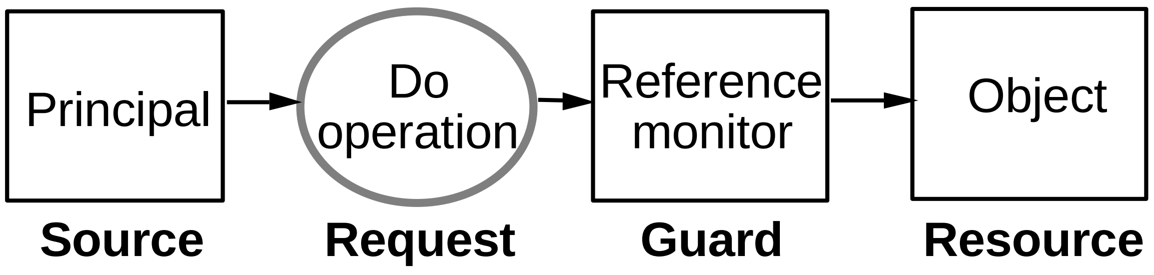
Security policies and access control

(continued)

Chaire Informatique et sciences numériques
Collège de France, cours du 23 mars 2011

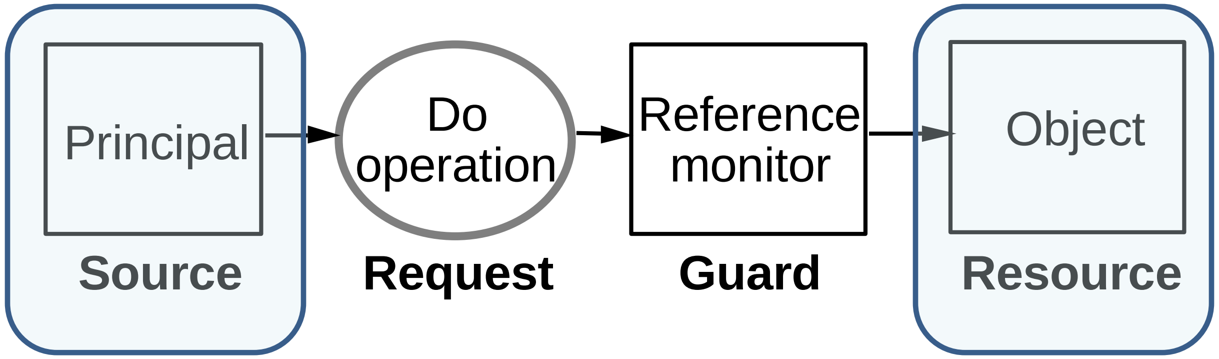
Access control and programs

Programs everywhere!



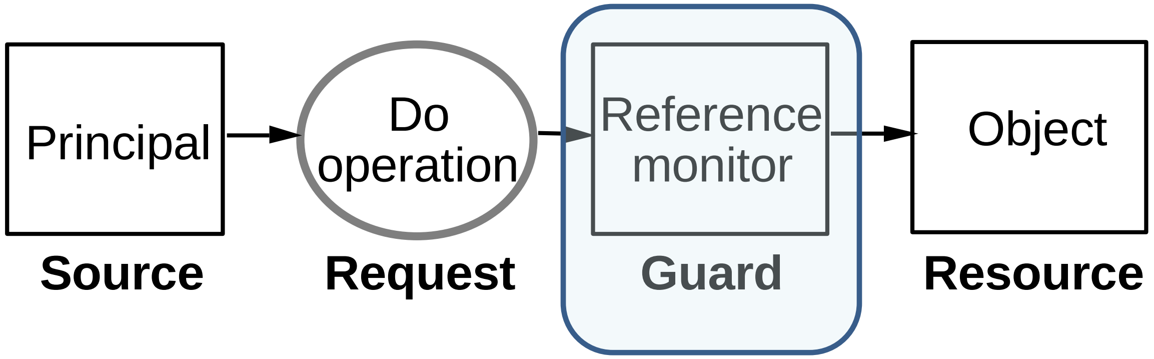
Programs everywhere!

- Programs are principals and objects.



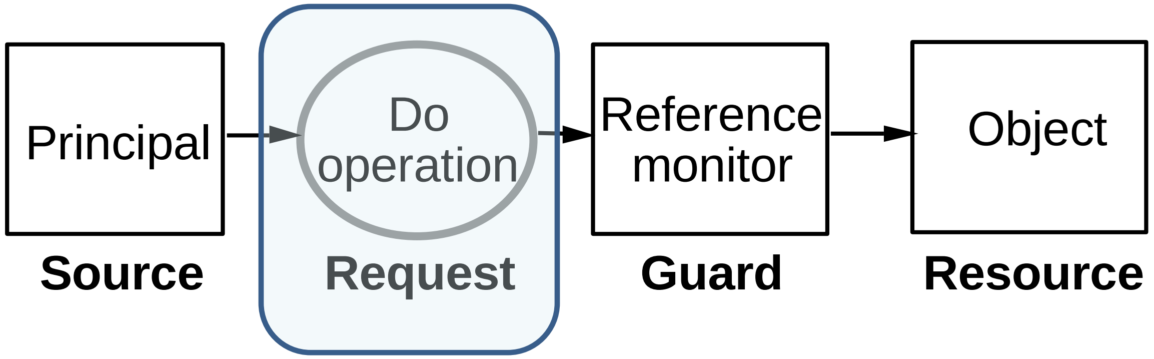
Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
  - Often, even some of the access control policy is baked into programs, for better or for worse.



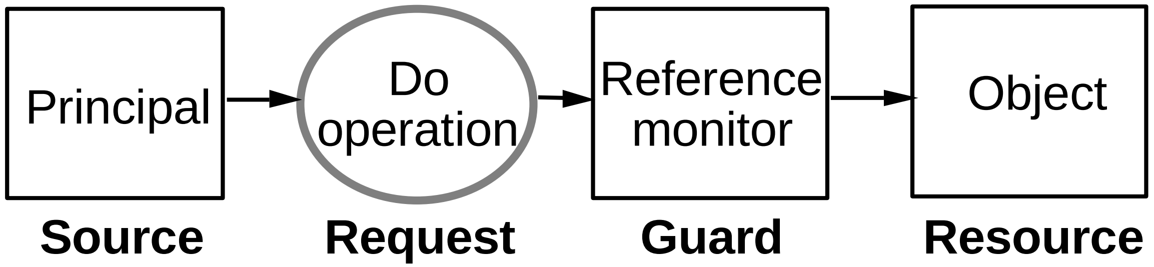
Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
  - Often, even some of the access control policy is baked into programs, for better or for worse.
- Programs implement the operations that are the concern of access control.



Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
  - Often, even some of the access control policy is baked into programs, for better or for worse.
- Programs implement the operations that are the concern of access control.



Bundling operations into programs

objects / principals	Data file	Log file	Program P's code
Alice			x
auditor		r	r
Program P	rw	rw	x

Conjoining users through programs

objects / principals	Data file	Program P's code
Alice		x
Bob		x
Program P	rw	x

where P checks that both Alice and Bob make the same request before forwarding the request.

Conjoining users through programs

(an alternative)

objects / principals	Data file	Data file's ACL	Program P's code
Alice			x
Bob			x
Program P		rw	x

where P modifies the matrix so that Alice has access when Bob requests it, and vice versa.

Modifying the matrix

- The access control matrix need not be static.
- It may be modified by programs like:

```
command CONFER (right, user, friend, file)
  if right in matrix[user, file]
  then enter right into matrix[friend,file]
end
```

How can we ensure safety?

Algorithmic analysis

[starting with Harrison, Ruzzo, and Ullman, 1976]

- A *system* has finite sets of rights and commands.
- A *command* is of the form "if conditions hold, then perform operations".
  - The conditions are predicates on the matrix.
  - Operations add/delete rights, principals, objects.

Let A be a principal and f an object.

In general, it is undecidable whether there is a reachable state such that A can access f.

Algorithmic analysis (cont.)

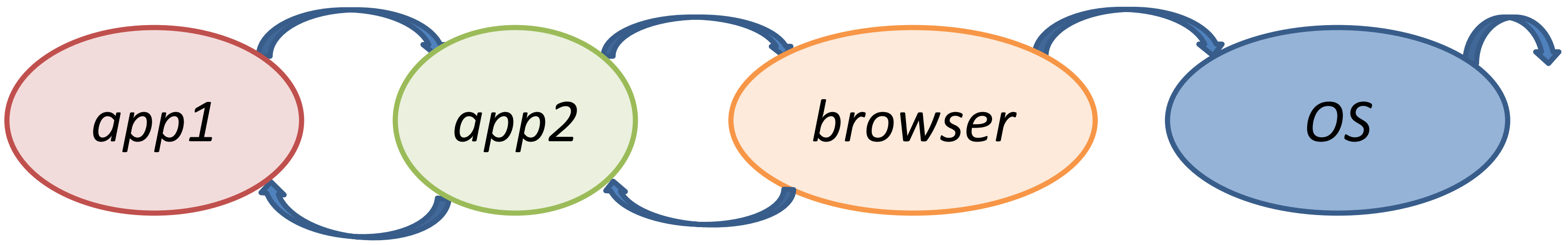
[in particular, Li, Winsborough, and Mitchell, 2003]

- Not all interesting problems are undecidable!
- Consider the containment problem:
  In every reachable state, does every principal that has one property (e.g., has access to a resource) also have another property (e.g., being an employee)?

For different classes of systems, this problem is decidable (in coNP or coNEXP).

Programs and other principals

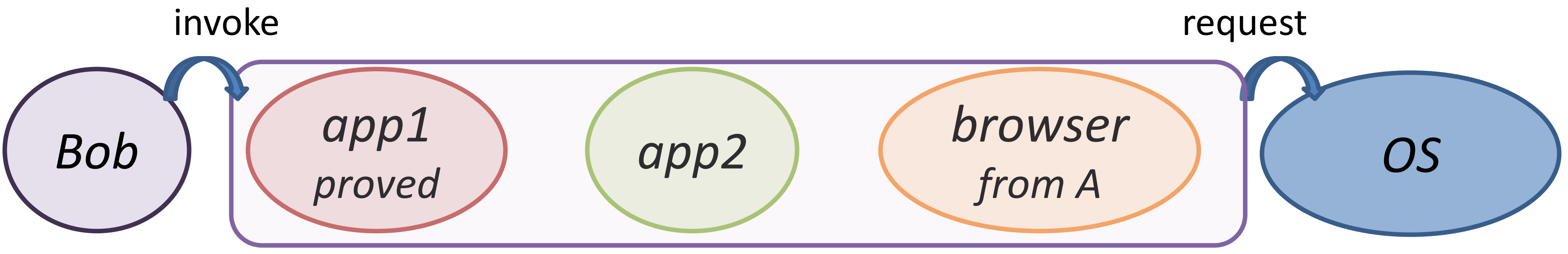
- So, programs may be principals too.
- But then:
  - we need to deal with program combinations,



  - we need to connect programs to other principals
    - who write them or edit them,
    - who provide them or install them,
    - who call them.

Running programs

- What are the run-time rights of a program P?
  – those of P's caller, or
  – those of some responsible user, or
  – something else, e.g, because of P's properties, or
  – some combination.
- The same factors appear in deciding whether to run a program.



Running programs (cont.)

Some approaches to combining authorities:

- setuid,
- code access security (with stack inspection or alternatives).

Some approaches to intrinsic properties:

- proofs (and proof-carrying code),
- types,
- dynamic checks (e.g., in sandboxes),
- their combinations (e.g., proofs about sandboxes).

Protection and isolation

- Programs must be protected (always) and limited to communicate on proper interfaces.
- This is often the job of the computing platform (OS + hardware).
  - It can implement address spaces so that programs in separate spaces cannot interact directly (e.g., cannot smash or snoop on one another).
- A language and its run-time system can provide fine-grained control.

More on this in a later lecture.

Examples

Access control in Unix (basics)

- Principals are users (plus root).
- Objects are files.
- Operations are read, write, and execute.
- Each file has an owner and a group.
- Each file has an ACL, which can be set by its owner and root.
- ACLs specify rights for the owner (“user”), group, and others (e.g., **rwxrw-r--**).

Access control in Unix (cont.)

- If a program file is marked as suid, then the program executes with the privilege of its owner (not that of the caller).
  - The usage of setuid is error-prone.
  - The details are complex and vary across systems.
- And there are other complications: sgid, capabilities in Linux, directories, …

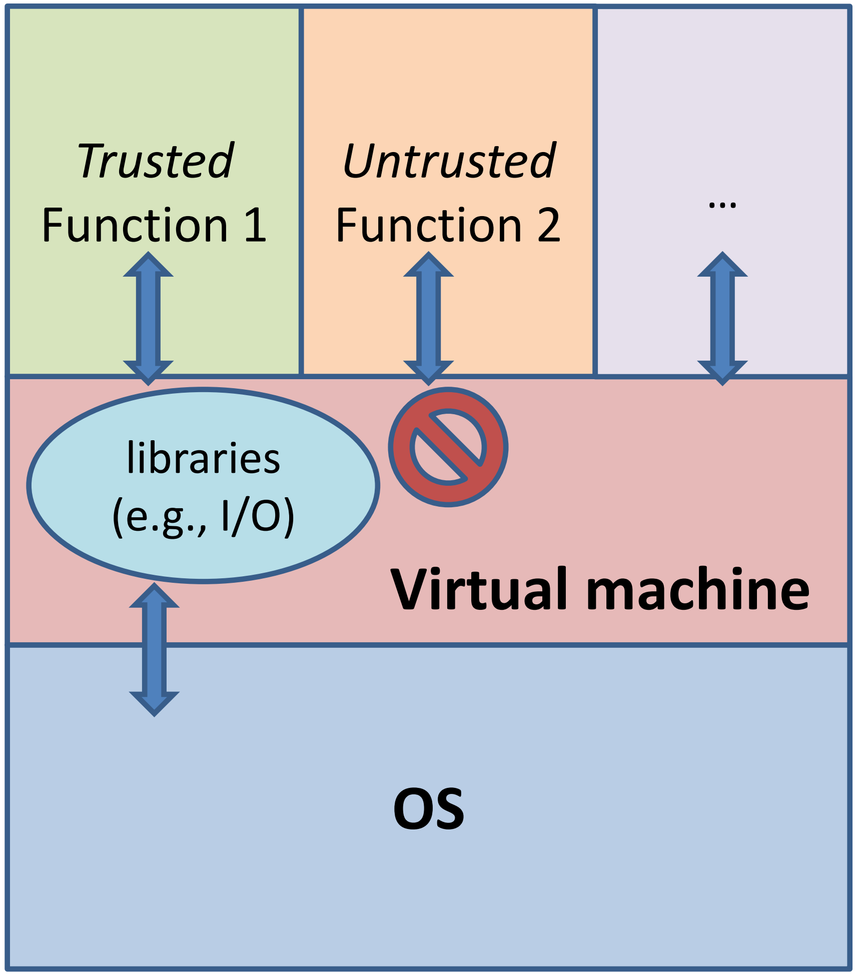
The basic sandbox policy

- *Trusted code* (e.g., local code) has the full power of the user that runs it.
- *Untrusted code* (e.g., foreign code) has very limited rights, e.g.:
  - no direct use of files,
  - network connections only to the code's origin.
- The sandbox is enforced at run-time:
  - A reference monitor ("security manager") is associated with code when the code is loaded.

The basic sandbox policy

Trusted code can access libraries and thereby the underlying OS services.

Untrusted code mostly cannot.

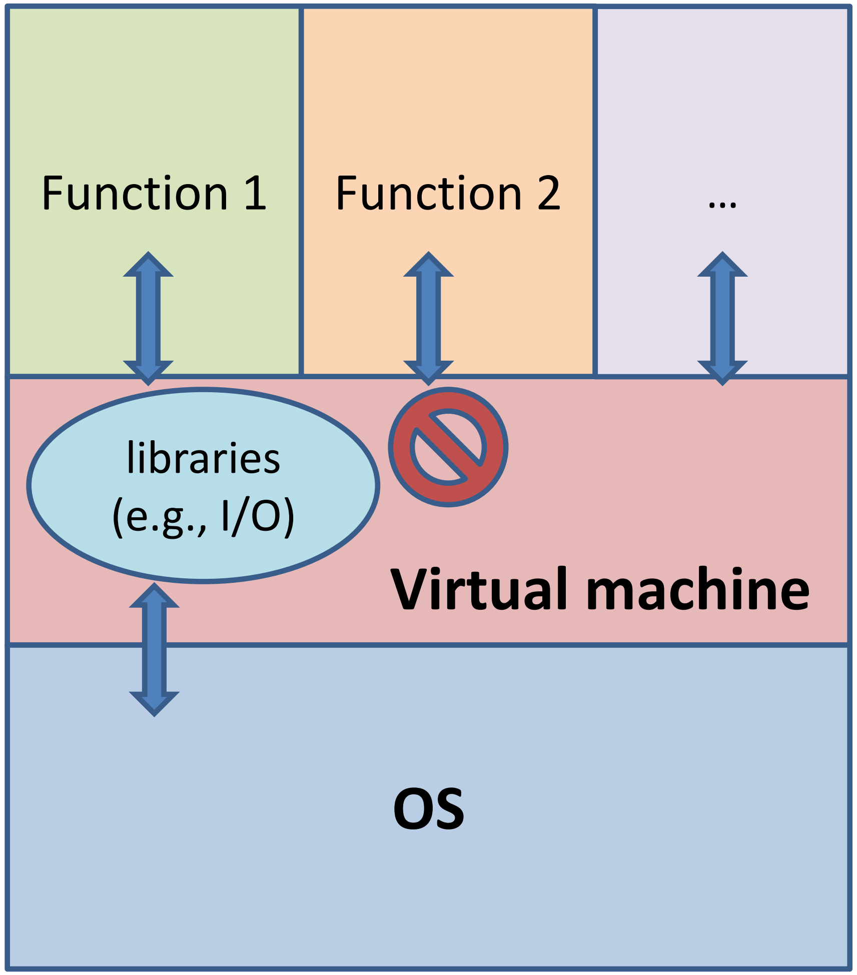


Permissions (as in Java)

Access to resources is expressed in terms of ***permissions***, such as “may perform screen I/O”.

Before execution, an annotation on each piece of code (e.g., function) indicates its permissions.

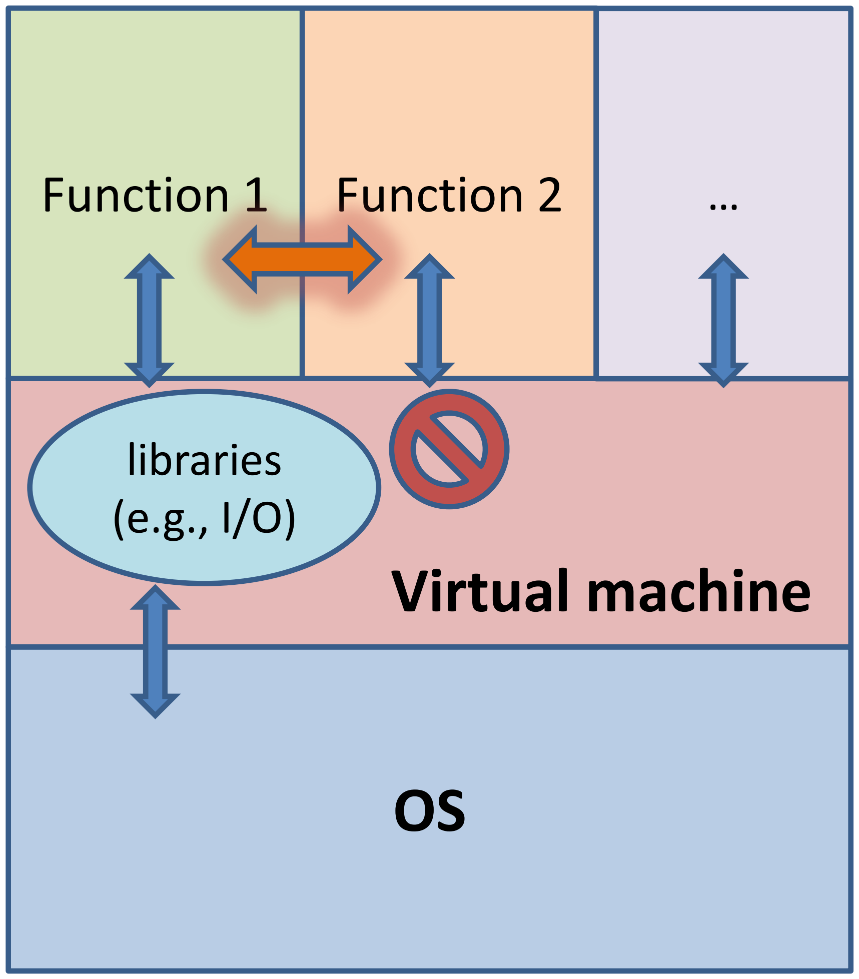
A ***configurable policy*** determines permissions depending on ***code origin***.



Permissions (cont.)

Code with a variety of origins, more or less trusted, may call one another or share data.

Should all of their permissions count in access decisions?



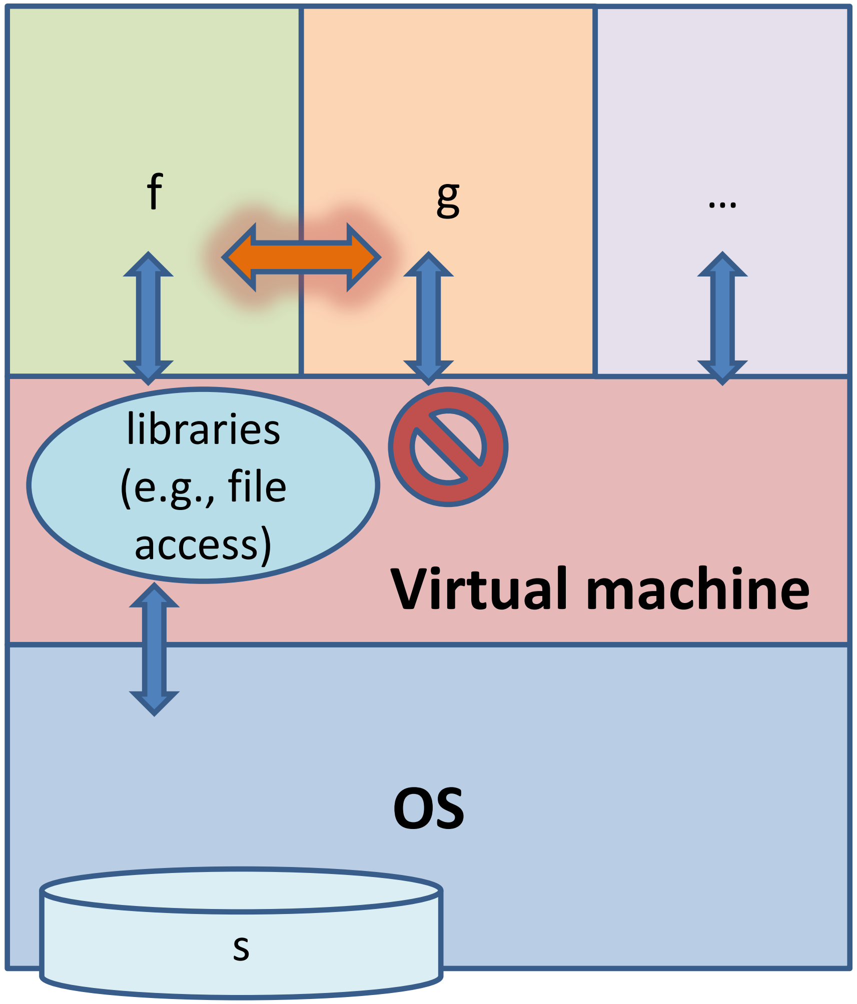
One answer, on a simple example

(also as in Java)

Suppose that f(s) modifies the file named s.

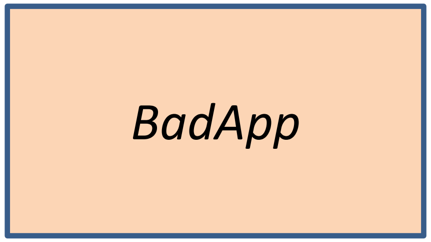
If g calls f(s), ***both*** should have permission to write to s.

(Otherwise, f may be used as a ***confused deputy***.)



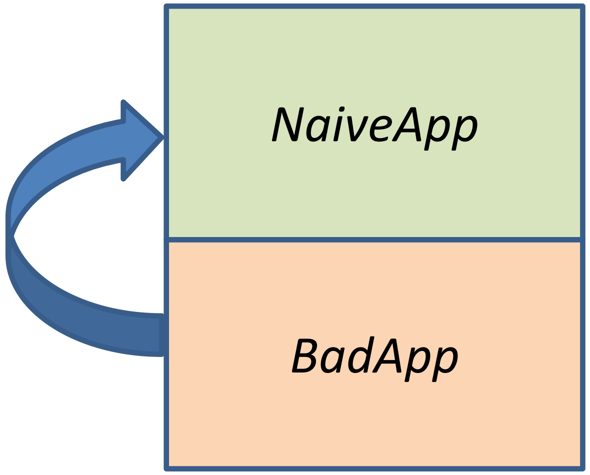
An example where looking at the stack suffices

```
// Fully trusted but naive: has all permissions
public class NaiveApp {
  public static void Write (string s, … ) {
    File.Write (s, … );
  }
}
// Untrusted: no FileIOPermission
class BadApp {
  public static void Main() {
    NaiveApp.Write (“..\\password”, …);
}
```



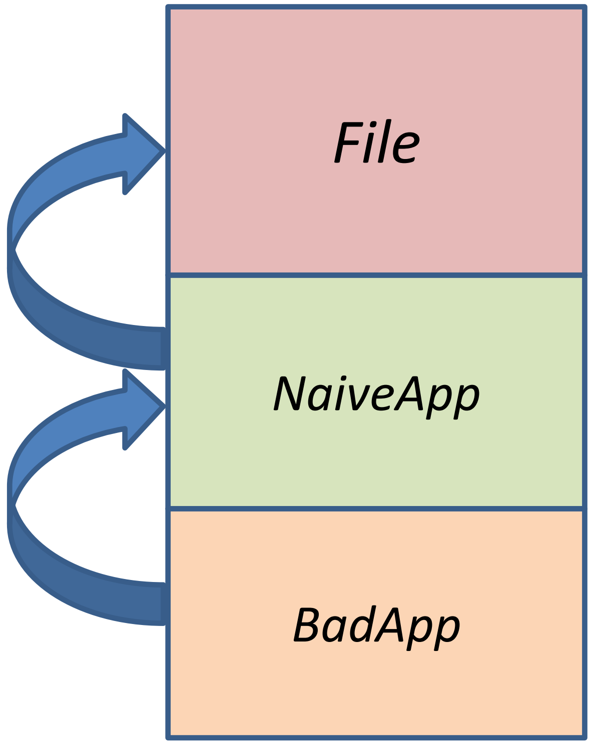
An example where looking at the stack suffices

```
// Fully trusted but naive: has all permissions
public class NaiveApp {
  public static void Write (string s, … ) {
    File.Write (s, … );
  }
}
// Untrusted: no FileIOPermission
class BadApp {
  public static void Main() {
    NaiveApp.Write ("..\\password", …);
}
```



An example where looking at the stack suffices

```
// Fully trusted but naive: has all permissions
public class NaiveApp {
  public static void Write (string s, … ) {
    File.Write (s, … );
  }
}
// Untrusted: no FileIOPermission
class BadApp {
  public static void Main() {
    NaiveApp.Write (“..\\password”, …);
}
```

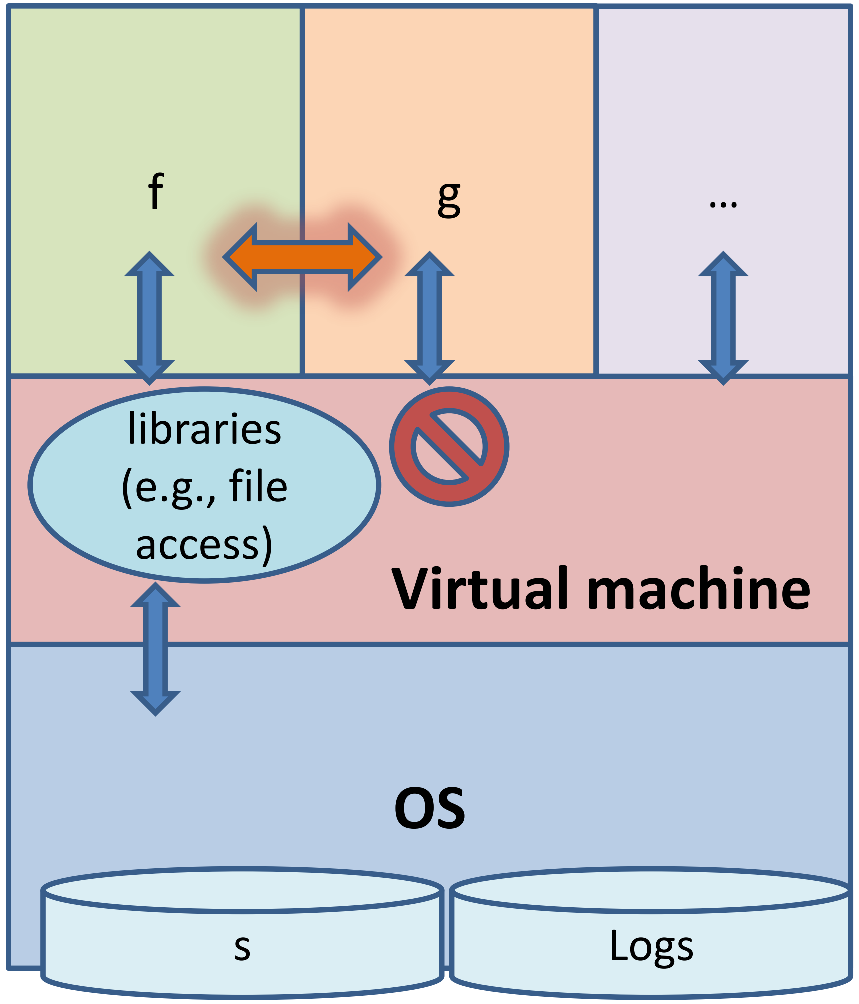


A twist

Suppose that f(s) wants to write to a log that g should not access.

If f is a trusted function, it can check that g's call is ok, assert it, and then use its own authority for writing to the log.

Afterwards, g's permissions do not matter, only f's.

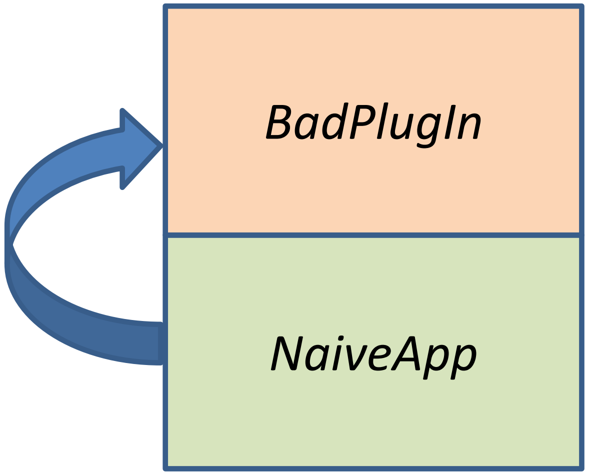


An example where looking at the stack does not suffice

```
// Fully trusted but naive: has all permissions
class NaiveApp {
 public static void Main() {
  string s = BadPlugIn.TempFile ();
  File.Write(s, … );
 }
}
// Untrusted: no FileIOPermission
public class BadPlugIn {
  public static string TempFile () {
   return "..\\ password";
 }
}
```

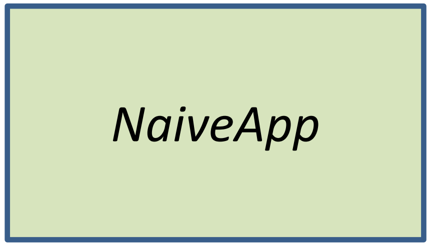
An example where looking at the stack does not suffice

```
// Fully trusted but naive: has all permissions
class NaiveApp {
  public static void Main() {
    string s = BadPlugIn.TempFile ();
    File.Write(s, … );
  }
}
// Untrusted: no FileIOPermission
public class BadPlugIn {
    public static string TempFile () {
      return "..\\ password";
  }
}
```



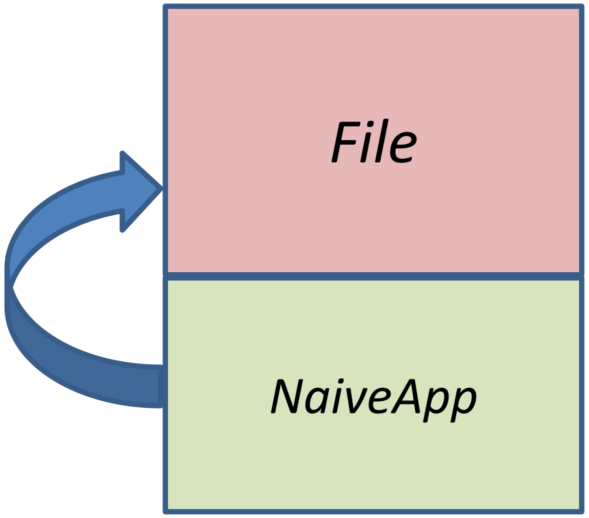
An example where looking at the stack does not suffice

```
// Fully trusted but naive: has all permissions
class NaiveApp {
  public static void Main() {
    string s = BadPlugIn.TempFile ();
    File.Write(s, … );
  }
}
// Untrusted: no FileIOPermission
public class BadPlugIn {
    public static string TempFile () {
      return “..\\ password”;
  }
}
```



An example where looking at the stack does not suffice

```
// Fully trusted but naive: has all permissions
class NaiveApp {
  public static void Main() {
    string s = BadPlugIn.TempFile ();
    File.Write(s, … );
  }
}
// Untrusted: no FileIOPermission
public class BadPlugIn {
    public static string TempFile () {
      return "..\\ password";
  }
}
```



Criticisms

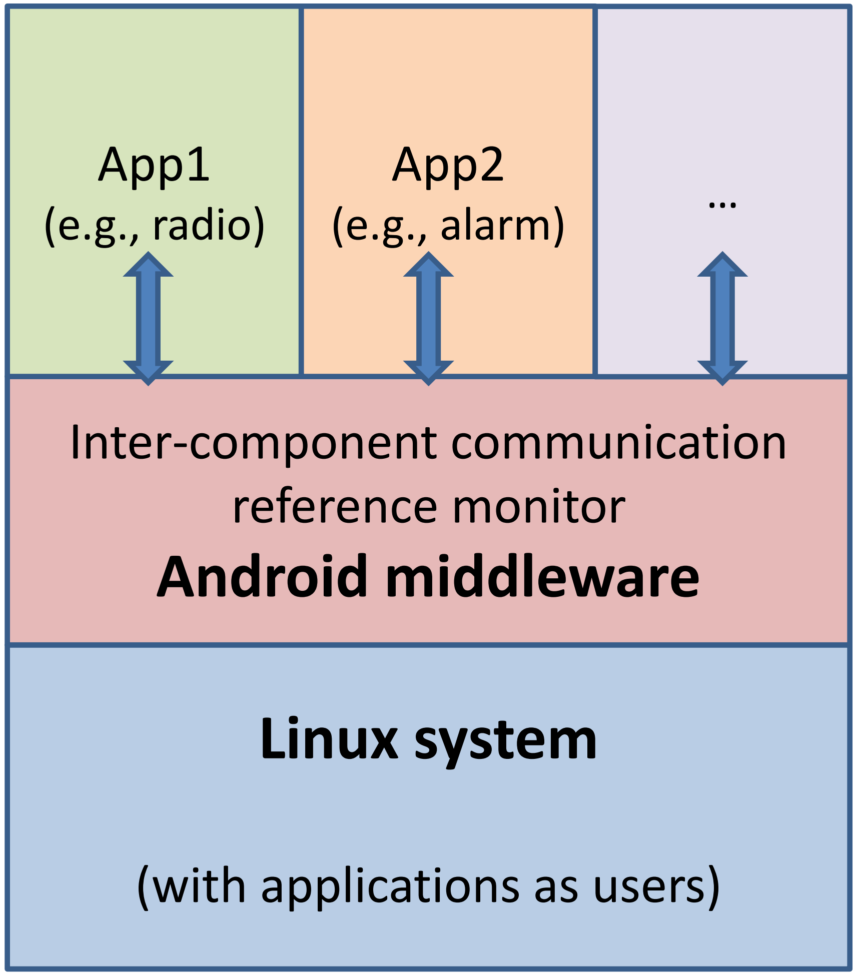
- Does this technique achieve real security?
  for what policy?
- Looking at chains of calls is not satisfactory.
  - Some other constructs require careful treatment.
  - A standard formulation (“stack inspection”) is tied to a particular stack implementation.
    $\Rightarrow$ It rules out or complicates optimizations.
- It can get hard to understand security.

Access control in Android

Applications are principals.

Each application comes with fixed permissions

- declared by developer;
- accepted by user at installation time;
- checked at run-time;
- some standard, e.g., access network;
- others defined by developers;
- over 100.


App1
(e.g., radio)
App2
(e.g., alarm)
…
Inter-component communication reference monitor
Android middleware
Linux system
(with applications as users)

(There are many other aspects to Android security.)

Languages and logics for access control policies

From matrices to rules

- An access control matrix may be represented with a ternary predicate symbol may-access.
- Other predicates may represent groups, etc..
- We may use standard logical operators.
- We may then write formulas such as:

  may-access(Alice, Foo.txt, Rd)

  and rules such as:

  may-access(p, o, Wr) $\rightarrow$ may-access(p, o, Rd)

  good(p) $\rightarrow$ may-access(p, o, Rd)

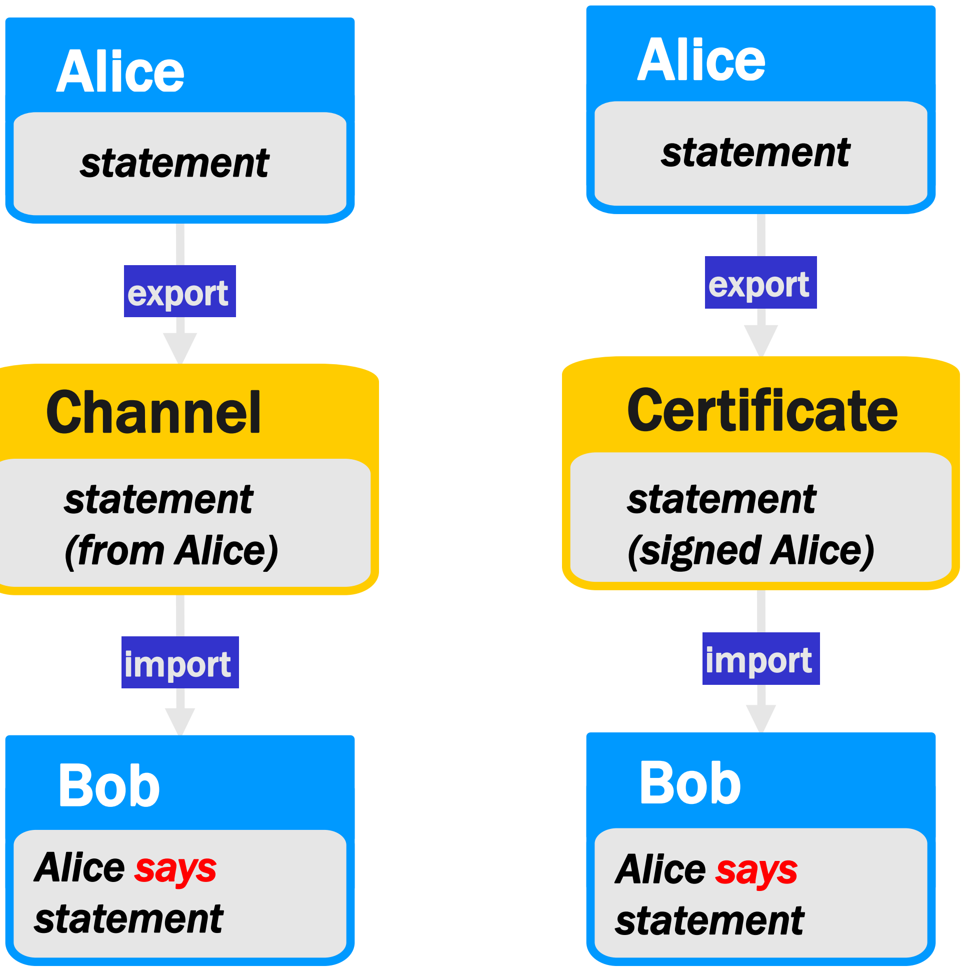
(see **XACML** and the like)

Going further: policies for distributed systems

- In distributed systems, there are multiple sources of assertions, trusted differently.
- This is reflected in some proposed public-key infrastructures, policy languages, and logics.
- One idea is to represent explicitly the principals that make assertions and to reason about them…

Says

- "says" represents communication across contexts.
- It abstracts from the details of authentication.
- The statement may be atomic or a more complex rule.



A calculus for access control

[with Burrows, Lampson, Plotkin, and Wobber 1991; and Garg, 2008]

- A simple notation for assertions:
  - A says s
  - A speaks for B
- With logical rules, for example:

$\vdash A \text{ says } (s \rightarrow t) \rightarrow (A \text{ says } s) \rightarrow (A \text{ says } t)$

$\vdash s \rightarrow (A \text{ says } s)$ $\quad\vdash (A \text{ says } A \text{ says } s) \rightarrow (A \text{ says } s)$

A calculus for access control

[with Burrows, Lampson, Plotkin, and Wobber 1991; and Garg, 2008]

- A simple notation for assertions:
  - A says s
  - A speaks for B
- With logical rules, for example:

$\vdash A \text{ says } (s \rightarrow t) \rightarrow (A \text{ says } s) \rightarrow (A \text{ says } t)$

$\vdash s \rightarrow (A \text{ says } s)$ $\quad$ $\vdash (A \text{ says } A \text{ says } s) \rightarrow (A \text{ says } s)$

$\vdash A \text{ speaks for } B \rightarrow (A \text{ says } s) \rightarrow (B \text{ says } s)$

$\vdash A \text{ speaks for } A$

$\vdash A \text{ speaks for } B \wedge B \text{ speaks for } C \rightarrow A \text{ speaks for } C$

$\vdash (B \text{ says } (A \text{ speaks for } B)) \rightarrow (A \text{ speaks for } B)$

A calculus for access control

[with Burrows, Lampson, Plotkin, and Wobber 1991; and Garg, 2008]

- A simple notation for assertions:
  - A says s
  - A speaks for B $\equiv$ for all X. ((A says X) $\rightarrow$ (B says X))
- With logical rules, for example:

$\vdash$ A says ($s \rightarrow t$) $\rightarrow$ (A says s) $\rightarrow$ (A says t)

$\vdash$ $s \rightarrow$ (A says s) $\vdash$ (A says A says s) $\rightarrow$ (A says s)

$\vdash$ A speaks for $B \rightarrow$ (A says s) $\rightarrow$ (B says s)

$\vdash$ A speaks for A

$\vdash$ A speaks for $B \land B$ speaks for $C \rightarrow A$ speaks for C

$\vdash$ (B says (A speaks for B)) $\rightarrow$ (A speaks for B)

A calculus for access control

[with Burrows, Lampson, Plotkin, and Wobber 1991; and Garg, 2008]

- A simple notation for assertions:
  - A says s
  - A speaks for B $\equiv$ for all X. ((A says X) $\rightarrow$ (B says X))

"same consequences"

- With logical rules, for example:

$\vdash$ A says ($s \rightarrow t$) $\rightarrow$ (A says s) $\rightarrow$ (A says t)

$\vdash$ $s \rightarrow$ (A says s) $\vdash$ (A says A says s) $\rightarrow$ (A says s)

$\vdash$ A speaks for B $\rightarrow$ (A says s) $\rightarrow$ (B says s)

$\vdash$ A speaks for A

$\vdash$ A speaks for B $\wedge$ B speaks for C $\rightarrow$ A speaks for C

$\vdash$ (B says (A speaks for B)) $\rightarrow$ (A speaks for B)

An example

- Let good-to-delete-file1 be a proposition.
- Let *B* controls *s* stand for (*B* says *s*) $\rightarrow$ *s*
- Assume that
  - *B* says (*A* speaks for *B*)
  - *B* controls good-to-delete-file1
  - *A* says good-to-delete-file1
- We can derive:
  - *B* says good-to-delete-file1
  - good-to-delete-file1

Applications

Several languages rely on logics for access control:

- D1LP and RT [Li, Mitchell, et al.]
- SD3 [Jim] and Binder [DeTreville]
- Daisy [Cirillo et al.]
- SecPAL [Becker, Fournet, and Gordon] and DKAL [Gurevich and Neeman]

“says” and “speaks for” play a role in other systems:

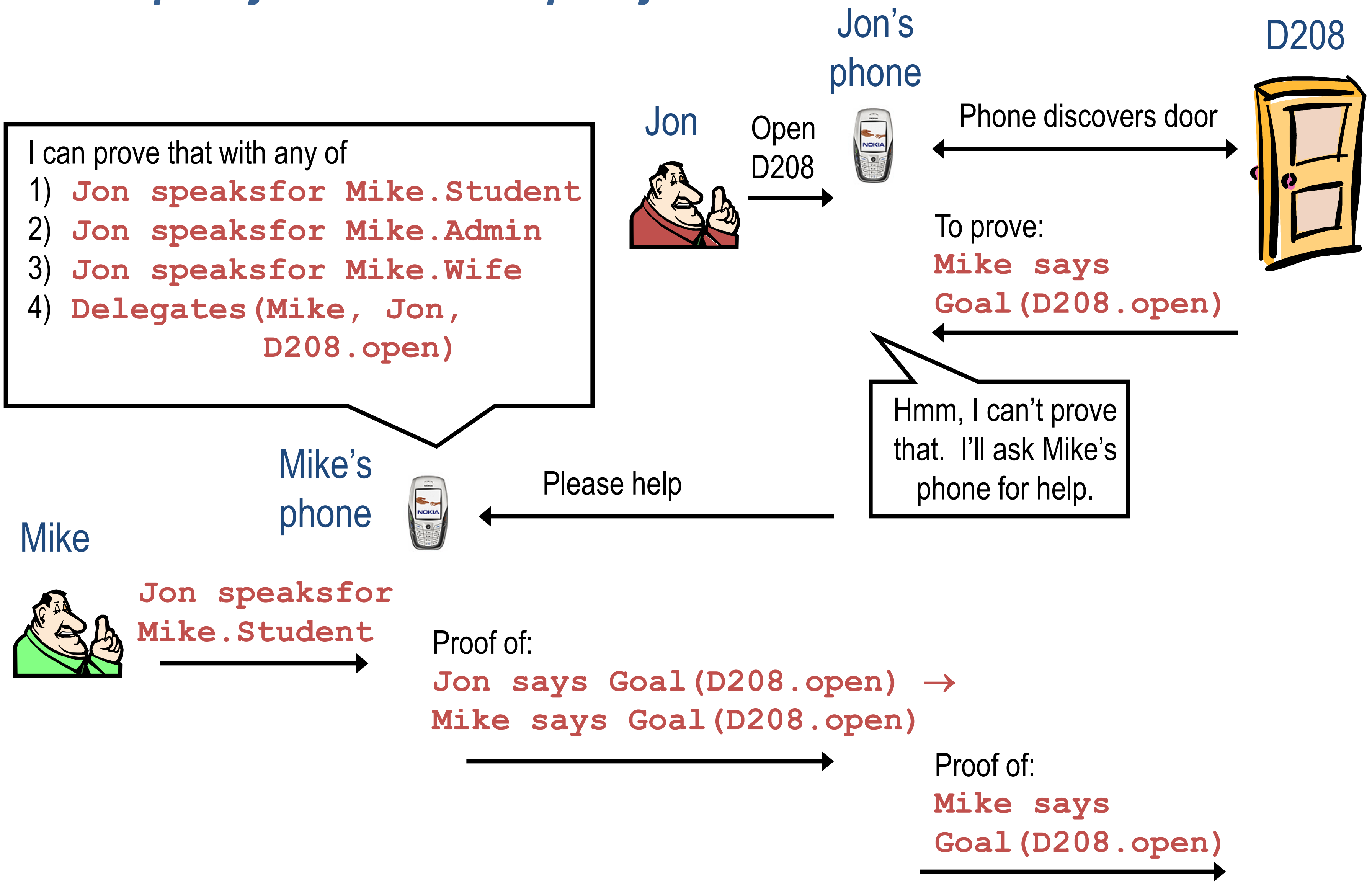
- SDSI and SPKI [Lampson and Rivest; Ellison et al.]
- Alpaca [Lesniewski-Laas et al.] and Aura [Vaughan et al.]
- PCFS (proof-carrying file system) [Garg and Pfenning]
- …

An example system: Grey

[Bauer, Reiter, et al., 2005–2008]

- Turns a cell phone into a tool for delegating and exercising authority.
- Uses cell phones to replace physical locks and key systems.
- Implemented in part of CMU.
- With access control based on logic and distributed proofs.

An example of a distributed proof:


Jon's phone
D208
Jon
Open D208
Phone discovers door
I can prove that with any of
1) Jon speaksfor Mike.Student
2) Jon speaksfor Mike.Admin
3) Jon speaksfor Mike.Wife
4) Delegates(Mike, Jon, D208.open)
To prove: Mike says Goal(D208.open)
Hmm, I can't prove that. I'll ask Mike's phone for help.
Mike's phone
Please help
Mike
Jon speaksfor Mike.Student
Proof of: Jon says Goal(D208.open) → Mike says Goal(D208.open)
Proof of: Mike says Goal(D208.open)

Slide originally from Mike Reiter

A small language: Binder

- Binder is a relative of Prolog.
- Like Datalog, it lacks function symbols.
- It also includes the special construct says.
- It does not include much else.

- Binder is not the most recent.
- Current systems (e.g., SecPAL) have similarities with Binder, but they are more complex.

An example

- Facts
  - owns(Alice, Foo.txt).
  - Alice says good(Bob).
- Rules
  - may_access(p, o, Rd) :- owns(q, o), blesses(q, p).
  - blesses(Alice, p) :- Alice says good(p).
- Conclusions
  - may_access(Bob, Foo.txt, Rd).

Proof rules

- Binder includes standard logical rules (“resolution”).
- E.g.,

  may_access(p, o, Rd) :- owns(q, o), blesses(q, p).

  owns(Alice, Foo.txt).

  imply

  may_access(p, Foo.txt, Rd) :- Alice says good(p).

Proof rules: importing

- In addition, formulas from a context F can be imported to a context D.
  - This adds “F says” in front of all atoms without a “says”.
  - It applies only to clauses where the head atom does not have “says”.

Importing by example

- Suppose F has the rules
  - may_access(p, o, Rd) :- owns(q, o), blesses(q, p).
  - blesses(Alice, p) :- Alice says good(p).
  - Alice says good(Bob).
- D may import the first two as:
  - F says may_access(p, o, Rd) :-
    F says owns(q, o), F says blesses(q, p).
  - F says blesses(Alice, p) :- Alice says good(p).
- D may import good(Bob) directly from Alice.

Importing by example (cont.)

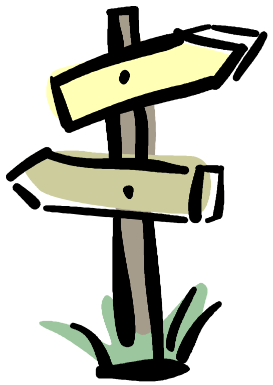
- Suppose F has the rule
  - blesses(Alice, p) :- Alice says good(p).
- D may import it as:
  - F says blesses(Alice, p) :- Alice says good(p).
- D and F should agree on Alice’s identity.
- But the meaning of predicates may vary, and it typically will.
  For example, F may also have:
  - blesses(Bob, p) :- Bob says excellent(p).

Important properties

- Policies can use application-specific predicates.
- Statements can be read declaratively.
- Queries are decidable (typically in PTime).

Issues and research directions

- What about algorithmic problems?
- What about more proof systems? Semantics?
- Can all reasonable policies be expressed? Can the simple ones be expressed simply?
- Is this a way of explaining other approaches? or something for direct use (e.g., as XACML)?



Some reading

- “Setuid Demystified”, by Chen, Wagner, and Dean.
- “Stack Inspection: Theory and Variants”, by Fournet and Gordon.
- “Understanding Android Security”, by Enck, Ongtang, and McDaniel.
- My “Logic in Access Control (Tutorial Notes)” and its references.