Decentralized Trust Management

Matt Blaze Joan Feigenbaum Jack Lacy
AT&T Research
Murray Hill, NJ 07974
{mab,jf,lacy}@research.att.com

Abstract

We identify the trust management problem as a distinct and important component of security in network services. Aspects of the trust management problem include formulating security policies and security credentials, determining whether particular sets of credentials satisfy the relevant policies, and deferring trust to third parties. Existing systems that support security in networked applications, including X.509 and PGP, address only narrow subsets of the overall trust management problem and often do so in a manner that is appropriate to only one application. This paper presents a comprehensive approach to trust management, based on a simple language for specifying trusted actions and trust relationships. It also describes a prototype implementation of a new trust management system, called PolicyMaker*, that will facilitate the development of security features in a wide range of network services.*

1. Introduction

The importance of cryptographic techniques in a wide range of network services is universally recognized. A service that uses cryptography must accommodate appropriate notions of users' security policies, their security credentials, and their trust relationships. For example, an electronic banking system must enable a bank to state that at least k bank officers are needed to approve loans of $1,000,000 or less (a policy), it must enable a bank employee to prove that he can be counted as 1 out of k approvers (a credential), and it must enable the bank to specify who may issue such credentials (a trust relationship).

It is our thesis that a coherent intellectual framework is needed for the study of security policies, security credentials, and trust relationships. We refer collectively to these components of network services as the *trust management problem.* Although certain aspects of trust management are dealt with satisfactorily by existing services in specialized ways that are appropriate to those services (*e.g.*, the PGP secure email system allows users to create security credentials by binding their IDs to their public keys), the trust management problem has not previously been identified as a general problem and studied in its own right. The goal of this paper is to identify the problem and to take the first step toward a comprehensive approach to solving it that is independent of any particular application or service.

To address trust management *per se*, as opposed to the security needs of one particular service, we have developed a general framework that can be applied to any service in which cryptography is needed. To facilitate the use of our approach, we are building a new type of tool, best described as a *trust management system.* Our system, called *PolicyMaker*, is suitable as a tool in the development of services whose main goal is privacy and authenticity (*e.g.*, a secure communication system) as well as services in which these features are merely enablers or enhancements (*e.g.*, an electronic shopping system).

Our approach to trust management is based on the following general principles.

- **Unified mechanism**: Policies, credentials, and trust relationships are expressed as programs (or parts of programs) in a "safe" programming language. Existing systems are forced to treat these concepts separately. By providing a common language for policies, credentials, and relationships, we make it possible for network applications to handle security in a comprehensive, consistent, and largely transparent manner.

- **Flexibility**: Our system is expressively rich enough to support the complex trust relationships that can occur in the very large-scale network applications currently being developed. At the same time, simple and standard policies, credentials, and relationships can be expressed succinctly and comprehensibly. In particular, PGP and X.509 "certificates" need only trivial modifications to be usable in our framework.

- **Locality of contol**: Each party in the network can decide in each circumstance whether to accept the credentials presented by a second party or, alternatively, on which third party it should rely for the appropriate "certificate." By supporting local control of trust relationships, we avoid the need for the assumption of a globally known, monolithic hierarchy of "certifying authorities." Such hierarchies do not scale beyond single "communities of interest" in which trust can be defined unconditionally from the top down.

- **Separation of mechanism from policy**: The mechanism for verifying credentials does not depend on the credentials themselves or the semantics of the applications that use them. This allows many different applications with widely varying policy requirements to share a single certificate verification infrastructure.

1.1. Review of Existing Approaches

Existing services that use cryptographic techniques do not use a "trust management system" as such and do not identify "trust management" as a problem in its own right. Usually, implicit notions of trust management are handled by applications. Most are based on public key "certificates" in which a trusted third party signs a specially formed message certifying the identity associated with a public key. How the certified identity is acted upon, however, is left to the application. The two best known certificate systems are those of PGP and X.509.

In the PGP system [7], a user generates a $(PublicKey, SecretKey)$ pair that is associated with his unique ID; usually an ID is of the form $(Name, EmailAddress)$. Keys are stored in *key records*. A public (resp. secret) key record contains an ID, a public (resp. secret) key, and a timestamp of when the key pair was created. Public keys are stored on *public key rings* and secret keys on *secret key rings*. Each user must store and manage a pair of key rings.

If user A has a good copy of user B's public-key record, *e.g.*, a copy that he is confident (for whatever reason) has not been tampered with since B generated it, then A can sign this copy and pass it on to user C. A thus acts as an *introducer* of B to C. A signed key record is called a *key certificate*,[1] and we sometimes use the word "certify" as a synonym for "sign." Each user must tell the PGP system which individuals he or she trusts as introducers and must certify the introducers' public-key records with his own secret key. Moreover, a user may specify the *degree of trust* that he has in each introducer; an individual may be designated *unknown, untrusted, marginally trusted*, or *completely trusted*. Each user stores his trust information on his key rings and tunes PGP so that it assigns a *validity score* to each certificate on a key ring and uses the key in that certificate only if the score is high enough. For example, a skeptical user may require two fully trusted signatures on a public-key record to judge the key it contains valid, and a less skeptical user may require only one fully trusted signature or two marginally trusted ones.

It is important to note that implicit in PGP is the assumption that the only notion of "security policy" that needs to be supported is that of verification of the ID of the sender of a message. Keys rings and degrees of trust allow each user to design his own policy *of this very limited form*. This narrow notion of policy is appropriate to PGP, which is designed specifically to provide secure email for individuals, but it is insufficient for the broader range of secure network services now being designed and implemented.

Note that A's signature on B's public-key record should not be interpreted to mean that A trusts B's personal integrity; the right interpretation is rather that A believes that the binding of B's identity to the key in the record is correct. Furthermore, note that trust is not transitive – the facts that A fully trusts B as an introducer and that B fully trusts C do not automatically imply anything about A's degree of trust in C.

As PGP has grown in popularity, a decentralized "web of trust" has emerged. Each individual is responsible for acquiring the public-key certificates he needs and for assigning degrees of trust to the introducers he gets them from. Similarly, each individual must create his own key pair and disseminate his own public key. This "grass roots" approach rejects the use of official *certifying authorities* that sign public keys of individuals (and those of other certifying authorities) and thereby act as "trust servers" for the users of those keys.

The X.509 authentication framework attempts to solve the same part of the trust management prob-

[1] The PGP User's Guide does not have distinct names for signed and unsigned key records; it refers to both of them as "certificates." We have chosen the term "record" to refer to an unsigned unit of key information so that we may use the term "certificate" as it is commonly used in the literature.

lem that PGP's introducer mechanism attempts to solve, namely the need to find a suitably trustworthy copy of the public key of someone with whom one wants to communicate.[2] As in PGP, X.509 certificates are signed records that associate users' IDs with their cryptographic keys; X.509 certificates contain more information than PGP certificates, *e.g.*, the names of the signature schemes used to create them and the time interval in which they are valid (see [3] for details), but their basic purpose is simply the binding of users to keys. However, X.509 differs sharply from PGP in its level of centralization of information. While anyone may sign public-key records and act as an introducer in PGP, the X.509 framework postulates that everyone will obtain certificates from an official *certifying authority* (CA). When user A creates a $(PublicKey, SecretKey)$ pair, he has it and the rest of the required information certified by one or more CAs and registers the resulting certificates with an official directory service. If A later wants to communicate securely with B, he obtains a certificate for B from one of the directory servers. If A and B have both been certified by the same CA, the directory server can just send B's certificate to A, who can verify its validity using the public key of this common CA. If A and B have not been directly certified by a common CA, then the directory service must create a *certification path* from A to B. This is a list of the form CA_1, $cert_1$, CA_2, $cert_2$, …, CA_n, $cert_n$, where $cert_i$, $1 \leq i < n$, is a certificate of CA_{i+1} that has been signed by CA_i, and $cert_n$ is a certificate of B. In order to use this path to obtain B's public key, A must know the public key of CA_1, the first authority in the path. Thus, the X.509 framework rests on the assumption that CAs are organized into a global "certifying authority tree" and that all users within a "community of interest" have keys that have been signed by CAs with a common ancestor in this global tree.

1.2. The PolicyMaker Approach

Despite the differences in the way the various certificate-based systems structure trust relationships, they all assume a similar, and, as we shall see, rather cumbersome, trust architecture in the applications that use them. In particular, identity-based certificates create an artifical layer of indirection between the information that is certified (which answers the question "who is the holder of this public key?") and the question that a secure application must answer ("can we trust this public key for this purpose?").

Consider the steps an application must go through to process a request based on a signed message from the holder of a traditional (X.509 or PGP) certificate. (Some of these steps might be performed by the operating system or otherwise hidden at a lower layer, as in the Taos operating system [6], but they are performed nevertheless):

1. Obtain certificates, verify signatures on certificates and on application request, determine public key of original signer(s).

2. Verify that certificates are unrevoked.

3. Attempt to find "trust path" from trusted certifier to certificate of public key in question.

4. Extract names from certificates.

5. Lookup names in database that maps names to the actions that they are trusted to perform.

6. Determine whether requested action is legal, based on the names extracted from certificates and whether the certification authorities are permitted to authorize such actions according to local policy.

7. Proceed if everything appears valid.

Observe that the final two steps are completely outside the scope of the certification framework and must be reimplemented for each application, despite being central to the problem that certificates are supposed to solve. The problem of reliably mapping names to the actions they are trusted to perform can represent as much of a security risk as the problem of mapping public keys to names, yet the certificates do not help the application map names to actions.

A more general system would integrate the specification of policy with the binding of public keys to the actions they are trusted to perform. That is, we would prefer a system in which the steps above could be reduced to:

1. Obtain certificates, verify signatures on certificates and on application request, determine public key of original signers.

2. Verify that certificates are unrevoked.

3. Submit request, certificates, and description of local policy to local "trust management engine."

4. Proceed if approved.

[2] Proposed future versions of X.509 include provisions for a "policy" attribute. However, the responsibility for interpreting the policy remains outside the scope of the X.509 mechanism [4].

PolicyMaker departs sharply from certificate-based security systems centered on the binding of identities to keys in that it allows requesters of secure services to prove directly that they hold credentials that authorize them to use those services. PolicyMaker binds public keys to predicates that describe the actions that they are trusted to sign for, rather than to the names of the keyholders as in current systems. Using the PolicyMaker language, for example, it is straightforward to authorize a cryptographic key to sign purchase orders for up to \$500 by itself or up to \$5000 when the transaction is countersigned by another authorized key. Considerations such as personal identity and organizational level of the approvers, which are only incidentally relevant to the question the application is trying to answer (whether to accept a purchase order), can be omitted altogther. This ability to express security credentials and policies without requiring the application to manage a mapping between personal identity and authority is especially convenient in systems that include anonymity as a security requirement. For example, an electronic voting system might require a requester to establish that he is a registered voter but might not be *allowed* to learn the requester's personal identity.

The expressiveness and generality that make the PolicyMaker language powerful does not come at the price of insecurity or incomprehensibility: Credentials can be presented by untrusted parties, because they are used by a "safe" interpreter. Moreover, simple policies and credentials can be stated simply, and existing PGP or X.509 certificates that merely bind keys to IDs can be used by PolicyMaker with only trivial modifications.

Trust relationships are also more general and flexible in PolicyMaker than they are in existing systems. Neither the completely anarchic PGP-style web of trust nor the monolithic X.509-style certifying authority trees suffices for many applications that require the use of cryptographic keys. In the PGP system, there is no official mechanism for creating, acquiring, and distributing certificates – one simply must acquire, by whatever *ad hoc* means one can devise, and store on one's key ring any certificates that are needed. If a recipient of a signed message does not have a valid copy of the public-key required to verify the signature, then the signature goes unverified until the recipient can find an introducer who has the certificate. Furthermore, there is no systematic mechanism that allows the sender of a message to know whether a signature will be acceptable to a recipient. This informal introduction mechanism may suffice for personal communication, but it is insufficiently reliable for commerce. On the other hand, the single, global certifying authority tree proposed in the X.509 authentication framework, no matter how reliable, is also insufficient for commerce, because it often forces competing entities to enter into unreasonable trust relationships (albeit possibly indirectly). The PolicyMaker system provides a simple language in which to express conditions under which an individual or an authority is trusted, as well as conditions under which trust may be deferred. Thus, a user may trust certificates signed by CA_1 and CA_2 for small transactions but may insist upon certificates from more reliable CA_3 for large transactions. One user may trust certificates signed directly by CA_1 but not those signed by authorities whom CA_1 trusts, while another user may trust certificates signed by CA_2 if CA_1 trusts CA_2. Similarly, one company may insist that its customers use CA_1's certificates, and another may insist upon CA_2's certificates. There is no assumption of a global tree in which all CAs have a common ancestor.

In addition to providing a richer language for expressing trust relationships, policies, and credentials, PolicyMaker greatly enhances the potential scope and form of security services by implementing trust management in a distinct software system. It frees the designers of such services from the need to handle security completely within applications (as in PGP) or completely within the operating system (as in Taos). It also allows implementations of "standard" security policies and credentials developed for one application to be reused in others.

2. The PolicyMaker Trust Management System

2.1. Architectural Framework

The interface to PolicyMaker reflects our goal of separating generic mechanism (provided by PolicyMaker) from application-specific policy (which is defined by each application). The PolicyMaker service appears to applications very much like a database query engine. PolicyMaker accepts as input a set of local policy statements, a collection of credentials, and a string describing a proposed trusted action. PolicyMaker evaluates proposed actions by interpreting the policy statements and credentials. Depending on the credentials and form of the query, it can return either a simple yes/no answer or additional restrictions that would make the proposed action acceptable. PolicyMaker can either be built into applications (through a linked library) or run as an separate "daemon" service.

In a simple application, certificates (and certificate revocation) will be obtained and managed by the application itself (*e.g.,* in an email system, the sender of a message might include the appropriate certificates in the message itself, which the receiving application would pass directly to PolicyMaker with each query). More complex applications might manage certificates with an external module, whose behavior (e.g., specifying certificate distribution and revocation authorities) might be specified in terms of PolicyMaker certificates. In this paper, however, we focus on the structure and language of the PolicyMaker interpreter itself.

Security policies and credentials are defined in terms of predicates, called *filters*, that are associated with public keys. Filters accept or reject action descriptions based on what the holders of the corresponding secret keys are trusted to do. Security policies and credentials consist of a binding between a filter and one or more public keys. Filters can be written in a variety of interpreted languages and are discussed in detail below. Any public key cryptosystem can be used; signature verification on credentials is handled by external agents (including, for example, PGP).

Trust may also be deferred. Since security information is often not completely defined or available locally, it is frequently necessary to rely on trusted third parties to provide additional security information. A local policy may defer to third parties who are trusted to issue credentials for others, and it may use filters that limit the extent to which these third parties are trusted. These third parties may themselves defer trust if they do not have all of the relevant security information, and they may use filters. Local policies may set bounds on the number of times trust may be deferred. Credentials themselves may also contain filters that limit the actions their holder is trusted to perform. Trust is monotonic; each policy statement or credential can only increase the capabilities granted to others. An action is considered acceptable according to local policy if there is a "chain" (defined in Section 2.3) from the policy to the key(s) requesting the action in which all the filters along the chain are satisfied. This model supports very precise and complex trust relationships, as we shall see below.

PolicyMaker is not tied to any particular notion of security policy or to any particular authentication or signature scheme. The form of action descriptions (called *action strings*) is not determined by or known to the PolicyMaker system itself. It is up to the application to generate and interpret the strings and up to the filters to accept or reject them. Similarly, PolicyMaker does not itself verify the signatures with which action strings are associated, allowing applications to employ virtually any authentication scheme. An application calls PolicyMaker after it has composed an action string and determined the authentication identifier(s) (*e.g.,* PGP public keys) from which the requested action originated. PolicyMaker then determines whether the action string is permitted according to the local security policy and credentials.

2.2. The PolicyMaker Language

The basic function of a PolicyMaker system is to process *queries*. A query is a request to determine whether a particular public key (or a sequence of public keys) is permitted to perform a particular action according to local policy. Queries are of the form

$key_1, key_2, ..., key_n$ **REQUESTS** *ActionString*

Action strings are application-specific messages that describe a trusted action requested by a (sequence of) public key(s). The semantics of action strings are determined by the applications that generate and interpret them and are not part of, or even known to, PolicyMaker. The action strings are interpreted only by the calling applications and might confer such diverse capabilities as signing electronic mail messages that claim to be from a particular individual, entering into contracts on behalf of an organization, logging into a computer system, or watching a pay-per-view movie.

PolicyMaker processes queries based on trust information contained in *assertions*. Assertions confer authority on keys. As discussed in the previous section, each assertion binds a predicate, called a *filter*, to a sequence of public keys, called an *authority structure*. The simplest filters are interpreted programs that can accept or reject action strings. More complex filters, discussed later, can also generate annotations to action strings. Assertions are of the form:

Source **ASSERTS** *AuthorityStruct* **WHERE** *Filter*

Here, *Source* indicates the source of the assertion (either the local policy in the case of policy assertions or the public key of a third party in the case of signed assertions). *AuthorityStruct* specifies the public key or keys to whom the assertion applies. In the simplest case, an authority structure is just a single public key, but more complex authority structures are also possible (such as "at least three of the following eight keys"; how this is done is discussed below). In this way, authority structures serve a purpose in our trust management system that is similar to the one served by "access structures" in a secret-sharing scheme. *Filter* is the predicate that action strings must satisfy for the assertion to hold. In other words, each assertion states that the assertion source trusts the public keys in the authority structure to be associated with action strings that satisfy the filter. (Any filters that apply to the source are recursively applied as well; we discuss query semantics in detail in the next section.)

There are two types of PolicyMaker assertions: *certificates* and *policies*. A certificate (also called a *signed assertion*) is a signed message that binds a particular authority structure to a filter. A policy also binds a particular authority structure to a filter. Policies, however, are not signed; instead, because they originate locally, they are unconditionally accepted locally. They are, semantically and syntactically, really just a special case of certificates. On any given system, the set of local policies forms the "trust root" of the machine and defines the context under which all queries are evaluated.

Authority structures are specified as filters that accept or reject a list of one or more public keys associated with an action string. The simplest authority structure matches exactly one key, but it is also possible to construct filters that implement complex requirements, such as k-out-of-n threshold schemes.

A more precise syntax is given in the Appendix.

2.3. Query semantics

A query is a request for information about the trust that can be placed in a particular (sequence of) public key(s). A PolicyMaker system must have at least one policy assertion before it can process queries. Typically, there will be several fixed policy assertions and a collection of signed assertions pertaining to the query at hand.

Recall that queries contain one or more keys and an action string and that assertions contain a source, an authority structure, and a filter. From a semantic point of view, the simplest case is that of a query with one key k and a set of assertions (both policies and certificates) in which all authority structures are just single keys. In this case, we may interpret the assertions as a directed graph D in which the vertices are labeled by keys or policy sources and the arcs are labeled by filters; if $v \rightarrow w$ is an arc in D that is labeled by f, then there must be an assertion whose source is the label of v, whose authority structure is the label of w, and whose filter is f. To process a query, the PolicyMaker system must find a *chain* $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_t$ in D in which v_1 is a local policy source and $v_t = k$. If the query contains multiple keys $k_1, \ldots, k_n$ and the assertions contain complex authority structures, then $V(D)$ must include nodes that are labeled by keys, policy sources, and complex authority structures, and the chain $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_t$ must be such that v_t is labeled by an authority structure that accepts the input $(k_1, \ldots, k_n)$.

The filters in certificate and policy assertions may take one of two forms. The simplest form is a program that simply accepts or rejects action strings. A query is then satisfied if the digraph given by these assertions contains a chain in which *all* of the filters accept the action string.

The second filter form not only accepts or rejects action strings but may also append *annotations* to an otherwise acceptable action string that indicate restrictions or information not present in the original query. Such assertions allow the querying application to generate a description of the trust capabilities of a key without needing to "probe" PolicyMaker with specific queries. For example, in an electronic mail system, a certificate for an organizational certifying authority might generate an "organization:" line to be added to action strings where one is not already present. Such assertions contain two filter programs: a *predicate*, which behaves exactly as the filters we have already described, and an *annotator*. Queries involving assertions that contain annotators are evaluated in two phases. Annotators behave just like predicates but are also able to emit an *annotation* that is appended to the action string. In the first pass, the action string is passed to each annotator along the chain, from policy to query, possibly acquiring more annotations. If all annotators are satisfied, we run a second pass with the fully annotated action string through the predicates in the chain. The predicates can either accept or reject but cannot add further annotations. This ensures that any annotations in the previous phase are acceptable to all certificates in the chain. If all predicates are satisfied, the annotated action string is returned to the application.

2.4. Signature Schemes and Filter Languages

PolicyMaker does not itself verify signatures on signed assertions or queries or even process the original signed messages. Instead, signatures are verified by some external program or function (*e.g.*, PGP, PEM, etc.). The external program guarantees that the signature was valid for the identified public key. The public key passed to the PolicyMaker interpreter identifies the program and the key (*e.g.*, "`PGP:0x01234567abcdefa0b1c2d3e4f5a6b7`"). By not interpreting signatures itself or insisting on a particular signature scheme or format, PolicyMaker makes it very easy to implement a certification authority that exploits existing infrastructure. For example, it is possible to have chains of trust that consist of a mixture of X.509 certificates (interpreted by a program that converts them into PolicyMaker certificates) and certificates consisting of simple text messages signed with a program such as PGP.

Similarly, PolicyMaker filters are interpreted programs that are run within a "safe" (I/O and resource limited) wrapper. Our implementation currently supports three filter languages: a regular expression system (similar to those used in Unix), an internally developed safe version of AWK [1], which we call AWKWARD, and a macro language that preprocesses into safe AWK. Other "safe" languages, such as Java [2] or Safe-TCL [5], are easily added as desired.

In general, any language that can be safely interpreted can be used as a filter language. A distinguishing feature of our system is that filters are allowed the full complexity and expressiveness of general programs. Designing and implementing a safe filter language is a much simpler task than designing a general-purpose language for remote agents (like Java), however, because filters generally have no need to issue "dangerous" system calls. (There is no need to open files or interact with the network, for example.) PolicyMaker wraps the filter-language interpreter in a resource-limited shell that prevents, *e.g.*, infinite loops in filters from consuming the entire host CPU. Most filters can be assumed to be very simple, so their resource allocation can be modest.

Input to filters consists of the current action string and an "environment" containing information about the current context (*e.g.*, date, time, application name, etc.). A filter can use the environment to enforce contextual constraints such as expiration times. A filter also has access to information about the rest of the chain in which it is being evaluated, which makes it possible to design certificates that limit the degree to which their authority can be deferred.

Although the interpreter for a filter language is external to PolicyMaker itself, the name of the language is given in assertions and must be known by anyone who needs to use the assertion. New languages can be added easily as needed, provided that all recipients of certificates using the new language are configured to interpret them. PolicyMaker ignores certificates written in unknown or unsupported filter languages.

3. Application Examples

Because the responsibility for defining and interpreting action strings rests entirely with the application, the most important consideration in integrating PolicyMaker into applications is identifying an appropriate "trust language" that captures the required security semantics. An application's action string language should be chosen so that it can be easily generated and acted upon by the application and so that recognizers can be easily programmed into policies and certificates. In general, it should be possible for a person who understands the application to examine an action string and understand what it does and to examine assertions and understand what kinds of actions satisfy them. In the sections that follow, we give (informal) examples of trust languages and assertions that show how PolicyMaker might be integrated into various applications.

3.1. Email system

Here, we propose an electronic mail system for the Internet, in which the security policy requires that we establish the identities of parties to messages. A natural, if somewhat simplistic, language for describing the trust properties of messages in such a system derives from the mail delivery headers used to route the message from sender to recipient. For example, headers containing the lines

```
From: Alice
Organization: Bob Labs
```

indicate that the message originated from an individual named "Alice" who is affiliated with "Bob Labs." The security policy in such a system aims to ensure that the headers displayed to the user along with each message are correct, based on certificates from (locally chosen) trusted authorities. It is easy to imagine a language of action strings for such a system:

```
From:  sender's name
Organization:  sender's organization
```

Given such a language, it is also easy to construct a policy that binds Alice's PGP public key to the ability to sign messages that claim to originate with Alice:

```
policy ASSERTS
  pgp:"0xf0012203a4b51677d8090aabb3cdd9e2f"
  WHERE PREDICATE=regexp:"(From: Alice) &&
    (Organization: Bob Labs)" ;
```

Because the policy is very simple and can be recognized by simple pattern matching, we used a regular expression for the filter. The expression simply checks that the expected fields are present and contain only the expected information. (We might have also added an ANNOTATOR filter that contains a simple AWK program that fills in any missing fields, but we did not in this example.) Note that, with this policy, the authority to certify identity is not deferred to a third party; the trust in Alice's key is embedded directly from the policy.

In most cases, we would prefer a level of indirection. For example, we might trust the public key belonging to Bob (the president of Bob Labs) to tell us which public keys belong to his employees. Here, local policy would associate his public key with a predicate that checks only that the "`Organization:`" field indeed says "`Bob Labs`". Bob can sign predicates on behalf of his employees that check their names in the "`From:`" field. These signed predicates are roughly analogous to the "certificates" of the X.509 system; Bob is trusted as the "certificate authority" but only with respect to his own employees.

It is simple to set a policy to trust Bob (whose key is `pgp:"0x01234567abcdefa0b1c2d3e4f5a6b7"`) in this role:

```
policy ASSERTS
 pgp:"0x01234567abcdefa0b1c2d3e4f5a6b7"
 WHERE
   PREDICATE=regexp:"Organization: Bob Labs";
```

This policy allows us to trust certificates from Bob (which are just messages signed with his PGP key):

```
pgp:"0x01234567abcdefa0b1c2d3e4f5a6b7"
 ASSERTS
   pgp:"0xf0012203a4b51677d8090aabb3cdd9e2f"
    WHERE PREDICATE=regexp:"From: Alice";
```

Together, this certificate and the deferring policy have the same meaning as the non-deferring policy given above. The query:

```
pgp:"0xf0012203a4b51677d8090aabb3cdd9e2f"
  REQUESTS "From: Alice
            Organization: Bob Labs" ;
```

would succeed, but

```
pgp:"0xf0012203a4b51677d8090aabb3cdd9e2f"
 REQUESTS "From: Alice
           Organization: Matt Labs" ;
```

and

```
pgp:"0xf0012203a4b51677d8090aabb3cdd9e2f"
 REQUESTS "From: John
           Organization: Bob Labs" ;
```

would both fail.

The preceding discussion shows how to use PolicyMaker to support authenticity of email messages. Note that it can also support privacy of messages. By querying a PolicyMaker daemon, a sender can obtain the appropriate key with which to encrypt a message, as well as the information about the recipient's security policy needed to prepare the outgoing message.

3.2. A Certificate Revocation Server

PolicyMaker does not itself implement certificate distribution or revocation services (*i.e.*, there are no "certificate revocation lists" built into the system). However, PolicyMaker can be used to specify services in which arbitrary certificate revocation, distribution, and freshness constraints are built into the policy. It is possible, for example, for a certificate issuer to specify authority structures that include not only the public key being certified but the public key of a certificate revocation service as well. The revocation service issues frequently broadcast certificates that contain predicates that are only satisfied by non-revoked keys. Other constraints, such as the required "freshness" of the revocation certificates, can also be specified and may be based on application-based criteria (*e.g.*, high-valued transactions require more recently issued assurance that the certificates are not revoked).

Detailed example is omitted for brevity.

3.3. X.509/PGP Certificates as PolicyMaker Assertions

It is possible to exploit existing certification infrastructure and still use the policy specification mechanisms of PolicyMaker. For example, if the majority of certificates in a system are still in X.509 or PGP format, it is simple to write an application-specific program that converts these certificates into PolicyMaker assertions. The predicate for such an assertion would include routines that check the application's action string language for, *e.g.*, the correct identity.

3.4. A Simple Workflow System

A company policy regarding signatures on contracts, checks, bids, purchase orders, etc. might require k out of n people to sign for the company. The signers have to be identifiable as legitimate co-signers for the various applications and corresponding qualifiers (such as dollar amount, liability, control, etc). The typical co-signature is a two-out-of-n protocol. In other cases, perhaps to recover an escrowed key that has been covered using a k-out-of-n threshold scheme, k signatures are needed.

In the following example, the company policy is that, for the purchase-order department to process purchase orders for amounts less than or equal to \$1,000,000, submitted orders must have the signatures of at least three of the company directors. The purchase-order application then requires that the amount, the organization, and the signers' names be present before it will verify that an order has been appropriately constructed.

The top level policy states that the security certificate authority key is allowed to create certificates in the name of the company. (in the interest of readablity in the examples that follow, we use labels like "`Security_CA`" in place of the actual public keys).

```
POLICY ASSERTS Security_CA WHERE
  PREDICATE=regexp:"Organization: Bob Labs";
```

This Security_CA then creates a certificate that states that any group of at least three legitimate corporate directors can sign purchase orders for ammounts not exceeding $1,000,000.

```
Security_CA ASSERTS
   "an AWKWARD program requiring at least
     3 keys directly certified by PersonnelKey"
  WHERE PREDICATE="an AWKWARD program
    that checks that Amount <= $1,000,000";
```

The PersonnelKey signs keys for the four directors, Jack, Joan, Matt, and Alice, certifying their names:

```
PersonnelKey ASSERTS JackKey WHERE
   PREDICATE=regexp:"Signer's Name: Jack";

PersonnelKey ASSERTS JoanKey WHERE
   PREDICATE=regexp:"Signer's Name: Joan";

PersonnelKey ASSERTS MattKey WHERE
   PREDICATE=regexp:"Signer's Name: Matt";

PersonnelKey ASSERTS AliceKey WHERE
   PREDICATE=regexp:"Signer's Name: Alice";
```

When the purchase-order application receives a PO of the form:

```
...
PO Amount: $800,000
Signed By:
    Jack, Joan, Matt
```

it checks the digital signatures and generates an appropriate action string with which it queries PolicyMaker:

```
{JackKey, JoanKey, MattKey}
    REQUESTS "PO amount = $800,000
              Organization: Bob Labs
              Signer's Name: Jack
              Signer's Name: Joan
              Signer's Name: Matt";
```

4. Current Status and Future Directions

We have implemented a prototype PolicyMaker interpreter that includes a built-in regular expression and AWKWARD interpreter. External programs recognize DSA and PGP-signed PolicyMaker assertions and can convert X.509 and PGP certificate formats into PolicyMaker assertions for a simple Internet email application. Performance is reasonable, with chains several nodes long evaluated in much less time than is required to verify the signatures on the certificates.

The PolicyMaker approach has a number of advantages compared with the traditional, *ad hoc* trust management approaches forced by such systems as X.509 and PGP. First, because certificates and policies are based on predicates written in a general programming language, the trust language for an application domain can be as simple or as complex as required without changing the trust management system itself or the interface to it. Second, trust descriptions can be in whatever form naturally occurs in the application and can be changed without altering the trust management system. In an email system, they might consist of messages header lines. In a system for signing and approving contracts, they might consist of strings indicating the amounts and types of expenditures. Third, applications that use PolicyMaker to implement trust management may be more secure, since the risks arising from one level of indirection (mapping of identities to their authority) are eliminated. Finally, responsibility is separated in a natural way. Applications are responsible for describing trusted actions and taking appropriate actions based on correct descriptions. Certificates and policies are responsible for describing who is trusted to perform actions according to the descriptions. The trust management system (PolicyMaker) is responsible for ensuring that described actions actually conform to the policies and certificates but need not actually understand them.

Of course, PolicyMaker does not solve the entire trust management problem or guarantee that systems that use it will be secure. Applications must define action description languages that accurately reflect the security semantics of the application. The predicates in policy and certificate assertions must be carefully written to reflect the intentions of the policy. Because there are few restrictions on predicates, it is possible to construct policies that have unfortunate or unexpected consequences. However, because PolicyMaker's trust management functions are encapsulated in only a few components (the certificates, policies, and application's action string management functions), it is probably easier to verify or at least debug security properties of systems than it is in traditional approaches in which trust management is spread across the entire system.

Our near-term plan for PolicyMaker has two important components. On one front, we plan to develop a formal model of trust management in which to investigate the power and limitations of the PolicyMaker approach in a mathematically rigorous manner. Simultaneously, we plan to experiment with our prototype implementation of the PolicyMaker system in diverse applications contexts. We also believe PolicyMaker will make a good framework for certificate distribution (*e.g.*, a certification server could provide appropriate certificates based on provided policy assertions). Both formal and experimental results will guide the development of future versions of the system.

In conclusion, PolicyMaker introduces the general "trust management layer" at what appears to be the right level of abstraction. An important benefit is that, by exposing a trust management interface, it requires designers and implementers of secure services and systems to consider trust management explicitly. It encourages the use of sophisticated notions of security when appropriate to the context, and it enforces the necessary coordination of the design of policy, credentials, and trust relationships.

5. Acknowledgements

We are grateful to Brian Kernighan for implementing AWKWARD and to Mike Reiter and Ron Rivest for their helpful comments on earlier drafts.

References

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, 1988.

[2] J. Gosling and H. McGilton, *The Java Language Environment, A White Paper*, Sun Microsystems, Inc., Mountain View, 1995.

[3] *Information Technology – Open Systems Interconnection – The Directory:Authentication Framework*, Recommendation X.509, ISO/IEC 9594-8.

[4] International Telegraph and Telephone Consultative Committee (CCITT). *The Directory – Authentication Framework, Recommendation X.509* 1993 update.

[5] John K. Ousterhout, *TCL and the TK Toolkit*, Addison-Wesley, Reading, 1994.

[6] E. Wobber and M. Abadi and M. Burrows and B. Lampson, "Authentication in the Taos Operating System," *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[7] P. Zimmermann, *PGP User's Guide*, MIT Press, Cambridge, 1994.

Appendix: PolicyMaker Syntax

The basic grammar accepted by the PolicyMaker interpreter follows. Terminals are given in `UPPERCASE`.

```
assertion : source ASSERTS
       authstruct WHERE
       filterlist SEMICOLON
   | source ASSERTS
       authstruct SEMICOLON

query : keylist REQUESTS
     string condition SEMICOLON

source : keyid
   | POLICY

keyid : system COLON string

authstruct : filterprog
   | keyid

filterlist : filtername EQUALS
             filterprog
   | filtername EQUALS
     filterprog COMMA filterlist

filtername : ANNOTATOR
   | PREDICATE
   | COMMENTARY
   | APPLICATION

filterprog : language
             COLON string

language : AWKWARD
   | REGEXP

keylist : keyid
   | keylist COMMA keyid

condition : <nullstring>
   | WHERE filterprog
```