1

Distributed Trust

John Ioannidis, *AT&T Labs – Research*

Angelos D. Keromytis, *Columbia University*

CONTENTS

1.1 Access Control and Trust Management 2
1.2 Technical Foundations 3
1.2.1 Authentication 3
1.2.2 Public Key Certificates 3
1.3 Distributed Trust Management 4
1.3.1 PolicyMaker 6
1.3.2 KeyNote 9
1.4 Applications of Trust Management Systems 11
1.4.1 Network-layer Access Control 11
1.4.2 Distributed Firewalls and the STRONGMAN Architecture 12
1.4.3 Grid Computing and Transferable Micropayments 13
1.4.4 Micropayments: Microchecks and Fileteller 13
1.4.5 Active Networking 13
1.5 Other Trust-Based Systems 14
1.6 Closing Remarks 15
1.7 Acknowledgements 15

Abstract

This chapter explores the concept of trust management in access control. We introduce the concepts behind trust management and discuss two such systems. The first, PolicyMaker [Blaze et al., 1996], first introduced the concepts of trust management, which were further explored in the work on the KeyNote credential language [Blaze et al., 1999b]. We discuss some applications of trust management systems, as well as other related work. Our focus is on the concepts and design, rather than the details of particular approaches or mechanisms. Our goal is to impart enough information to the readers to make informed decisions as to how best to use the power and expressiveness of trust management.

X-XXXX-XXXX-X/XX/$X.XX+$X.XX

1.1 Access Control and Trust Management

Authorization and access control is the process by which a security enforcement point determines whether an entity should be allowed to perform a certain action. Authorization takes place after said entity has been authenticated. Furthermore, authorization occurs within the scope of an access control policy. In simpler terms, the first step in making an access control decision is determining who is making a request; the second step is determining, based on the result of the authentication as well as additional information (the access control policy), whether that request should be allowed.

One security mechanism often used in operating systems is the Access Control List (ACL). Briefly, an ACL is a list describing which access rights a principal has on an object (resource). For example, an entry might read "User Foo can Read File Bar." Such a list (or table) need not physically exist in one location but may be distributed throughout the system. The $Unix^{TM}$-filesystem "permissions" mechanism is essentially an ACL.

ACLs have been used in distributed systems, because they are conceptually easy to grasp and because there is an extensive literature about them. However, there are a number of fundamental reasons that ACLs are inadequate for distributed-system security, *e.g.*,

- *Authentication:* In an operating system, the identity of a principal is well known. This is not so in a distributed system, where some form of authentication has to be performed before the decision to grant access can be made.

- *Delegation* is necessary for scalability of a distributed system. It enables *decentralization* of administrative tasks. Existing distributed-system security mechanisms usually delegate directly to a "certified entity." In such systems, policy (or authorizations) may only be specified at the last step in the delegation chain (the entity enforcing policy), most commonly in the form of an ACL. The implication is that high-level administrative authorities cannot directly specify overall security policy; rather, all they can do is "certify" lower-level authorities. This authorization structure leads easily to inconsistencies among locally-specified sub-policies.

- *Expressibility and Extensibility:* A generic security mechanism must be able to handle new and diverse conditions and restrictions. The traditional ACL approach has not provided sufficient expressibility or extensibility. Thus, many security policy elements that are not directly expressible in ACL form must be hard-coded into applications. This means that changes in security policy often require reconfiguration, rebuilding, or even rewriting of applications.

- *Local trust policy:* The number of administrative entities in a distributed system can be quite large. Each of these entities may have a different trust model for different users and other entities. For example, system A may trust system B to authenticate its users correctly, but not system C; on the other hand, system B may trust system C. It follows that the security mechanism should not enforce uniform and implicit policies and trust relations.

The *trust-management approach* to distributed-system security was developed as an answer to the inadequacy of traditional authorization mechanisms. Trust-management engines avoid the need to resolve "identities" in an authorization decision. Instead, they express privileges and restrictions in a programming language. This allows for increased flexibility and expressibility, as well as standardization of modern, scalable security mechanisms. Further advantages of the trust-management approach include proofs that requested transactions comply with local policies and system architectures that encourage developers and administrators to consider an application's security policy carefully and specify it explicitly.

Section 1 provides some background material on the concepts of authentication and public key certificates, which form the basis on which trust management systems are built. Section 1.2.2 describes the trust-management approach to authorization and access control, focusing on the Policy-

Maker and KeyNote systems. Section 1.3.2 describes various applications of trust management systems (especially KeyNote), while Section 1.4.5 briefly describes some related work.

1.2 Technical Foundations

1.2.1 Authentication

The term authentication in network security and cryptography is used to indicate the process by which one entity convinces another that it has possession of some secret information, for the purpose of "identifying" itself. This identification does not necessarily correspond to a real-world entity; rather, it implies the continuity of a relationship (or, as stated in [Schneier, 2000], knowing who to trust and who not to trust).

In computer networks, strong authentication is achieved through cryptographic protocols and algorithms. In particular, public key cryptography forms the basis for many protocols and proposals for scalable network-based authentication. A discussion of the various constraints and goals in these protocols is beyond the scope of this chapter. The interested reader is referred to [Schneier, 1996].

1.2.2 Public Key Certificates

Public key certificates are statements made by a principal (an entity, such as a user or a process acting on behalf of a user, that can undertake an action in the system and is identified by a cryptographic public key) about another principal (also identified by a public key). Public key certificates are cryptographically signed, such that anyone can verify their integrity (the fact that they have not been modified since the signature was created). Public key certificates are utilized in authentication because of their natural ability to express delegation (more on this in Section 1.3.1).

A traditional public key certificate cryptographically binds an identity to a public key. In the case of the X.509 standard [CCITT, 1989], an identity is represented as a "distinguished name," *e.g.,*

```
/C=US/ST=PA/L=Philadelphia/O=University of Pennsylvania/
OU=Department of Computer and Information Science/CN=Jonathan M. Smith
```

In more recent public key certificate schemes [Ellison et al., 1999; Blaze et al., 1999b] the identity is the public key, and the binding is between the key and the permissions granted to it. Public key certificates also contain expiration and revocation information.

Revoking a public key certificate means notifying entities that might try to use it that the information contained in it is no longer valid, even though the certificate itself has not expired. Possible reasons for this include theft of the private key used to sign the certificate (in which case, all certificates signed by that key need to be revoked), or discovery that the information contained in the certificate has become inaccurate. This happened when Verisign, a commercial Certification Authority (CA), mistakenly issued an X.509 certificate to an unknown person with the common name "Microsoft Corporation." There exist various revocation methods (Certificate Revocation Lists (CRLs), Delta-CRLs, Online Certificate Status Protocol (OCSP), refresher certificates), each with its own tradeoffs in terms of the amount of data that needs to be kept around and transmitted, any online availability requirements, and the window of vulnerability.

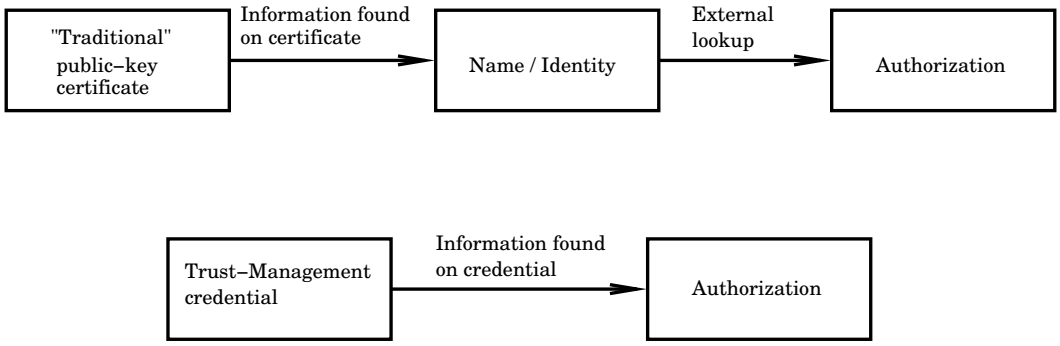

Figure 1.1: The difference between access control using traditional public-key certificates and trust management.

1.3 Distributed Trust Management

A traditional "system-security approach" to the processing of a signed request for action (such as access to a controlled resource) treats the task as a combination of *authentication* and *authorization*. The receiving system first determines *who* signed the request and then queries an internal database to decide *whether* the signer should be granted access to the resources needed to perform the requested action. It has been argued that this is the wrong approach for today's dynamic, internetworked world [Blaze et al., 1996, 1999a; Ellison, 1999; Ellison et al., 1999]. In a large, heterogeneous, distributed system, there is a huge set of people (and other entities) who may make requests, as well as a huge set of requests that may be made. These sets change often and cannot be known in advance. Even if the question "who signed this request?" could be answered reliably, it would not help in deciding whether or not to take the requested action if the requester is someone or something from whom the recipient is hearing for the first time.

The right question in a far-flung, rapidly changing network becomes "is the key that signed this request *authorized* to take this action?" Traditional name-key mappings and pre-computed access-control matrices are inadequate. The former because they do not convey any access control information, the latter because of the amount of state required: given N users, M objects to which access needs to be restricted, X variables which need to be considered when making an access control decision. We would need access control lists of minimum size $N \times X$ associated with each object, for a total of $N \times M$ policy rules of size X in our system. As the conditions under which access is allowed or denied become more refined (and thus larger), these products increase. In typical systems, the number of users and objects (services) is large, whereas the number of variables is small; however, the combinations of variables in expressing access control policy can be non-trivial (and arbitrarily large, in the worst case). Furthermore, these rules have to be maintained, securely distributed, and stored across the entire network. Thus, one needs a more flexible, more "distributed" approach to authorization.

The *trust-management approach*, initiated by Blaze *et al.* [1996], frames the question as follows: "Does the set C of *credentials* prove that the *request* r *complies* with the local security *policy* P?" This difference is shown graphically in Figure 1.1.

Each entity that receives requests must have a policy that serves as the ultimate source of authority in the local environment. The entity's policy may directly authorize certain keys to take certain actions, but more typically it will *delegate* this responsibility to credential issuers that it trusts to have the required domain expertise as well as relationships with potential requesters. The *trust-management engine* is a separate system component that takes (r, C, P) as input, outputs a decision about whether compliance with the policy has been proven, and may also output some additional

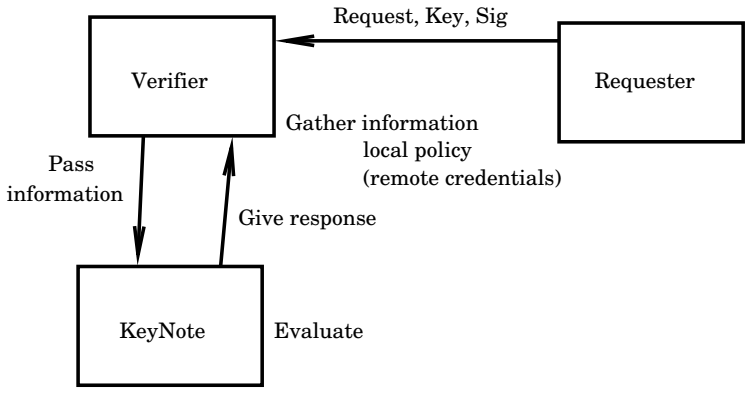

Figure 1.2: Interaction between an application and a trust-management system.

information about how to proceed if the required proof has not been achieved. Figure 1.2 shows an example of the interactions between an application and a trust-management system.

An essential part of the trust-management approach is the use of a *general-purpose, application independent* algorithm for checking proofs of compliance. Why is this a good idea? Any product or service that requires some form of proof that requested transactions comply with policies, could use a special-purpose algorithm or language implemented from scratch. Such algorithms/languages could be made more expressive and tuned to the particular intricacies of the application. Compared to this, the trust-management approach offers two main advantages.

The first is simply one of engineering: it is preferable (in terms of simplicity and code reuse) to have a "standard" library or module, and a consistent API, that can be used in a variety of different applications.

The second, and perhaps most important gain is in soundness and reliability of both the definition and the implementation of "proof of compliance." Developers who set out to implement a "simple," special-purpose compliance checker (in order to avoid what they think are the overly "complicated" syntax and semantics of a universal "meta-policy") discover that they have underestimated their application's need for proof and expressiveness. As they discover the full extent of their requirements, they may ultimately wind up implementing a system that is as general and expressive as the "complicated" one they set out to avoid. A general-purpose compliance checker can be explained, formalized, proven correct, and implemented in a standard package, and applications that use it can be assured that the answer returned for any given input (r, C, P) depends only on the input and *not* on any implicit policy decisions (or bugs) in the design or implementation of the compliance checker.

Basic questions that must be answered in the design of a trust-management engine include:

- How should "proof of compliance" be defined?
- Should policies and credentials be fully or only partially programmable? In which language or notation should they be expressed?
- How should responsibility be divided between the trust-management engine and the calling application? For example, which of these two components should perform the cryptographic signature verification? Should the application fetch all credentials needed for the compliance proof before the trust-management engine is invoked, or may the trust-management engine fetch additional credentials while it is constructing a proof?

At a high level of abstraction, trust-management systems have five components:

- A language for describing *actions*, which are operations with security consequences that are to be controlled by the system.
- A mechanism for identifying *principals*, which are entities that can be authorized to perform actions.
- A language for specifying application *policies*, which govern the actions that principals are authorized to perform.
- A language for specifying *credentials*, which allow principals to delegate authorization to other principals.
- A *compliance checker*, which provides a service to applications for determining how an action requested by principals should be handled, given a policy and a set of credentials.

By design, trust management unifies the notions of security policy, credentials, access control, and authorization. An application that uses a trust-management system can simply ask the compliance checker whether a requested action should be allowed. Furthermore, policies and credentials are written in standard languages that are shared by all trust-managed applications; the security configuration mechanism for one application carries exactly the same syntactic and semantic structure as that of another, even when the semantics of the applications themselves are quite different.

1.3.1 PolicyMaker

PolicyMaker was the first example of a "trust-management engine." That is, it was the first tool for processing signed requests that embodied the "trust-management" principles articulated in Section 1.2.2. It addressed the authorization problem directly, rather than handling the problem indirectly via authentication and access control, and it provided an application-independent definition of "proof of compliance" for matching up requests, credentials, and policies. PolicyMaker was introduced in the original trust-management paper by Blaze *et al.* [1996], and its compliance-checking algorithm was later fleshed out in [Blaze et al., 1998]. A full description of the system can be found in [Blaze et al., 1996, 1998], and experience using it in several applications is reported in [Blaze et al., 1997; Lacy et al., 1997].

PolicyMaker credentials and policies (collectively referred to as "assertions") are programmable: they are represented as pairs (f, s), where s is the *source* of authority, and f is a program describing the nature of the authority being granted as well as the party or parties to whom it is being granted. In a policy assertion, the source is always the keyword **POLICY**. For the PolicyMaker trust-management engine to be able to make a decision about a requested action, the input supplied to it by the calling application must contain one or more policy assertions; these form the "trust root," *i.e.*, the ultimate source of authority for the decision about this request, as shown in Figure 1.3. In a credential assertion, the source of authority is the public key of the issuing entity. Credentials must be signed by their issuers, and these signatures must be verified before the credentials can be used.

PolicyMaker assertions can be written in any programming language that can be "safely" interpreted by a local environment that has to import credentials from diverse (and possibly untrusted) issuing authorities. A version of AWK without file I/O operations and program execution time limits (to avoid denial of service attacks on the policy system) was developed for early experimental work on PolicyMaker (see [Blaze et al., 1996]), because AWK's pattern-matching constructs are a convenient way to express authorizations. For a credential assertion issued by a particular authority to be useful in a proof that a request complies with a policy, the recipient of the request must have an interpreter for the language in which the assertion is written (so that the program contained in the assertion can be executed). Thus, it would be desirable for assertion writers ultimately to converge on a small number of assertion languages so that receiving systems have to support only a small number of interpreters and so that carefully crafted credentials can be widely used. However, the

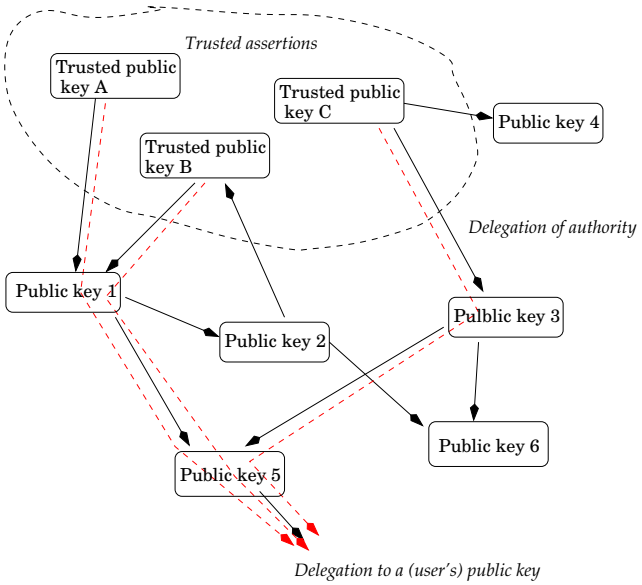

Figure 1.3: Delegation in PolicyMaker, starting from a set of trusted assertions. The dotted lines indicate a delegation path from a trusted assertion (public key) to the user making a request. If all the assertions along that path authorize the request, it will be granted.

question of which languages these will be was left open by the PolicyMaker project. A positive aspect of PolicyMaker's not insisting on a particular assertion language is that all of that work that has gone into designing, analyzing, and implementing the PolicyMaker compliance-checking algorithm will not have to be redone every time an assertion language is changed or a new language is introduced. The "proof of compliance" and "assertion-language design" problems are orthogonal in PolicyMaker and can be worked on independently.

One goal of the PolicyMaker project was to make the trust-management engine as small as possible and analyzable. Architectural boundaries were drawn so that a fair amount of responsibility was placed on the calling application rather than the trust-management engine. In particular, the calling application was made responsible for all cryptographic verification of signatures on credentials and requests. One pleasant consequence of this design decision is that the application developer's choice of signature scheme(s) can be made independently of his choice of whether or not to use PolicyMaker for compliance checking. Another important responsibility that was assigned to the calling application is credential gathering. The input (r, C, P) supplied to the trust-management module is treated as a claim that credential set C contains a proof that request r complies with Policy P. The trust-management module is *not* expected to be able to discover that C is missing just one credential needed to complete the proof and to go fetch that credential from *e.g.*, the corporate database, the issuer's web site, the requester himself, or elsewhere. Later trust-management engines, including KeyNote [Blaze et al., 1999b] and REFEREE [Chu et al., 1997] divide responsibility between the calling application and the trust-management engine differently than the way PolicyMaker divides it.

The main technical contribution of the PolicyMaker project is a notion of "proof of compliance" that is fully specified and analyzed. We give an overview of PolicyMaker's approach to compliance checking here; a complete treatment of the compliance checker can be found in [Blaze et al., 1998].

The PolicyMaker runtime system provides an environment in which the policy and credential assertions fed to it by the calling application can cooperate to produce a proof that the request complies

with the policy (or can fail to produce such a proof). Among the requirements for this cooperation are a method of inter-assertion communication and a method for determining that assertions have collectively succeeded or failed to produce a proof.

Inter-assertion communication in PolicyMaker is done via a simple, append-only data structure on which all participating assertions record intermediate results. Specifically, PolicyMaker initializes the proof process by creating a "blackboard" containing only the request string r and the fact that no assertions have thus far approved the request or anything else. Then PolicyMaker runs the various assertions, possibly multiple times each. When assertion (f_i, s_i) is run, it reads the contents of the blackboard and then adds to the blackboard one or more *acceptance records* (i, s_i, R_{ij}). Here R_{ij} is an application-specific action that source s_i approves, based on the partial proof that has been constructed thus far. R_{ij} may be the input request r, or it may be some related action that this application uses for inter-assertion communication. Note that the meanings of the action strings R_{ij} are understood by the application-specific assertion programs f_i, but they are not understood by PolicyMaker. All PolicyMaker does is run the assertions and maintain the global blackboard, making sure that the assertions do not erase acceptance records previously written by other assertions, fill up the entire blackboard so that no other assertions can write, or exhibit any other non-cooperative behavior. PolicyMaker never tries to interpret the action strings R_{ij}.

A proof of compliance is achieved if, after PolicyMaker has finished running assertions, the blackboard contains an acceptance record indicating that a policy assertion approves the request r. For example, consider the following two assertions, the first of which is a policy and the second a credential:

```
Source: POLICY
if (has_posted(KEY_ALICE) == "accept") post("accept");

Source: KEY_ALICE
if (action_string("file") == "/home/angelos/html/index.html" &&
    action_string("operation") == "read") post("accept");
Signature: ...
```

The policy assertion will approve a request if an assertion that has been signed by KEY_ALICE approves it. The second assertion will approve any "read" request to the given file. The function *action_string()* is used to read the action strings, *post()* is used to issue assertion acceptance records, and *has_posted()* reads acceptance records posted on the board.

Among the nontrivial decisions that PolicyMaker must make are (1) in what order assertions should be run, (2) how many times each assertion should be run, and (3) when an assertion should be discarded because it is behaving in a non-cooperative fashion. Blaze *et al.* [1998] provide:

- A mathematically precise formulation of the PolicyMaker compliance-checking problem.
- Proof that the problem is undecidable in general and is NP-hard even in certain natural special cases.
- One special case of the problem that is solvable in polynomial-time, is useful in a wide variety of applications, and is implemented in the current version of PolicyMaker, as described below.

Although the most general version of the compliance-checking problem allows assertions to be arbitrary functions, the computationally tractable version that is analyzed in [Blaze et al., 1998] and implemented in PolicyMaker is guaranteed to be correct only when all assertions are monotonic. (Basically, if a monotonic assertion approves action a when given evidence set E, then it will also approve action a when given an evidence set that contains E; see [Blaze et al., 1998] for a formal definition.) In particular, correctness is guaranteed only for monotonic *policy* assertions, and this excludes certain types of policies that are used in practice, most notably those that make explicit

use of "negative credentials" such as revocation lists. Although it is a limitation, the monotonicity requirement has certain advantages. One of them is that, although the compliance checker may not handle all potentially desirable policies, it is at least analyzable and provably correct on a well-defined class of policies. Furthermore, the requirements of many non-monotonic policies can often be achieved by monotonic policies. For example, the effect of requiring that an entity *not* occur on a revocation list can also be achieved by requiring that it present a "certificate of non-revocation"; the choice between these two approaches involves trade-offs among the (system-wide) costs of the two kinds of credentials and the benefits of a standard compliance checker with provable properties. Finally, restriction to monotonic assertions encourages a conservative, prudent approach to security: In order to perform a potentially dangerous action, a user must present an adequate set of affirmative credentials; no potentially dangerous action is allowed "by default," simply because of the absence of negative credentials.

1.3.2 KeyNote

KeyNote [Blaze et al., 1999b] was designed according to the same principles as PolicyMaker, using credentials that directly authorize actions instead of dividing the authorization task into authentication and access control. Two additional design goals for KeyNote were standardization and ease of integration into applications. To address these goals, KeyNote assigns more responsibility to the trust-management engine than PolicyMaker does and less to the calling application; for example, cryptographic signature verification is done by the trust-management engine in KeyNote and by the application in PolicyMaker. KeyNote also requires that credentials and policies be written in a specific assertion language, designed to work smoothly with KeyNote's compliance checker. By fixing a specific assertion language that is flexible enough to handle the security policy needs of different applications, KeyNote goes further than PolicyMaker toward facilitating efficiency, interoperability, and widespread use of carefully written credentials and policies, at the cost of reduced expressibility and interaction between different policies (compared to PolicyMaker).

A calling application passes to a KeyNote evaluator a list of credentials, policies, and requester public keys, and an "Action Attribute Set." This last element consists of a list of attribute/value pairs, similar in some ways to the $Unix^{TM}$ shell environment (described in the *environ(7)* manual page of most Unix installations). The action attribute set is constructed by the calling application and contains all information deemed relevant to the request and necessary for the trust decision. The action-environment attributes and the assignment of their values must reflect the security requirements of the application accurately. Identifying the attributes to be included in the action attribute set is perhaps the most important task in integrating KeyNote into new applications. The result of the evaluation is an application-defined string (perhaps with some additional information) that is passed back to the application. In the simplest case, the result is of the form "authorized."

The KeyNote assertion format resembles that of e-mail headers. An example (with artificially short keys and signatures for readability) is given in Figure 1.4.

As in PolicyMaker, policies and credentials (collectively called assertions) have the same format. The only difference between policies and credentials is that a policy (that is, an assertion with the keyword **POLICY** in the *Authorizer* field) is locally trusted (by the compliance-checker) and thus needs no signature.

KeyNote assertions are structured so that the *Licensees* field specifies explicitly the principal or principals to which authority is delegated. Syntactically, the Licensees field is a formula in which the arguments are public keys and the operations are conjunction, disjunction, and threshold. Intuitively, this field specifies which combinations of keys must approve a request to satisfy the issuer of the assertion. Thus, for example, a request may be granted if the CEO, or all three vice-presidents together, or any 5 out of 10 members of the Board approve it. The full semantics of these expressions are specified in [Blaze et al., 1999b].

The programs in KeyNote are encoded in the *Conditions* field and are essentially tests of the

```
KeyNote-Version: 2
Authorizer: "rsa-hex:1023abcd"
Licensees: "dsa-hex:986512a1" || "rsa-hex:19abcd02"
Comment: Authorizer delegates read access to either of the
         Licensees
Conditions: (file == "/etc/passwd" &&
             access == "read") -> "true";
Signature: sig-rsa-md5-hex:"f00f5673"
```

Figure 1.4: Sample KeyNote assertion authorizing the two keys appearing in the Licensees field to read the file "/etc/passwd." Noone else is authorized to read this file, based on this credential.

action attributes. These tests are string comparisons, numerical operations and comparisons, and pattern-matching operations.

The design choice of using a simple language for KeyNote assertions was based on the following reasons:

- AWK, one of the first assertion languages used by PolicyMaker, was criticized as too heavy-weight for most relevant applications. Because of AWK's complexity, the footprint of the interpreter is considerable, and this discourages application developers from integrating it into a trust-management component. The KeyNote assertion language is simple and has a small-size interpreter.

- In languages that permit loops and recursion (including AWK), it is difficult to enforce resource-usage restrictions, but applications that run trust-management assertions written by unknown sources often need to limit their memory and CPU usage.

  In retrospect, a language without loops, dynamic memory allocation, and certain other features seems sufficiently powerful and expressive [Blaze et al., 2001b,a]. The KeyNote assertion syntax is restricted so that resource usage is proportional to the program size. Similar concepts have been successfully used in other contexts [Hicks and Keromytis, 1999].

- Assertions should be both understandable by human readers and easy for a tool to generate from a high-level specification. Moreover, they should be easy to analyze automatically, so that automatic verification and consistency checks can done. This is currently an area of active research.

- One of the design goals is to use KeyNote as a means of exchanging policy and distributing access control information otherwise expressed in an application-native format. Thus the language should be easy to map to a number of such formats (*e.g.*, from a KeyNote assertion to packet-filtering rules).

- The language chosen was adequate for KeyNote's evaluation model.

This last point requires explanation.

In PolicyMaker, compliance proofs are constructed via repeated evaluation of assertions, along with an arbitrated "blackboard" for storage of intermediate results and inter-assertion communication.

In contrast, KeyNote uses an algorithm that attempts (recursively) to satisfy at least one policy assertion. Referring again to Figure 1.3, KeyNote treats keys as vertices in the graph, with (directed) edges representing assertions delegating authority. The prototype implementation uses a Depth First Search algorithm, starting from the set of trusted ("POLICY") assertions and trying to construct a

path to the key of the user making the request. An edge between two vertices in the graph exists only if:

- There exists an assertion where the *Authorizer* and the *Licensees* are the keys corresponding to the two vertices.
- The predicate encoded in the *Conditions* field of that KeyNote assertion authorizes the request.

Thus, satisfying an assertion entails satisfying both the *Conditions* field and the *Licensees* key expression. If no such graph exists (due to missing credentials or requests that were not accepted by an assertion's predicate), the request will be denied. Thus, as in PolicyMaker, it is the responsibility of the requester to provide all necessary material for the request to succeed.

Note that there is no explicit inter-assertion communication as in PolicyMaker; the *acceptance records* returned by program evaluation are used internally by the KeyNote evaluator and are never seen directly by other assertions. Because KeyNote's evaluation model is a subset of PolicyMaker's, the latter's compliance-checking guarantees are applicable to KeyNote. In PolicyMaker, the programs contained in the assertions and credentials can interact with each other by examining the values written on the blackboard, and reacting accordingly. This, for example, allows for a negotiation of sorts: "I will approve, if you approve," "I will also approve if you approve," "I approve," "I approve as well"... In KeyNote, each assertion is evaluated exactly once, and cannot directly examine the result of another assertion's evaluation. Whether the more restrictive nature of KeyNote allows for stronger guarantees to be made, *e.g.*, a tighter space/time evaluation bound, is an open question requiring further research.

Ultimately, for a request to be approved, an assertion graph must be constructed between one or more policy assertions and one or more keys that signed the request. Because of the evaluation model, an assertion located somewhere in a delegation graph can effectively only refine (or pass on) the authorizations conferred on it by the previous assertions in the graph. (This principle also holds for PolicyMaker, although its evaluation model differs.) For more details on the evaluation model, see [Blaze et al., 1999b].

It should be noted that PolicyMaker's restrictions regarding "negative credentials" also apply to KeyNote. Certificate revocation lists (CRLs) are not built into the KeyNote (or the PolicyMaker) system; these can be provided at a higher (or lower) level, perhaps even transparently to KeyNote. (Note that the decision to consult a CRL is (or should be) a matter of local policy.) The problem of credential discovery is also not explicitly addressed in KeyNote.

Finally, note that KeyNote, like other trust-management engines, does not directly *enforce* policy; it only provides "advice" to the applications that call it. KeyNote assumes that the application itself is trusted and that the policy assertions are correct. Nothing prevents an application from submitting misleading assertions to KeyNote or from ignoring KeyNote altogether.

1.4 Applications of Trust Management Systems

In this section we briefly describe the use of KeyNote in various systems. Although the ability to use KeyNote in such a wide range of applications validates its generality, we also identify several shortcomings.

1.4.1 Network-layer Access Control

One of the first applications of KeyNote was providing access control services for the IPsec [Kent and Atkinson, 1998] architecture. The IPsec protocol suite, which provides network-layer security for the Internet, has been standardized in the IETF and is beginning to make its way into commercial

implementations of desktop, server, and router operating systems. IPsec does not itself address the problem of managing the *policies* governing the handling of traffic entering or leaving a node running the protocol. By itself, the IPsec protocol can protect packets from external tampering and eavesdropping, but does nothing to control which nodes are authorized for particular kinds of sessions or for exchanging particular kinds of traffic. In many configurations, especially when network-layer security is used to build firewalls and virtual private networks, such policies may necessarily be quite complex.

[Blaze et al., 2001b, 2002] introduced a new policy management architecture for IPsec. A *compliance check* was added to the IPsec architecture that tests packet filters proposed when new security associations are created for conformance with the local security policy, based on credentials presented by the peer node. Security policies and credentials can be quite sophisticated (and specified in KeyNote), while still allowing very efficient packet-filtering for the actual IPsec traffic. The resulting implementation [Hallqvist and Keromytis, 2000] has been in use in the OpenBSD [de Raadt et al., 1999] operating system for several years. This system has formed the basis of several commercial VPN and SoHo firewall products.

1.4.2 Distributed Firewalls and the STRONGMAN Architecture

Conventional firewalls rely on topology restrictions and controlled network entry points to enforce traffic filtering. The fundamental limitation of the firewall approach to network security is that a firewall cannot filter traffic it does not see; by implication, everyone on the protected side has to be considered trusted. While this model has worked well for small to medium size networks, networking trends such as increased connectivity, higher line speeds, extranets, and telecommuting threaten to make it obsolete. To address the shortcomings of traditional firewalls, the concept of a *distributed firewall* has been proposed [Bellovin, 1999]. In this scheme, security policy is still centrally defined, but enforcement is left up to the individual endpoints. Credentials distributed to every node express parts of the overall network policy. The use of KeyNote for access control at the network layer enabled us to develop a prototype distributed firewall [Ioannidis et al., 2000]. Under certain circumstances, the prototype exhibited better performance than the traditional-firewall approach, as well as handle the increasing protocol complexity and the use of end-to-end encryption.

This functionality has been used in other projects where dynamic access control was necessary. In [Keromytis et al., 2002], the ability to effectively control a large number of firewalls, any of which can be contacted by any of a large number of potentially users was allowed to build a distributed denial of service (DDoS) resistant architecture for allowing authorized users to contact sites that are under attack.

The distributed firewall concept was later generalized in the STRONGMAN architecture, which allowed coordinated and decentralized management of a large number of nodes and services throughout the network stack [Keromytis et al., 2003; Keromytis, 2001]. STRONGMAN offers three new approaches to scalability, applying the principle of local policy enforcement complying with global security policies. First is the use of a compliance checker to provide great local autonomy within the constraints of a global security policy. Second is a mechanism to compose policy rules into a coherent enforceable set, *e.g.*, at the boundaries of two locally autonomous application domains. Third is the lazy instantiation of policies to reduce the amount of state that enforcement points need to maintain. STRONGMAN is capable of managing such diverse resources and protocols as firewalls, web access control (discussed later), filesystem accesses, and process sandboxing. Work on STRONGMAN is continuing, focusing on the ease of management and correctness components of the system [Ioannidis et al., 2003].

Another use of KeyNote has been in web access control, where it is used to mediate requests for pages or access to CGI scripts [Levine et al., 2003]. This is implemented as a module for the Apache web server, *mod_keynote*, which performs the compliance checking functions on a per-request basis. This module has also been distributed with the OpenBSD operating system for several years, and

the functionality has been folded into the STRONGMAN architecture.

1.4.3 Grid Computing and Transferable Micropayments

KeyNote is used to manage the authorization relationships in the Secure WebCom Metacomputer [Foley et al., 2001, 2002]. WebCom [Morrison et al., 1999] is a client/server based system that may be used to schedule mobile application components for execution across a network. In Secure WebCom, KeyNote credentials are used to determine the authorization of X.509-authenticated SSL connections between WebCom masters and clients. Client credentials are used by WebCom masters to determine what operations the client is authorized to execute; WebCom master credentials are used by clients to determine if the master has the authorization to schedule the (trusted) mobile computation that the client is about to execute.

Systems that provide access to their resources can be paid using hash-chain based micropayments [Foley and Quillinan, 2002]. KeyNote credentials are used to codify hash-chain micropayment contracts; determining whether a particular micropayment should be accepted amounts to a KeyNote compliance check that the micropayment is authorized. This scheme is generalized in [Foley, 2003] to support the efficient transfer of micropayment contracts whereby a transfer amounts to delegation of authorization for the contract. Characterizing a payment scheme as a trust management problem means that trust policies that are based on both monetary and conventional authorization concerns can be formulated.

1.4.4 Micropayments: Microchecks and Fileteller

One of the more esoteric uses of KeyNote has been as a micropayment scheme that requires neither online transactions nor trusted hardware for either the payer or payee. Each payer is periodically issued certified credentials that encode the type of transactions and circumstances under which payment can be guaranteed. A risk management strategy, taking into account the payer's history, and other factors, can be used to generate these credentials in a way that limits the aggregated risk of uncollectible or fraudulent transactions to an acceptable level. [Blaze et al., 2001a] showed a practical architecture for such a system that used KeyNote to encode the credentials and policies, and described a prototype implementation of the system in which vending machine purchases were made using off-the-shelf consumer PDAs.

[Ioannidis et al., 2002] uses this micropayment architecture to build a credential-based network file storage system with provisions for paying for file storage and getting paid when others access files. Users get access to arbitrary amounts of storage anywhere in the network, and use a micropayments system to pay for both the initial creation of the file and any subsequent accesses. Wide-scale information sharing requires that a number of issues be addressed; these include distributed access, access control, payment, accounting, and delegation (so that information owners may allow others to access their stored content). Utilizing the same mechanism for both access control and payment results in an elegant and scalable architecture.

Ongoing work in this area is examining at distributed peer-to-peer filesystems and pay-per-use access to 802.11 networks.

1.4.5 Active Networking

Finally, STRONGMAN has been used in the context of active networks [Alexander et al., 1998a] to provide access control services to programmable elements [Alexander et al., 1998b, 2000, 2001]. An active network is a network infrastructure that is programmable on a per-user or even per-packet basis. Increasing the flexibility of such network infrastructures invites new security risks. Coping with these security risks represents the most fundamental contribution of active network research. The security concerns can be divided into those which affect the network as a whole and those which

affect individual elements. It is clear that the element problems must be solved first, as the integrity of network-level solutions will be based on trust of the network elements. In the SANE architecture, KeyNote was used to limit the privileges of network users and their mobile code, by specifying the operations such code was allowed to perform on any particular active node. KeyNote was used in a similar manner in the FLAME architecture [Anagnostakis et al., 2001, 2002b,a], and to provide an economy for resources in an active network [Anagnostakis et al., 2000].

1.5 Other Trust-Based Systems

The REFEREE system of Chu *et al.* [1997] is like PolicyMaker in that it supports full programmability of assertions (policies and credentials). However, it differs in several important ways. It allows the trust-management engine, while evaluating a request, to fetch additional credentials and to perform cryptographic signature-verification. (Recall that PolicyMaker places the responsibility for both of these functions on the calling application and insists that they be done before the evaluation of a request begins.) Furthermore, REFEREE's notion of "proof of compliance" is more complex than PolicyMaker's; for example, it allows non-monotonic policies and credentials. The REFEREE proof system also supports a more complicated form of inter-assertion communication than PolicyMaker does. In particular, the REFEREE execution environment allows assertion programs to call each other as subroutines and to pass different arguments to different subroutines, whereas the PolicyMaker execution environment requires each assertion program to write anything it wants to communicate on a global "blackboard" that can be seen by all other assertions.

REFEREE was designed with trust management for web browsing in mind, but it is a general-purpose language and could be used in other applications. Some of the design choices in REFEREE were influence by experience (reported in [Blaze et al., 1997]) with using PolicyMaker for web-page filtering based on PICS [Resnick and Miller, 1996] labels and users' viewing policies. It is unclear whether the cost of building and analyzing a more complex trust-management environment such as REFEREE is justified by the ability to construct more sophisticated proofs of compliance than those constructible in PolicyMaker. Assessing this tradeoff would require more experimentation with both systems, as well as a rigorous specification and analysis of the REFEREE proof system, similar to the one for PolicyMaker given in [Blaze et al., 1998].

The Simple Public Key Infrastructure (SPKI) project of Ellison *et al.* [Ellison, 1999] has proposed a standard format for authorization certificates. SPKI shares with our trust-management approach the belief that certificates can be used directly for authorization rather than simply for authentication. However, SPKI certificates are not fully programmable; they are data structures with the following five fields: "Issuer" (the source of authority), "Subject" (the entity being authorized to do something), "Delegation" (a boolean value specifying whether or not the subject is permitted to pass the authorization on to other entities), "Authorization" (a specification of the power that the issuer is conferring on the subject), and "Validity dates."

The SPKI documentation [Ellison, 1999] states that the processing of certificates and related objects to yield an authorization result is the province of the developer of the application or system. The processing plan presented in that document is an example that may be followed, but its primary purpose is to clarify the semantics of an SPKI certificate and the way it and various other kinds of certificate might be used to yield an authorization result.

Thus, strictly speaking, SPKI is not a trust-management engine according to our use of the term, because compliance checking (referred to above as "processing of certificates and related objects") may be done in an application-dependent manner. If the processing plan presented in [Ellison, 1999] were universally adopted, then SPKI would be a trust-management engine. The resulting notion of "proof of compliance" would be considerably more restricted than PolicyMaker's; essentially,

proofs would take the form of chains of certificates. On the other hand, SPKI has a standard way of handling certain types of non-monotonic policies, because validity periods and simple CRLs are part of the proposal.

1.6 Closing Remarks

Trust management is a powerful approach to specifying and enforcing access control policies. The fundemental concepts behind trust management are the inherent constrained-delegation capability, *assertion monotonicity*, and a policy evaluation model that ensures safety and correctness. We briefly identified some uses of trust management systems in various applications, which should demonstrate the versatility and adaptability of the concepts and mechanisms presented.

The opportunity to use trust-management techniques exists in many projects that require some security component. Although the specific approaches we discussed may not be appropriate for any given application, the concepts are general enough and should be applicable in any context. The designer should carefully consider the system's needs and determine how best to use trust management. Doing so will allow them to easily manage fine-grained authorization and access control in a scalable and powerful way.

1.7 Acknowledgements

We would like to thank our collaborators in this work, Matt Blaze and Joan Feigenbaum.

References

D. S. Alexander, W. A. Arbaugh, M. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare Active Network Architecture. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):29–36, 1998a.

D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, S. Muir, and J. M. Smith. Secure Quality of Service Handling (SQoSH). *IEEE Communications*, 38(4):106–112, April 2000.

D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. A Secure Active Network Environment Architecture: Realization in SwitchWare. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):37–45, 1998b.

D.S. Alexander, P.B. Menage, A.D. Keromytis, W.A. Arbaugh, K.G. Anagnostakis, and J.M. Smith. The Price of Safety in an Active Network. *Journal of Communications (JCN), special issue on programmable switches and routers*, 3(1):4–18, March 2001.

K. G. Anagnostakis, M. B. Greenwald, S. Ioannidis, and S. Miltchev. Open Packet Monitoring on FLAME: Safety, Performance and Applications. In *Proceedings of the 4rd International Working Conference on Active Networks (IWAN)*, December 2002a.

K. G. Anagnostakis, M. W. Hicks, S. Ioannidis, A. D. Keromytis, and J. M. Smith. Scalable Resource Control in Active Networks. In *Proceedings of the Second International Working Conference on Active Networks (IWAN)*, pages 343–357, October 2000.

K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, Michael B. Greenwald, and J. M. Smith. Efficient Packet Monitoring for Network Management. In *Proceedings of IFIP/IEEE Network Operations and Management Symposium (NOMS) 2002*, April 2002b.

K. G. Anagnostakis, S. Ioannidis, S. Miltchev, and J. M. Smith. Practical Network Applications on a Lightweight Active Management Environment. In *Proceedings of the 3rd International Working Conference on Active Networks (IWAN)*, October 2001.

S. M. Bellovin. Distributed Firewalls. *;login: magazine, special issue on security*, pages 37–39, November 1999.

M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The Role of Trust Management in Distributed Systems Security. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer-Verlag Inc., 1999a. ISBN 3-540-66130-1.

M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System Version 2. Internet RFC 2704, September 1999b.

M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 17th IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

M. Blaze, J. Feigenbaum, P. Resnick, and M. Strauss. Managing Trust in an Information Labeling System. In *European Transactions on Telecommunications, 8*, pages 491–501, 1997.

M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust-Management System. In *Proceedings of the Financial Cryptography Conference, Lecture Notes in Computer Science, vol. 1465*, pages 254–274. Springer, 1998.

M. Blaze, J. Ioannidis, and A. D. Keromytis. Offline Micropayments without Trusted Hardware. In *Proceedings of the Fifth International Conference on Financial Cryptography*, pages 21–40, February 2001a.

M. Blaze, J. Ioannidis, and A.D. Keromytis. Trust Management for IPsec. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, pages 139–151, February 2001b.

M. Blaze, J. Ioannidis, and A.D. Keromytis. Trust Management for IPsec. *ACM Transactions on Information and System Security (TISSEC)*, 32(4):1–24, May 2002.

CCITT. *X.509: The Directory Authentication Framework*. International Telecommunications Union, Geneva, 1989.

Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust Management for Web Applications. In *World Wide Web Journal, 2*, pages 127–139, 1997.

T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in OpenBSD: An Overview. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 93 – 101, June 1999.

C. Ellison. SPKI Requirements. Request for Comments 2692, Internet Engineering Task Force, September 1999. URL `ftp://ftp.isi.edu/in-notes/rfc2692.txt`.

C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. Request for Comments 2693, Internet Engineering Task Force, September 1999. URL `ftp://ftp.isi.edu/in-notes/rfc2693.txt`.

S.N. Foley. Using Trust Management to Support Transferable Hash-Based Micropayments. In *Proceedings of the International Financial Cryptography Conference*, January 2003.

S.N. Foley and T.B Quillinan. Using Trust Management to Support MicroPayments. In *Proceedings of the Annual Conference on Information Technology and Telecommunications*, October 2002.

S.N. Foley, T.B. Quillinan, and J.P. Morrison. Secure Component Distribution Using WebCom. In *Proceedings of the 17th International Conference on Information Security (IFIP/SEC)*, May 2002.

S.N. Foley, T.B. Quillinan, J.P. Morrison, D.A. Power, and J.J. Kennedy. Exploiting KeyNote in WebCom: Architecture Neutral Glue for Trust Management. In *Fifth Nordic Workshop on Secure IT Systems*, Oct 2001.

Niklas Hallqvist and Angelos D. Keromytis. Implementing Internet Key Exchange (IKE). In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, pages 201–214, June 2000.

Michael Hicks and Angelos D. Keromytis. A Secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Working Conference on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999. URL `http://www.cis.upenn.edu/ switchware/papers/iwan99.ps`.

John Ioannidis, Sotiris Ioannidis, Angelos Keromytis, and Vassilis Prevelakis. Fileteller: Paying and Getting Paid for File Storage. In *Proceedings of the Sixth International Conference on Financial Cryptography*, March 2002.

S. Ioannidis, S. M. Bellovin, I. Ioannidis, A. D. Keromytis, and J. M. Smith. Design and Implementation of Virtual Private Services. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security, Special Session on Trust Management in Collaborative Global Computing*, June 2003.

S. Ioannidis, A.D. Keromytis, S.M. Bellovin, and J.M. Smith. Implementing a Distributed Firewall. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 190–199, November 2000.

S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. Request for Comments (Proposed Standard) 2401, Internet Engineering Task Force, November 1998. URL `ftp://ftp.isi.edu/in-notes/rfc2401.txt`.

A. D. Keromytis. *STRONGMAN: A Scalable Solution To Trust Management In Networks*. PhD thesis, University of Pennsylvania, Philadelphia, November 2001.

A.D. Keromytis, S. Ioannidis, M.B. Greenwald, and J.M. Smith. The STRONGMAN Architecture. In *Proceedings of DISCEX III*, April 2003.

Angelos D. Keromytis, Vishal Misra, and Daniel Rubenstein. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM*, pages 61–72, August 2002.

J. Lacy, J. Snyder, and D. Maher. Music on the Internet and the Intellectual Property Protection Problem. In *Proceedings of the International Symposium on Industrial Electronics*, pages SS77–83, 1997.

A. Levine, V. Prevelakis, J. Ioannidis, S. Ioannidis, and A. D. Keromytis. WebDAVA: An Administrator-Free Approach to Web File-Sharing. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Distributed and Mobile Collaboration*, June 2003.

J.P. Morrison, D.A. Power, and J.J. Kennedy. WebCom: A Web Based Distributed Computation Platform. In *Proceedings of Distributed computing on the Web*, June 1999.

P. Resnick and J. Miller. PICS: Internet Access Controls Without Censorship. *Communications of the ACM*, pages 87–93, October 1996.

B. Schneier. *Applied Cryptography*. John Wiley, 1996.

B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, 2000. ISBN 0471253111.