Practical Principles for Computer Security

Butler Lampson

Marktoberdorf

August 2006

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

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`

label(f (x)) = max(label(f), label(x))

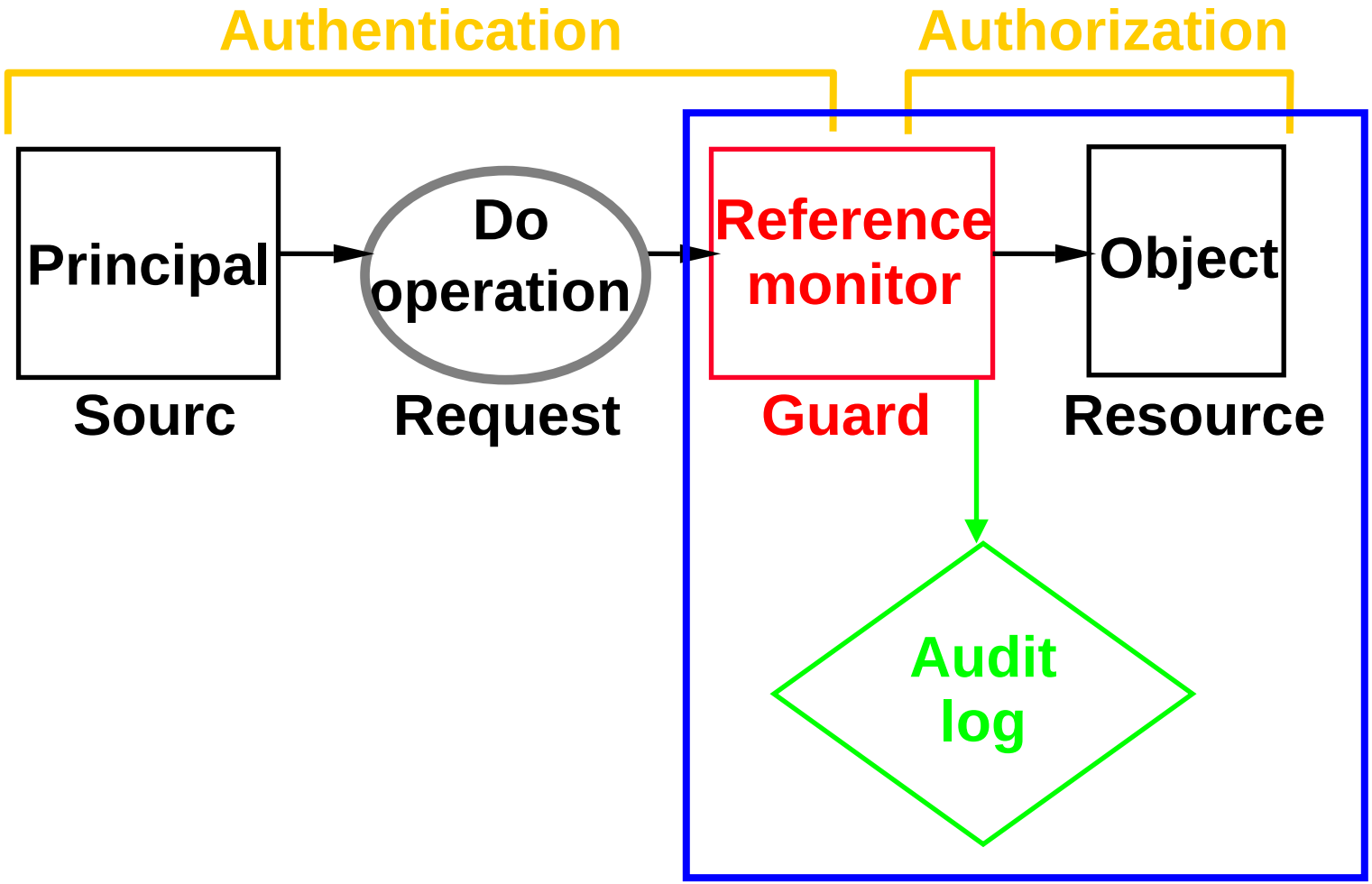
Labels can keep track of data properties:

- 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

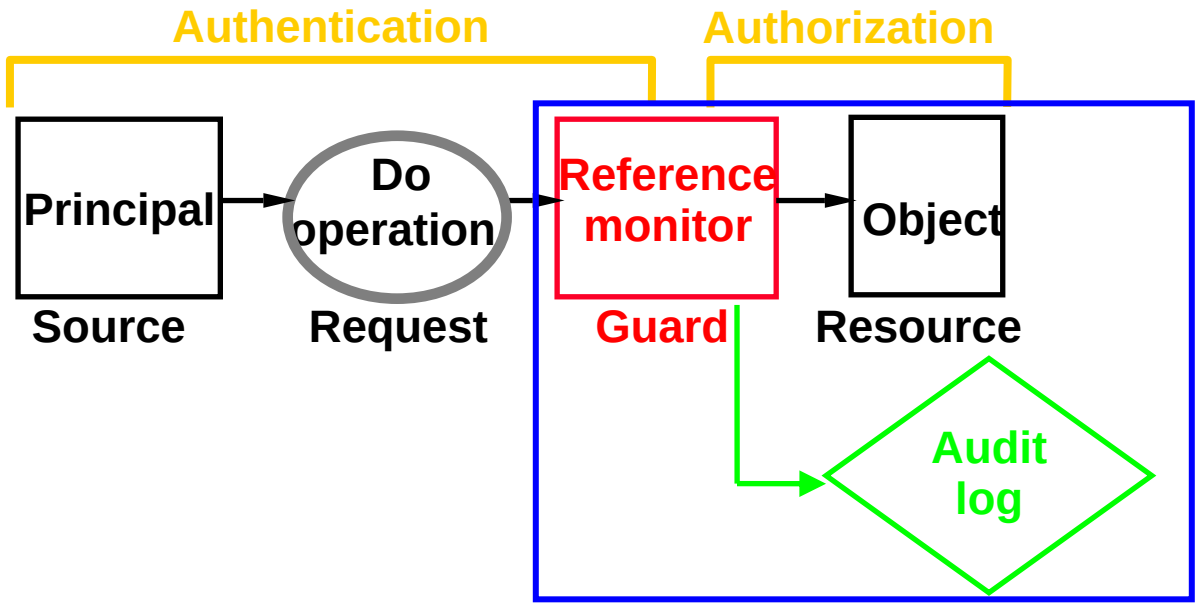
Guards control access to valued resources.

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

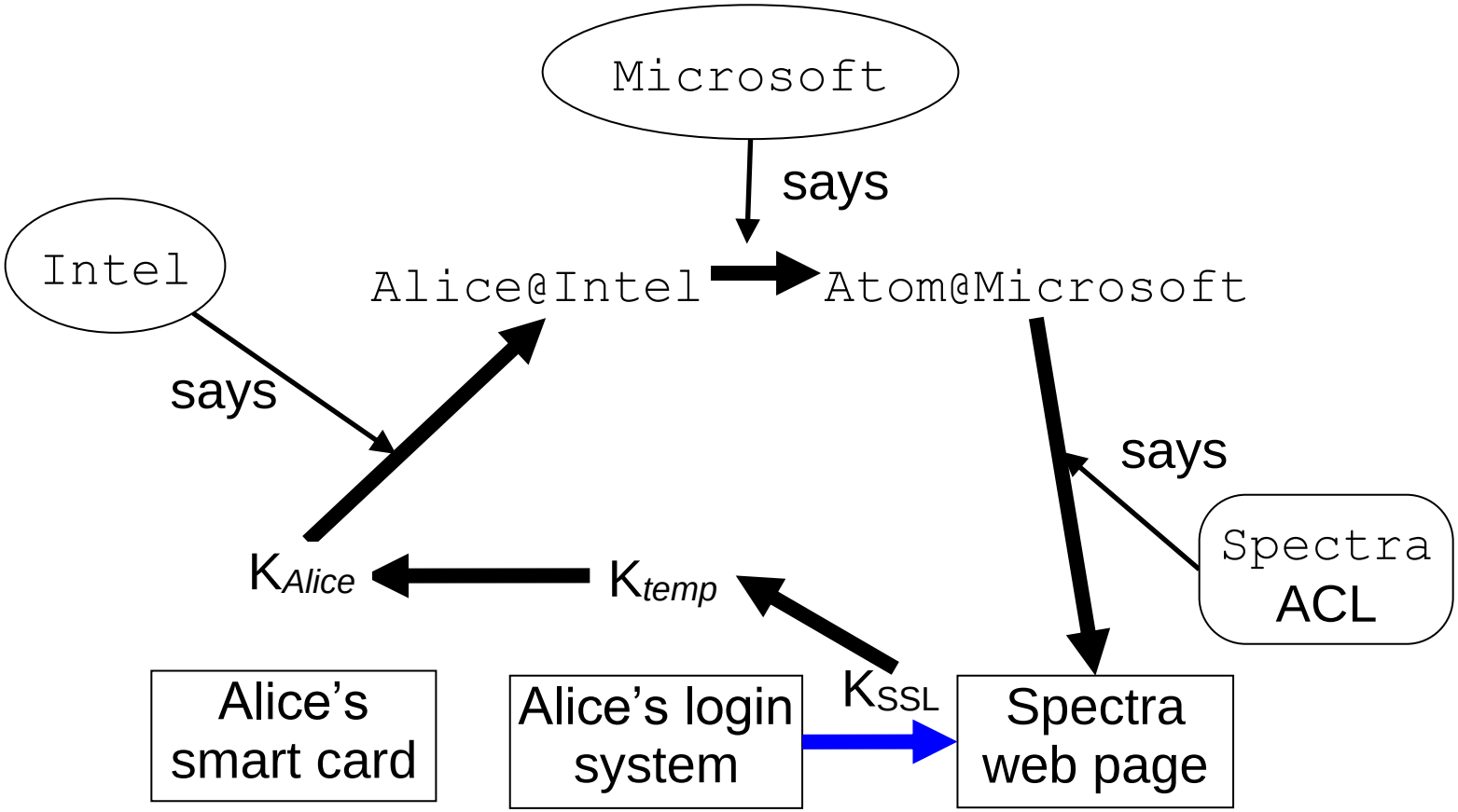
Assurance

- Trusted computing base

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



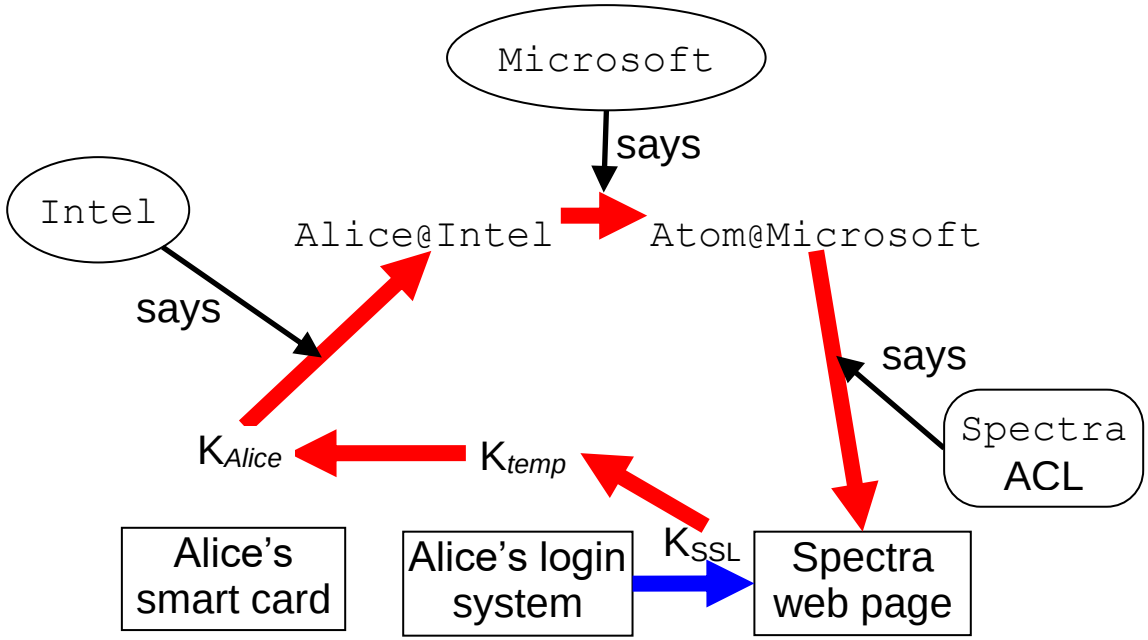
Chain of responsibility

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

Chain of responsibility:

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

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



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 /MSR/Lampson/foo`"

MSR-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* $\boxed{A \Rightarrow_T B}$

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*

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** $Q \Rightarrow P$ then $Q \Rightarrow P$

Reasonable if *P* is competent and accessible.

Restrictions are possible

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`

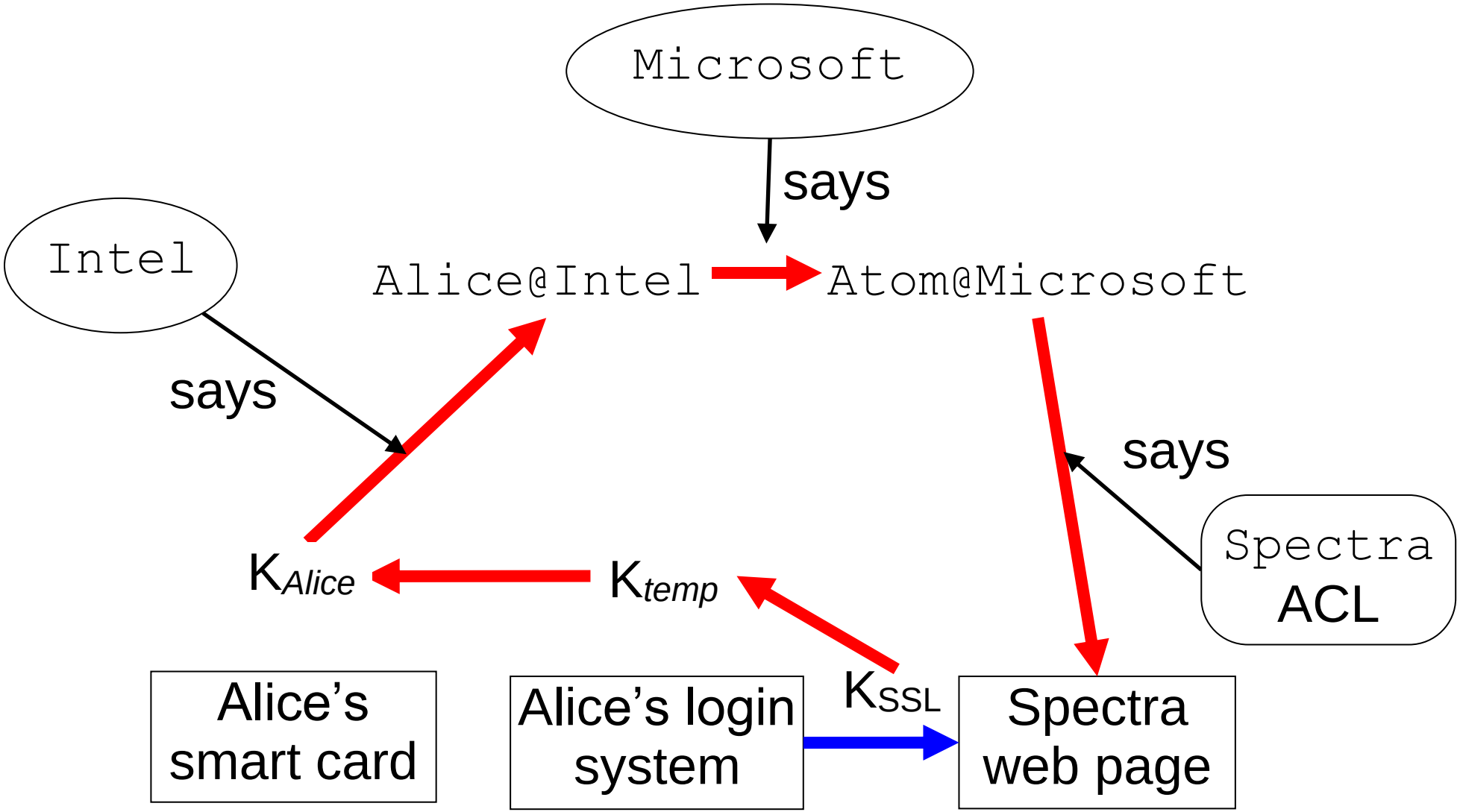
  Of course, only someone who believes $S \Rightarrow$ `SQLv71` will believe this

  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*

End-To-End Example



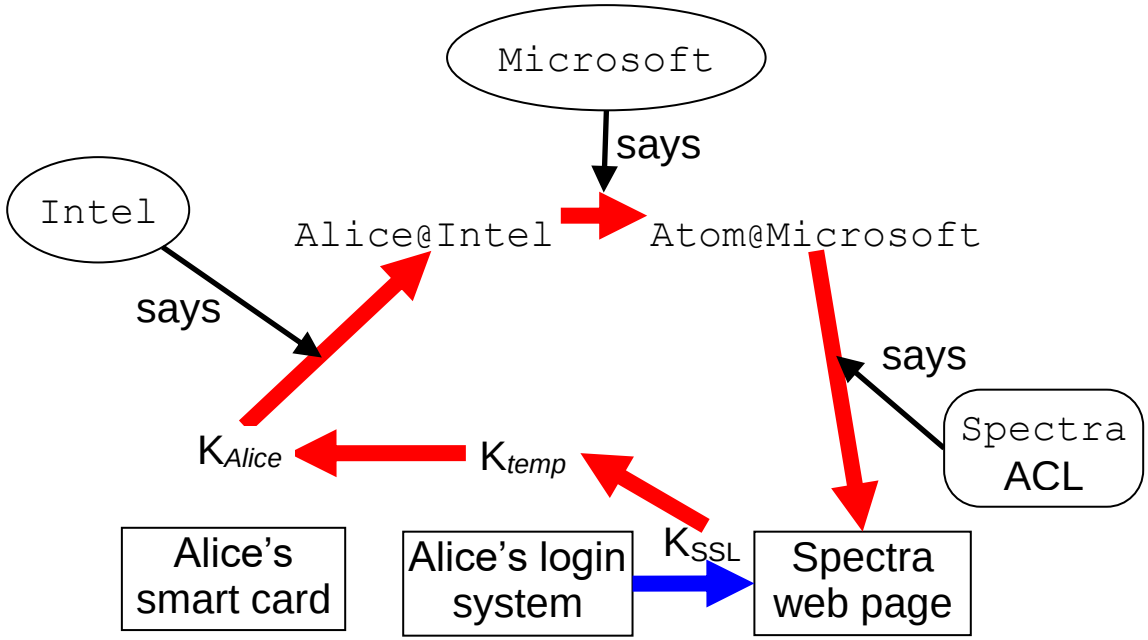
Chain of Responsibility

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

Chain of responsibility:

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

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



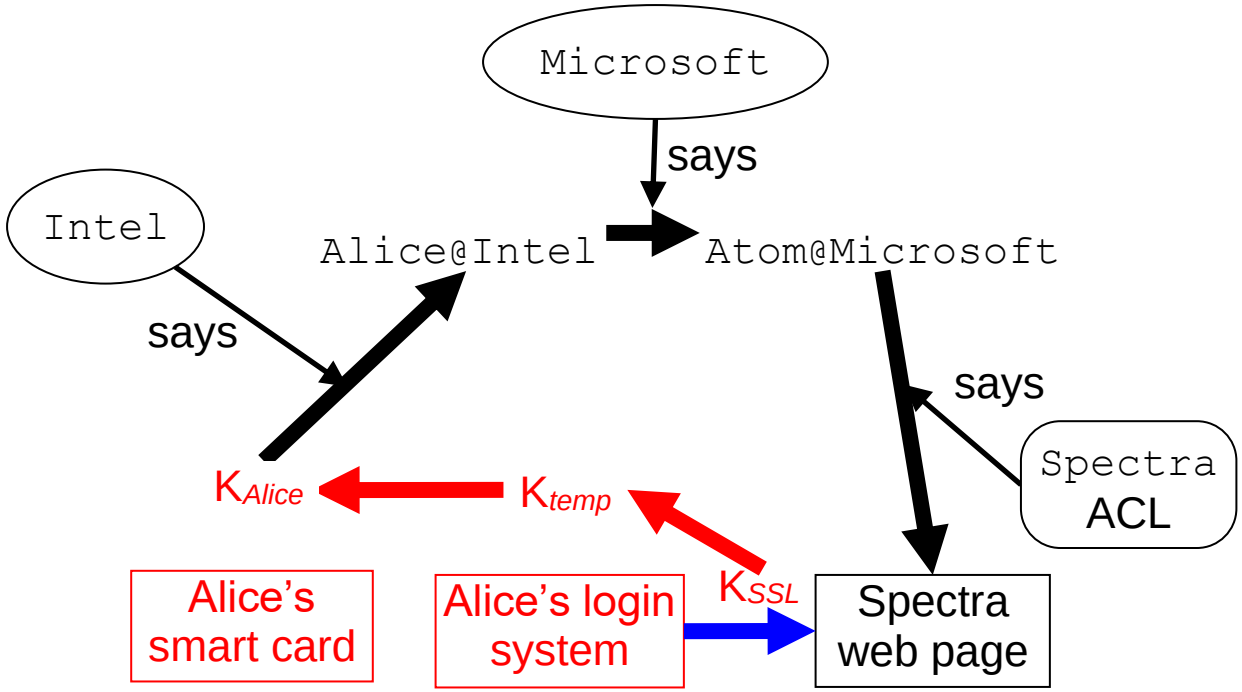
Authenticating Channels

Chain of responsibility:

K_{SSL} ⇒ K_{temp} ⇒ K_{Alice} ⇒ `Alice@Intel` ⇒ ...

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

(SSL setup) (via smart card)



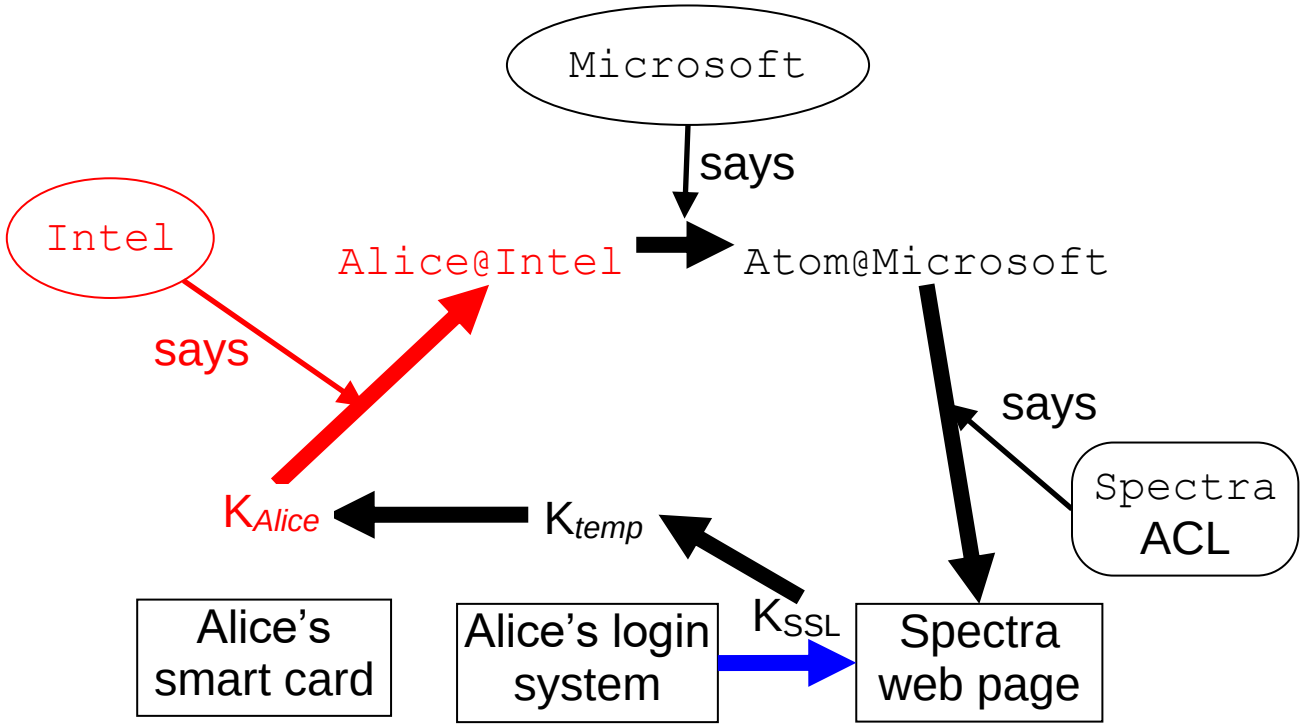
Authenticating Names: SDSI

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`)

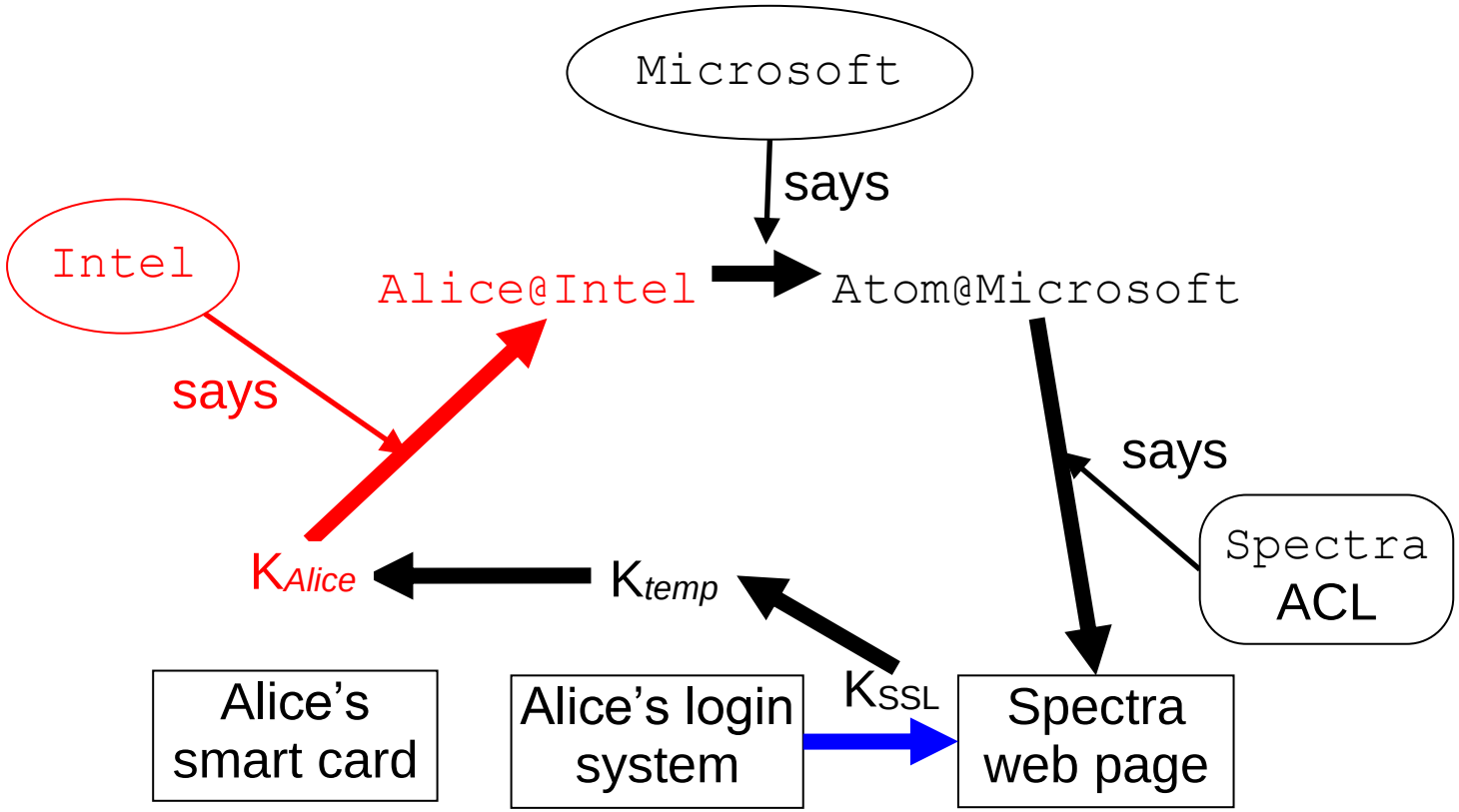


Authenticating Names

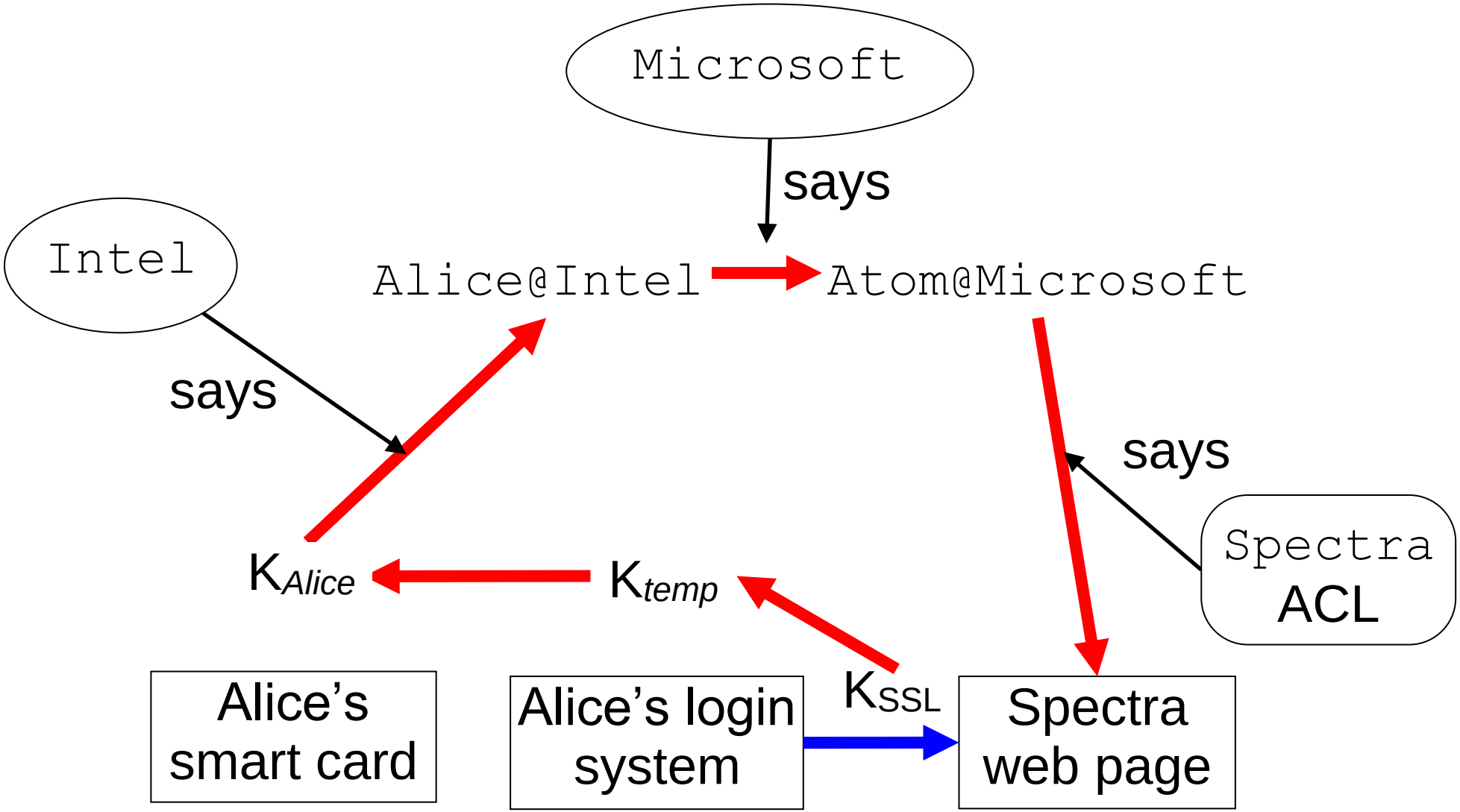
$K_{Intel} \Rightarrow$ `Intel` $\Rightarrow$ `Intel/Alice` (= `Alice@Intel`)

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

K_{Intel} **says**



End-To-End Example



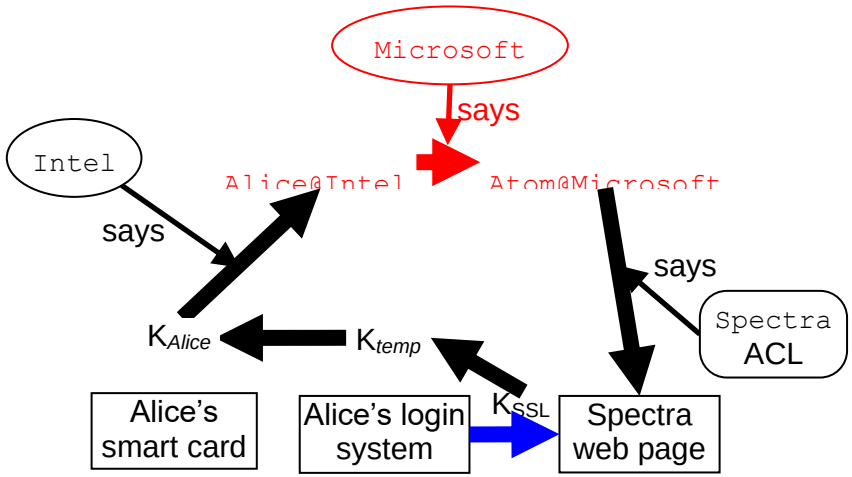
Authenticating Groups

A group is a principal; its members speak for it

- `Alice@Intel` ⇒ `Atom@Microsoft`
- `Bob@Microsoft` ⇒ `Atom@Microsoft`
- …

Evidence for groups: Just like names and keys.

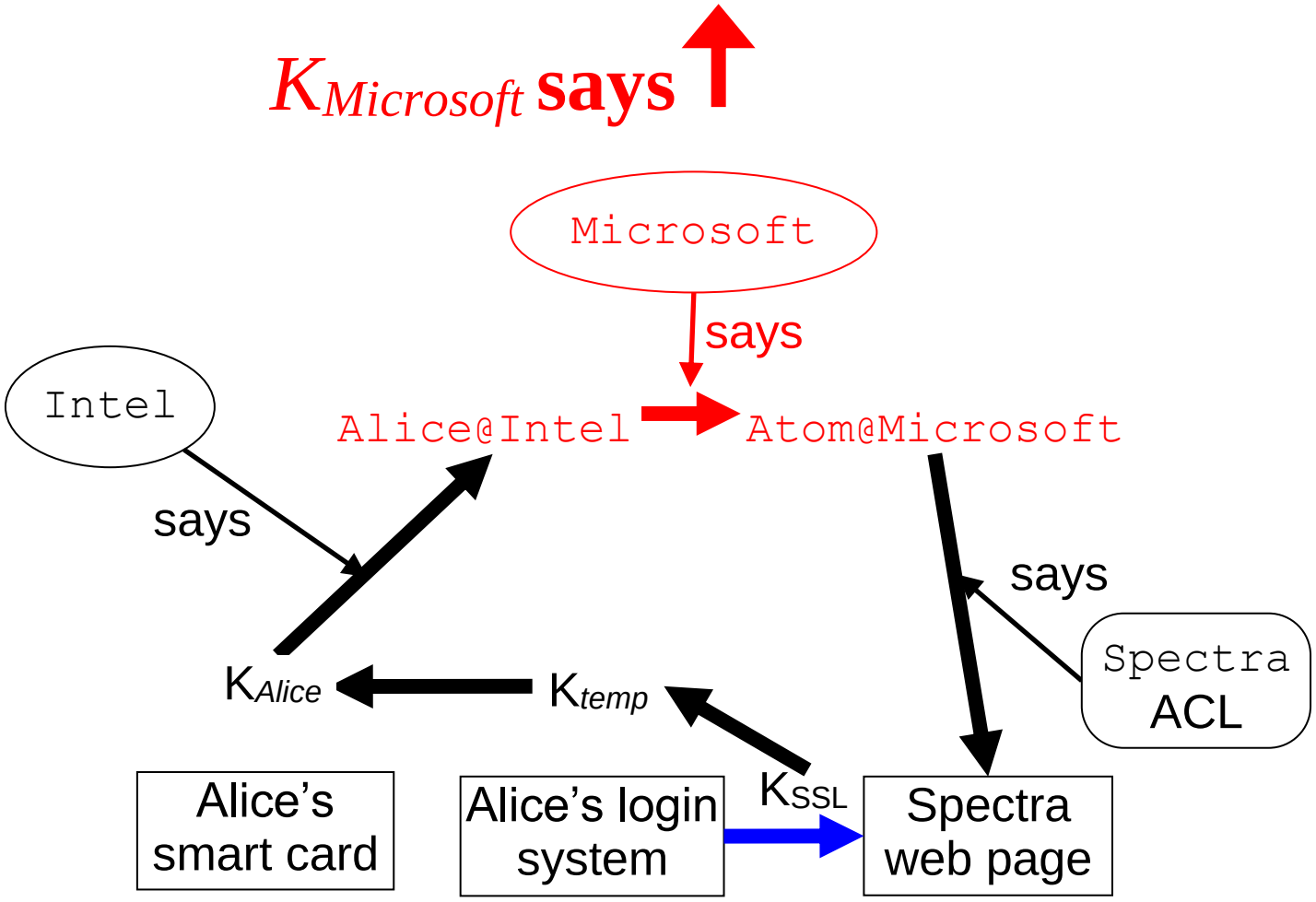
$K_{Microsoft}$ ⇒ `Microsoft` ⇒ `Microsoft/Atom`
(= `Atom@Microsoft`)



Authenticating Groups

$K_{Microsoft}$ ⇒ Microsoft ⇒ Atom@Microsoft

... ⇒ K_{Alice} ⇒ Alice@Intel ⇒ Atom@Microsoft ⇒ ...



Authorization with ACLs

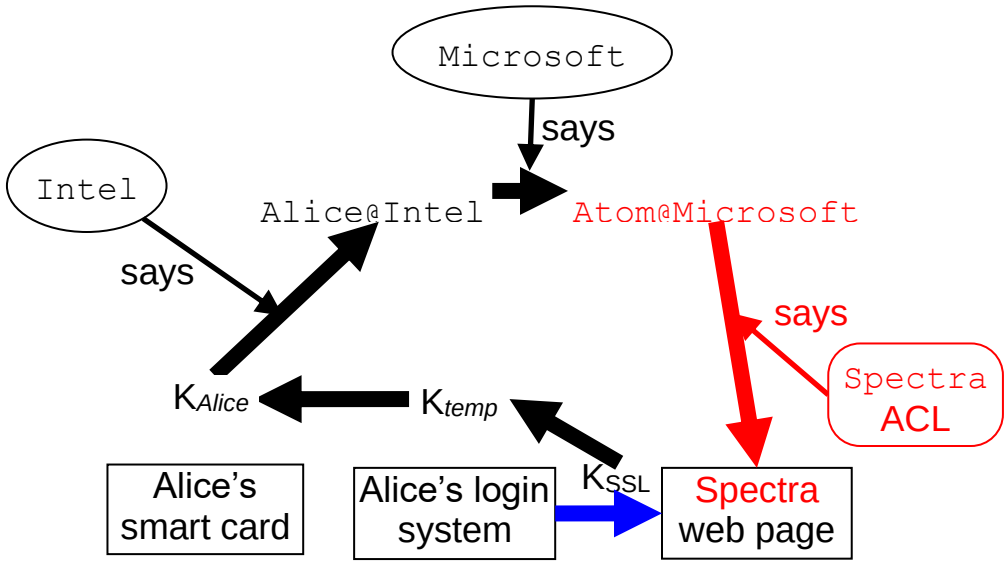
View a resource object *O* as a principal

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`

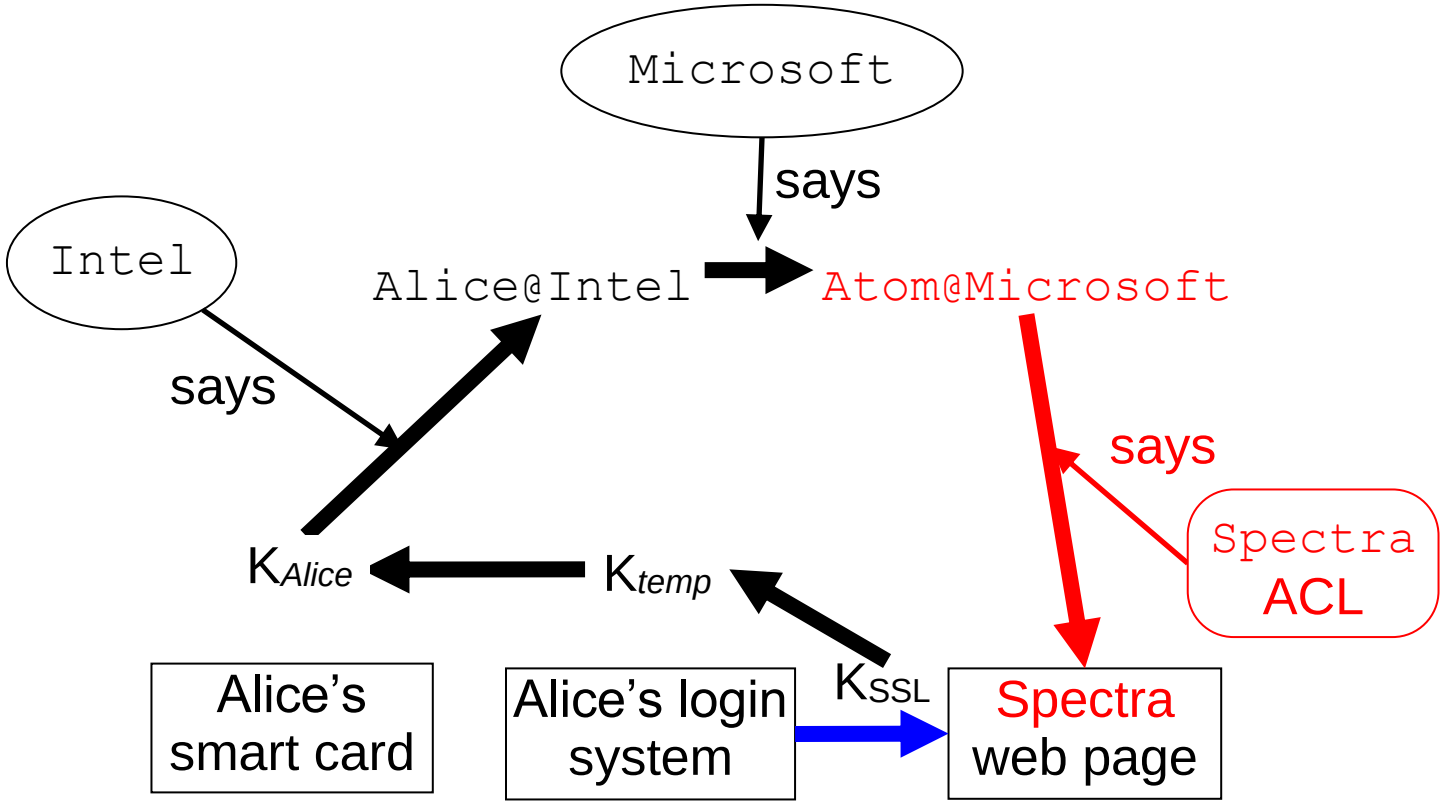


Authorization with ACLs

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

...⇒ Alice@Intel ⇒ Atom@Microsoft ⇒$_{r/w}$ Spectra

Spectra **says**



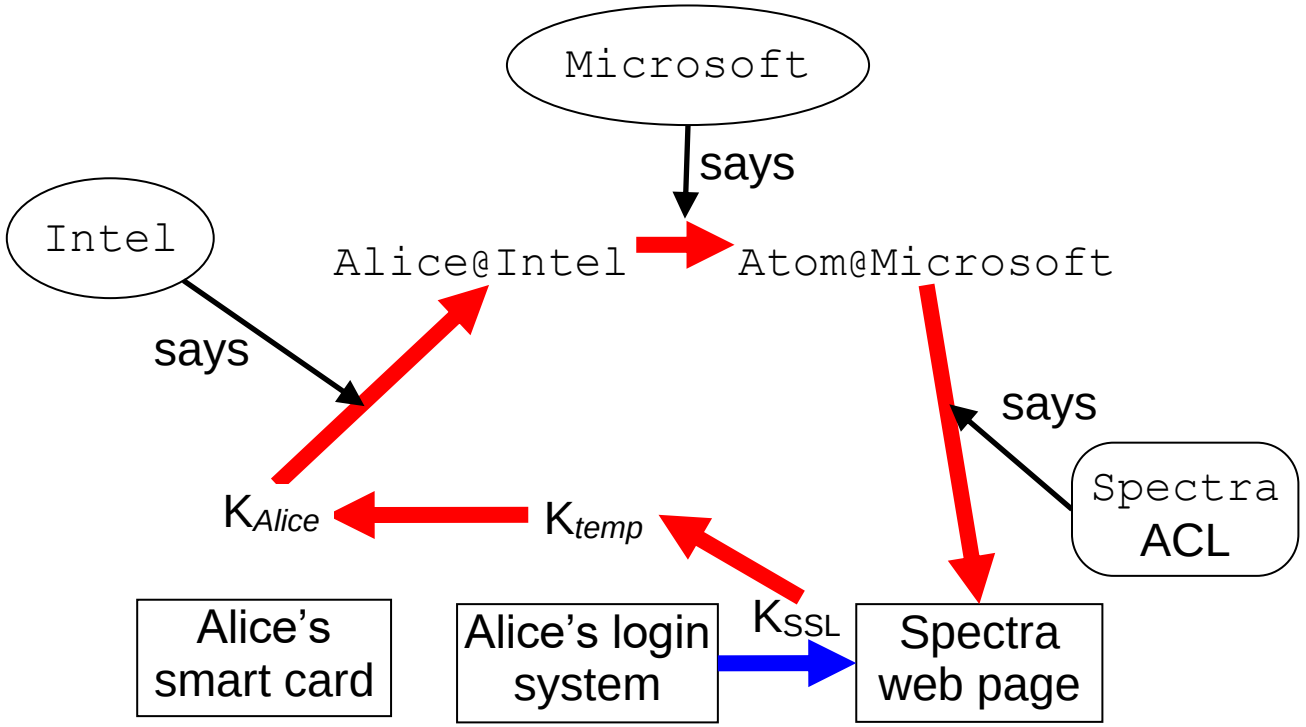
End-to-End Example: Summary

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

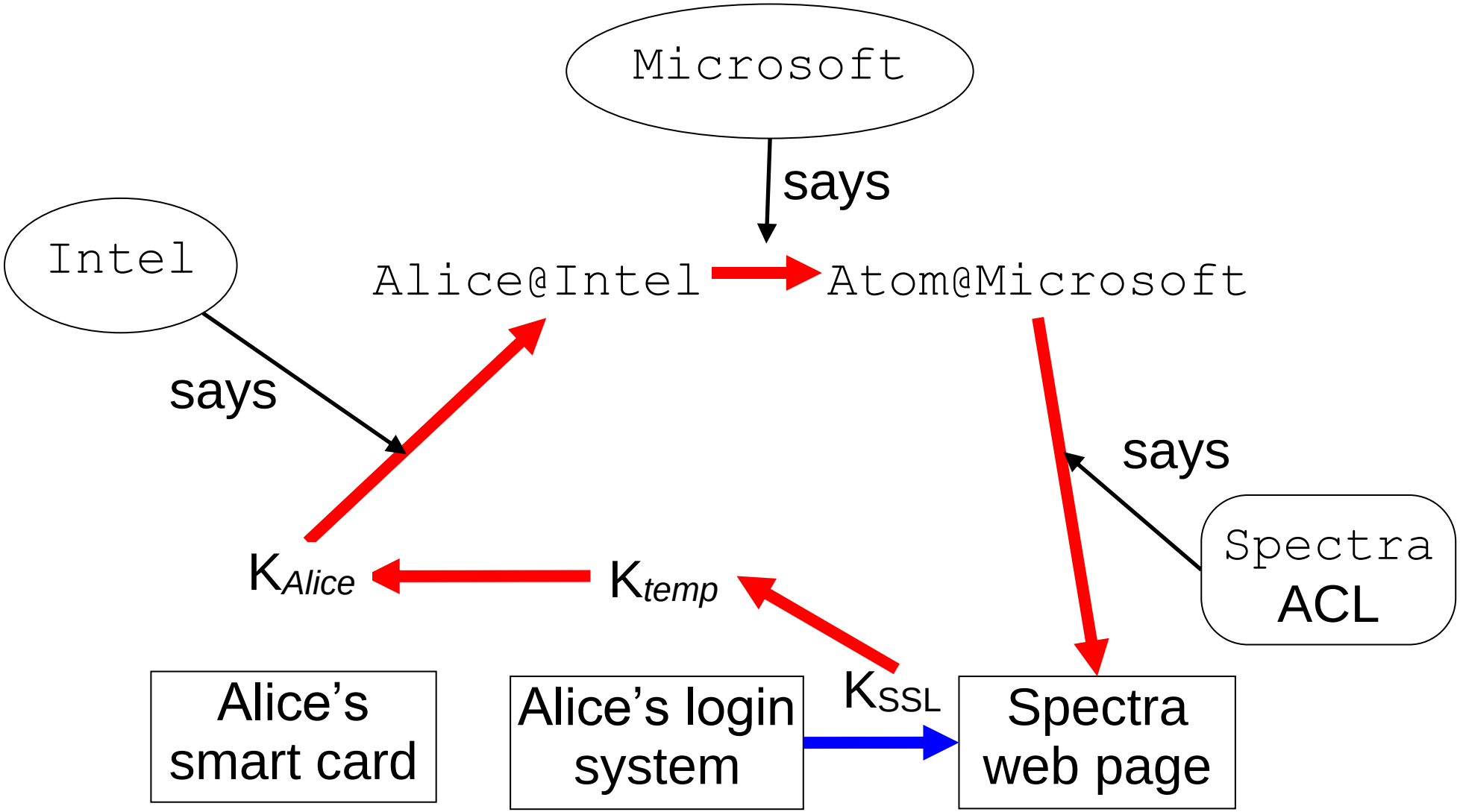
Chain of responsibility:

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

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



End-To-End Example



Compatibility with Local OS?

(1) Put network principals on OS ACLs

(2) Let network principal speak for local one

- `Alice@Intel` ⇒ `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` $\Rightarrow_{r/w}$ `Spectra`

Can replace a long chain with one summary certificate

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

Need a principal who speaks for the end of the chain

This is often called a capability

Lattice of Principals

$\Rightarrow$ is the lattice's partial order

A **and** B	max, least upper bound
A **or** B	min, greatest lower bound

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

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

(A **or** B) **says** $s \Leftarrow$ (A **says** s) **or** (B **says** s)

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 ⇒ represent clearance

- `lampson` ⇒ `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 $\Rightarrow K_{Alice}$"

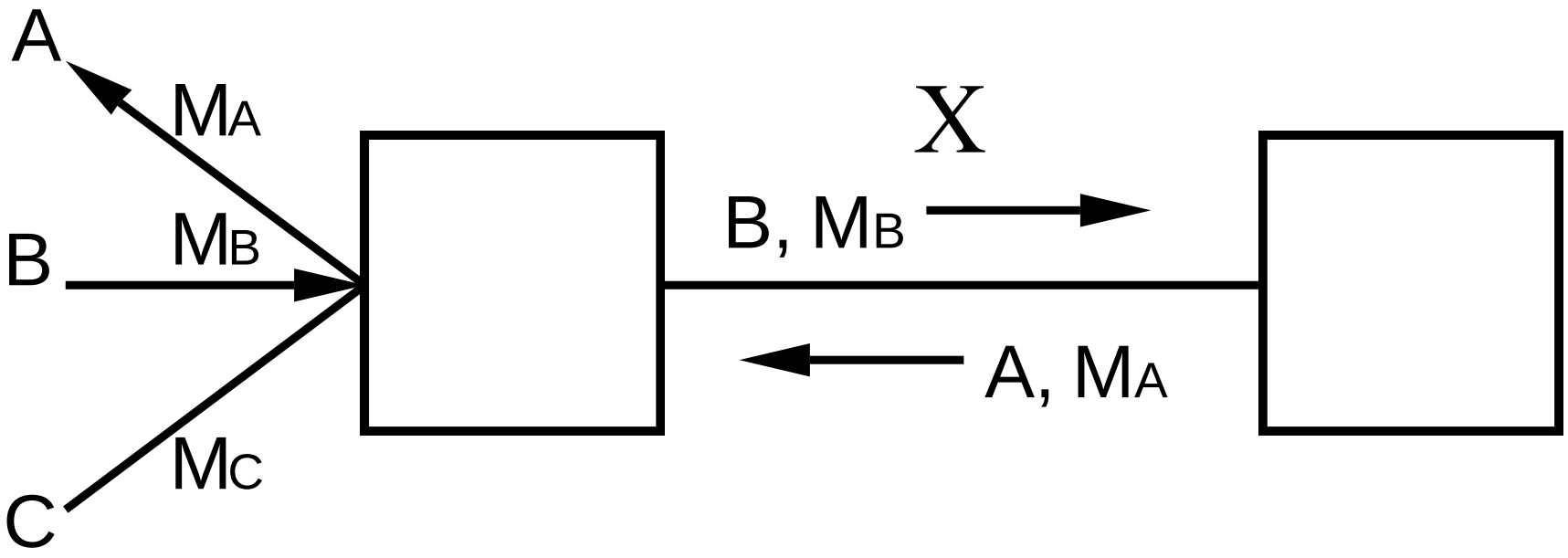
unless you trust the receiver not to send this on another channel!

- 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**

| is idempotent: $A \mid A = A$

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

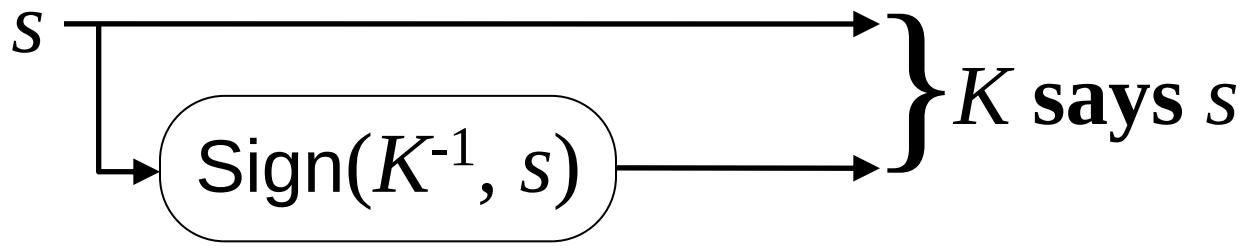
Multiplexing a Channel: Examples

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!

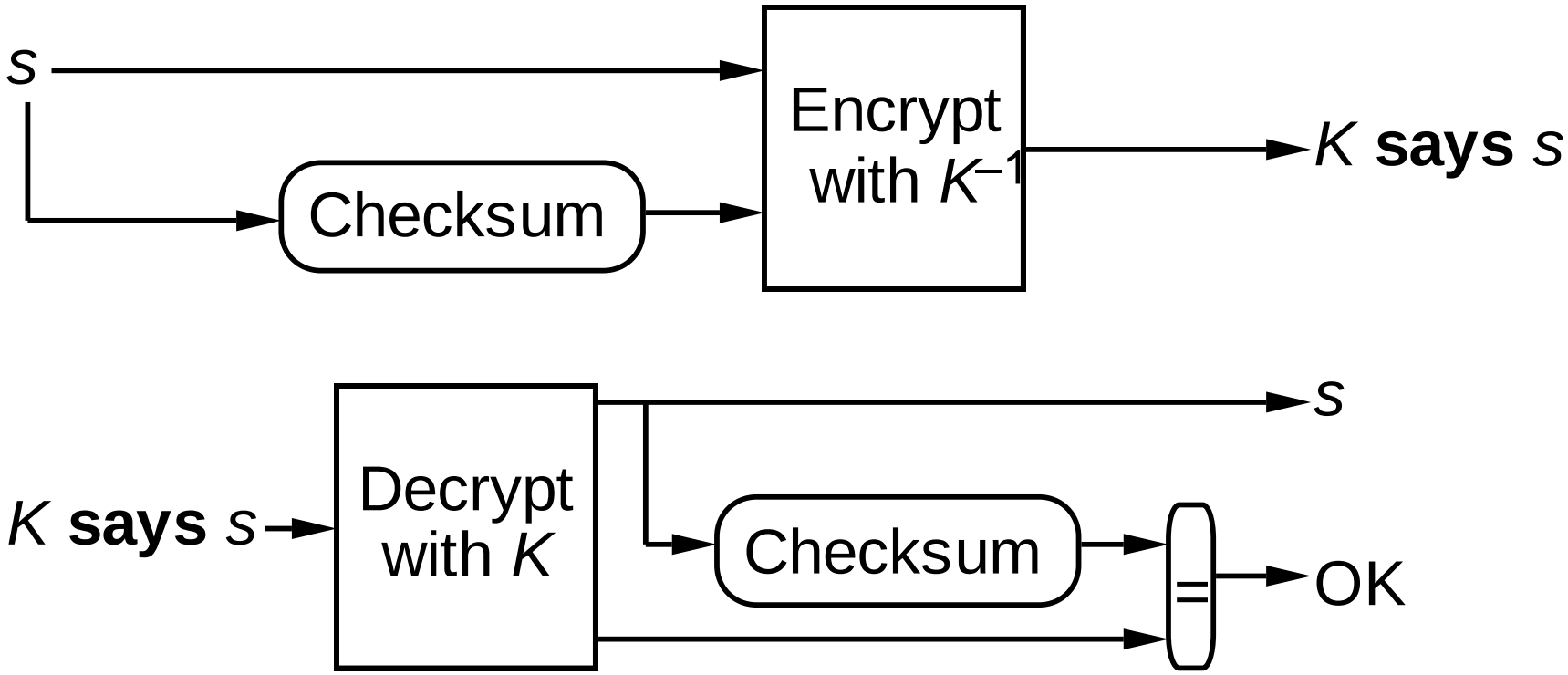
Sign and Seal

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

  Temp key ⇒ 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—100-3000x public key

Can simulate public key with trusted on-line server

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, $\{K\}_{K_C}$ means C **says** "K is a key". C says nothing about A and B. This is useless
- $\{A, B, K\}_{K_C}$ means C **says** "K is a key for A to talk to B". C says nothing about when K is valid
- $\{A, B, K, T\}_{K_C}$ 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

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

Roles: *P* as *R*

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

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_{TUM} **says** (*H* **as** `SQL`) $\Rightarrow$ `TUM / SQL`

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

- *H* **says** *C* $\Rightarrow$ *H* **as** `SQL`
- Hence *C* $\Rightarrow$ `TUM / 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`.

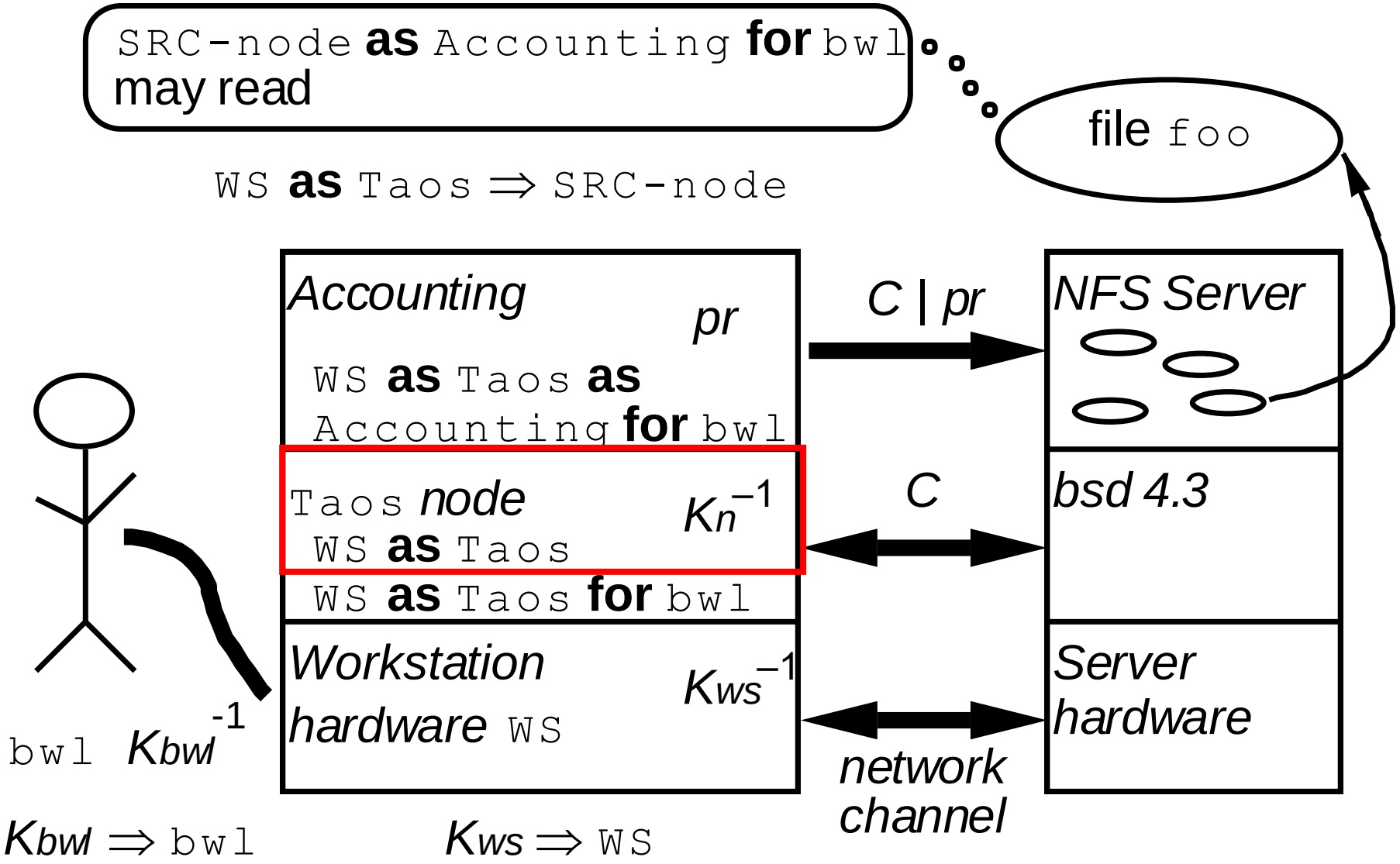
Generates K_n^{-1}, the node key.

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}.

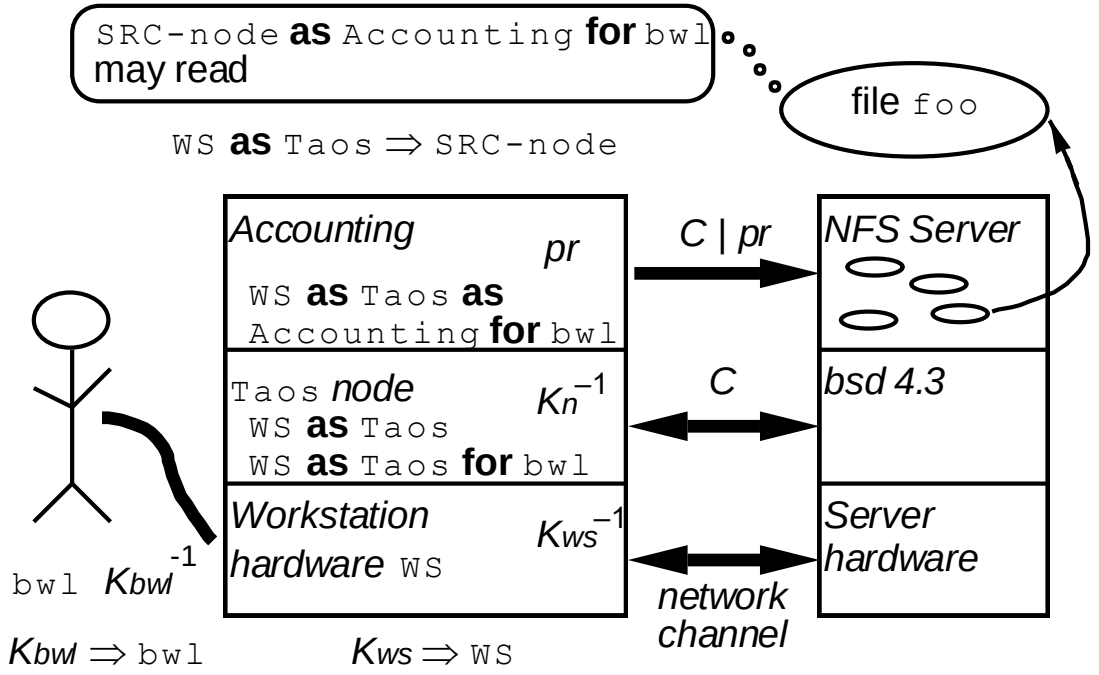
Other systems are similar: K_{ws} **as** `Taos` **as** `Accounting`

Node Credentials: Example



Example: Server's Access Control

Kws **says** *Kn* ⇒ *Kws* **as** `Taos`	node	*credentials*
Kbwl **says** *Kn* ⇒	login	
(*Kws* **as** `Taos`) ∧ *Kbwl*	session	
Kn says *C* ⇒ *Kn*	channel	
C says *C* | *pr* ⇒	process	
(*Kws* **as** `Taos` **as** `Accounting`) ∧ *Kbwl*		
C | *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: K_{MS} **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

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.

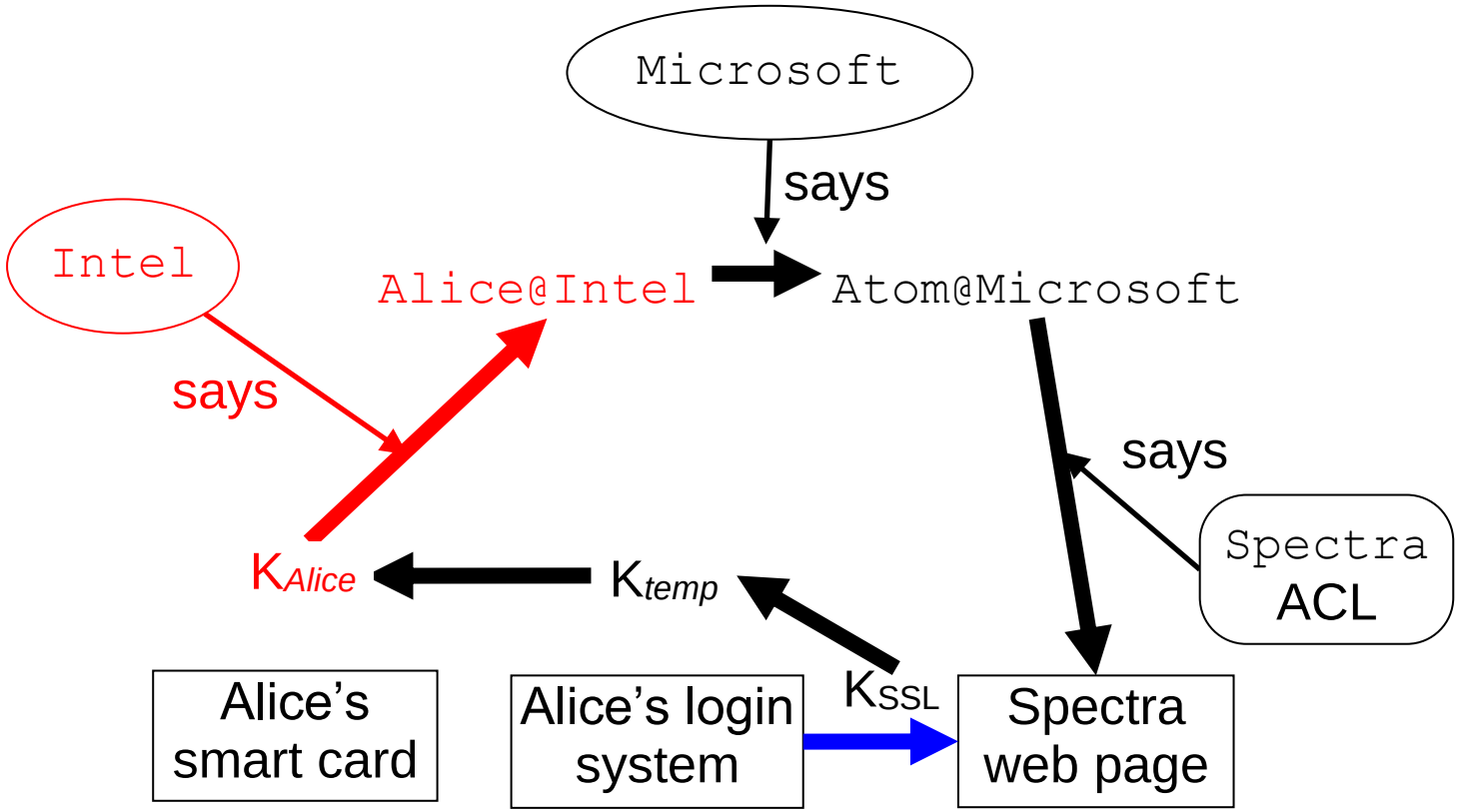
`/DEC/SRC/Lampson` may read file `/DEC/SRC/udir/Lampson/report`

Authenticating Names

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

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

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:

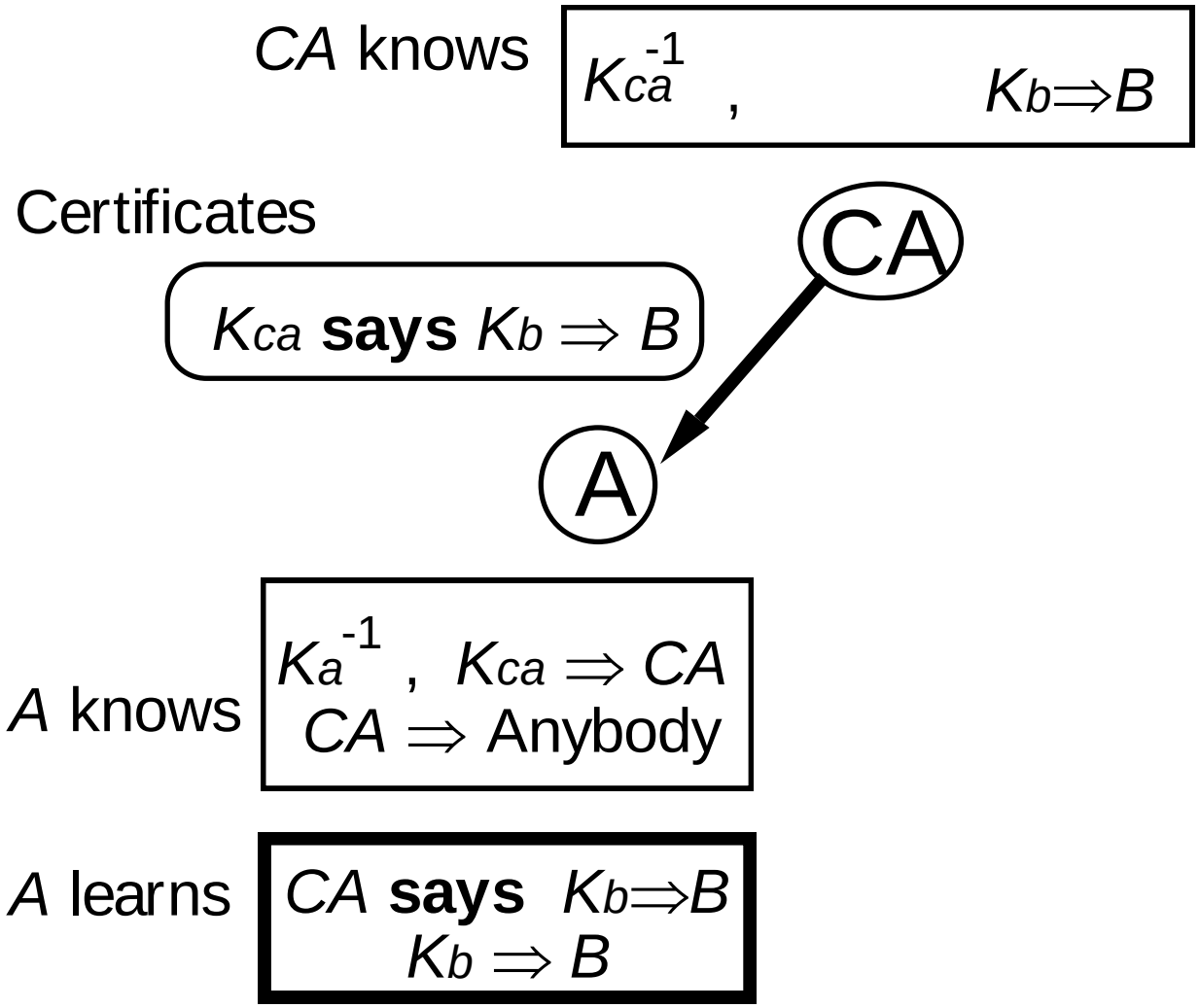
$\boxed{CA \Rightarrow \text{Anybody}}$

Then *CA* can authenticate channels:

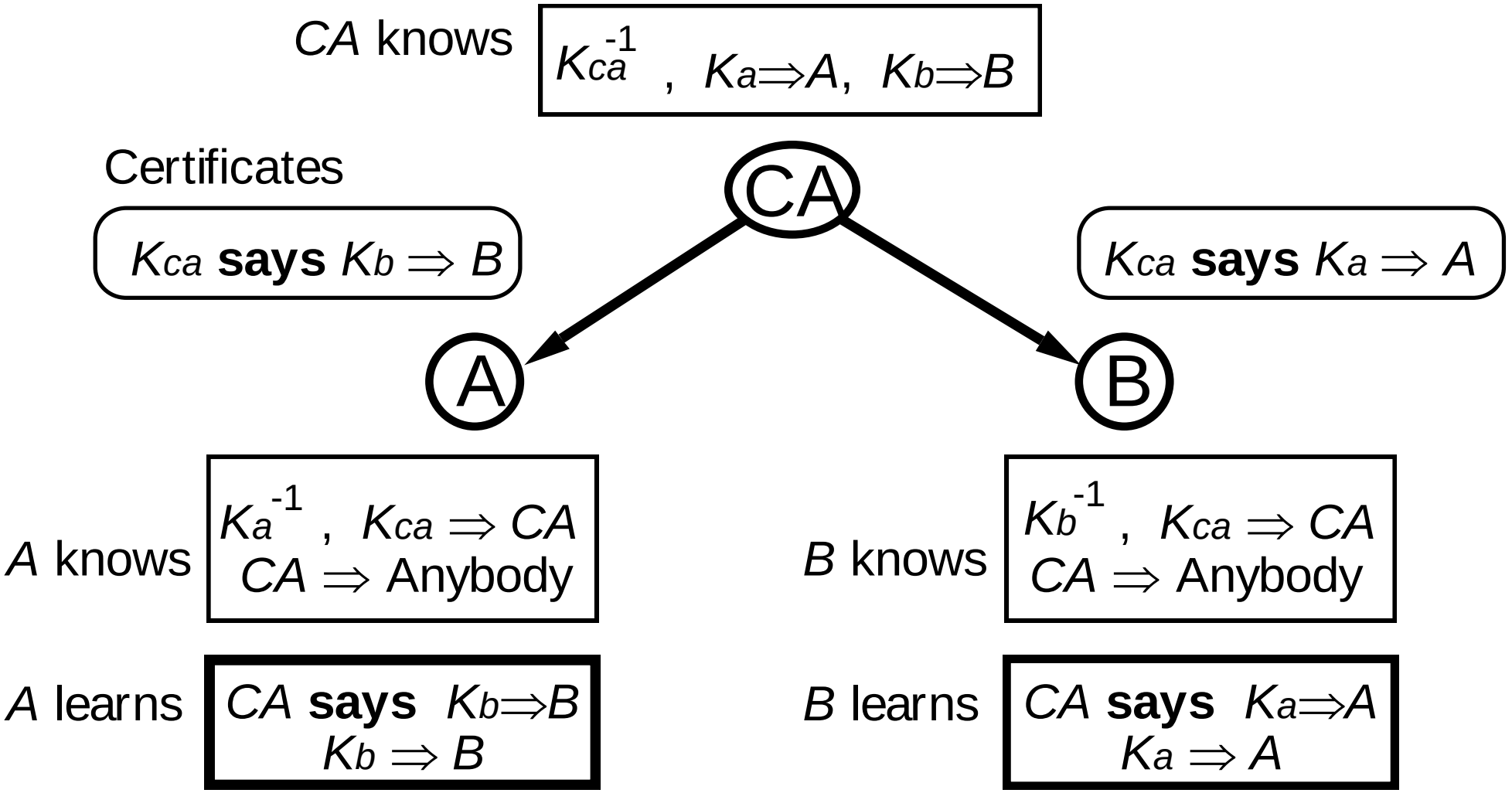
K_{ca} **says** $K_{ws} \Rightarrow$ `WS`

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

One-Way Authentication



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.,

$$CA_{assert} \textbf{ and } CA_{revoke} \Rightarrow \text{Anybody}$$

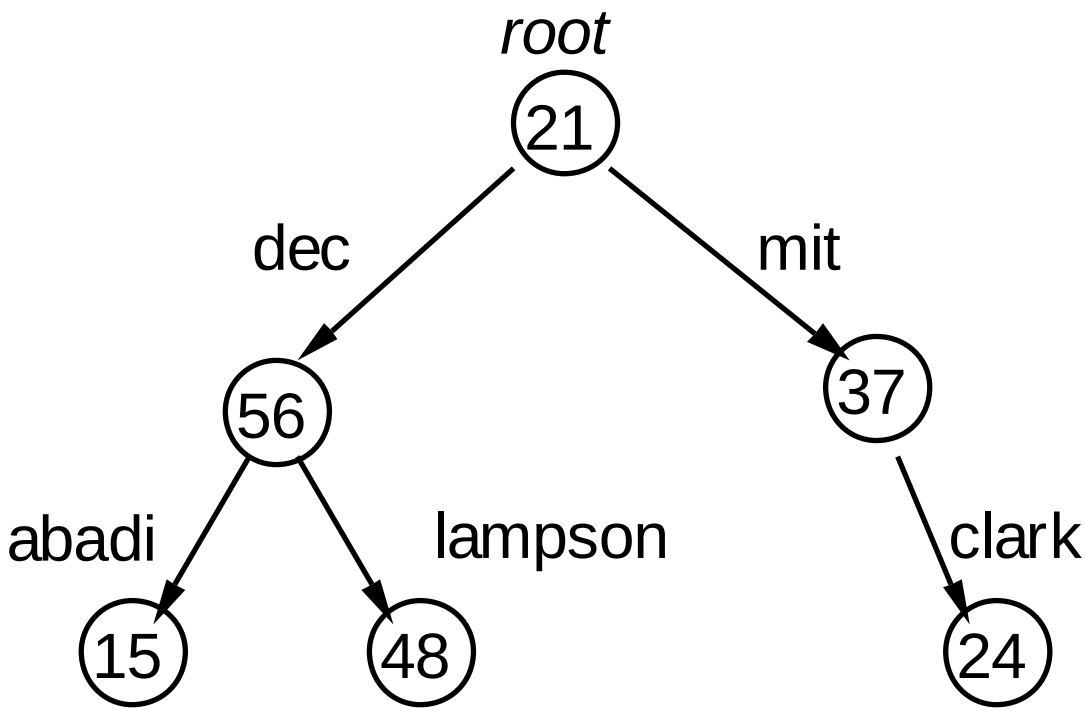
Large-Scale Authentication

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



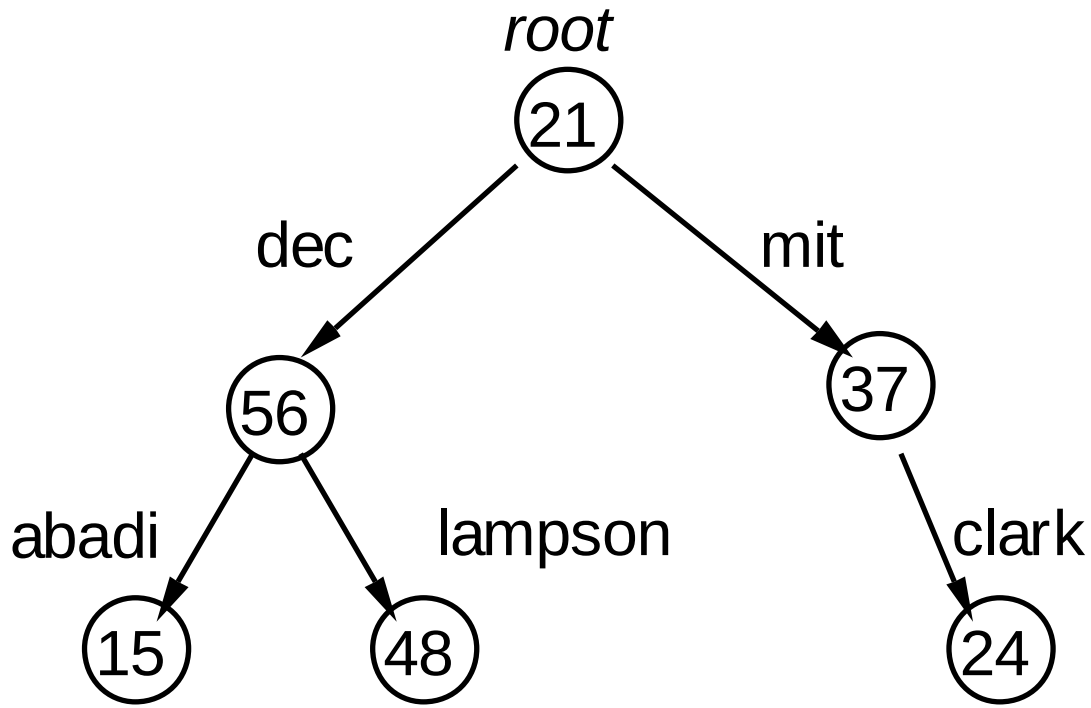
Large-Scale Authentication: Example

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`

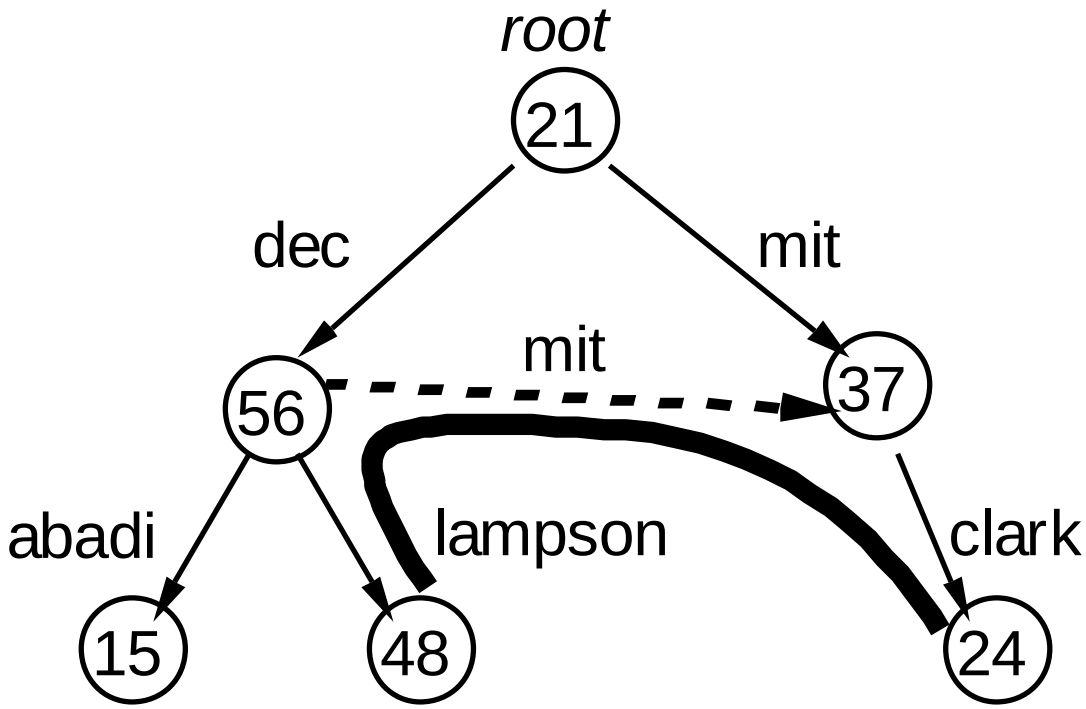


Trusting Fewer Authorities: Cross-Links

For less trust, add links to the tree

Now `lampson` trusts only `dec` for

`/dec/lampson` → `/dec/mit/clark`



GROUPS and Group Credentials

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

`Alice@Intel` ⇒ `Atom@Microsoft`

`Bob@Microsoft` ⇒ `Atom@Microsoft`

. . .

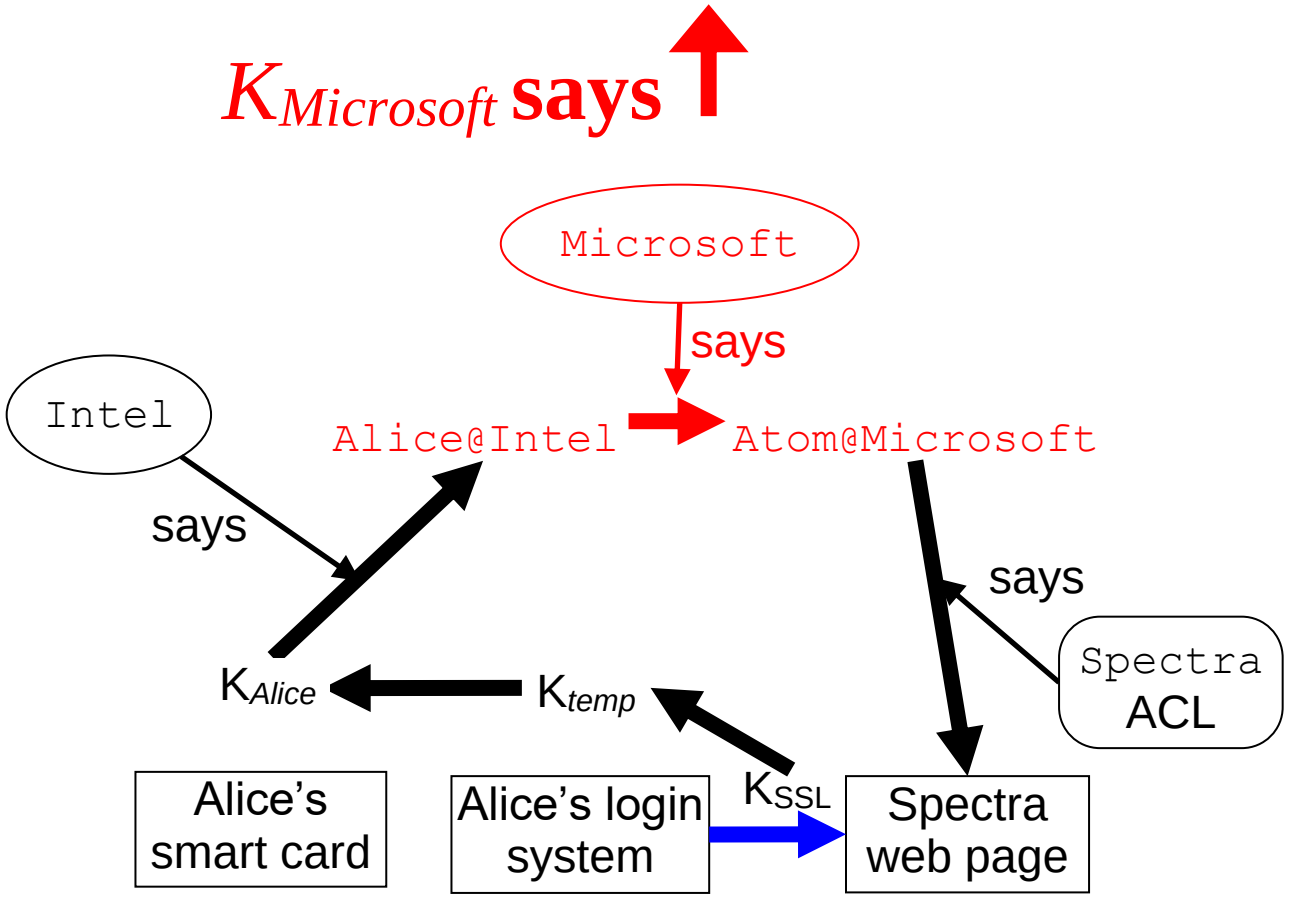
Proving group membership: Use certificates

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

Authenticating Groups

$K_{Microsoft}$ ⇒ Microsoft ⇒ Atom@Microsoft

... ⇒ K_{Alice} ⇒ Alice@Intel ⇒ Atom@Microsoft ⇒ ...



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 / Attributes

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.

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

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

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

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

`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 min, greatest lower bound

$A \Rightarrow B \equiv (\ A = A$ **and** $B\ ) \equiv (\ B = A$ **or** $B\ )$

(A **or** B) **says** $s \Leftarrow (A$ **says** $s) \vee (B$ **says** $s)$

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

AUDITING

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 are OK	`read/write` $\geq$ `read`

Auditing

Each step is justified by

- a signed statement, or
- a rule

Summary: The "Speaks for" Relation $\Rightarrow$

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

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*

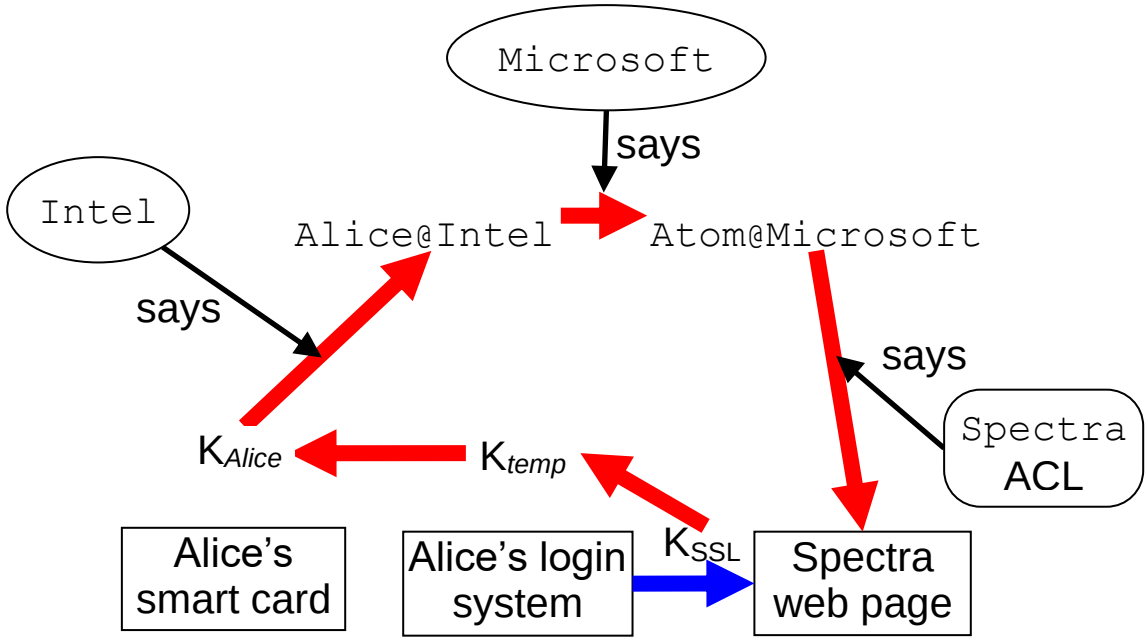
Summary: Chain of Responsibility

`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*


K_{Alice}