



Making certificates programmable

John DeTreville
Microsoft Research
April 24, 2002

Why haven't PKIs "happened yet"?

- “To a large extent, the hoped-for public key infrastructure has not ‘happened yet.’ PKI for large, eclectic populations has not materialized; PKI for smaller, less diverse ‘enterprise’ populations is beginning to emerge, but at a slower rate than many would like or had expected.”
- We argue a structural reason, based on the restricted expressiveness of current PKIs

How expressive are certificates?

- **Current certificates—X.509, SDSI/SPKI, etc.—are limited in their expressiveness**
 - Many interesting relations are inexpressible
 - It is currently too hard to build a PKI from small, loosely-coupled parts
 - This restricts the global PKI ecosystem
- **Open systems need open certificates**
 - We must reduce the barriers to entry
- **“Every message should say what it means: the interpretation of a message should depend only on its content. It should be possible to write down a straightforward English sentence describing the content.” [Abadi & Needham 1996]**

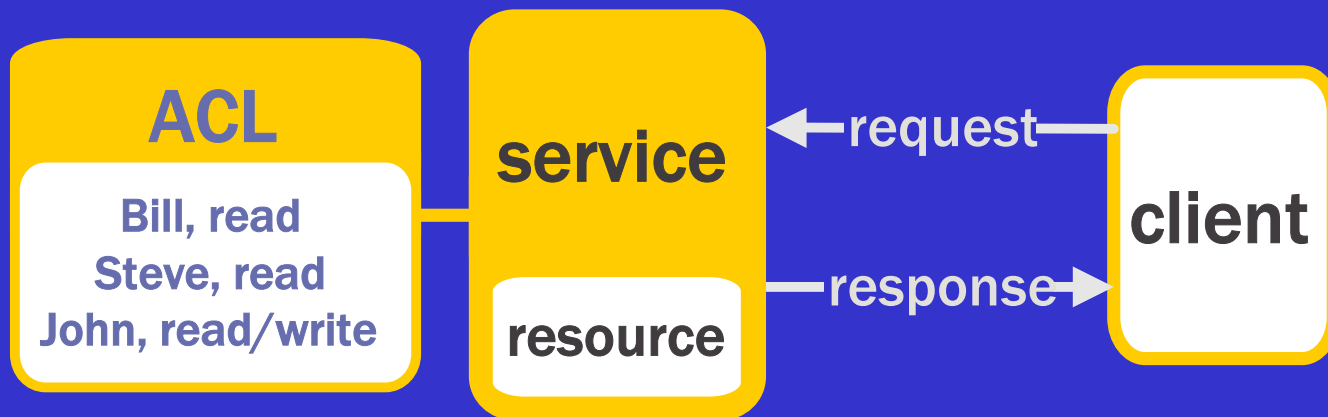
How expressive should certificates be?

- The expressiveness of our certificates is a cause for concern
- If our certificates are highly regular data structures that can work together only in a few restricted ways, it might be impossible to state a rich variety of relations
- This may be an advantage in some contexts
- Open systems need open certificates

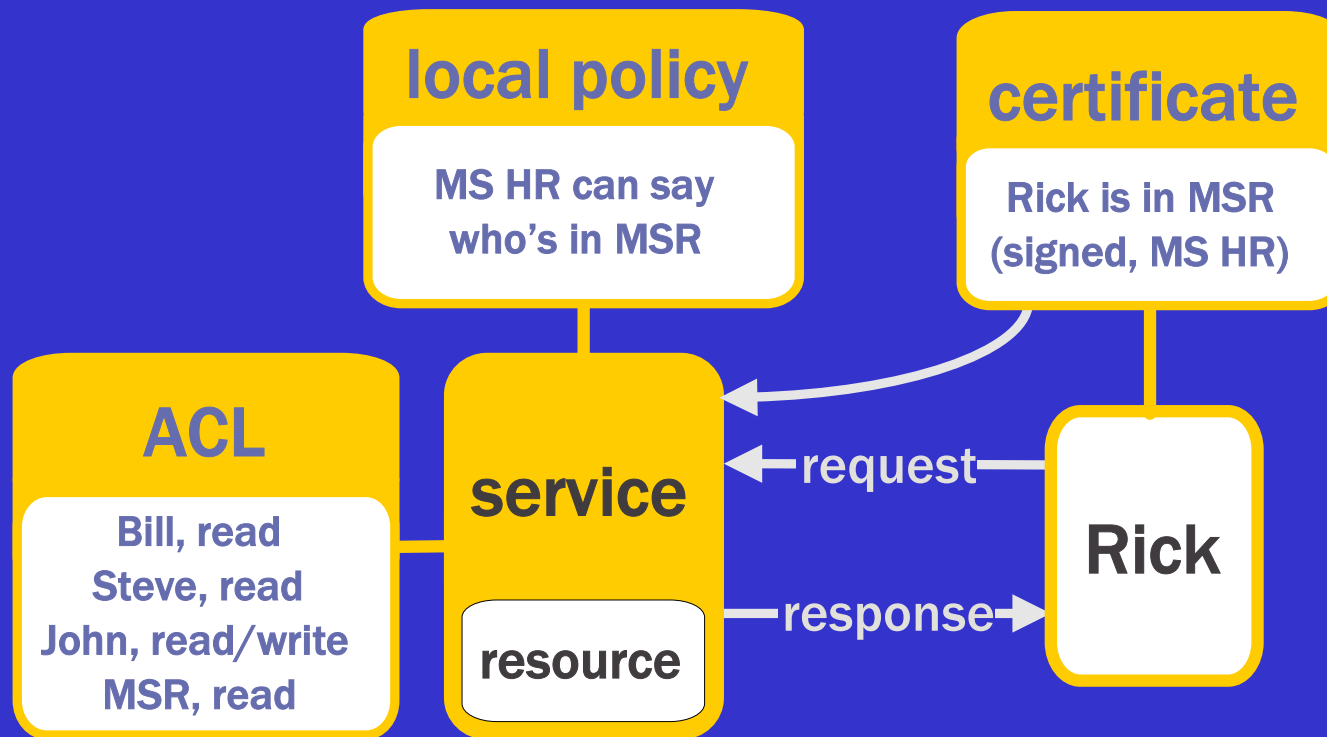
Security languages

- Security languages encode security statements (e.g., certificates) as schematized data structures, like ACLs or X.509 certificates
- The schema and its accompanying decision procedure define a *security language*
 - Our certificates, policies, and ACLs are formed from security statements written in our security language and interpreted by its decision procedure
- Most security languages are *ad hoc* and task-specific

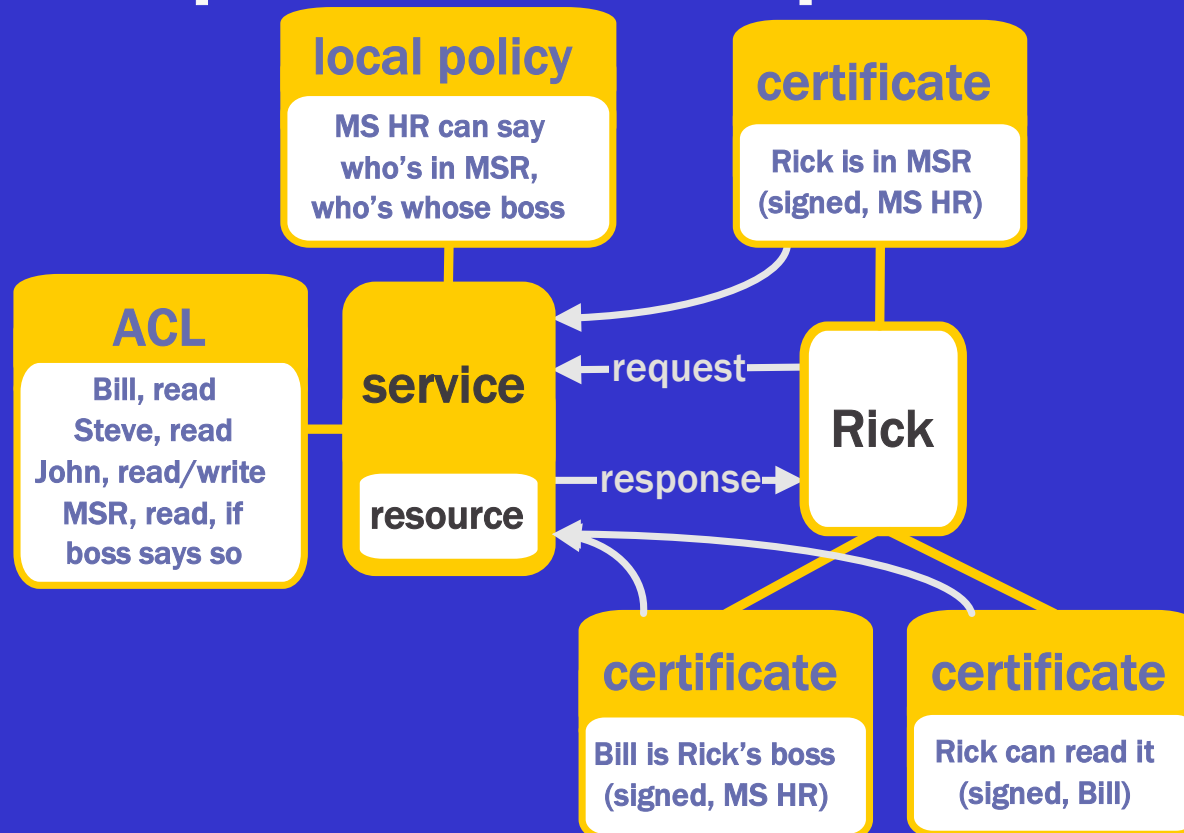
Example 1: simple



Example 2: more complex



Example 3: too complex



Making certificates programmable

8



Open systems need open security languages

- For a closed system with known requirements, we can choose a minimalist security language, closely matched to our needs
- For an open system to be used in unexpected ways and to evolve in unknown directions, we must make our language more expressive
 - An open distributed system with multiple administrative domains will have multiple interoperating policies
 - Independent security policies must interoperate fluidly and efficiently and correctly

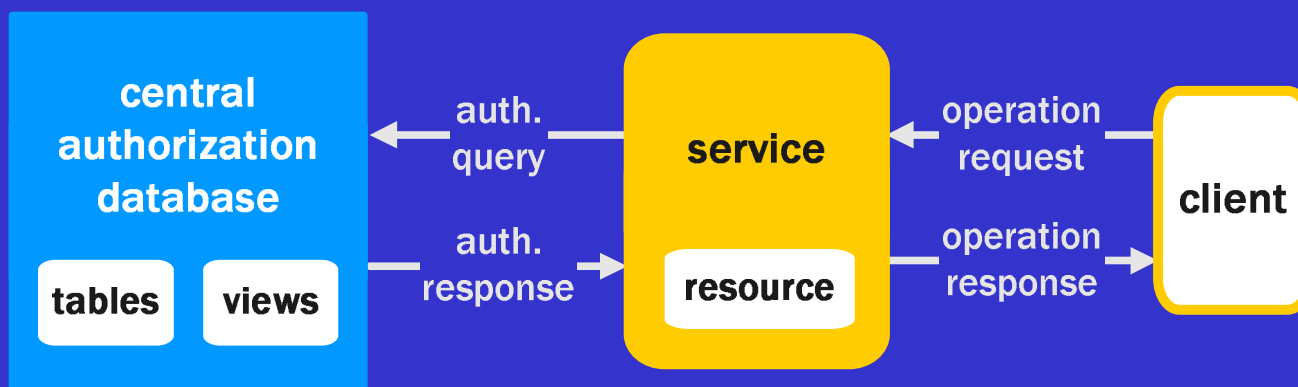
More expressiveness is good

- The more expressive our security language, the easier it is to say the wrong thing
- Of all the possible security relations among clients, services, and resources, which can we model?
- And perhaps most importantly, of all the simple relations, which ones can we model simply?
- Open systems need open security languages
- Our language is declarative
 - Multiple policies must interoperate fluidly and efficiently and correctly
- Our language is PTIME-complete

A hypothetical centralized system with maximal expressiveness

- We might imagine storing all our security statements as data in a single shared, centralized database, and using database queries to answer our access control questions
- Given a particular request, we would construct an appropriate query to discover whether the principal issuing the request should be allowed to access the resource named

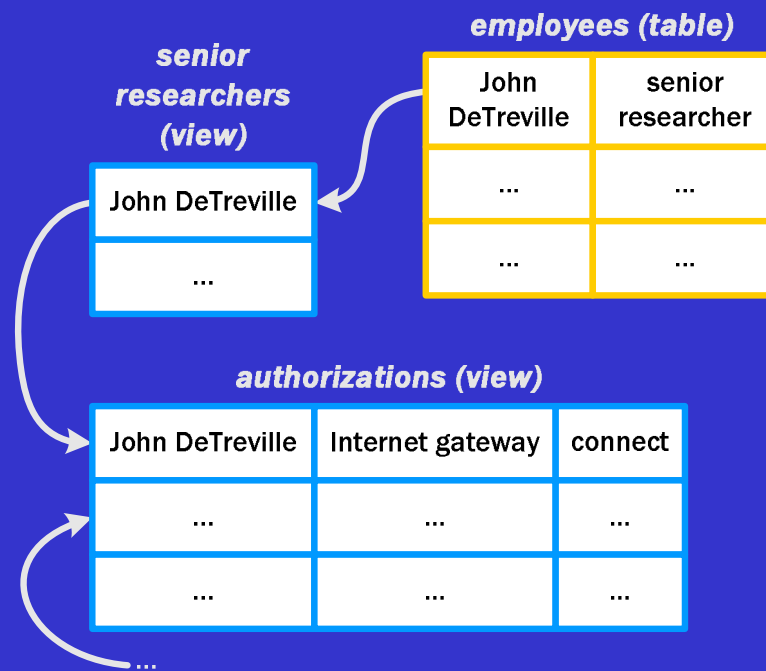
A hypothetical central database



Using a relational database to represent state and logic

- The central authorization database contains tables and views
 - Tables store data
 - Views are defined in terms of data that appear in tables and other views
- We can define the database views and queries in terms of relational algebra, which operates on tables and views using operators like *select*, *project*, and *join*
- We can also define them in terms of the logic-programming language *datalog*

Authorization via tables and views



Making certificates programmable

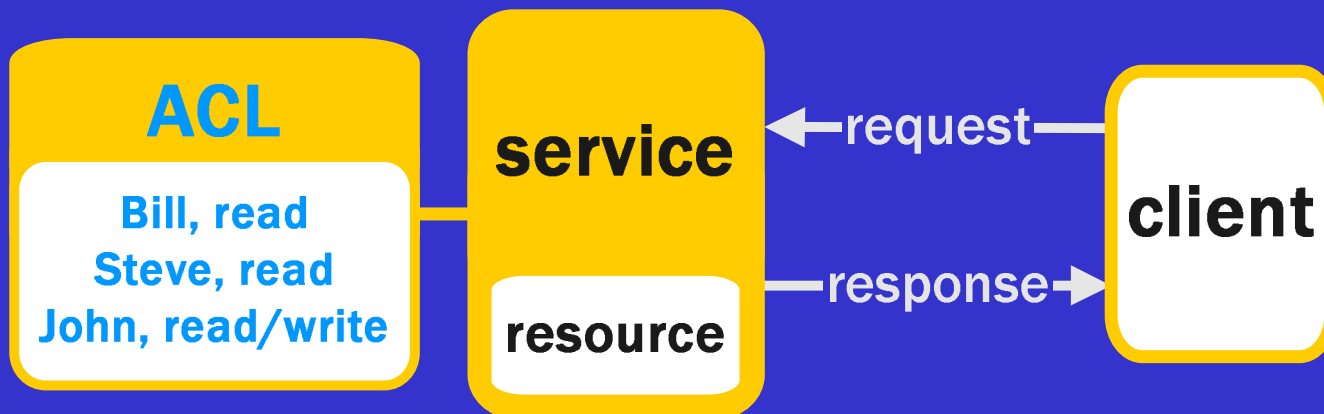
14

Distributed logic programming

- We extend datalog with authenticated communication across a distributed environment
 - `employee(john, msr).`
 - `employee(john, msr) (signed, ms_hr).`
 - `ms_hr says employee(john, msr).`



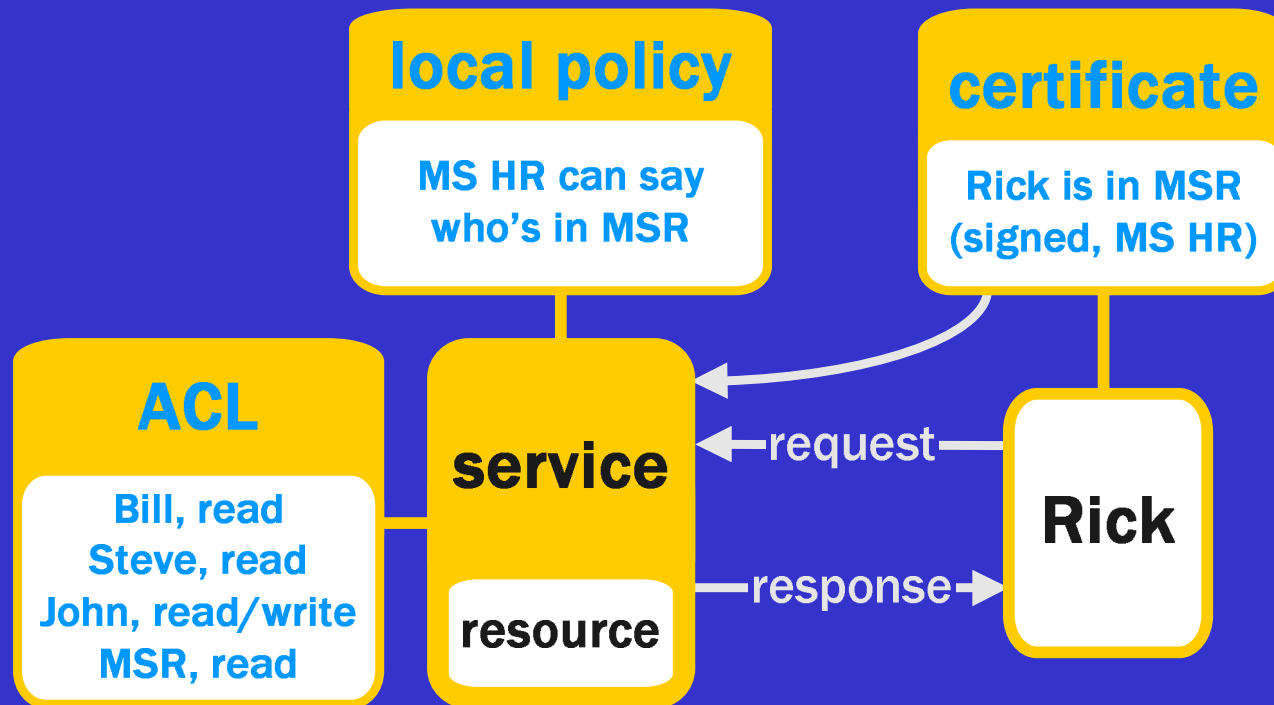
Example 1



Example 1 in datalog

#	English statement	datalog statement
1	“Bill can read R”	<code>can(bill, read, r).</code>
2	“Steve can read R”	<code>can(steve, read, r).</code>
3	“John can read and write R”	<code>can(john, read, r).</code> <code>can(john, write, r).</code>

Example 2



Making certificates programmable

18

Example 2 in datalog

#	English statement	datalog statement
1	"Bill can read R"	<code>can(bill, read, r).</code>
2	"Steve can read R"	<code>can(steve, read, r).</code>
3	"John can read and write R"	<code>can(john, read, r).</code> <code>can(john, write, r).</code>
4	"Members of MSR can read R"	<code>can(X, read, r)</code> <code>:- member(X, msr).</code>

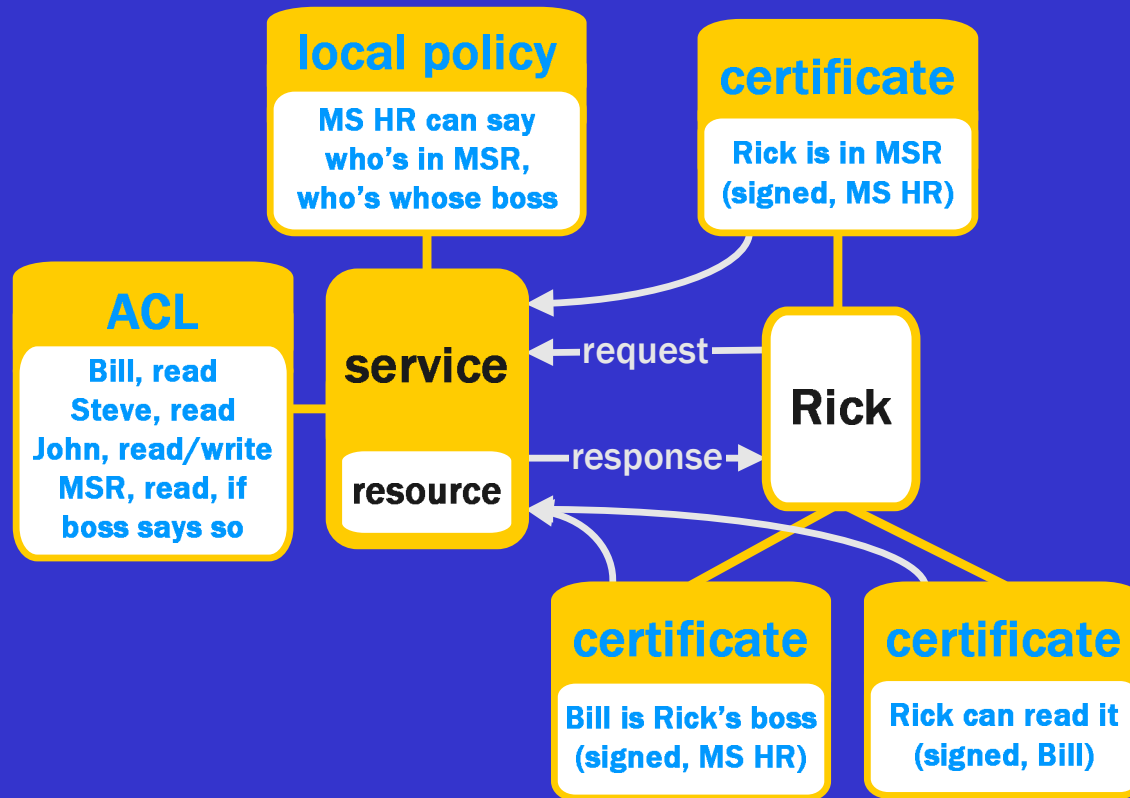
Example 2 in datalog (continued)

#	English statement	datalog statement
5	“MS HR can say who’s in MSR”	<pre>member(X, msr) :- ms_hr says member(X, msr).</pre>

Example 2 in datalog (continued)

#	English statement	datalog statement
6	"Rick is in MSR"	<code>member(rick, msr).</code> <i>(at MS HR, before export)</i>
		<code>ms_hr says</code> <code> member(rick, msr).</code> <i>(at service, after import)</i>

Example 3



Making certificates programmable

22

Example 3 in datalog

#	English statement	datalog statement
1	"Bill can read R"	<code>can(bill, read, r).</code>
2	"Steve can read R"	<code>can(steve, read, r).</code>
3	"John can read and write R"	<code>can(john, read, r).</code> <code>can(john, write, r).</code>
4	"Members of MSR can read R if their boss says so"	<code>can(X, read, r)</code> <code>:- member(X, msr),</code> <code>boss(B, X),</code> <code>B says</code> <code>can(X, read, r).</code>

Example 3 in datalog (continued)

#	English statement	datalog statement
5	“MS HR can say who’s in MSR”	<pre>member(X, msr) :- ms_hr says member(X, msr).</pre>

Example 3 in datalog (continued)

#	English statement	datalog statement
6	“MS HR can say who’s whose boss”	<pre>boss(B, X) :- ms_hr says boss(B, X).</pre>

Example 3 in datalog (continued)

#	English statement	datalog statement
7	"Rick is in MSR"	<code>member(rick, msr).</code> <i>(at MS HR, before export)</i>
		<code>ms_hr says</code> <code> member(rick, msr).</code> <i>(at service, after import)</i>

Example 3 in datalog (continued)

#	English statement	datalog statement
8	"Bill is Rick's boss"	<code>boss(bill, rick).</code> <i>(at MS HR, before export)</i>
		<code>ms_hr says</code> <code> boss(bill, rick).</code> <i>(at service, after import)</i>

Example 3 in datalog (continued)

#	English statement	datalog statement
9	"Rick can read R"	<code>can(rick, read, r).</code> <i>(at Bill, before export)</i>
		<code>bill says</code> <code> can(rick, read, r).</code> <i>(at service, after import)</i>

Indirection

- It is a folk theorem that any problem in computing can be solved by adding another level of indirection
- We reify tables and views and services
 - References to public keys need not be constant, but can themselves be drawn from tables and views.
- Allowing views in one service to refer to tables or views in another allows the PKI designer to use an arbitrary number of levels of indirection
 - This will be a powerful technique, and it can make explicit and extend many security assumptions that would otherwise be wired into the system architecture

Trust

- Delegation and trust are the tools for the indirection of security decisions in traditional security systems
- We model delegation and trust using the controlled import of datalog statements and rules
 - One service trusts another if its views depend on tables or views from that other service
 - For example, chains of delegation are modeled as chainable import rules
- Because the database can hold the names of services (e.g., their public keys), we can organize services into groups or other more complex relations
 - We can have a table or view of which services “trust” which others

Conclusions and future work

- Most security languages to date have been *ad hoc* and task-specific
- Certificates can be made more expressive and precise using program code written in an enhanced relational algebra or datalog
- The choice of a security language involves an engineering tradeoff between increasing generality and maintaining usability
 - Indirection is a tool
 - Open systems need open security languages
 - Security statements should be expressible in English



Microsoft[®]

Making certificates programmable

32