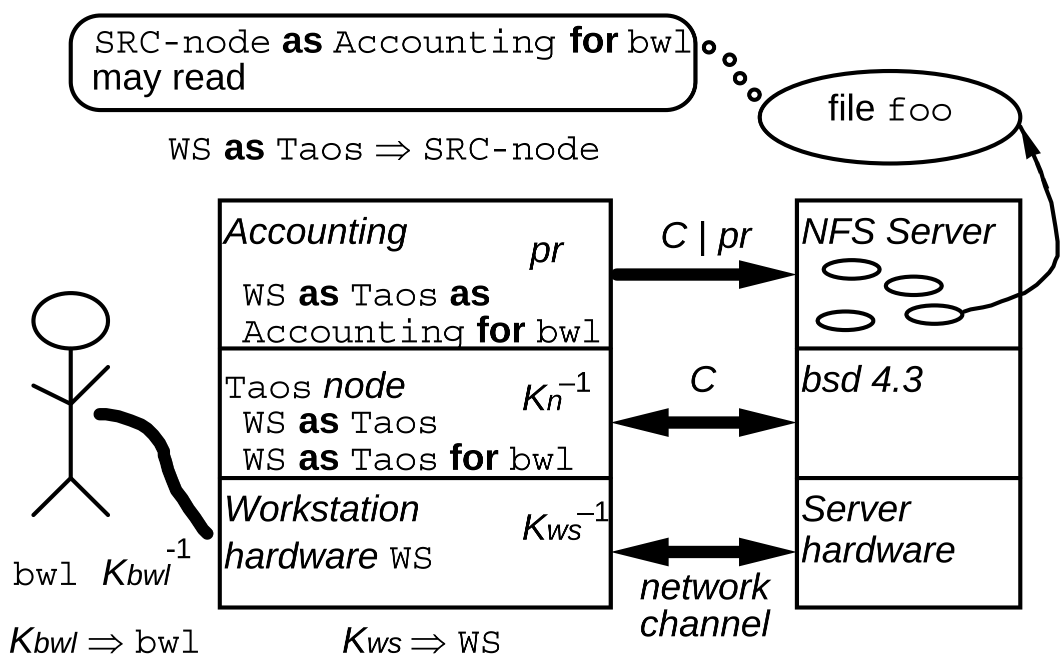
- It will be refreshed by pull if needed

Example: Details



The Example Reviewed

K_{ws} **says** $K_n \Rightarrow K_{ws}$ **as** `Taos`	node	*credentials*
K_{bwl} **says** $K_n \Rightarrow$	login	
(K_{ws} **as** `Taos`) **for** K_{bwl}	session	
K_n says $C \Rightarrow K_n$	channel	
C says $C \mid pr \Rightarrow$ (K_{ws} **as** `Taos` **as** `Accounting`) **for** K_{bwl}	process	
$C \mid pr$ says "`read file foo`"		*request*



Bytes vs. Secure Data

Can choose the the flow and storage of encrypted bytes optimize

- –simplicity
- –performance
- –availability.

Public key = off-line broadcast channel.

- –Write certificate on a tightly secured offline system
- – Store it in untrusted system; anyone can verify it.

Certificates are secure answers to pre-determined queries, (for example, “What is Alice’s key?”) not magic.

- –It’s the same to query an on-line secure database (say Kerberos KDC) over a secure channel

Caching Secure Data

Caching can greatly improve performance

It doesn't affect security or availability

- as long as there's always a way to reload the cache if gets cleared or invalidated

Auditing

Checking access:

Given a request Q **says** `read` O

an ACL P may `read/write` O

Check that Q speaks for P $\boxed{Q \Rightarrow P}$

rights are enough `read/write` $\geq$ `read`

Auditing

Each step is justified by

a signed statement, or

a rule

Implement: Tools and Assurance

Services — tools for implementation

Authentication	Who said it?
Authorization	Who is trusted?
Auditing	What happened?

Trusted computing base

Keep it small and simple

Validate each component carefully

The “Speaks for” Relation $\Rightarrow$

Principal *A* speaks for *B* about *T* $A \Rightarrow_T B$

If *A* says something in set *T*, *B* does too:

Thus, *A* is stronger than *B*, or responsible for *B*, about *T*

Precisely: (*A* **says** *s*) $\wedge$ ($s \in T$) implies (*B* **says** *s*)

These are the links in the chain of responsibility

Examples

`Alice`	$\Rightarrow$ `Atom`	*group of people*
Key `#7438`	$\Rightarrow$ `Alice`	*key for Alice*

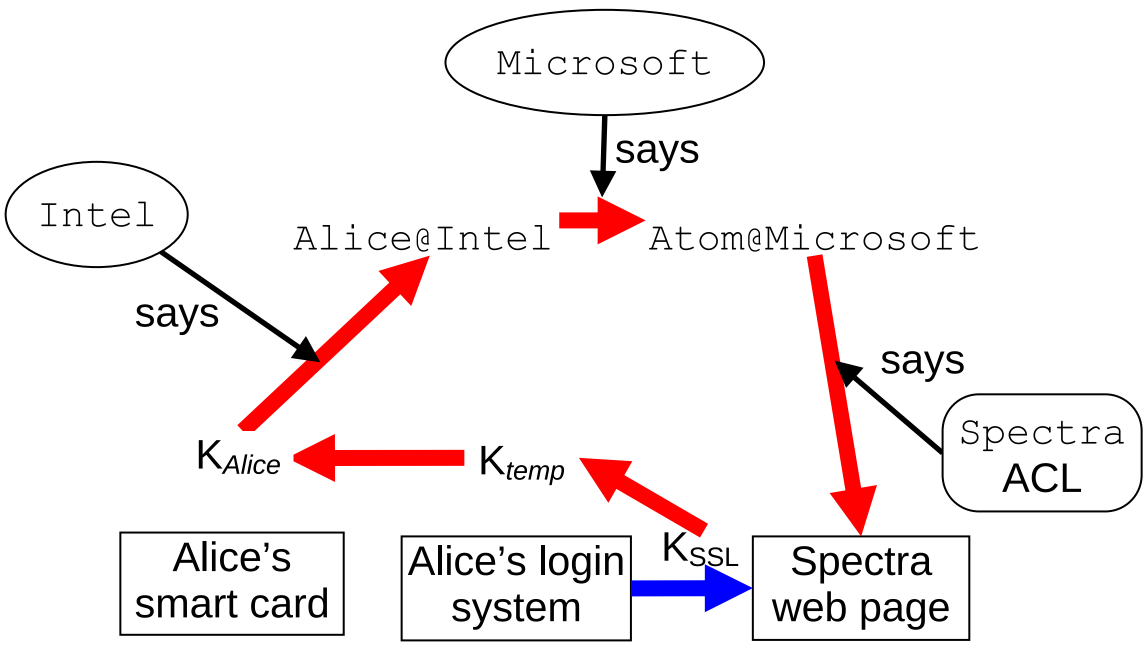
Chain of responsibility

Alice at Intel, working on Atom, connects to Spectra, Atom's web page, with SSL

Chain of responsibility:

$$K_{SSL} \Rightarrow K_{temp} \Rightarrow K_{Alice}$$

$$\Rightarrow \texttt{Alice@Intel} \Rightarrow \texttt{Atom@Microsoft} \Rightarrow \texttt{Spectra}$$



References

Look at my web page for these:
`research.microsoft.com/lampson`

Computer security in the real world. At ACSAC 2000. A shorter version is in *IEEE Computer*, June 2004

Authentication in distributed systems: Theory and practice. *ACM Trans. Computer Sys.* **10**, 4 (Nov. 1992)

Authentication in the Taos operating system. *ACM Trans. Computer Systems* **12**, 1 (Feb. 1994)

SDSI—A Simple Distributed Security Infrastructure, Butler W. Lampson and Ronald L. Rivest.

References

Jon Howell and David Kotz. End-to-end authorization. In *Proc. OSDI 2000*

Paul England et al. A Trusted Open Platform, *IEEE Computer*, July 2003

Ross Anderson—www.cl.cam.ac.uk/users/rja14

Bruce Schneier—*Secrets and Lies*

Kevin Mitnick—*The Art of Deception*