The Role of Trust Management in Distributed Systems Security*

Matt Blaze[1], Joan Feigenbaum[1], John Ioannidis[1], and Angelos D. Keromytis[2]

[1] AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932 USA
{mab,jf,ji}@research.att.com

[2] Distributed Systems Lab
CIS Department, University of Pennsylvania
200 S. 33rd Str., Philadelphia, PA 19104 USA
angelos@dsl.cis.upenn.edu

Abstract. Existing authorization mechanisms fail to provide powerful and robust tools for handling security at the scale necessary for today's Internet. These mechanisms are coming under increasing strain from the development and deployment of systems that increase the programmability of the Internet. Moreover, this "increased flexibility through programmability" trend seems to be accelerating with the advent of proposals such as Active Networking and Mobile Agents.

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.

In this paper, we examine existing authorization mechanisms and their inadequacies. We introduce the concept of trust management, explain its basic principles, and describe some existing trust-management engines, including *PolicyMaker* and *KeyNote*. We also report on our experience using trust-management engines in several distributed-system applications.

* To appear in "Secure Internet Programming: Security Issues for Mobile and Distributed Objects," *ed.* Jan Vitek and Christian Jensen. Springer-Verlag Inc., New York, NY, USA.

1 Introduction

With the advent of the Internet, distributed computing has become increasingly prevalent. Recent developments in programming languages, coupled with the increase in network bandwidth and end-node processing power, have made the Web a highly dynamic system. Virtually every user of the Internet is at least aware of languages such as Java [GJS96], JavaScript, Active-X, and so on. More "futuristic" projects involve computers running almost exclusively downloaded interpreted-language applications (Network PC), or on-the-fly programmable network infrastructures (Active Networks). On a more mundane level, an increasing number of organizations use the Internet (or large Intranets) to connect their various offices, branches, databases, *etc.*

All of these emerging systems have one thing in common: the need to grant or restrict access to resources according to some security policy. There are several issues worth noting.

First, different systems and applications have different notions of what a resource is. For example, a web browser may consider CPU cycles, network bandwidth, and perhaps private information to be resources. A database server's notion of "resource" would include individual records. Similarly, a banking application would equate resources with money and accounts. While most of these resources can be viewed as combinations of more basic ones (such as CPU cycles, I/O bandwidth, and memory), it is often more convenient to refer to those combinations as single resources, abstracting from lower-level operations. Thus, a generic security mechanism should be able to handle any number and type of resources.

What should also be obvious from the few examples mentioned above is that different applications have different access-granting or -restricting policies. The criteria on which a decision is based may differ greatly among different applications (or even between different instances of the same application). The security mechanism should be able to handle those different criteria.

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. Typically, authentication is accomplished via a username/password mechanism. Simple password-based protocols are inadequate in networked comput-

ing environments, however, even against unsophisticated adversaries; simple eavesdropping can destroy security. Other recently developed mechanisms include:

  - One-Time passwords, which do not secure the rest of the session.
  - Centralized ticket-based systems, such as Kerberos [MNSS87]. Problems with such systems include the necessity for an authentication server (and for frequent communication with it) and implicit trust assumptions.
  - Public-key based authentication protocols, which are considered the "state of the art" for scalable authentication systems.

- *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.

We believe that these points constitute a forceful argument that *Authenticode*, X.509, and, generally, the use of identity-based public-key systems in conjunction with ACLs[1] are inadequate solutions to distributed (and programmable) system-security problems. Even modern ACL-based systems like DCE fall somewhat short of satisfyingly addressing the extensibility, expressibility, and delegation issues, despite some work in these directions [CS96].

Furthermore, one finds insecure, inadequate, or non-scalable authentication mechanisms (such as username/password, One-Time passwords, and hardware token authentication) used in conjunction with ACLs. Finally, many policy problems are left unsolved by the "binary" authorization model employed widely in

[1] Sometimes the public keys are hardcoded into the application, which deprives the environment of even the limited flexibility provided by ACLs. Such an example is the under-development IEEE 1394 Digital Content Protection standard.

Web security (and elsewhere): access is granted on the condition that the requesting principal has a certificate by a particular Certification Authority (CA). We call this a binary authorization model because it means essentially "all-or-nothing" access. While this is sufficient in a small number of cases (*e.g.,* when read-only access to a web page is the only decision that need be made), it is obvious that this approach neither scales nor is extensible.

We believe that these unintuitive and in many ways problematic mechanisms are in use simply because of the lack of alternatives that are better suited to distributed systems. Developers have tried to adapt existing tools to their security model or *vice versa*, and neither strategy has worked particularly well. Accordingly, misconfiguration and unexpected component interactions are causing security breaches and, inevitably, loss of confidence in the security tools.

We also believe that the new Internet services now emerging as a result of increased programmability will require powerful and expressive authorization mechanisms without the problems present in ACL-based systems[2]. Naturally, existing services and protocols can also make good use of such mechanisms.

Trust Management, introduced by Blaze *et al.* [BFL96], is a unified approach to specifying and interpreting security policies, credentials, and relationships that allows direct authorization of security-critical actions. In particular, a trust-management system combines the notion of specifying security policy with the mechanism for specifying security credentials. Credentials describe specific delegations of trust among public keys; unlike traditional certificates, which bind keys to names, trust-management credentials bind keys directly to authorizations to perform specific tasks. Trust-management systems support delegation, and policy specification and refinement at the different layers of a policy hierarchy, thus solving to a large degree the consistency and scalability problems inherent in traditional ACLs. Furthermore, trust-management systems are by design extensible and can express policies for different types of applications.

Section 2 gives an overview of Trust Management and briefly describes two tools that we have developed (*PolicyMaker* and *KeyNote*). Section 3 presents some applications of Trust Management. Finally, Section 4 discusses future work and concludes this paper.

2 Trust Management

This section explains the essence of the trust-management approach, describes PolicyMaker [BFL96, BFS98] and KeyNote [BFIK98], and ends with a brief discussion of related trust-management work.

2.1 Basics

A traditional "system-security approach" to the processing of a signed request for action treats the task as a combination of *authentication* and *access control*.

[2] For the purposes of this discussion, we consider the traditional capability systems as ACLs, as they exhibit most of the weaknesses we mention.

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. We believe that this is the wrong approach for today's dynamic, internetworked world. 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?" Because name-key mappings and pre-computed access-control matrices are inadequate, one needs a more flexible, more "distributed" approach to authorization. The *trust-management approach*, initiated by Blaze *et al.* [BFL96], frames the question as follows: "Does the set C of *credentials* prove that the *request* r *complies* with the local security *policy* P?" Each entity that receives requests must have a policy that serves as the ultimate source of authority in the local environment. The 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 policy has been proven, and may also output some additional information about how to proceed if it hasn't.

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? Since any product or service that requires some form of proof that requested transactions comply with policies could use a special-purpose algorithm implemented from scratch, what do developers, administrators, and users gain by using a general-purpose compliance checker?

The 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") may 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?

In the rest of this section, we survey several recent and ongoing trust-management projects in which different answers to these questions are explored.

2.2 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 2.1. 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.* [BFL96], and its compliance-checking algorithm was later fleshed out in [BFS98]. We give a high-level overview of the design decisions that went into PolicyMaker in this section and some technical details of the PolicyMaker compliance checker in Appendix A below. A full description of the system can be found in [BFL96, BFS98], and experience using it in several applications is reported in [BFRS97, LSM97, LMB].

PolicyMaker credentials and policies are fully programmable; together credentials and policies are referred to as "assertions." Roughly speaking, assertions 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. In a credential assertion, the source is the public key of the issuing authority. 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 safe version of

AWK was developed for early experimental work on PolicyMaker (see Blaze *et al.* [BFL96]), 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. 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 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 minimal 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 [BFIK98] and REFEREE [CFL+97] divide responsibility between the calling application and the trust-management engine differently from 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 and some details in Appendix A; a complete treatment of the compliance checker can be found in [BFS98].

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, write-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. 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.* [BFS98] 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 polynomial-time solvable, is useful in a wide variety of applications, and is implemented in the current version of PolicyMaker.

Although the most general version of the compliance-checking problem allows assertions to be arbitrary functions, the computationally tractable version that is analyzed in [BFS98] 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 Appendix A 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.

2.3 KeyNote

KeyNote [BFIK98] 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 and appropriate assertion language, KeyNote goes further than PolicyMaker toward facilitating efficiency, interoperability, and widespread use of carefully written credentials and policies.

A calling application passes to a KeyNote evaluator a list of credentials, policies, and requester public keys, and an "Action Environment." This last element consists of a list of attribute/value pairs, similar in some ways to the $Unix^{TM}$ shell environment. The action environment 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 environment 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 something like "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.

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-Version: 1
Authorizer: rsa-pkcs1-hex:"1023abcd"
Licensees: dsa-hex:"986512a1" ||
           rsa-pkcs1-hex:"19abcd02"
Comment: Authorizer delegates read
         access to either of the
         Licensees
Conditions: ($file == "/etc/passwd" &&
             $access == "read") ->
                          {return "ok"}
Signature: rsa-md5-pkcs1-hex:"f00f5673"
```

Fig. 1. Sample KeyNote assertion

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. The semantics of these expressions are specified in [BFIK98].

The programs in KeyNote are encoded in the *Conditions* field and are essentially tests of the action environment variables. These tests are string comparisons, numerical operations and comparisons, and pattern-matching operations.

We chose such a simple language for KeyNote assertions for the following reasons:

- AWK, one of the first assertion languages used by PolicyMaker, was criticized as too heavyweight 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 minimal-sized 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.
  We believe that for out purposes a language without loops, dynamic memory allocation, and certain other features is sufficiently powerful and expressive. 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 [HKM+98].
- 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 our goals is to use KeyNote as a means of exchanging policy and distributing access control information otherwise expressed in an applica-

tion-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 a depth-first search (DFS) algorithm that attempts (recursively) to satisfy at least one policy assertion. Satisfying an assertion entails satisfying both the *Conditions* field and the *Licensees* key expression. 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. Whether the more restrictive nature of KeyNote allows for stronger guarantees to be made 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.) For more details on the evaluation model, see [BFIK98].

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 however could be provided at a higher (or lower) level, perhaps even transparently to KeyNote[3]. The problem of credential discovery is also not explicitly addressed in KeyNote. We hope that the fact that KeyNote provides an explicit, descriptive credential format will facilitate research on both credential discovery and revocation.

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.

2.4 Related Work on Trust Management

We close with a brief discussion of two general-purpose trust-management systems that share much of the basic approach initiated by PolicyMaker but depart from it in some notable ways.

The REFEREE system of Chu *et al.* [CFL+97] 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,

[3] Note that the decision to consult a CRL is (or should be) a matter of local policy.

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 [BFRS97]) with using PolicyMaker for web-page filtering based on PICS labels [RM96] 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 [BFS98].

The Simple Public Key Infrastructure (SPKI) project of Ellison *et al.* [EFR+97] 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 certificate format is compatible with the Simple Distributed Security Infrastructure (SDSI) local-names format proposed by Rivest and Lampson [LR97], and Ellison *et al.* [EFR+97] explain how to integrate the two.

The SPKI documentation [EFR+97] 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 here 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 "process-

ing of certificates and related objects") may be done in an application-dependent manner. If the processing plan presented in [EFR+97] 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.

3 Applications of Trust Management

In this section, we give a brief overview of experiences with the trust-management approach in mobile-code security, active networking, nd distributed access-control.

3.1 Active Networks

There has been a great deal of interest in the problem of exposing the ability to control of network infrastructure. Much of this interest has been driven by a the desire to accelerate service creation. Sometimes services can be created using features of existing systems. One of the most aggressive proposals is the notion of *programmable* network infrastructure or "active networking." In an active network, the operator or user has facilities for directly modifying the operational semantics of the network itself. Thus, the role of network and endpoint become far more malleable for the construction of new applications. This is in contrast to the "service overlay model" as employed, for example, in present-day Internet, where service introduction at the "edge" of the virtual infrastructure is very easy, but changes in the infrastructure itself have proven very difficult (*e.g.,* RSVP [BZB+97] and multicasting [Dee89]). A number of active network architectures have been proposed and are under investigation [AAH+98, WGT98, HKM+98, WLG98, CSAA98, PJ96].

A programmable network infrastructure is potentially more vulnerable to attacks since a portion of the control plane is *intentionally* exposed, and this can lead to far more complex threats than exist with an inaccessible control plane. For example, a denial-of-service attack on the transport plane may also inhibit access to the control plane. Unauthenticated access to the control plane can have severe consequences for the security of the whole network.

It is therefore especially necessary for an active network to use a robust and powerful authorization mechanism. Because of the many interactions between network nodes and switchlets (pieces of code dynamically loaded on the switches, or code in packets that is executed on every active node they encounter), a versatile, scalable, and expressive mechanism is called for.

We have applied KeyNote [BFIK98] in one proposed active network security architecture, in the Secure Active Network Environment (SANE) [AAKS98] [AAKS99] developed at the University of Pennsylvania as part of the SwitchWare project [AAH+98].

In SANE, the principals involved in the authorization and policy decisions in the security model are users, programmers and administrators and network elements. The network elements are presumed to be under physical control of an administrator. Programmers may not have physical access to the network element, but may possess considerable access rights to resources present in the network elements. Users may have access to basic services (*e.g.,* transport), but also resources that the network elements are willing to export to all users, at an appropriate level of abstraction. Users may also be allowed to introduce their own services, or load those written by others. In such a dynamic environment, KeyNote is used to supply policy and authorization credentials for those components of the architecture that enforce resource usage and access control limits.

In particular, KeyNote policies and credentials are used to:

- Authorize principals to load code on active routers. This happens as a result of an authentication mechanism, further described in [AAKS98].
- Intimately related to the previous item, KeyNote sets resource limits (*e.g.,* memory allocated, CPU cycles consumed, memory bandwidth) that the execution environment[4] enforces. The KeyNote evaluator is initially invoked just before a switchlet begins execution, and is provided with the switchlet owner's credentials and the node's policies. It then determines whether the switchlet is allowed to execute on the node (addressing the previous item), and what the initial resource limits are. If these are exceeded, another call to the KeyNote system is made, whereupon the decision may be to grant more resources, kill the switchlet, or raise an exception.
- Fine-grained control of what actions a switchlet may take on the active node. In our current approach, loaded code may call other loaded or resident functions (resembling inter-process communication and system call invocation respectively). Which of these calls are allowed by policy to occur is (or rather, can optionally be) controlled by KeyNote. For performance reasons, and since the switchlet language used lends itself to some degree of static analysis, this authorization step happens at link time (when a newly-loaded switchlet is dynamically linked in the runtime system). An alternative way of achieving this would be to trap into the KeyNote evaluator every time an external (to the switchlet) function was called, by generating appropriate function stubs at link time. This could lead to considerable performance problems, especially for time-critical applications (such as real-time content processing and distribution). Such functionality, however, is necessary if one needs to restrict the range of arguments that may be passed to external functions. This is currently an area of open research.
- KeyNote credentials can be used by "active firewalls" to notify nodes behind the firewall that a particular piece of active code should (or should not) be allowed to perform specific tasks [KBIS98]. In this context, KeyNote defines trust relations between active nodes (*e.g.,* "firewall", "protected host", *etc.*)

Trust management for active networks is an area of continuing research.

[4] In the case of SANE, the execution environment is a modified and restricted runtime of the Caml [Ler] programming language.

3.2 Mobile Code Security

Another area of broad recent interest is the security and containment of untrusted "mobile" code. That is, executable content or mobile code is received by a host with a request to a execute it; lacking any automatic mechanisms for evaluating the security implications of executing such a piece of code, the host needs to find some other way of determining the trustworthiness of that code.

Failure to properly contain mobile code may result in serious damage or leakage or information resident on the host. Such damage can be the result of malicious intent (*e.g.,* industrial or otherwise espionage or vandalism), or unintentional (*e.g.,* programming failures or unexpected interactions with other system components or programs). Other consequences of failing to contain mobile code include denial-of-service attacks (the now familiar phenomenon of Java and JavaScript applets using all the system memory or CPU cycles, usually because of bugs), or infiltration of a company network through a downloaded trojan horse or virus.

Of course, these threats existed long before mobile code, such as Java applets, became popular. Downloading software from a bulletin board service or through the Internet and then executing it without any form of code review or execution restrictions has almost exactly the same security implications. The problem, however, has been made more prominent because of the scale and ease with which attacks to individuals and hosts can be carried out, especially in environments where mobile code is run automatically.

As we mentioned in the introduction, various proposals exist that tie code "trustworthiness" to the existence of a digital signature or other form of authentication. Such mechanisms, however, do not address the underlying problem; a signature has value only to the extent that the verifier can evaluate the signer's trustworthiness (which may be a result of poorly-defined, or incorrectly understood, social factors, technical competence, personal knowledge, *etc.*). In that respect, signature-based mobile code schemes do not scale. Furthermore, some programs may be "safe" to run under restricted environments or a small set of platforms. Simple digital signature schemes as have been proposed cannot readily accommodate such conditions.

Trust Management has at least two different roles in mobile code security:

- Express trust relations between code-certifying entities, and the conditions under which their certification has meaning. So, for example, code-certifier A could state that his certification of safety of a particular piece of code is predicated by some set of conditions (presumably the conditions under which the code evaluation took place, or the assumptions made by the code that may have security implications if violated). Trust management could also be used to express the local user's (or corporate) policy on executing mobile code (*e.g.,* "must be signed by the company security officer or must be accompanied by a PCC proof" [NL96, Nec97]).
- Trust-management credentials could be used to describe the minimal set of capabilities the host environment must grant to enable the code to perform

its tasks. This would then be combined with the relevant local policy[5] to restrict the system interface presented to the mobile code. This is similar in concept to the approach mentioned in Section 3.1. We are also investigating the application of this method in a traditional $Unix^{TM}$ kernel, in an environment where language-based protection is not available.

3.3 Access Control Distribution

An indirect, but architecturally very important, benefit of trust management involves the distribution of traditional access control list (ACL) databases. Some applications lend themselves naturally to a well-defined ACL-style interface for describing their security characteristics that do not obviously require the rich expressiveness of a trust-management approach. For example, it may be natural to describe who is allowed to log in to a computer according to a standard list of login accounts. Even when an ACL provides sufficient expressiveness, however, it is often architecturally beneficial to implement the ACL on top of a trust-management system.

Architectures based on a trust-management system can be easily extended if, in the future, it becomes necessary to base access decisions on more complex rules than are captured by an ACL. For example, it is natural and easy to add "time-of-day" restrictions as part of a policy or individual user credentials, even if the need for such restrictions had not been anticipated at the time the system was first designed. Of course, extensibility is a general benefit of trust management that is not limited to ACL applications.

More importantly, a trust-management system decouples the specification of access control policies from the mechanism used to distribute and implement them. For example, consider a traditional access control list maintained as a simple database in a computer system. A trust-management tool could convert ACL entries into credentials automatically, with a general credential (certificate) distribution mechanism used to ensure that updated credentials are sent to the appropriate place. Distributed, reliable access control can be a very difficult problem when implemented by itself, and is greatly simplified using a credential-based trust-management scheme as a back end to a non-distributed access control database. Even if the expressive power is limited to simple ACLs, trust management allows the use of an existing credential distribution scheme to solve the most difficult aspects of the problem.

4 Conclusion and Future Work

In the time since "trust management" first appeared in the literature in [BFL96], the concept has gained broad acceptance in the security research community. Trust management has a number of important advantages over traditional approaches such as distributed ACLs, hardcoded security policies, and global identity certificates. A trust-management system provides direct authorization of

[5] This would happen automatically, in the process of compliance checking.

security-critical actions and decouples the problem of specifying policy and authorization from that of distributing credentials.

Our work on trust management has focused on designing languages and compliance checkers, identifying applications, and building practical toolkits. There are important areas that we have not yet addressed. Foremost among those is automated credential discovery; in our current systems, it is the responsibility of the requester to submit all necessary credentials, under the assumption that he holds all credentials relevant to him. Even then, however, intermediate credentials that form the necessary connections between the verifier's policy and the requester's credentials must be acquired and submitted. The various solutions to this problem range from "leave it to the application" to using distributed databases and lookup mechanisms for credential discovery. A solution along the lines of a distributed database can also assist large organizations in auditing their security and access policies.

Another area of future work is providing functionality similar to that of certificate revocation lists (CRLs), keeping in mind the constraints about "negative credentials" mentioned in Section 2.2. While adoption and deployment of a scheme similar to that of X.509 is fairly straightforward, we are still investigating the security and operational implications.

Finally, we are examining higher-level policy languages that are even more human-understandable and capable of higher levels of abstraction. Such high-level policy would be combined with network- and application-specific information and compiled into a set to trust-management credentials. Similarly, a tool for translating trust-management credentials into application-native forms would give us all the advantages of trust management (delegation, formal proof of compliance, *etc.*) while requiring minimal changes to applications.

References

[AAH+98] 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, 1998.

[AAKS98] 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, 1998.

[AAKS99] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. Security in active networks. In *Secure Internet Programming*, Lecture Notes in Computer Science, pages ??–?? Springer-Verlag Inc., New York, NY, USA, 1999.

[BFIK98] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System. Work in Progress, http://www.cis.upenn.edu/~angelos/keynote.html, June 1998.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, Los Alamitos, 1996.

[BFRS97] 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.

[BFS98] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust-Management System. In *Proc. of the Financial Cryptography '98, Lecture Notes in Computer Science, vol. 1465*, pages 254–274. Springer, Berlin, 1998.

[BZB+97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. Internet RFC 2208, 1997.

[CFL+97] 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.

[CS96] J. Chinitz and S. Sonnenberg. A Transparent Security Framework For TCP/IP and Legacy Applications. Technical report, Intellisoft Corp., August 1996.

[CSAA98] M. Calderon, M. Sedano, A. Azcorra, and C. Alonso. The Support of Active Networks for Fuzzy-Tolerant Multicast Applications. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):20–28, 1998.

[Dee89] S. E. Deering. Host extensions for IP multicasting. Internet RFC 1112, 1989.

[EFR+97] C. M. Ellison, B. Frantz, R. Rivest, B. M. Thomas, and T. Ylonen. Simple Public Key Certificate. Work in Progress, http://www.pobox.com/~cme/html/spki.html, April 1997.

[ESY84] S. Even, A. Selman, and Y. Yacobi. The Complexity of Promise Problems with Applications to Public-Key Cryptography. *Information and Control*, 61:159–174, 1984.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.

[HKM+98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[KBIS98] Angelos D. Keromytis, Matt Blaze, John Ioannidis, and Jonathan M. Smith. Firewalls in Active Networks. Technical Report MS-CIS-98-03, University of Pennsylvania, February 1998.

[Ler] Xavier Leroy. The Caml Special Light System (Release 1.10). http://pauillac.inria.fr/ocaml.

[LMB] R. Levien, L. McCarthy, and M. Blaze. Transparent Internet E-mail Security. http://www.cs.umass.edu/~lmccarth/crypto/papers/email.ps.

[LR97] B. Lampson and R. Rivest. Cryptography and Information Security Group Research Project: A Simple Distributed Security Infrastructure. Technical report, MIT, 1997.

[LSM97] J. Lacy, J. Snyder, and D. Maher. Music on the Internet and the Intellectual Property Protection Problem. In *Proc. of the International Symposium on Industrial Electronics*, pages SS77–83. IEEE Press, 1997.

[MNSS87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, December 1987.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–119. ACM Press, New York, January 1997.

[NL96] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating System Design and Implementation (OSDI)*, pages 229–243. Usenix, Seattle, 1996.

[PJ96] C. Partridge and A. Jackson. Smart Packets. Technical report, BBN, 1996. http://www.net-tech.bbn.com-/smtpkts/smtpkts-index.html.

[RM96] P. Resnick and J. Miller. PICS: Internet Access Controls Without Censorship. *Communications of the ACM*, pages 87–93, October 1996.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *IEEE OpenArch Proceedings*. IEEE Computer Society Press, Los Alamitos, April 1998.

[WLG98] D. Wetherall, U. Legedza, and J. Guttag. Introducing New Internet Services: Why and How. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):12–19, 1998.

A PolicyMaker Compliance Proofs

We now give some technical details about the PolicyMaker compliance checker. This discussion is largely excerpted (with the authors' permission) from the paper of Blaze *et al.* [BFS98].

The general problem we are concerned with is *Proof of Compliance* (POC). The question is whether a *request* r complies with a *policy*. The policy is simply a function f_0 encoded in some well understood programming system or language and labeled by the keyword POLICY. In addition to the request and the policy, a POC instance contains a set of *credentials*, also general functions, each labeled by its source. Policies and credentials are collectively referred to as *assertions*.

Credentials are issued by *sources*. Formally, a credential is a pair (f_i, s_i) of function f_i and *source-ID* s_i, which is just a string over some appropriate alphabet. Important examples of source-IDs include public keys of credential issuers, URLs, names of people, and names of companies. With the exception of the keyword POLICY, the interpretation of source-IDs is part of the application-specific semantics of an assertion, and it is not the job of the compliance checker. From the compliance checker's point of view, the source-IDs are just strings, and the assertions encode a set of (possibly indirect and possibly conditional) trust relationships among the issuing sources. Associating each assertion with the correct source-ID is the responsibility of the calling application, as explained in Section 2.2.

The request r is a string encoding an *action* for which the calling application seeks a proof of compliance. In the course of deciding whether the credentials $(f_1, s_1), \ldots, (f_{n-1}, s_{n-1})$ constitute a proof that r complies with the policy (f_0, POLICY), the compliance checker's domain of discourse may need to include other action strings. For example, if POLICY requires that r be approved by credential issuers s_1 and s_2, the credentials (f_1, s_1) and (f_2, s_2) may want a way to say that they approve r *conditionally*, where the condition is that the other credential also approve it. A convenient way to formalize this is to use strings R, R_1, and R_2 over some finite alphabet Σ. The string R corresponds to the requested action r. The strings R_1 and R_2 encode "conditional" versions of R that might be approved by s_1 and s_2 as intermediate results of the compliance-checking procedure.

More generally, for each request r and each assertion (f_i, s_i), there is a set $\{R_{ij}\}$ of *action strings* that might arise in a compliance check. By convention, there is a distinguished string R that corresponds to the input request r. The range of assertion (f_i, s_i) is made up of *acceptance records* of the form (i, s_i, R_{ij}), the meaning of which is that, based on the information at its disposal, assertion number i, issued by source s_i, approves action R_{ij}. A set of acceptance records is referred to as an *acceptance set*. It is by maintaining acceptance sets and making them available to assertions that the PolicyMaker compliance checker manages "inter-assertion communication," giving assertions the chance to make decisions based on conditional decisions by other assertions. The compliance checker will start with *initial acceptance set* $\{(\Lambda, \Lambda, R)\}$, in which the one acceptance record means that the action string for which approval is sought is R and that no as-

sertions have yet signed off on it (or anything else). The checker will run the assertions $(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})$ that it has received as input, not necessarily in that order and not necessarily once each, and see which acceptance records are produced. Ultimately, the compliance checker approves the request r if the acceptance record $(0, \text{POLICY}, R)$, which means "policy approves the initial action string," is produced.

Thus, abstractly, an assertion is a mapping from acceptance sets to acceptance sets. Assertion (f_i, s_i) looks at an acceptance set A encoding the actions that have been approved so far and the numbers and sources of the assertions that approved them. Based on this information about what the sources it trusts have approved, (f_i, s_i) outputs another acceptance set A'.

The following concrete examples show why PolicyMaker assertions are allowed to approve multiple action strings for each possible request. That is, for a given input request r, why do assertions need to do anything except say "I approve r" or refuse to say it?

First, consider the following "co-signing required" assertion (f_0, POLICY): "All expenditures of \$500 or more require approval by A and B." Suppose that A's policy is to approve such expenditures if and only if B approves them and that B's is to approve them if and only if A approves them. Our acceptance record structure makes such approvals straightforward. The credential (f_1, A), can produce acceptance records of the form $(1, A, R)$ and $(1, A, R_B)$, where R corresponds to the input request r; the meaning of the second is "I will approve R if and only if B approves it." Similarly, the credential (f_2, B), can produce records of the form $(2, B, R)$ and $(2, B, R_A)$. On input $\{(\Lambda, \Lambda, R)\}$, the sequence of acceptance records $(1, A, R_B)$, $(2, B, R_A)$, $(1, A, R)$, $(2, B, R)$, $(0, \text{POLICY}, R)$ would be produced if the assertions were run in the order (f_1, A), (f_2, B), (f_1, A), (f_2, B), (f_0, POLICY), and the request r would be approved. If assertions could only produce binary approve/disapprove decisions, no transactions would ever be approved, unless the trust management system had some way of understanding the semantics of the assertions and knowing that it had to ask A's and B's credentials explicitly for a conditional approval. This would violate the goal of having a general-purpose, trust management system that processes requests and assertions whose semantics are only understood by the calling applications and that vary widely from application to application.

Second, consider the issue of "delegation depth." A very natural construction to use in assertion (f_0, POLICY) is "I delegate authority to A. Furthermore, I allow A to choose the parties to whom he will re-delegate the authority I've delegated to him. For any party B involved in the approval of a request, there must be a delegation chain of length at most two from me to B." Various "domain experts" $B_1, \ldots, B_t$ could issue credentials $(f_1, B_1), \ldots, (f_t, B_t)$ that *directly* approve actions in their areas of expertise by producing acceptance records of the form (i, B_i, R_0^i). An assertion (g_j, s_j) that sees such a record and explicitly trusts B_i could produce an acceptance record of the form (j, s_j, R_1^i), the meaning of which is that "B_i approved R^i directly, I trust B_i directly, and so I also approve R^i." More generally, if an assertion (g_l, s_l) trusts s_k directly and sees an

acceptance record of the form (k, s_k, R^i_d), it can produce the acceptance record (l, s_l, R^i_{d+1}). The assertion (f_0, POLICY) given above would approve an action R^i if and only if it were run on an acceptance set that contained a record of the form (k, A, R^i_1), for some k. Note that (f_0, POLICY) need not know *which* credential (f_i, B_i) directly approved R^i by producing (i, B_i, R^i_0). All it needs to know is that it trusts A and that A trusts *some* B_i whose credential produced such a record.

The most general version of the compliance-checking problem is:

Proof of Compliance (POC):
Input : A request r and a set $\{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ of assertions.
Question : Is there a finite sequence $i_1, i_2, \ldots, i_t$ of indices such that each i_j is in $\{0, 1, \ldots, n-1\}$, but the i_j's are not necessarily distinct and not necessarily exhaustive of $\{0, 1, \ldots, n-1\}$ and such that

$$(0, \text{POLICY}, R) \in (f_{i_t}, s_{i_t}) \circ \cdots \circ (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\}),$$

where R is the action string that corresponds to the request r?

This most general version of the problem is clearly undecidable. A compliance checker cannot even decide whether an arbitrary assertion (f_i, s_i) halts when given an arbitrary acceptance set as input, much less whether some sequence containing (f_i, s_i) produces the desired output.

When we say that "$\{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ contains a proof that r complies with POLICY," we mean that $(r, \{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\})$ is a yes-instance of this unconstrained, most general form of POC. If F is a (possibly proper) subset of $\{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ that contains all of the assertions that actually appear in the sequence $(f_{i_t}, s_{i_t}) \circ \cdots \circ (f_{i_1}, s_{i_1})$, then we say that "$F$ contains a proof that r complies with POLICY."

Restricted versions of POC are obtained by adding various pieces of information to the problem instances. Specifically, consider augmenting the instance $(r, \{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\})$ in one or more of the following ways:

Global runtime bound: An instance may contain an integer d such that a sequence of assertions $(f_{i_1}, s_{i_1}), \ldots, (f_{i_t}, s_{i_t})$ is only considered a valid proof that r complies with POLICY if the total amount of time that the compliance checker needs to compute $(f_{i_t}, s_{i_t}) \circ \cdots \circ (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\})$ is $O(N^d)$. Here N is the length of the original problem instance, *i.e.*, the number of bits needed to encode r, $(f_0, \text{POLICY}), \ldots, (f_{n-1}, s_{n-1})$, and d in some standard fashion.

Local runtime bound: An instance may contain an integer c such that $(f_{i_1}, s_{i_1}), \ldots, (f_{i_t}, s_{i_t})$ is only considered a valid proof that r complies with POLICY if each (f_{i_j}, s_{i_j}) runs in time $O(N^c)$. Here N is the length of the actual acceptance set that is input to (f_{i_j}, s_{i_j}) when it is run by the compliance checker. Note that the length of the input fed to an individual assertion (f_{i_j}, s_{i_j}) in the course of checking a proof may be considerably bigger than the length of the original

problem instance $(r, \{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}, c)$, because the running of assertions (f_{i_1}, s_{i_1}), ..., $(f_{i_{j-1}}, s_{i_{j-1}})$ may have caused the creation of many new acceptance records.

Bounded number of assertions in a proof: An instance may contain an integer l such that (f_{i_1}, s_{i_1}), ..., (f_{i_t}, s_{i_t}) is only considered a valid proof if $t \leq l$.

Bounded output set: An instance may contain integers m and s such that an assertion (f_i, s_i) can only be part of a valid proof that r complies with POLICY if there is a set $O_i = \{R_{i1}, \ldots, R_{im}\}$ of m action strings, such that $(f_i, s_i)(A) \subseteq O_i$ for any input set A, and the maximum size of an acceptance record (i, s_i, R_{ij}) is s. Intuitively, for any user-supplied request r, the meaningful "domain of discourse" for assertion (f_i, s_i) is of size at most m — there are at most m actions that it would make sense for (f_i, s_i) to sign off on, no matter what the other assertions in the instance say about r.

Monotonicity: Important variants of POC are obtained by restricting attention to instances in which the assertions have the following property: (f_i, s_i) is *monotonic* if, for all acceptance sets A and B, $A \subseteq B \Rightarrow (f_i, s_i)(A) \subseteq (f_i, s_i)(B)$. Thus, if (f_i, s_i) approves action R_{ij} when given a certain set of "evidence" that R_{ij} is ok, it will also approve R_{ij} when given a superset of that evidence — it does not have a notion of "negative evidence."

Any of the parameters l, m, and s that are present in a particular instance should be written in unary so that they play an analogous role to n (the number of assertions) in the calculation of the total size of the instance. The parameters d and c are exponents in a runtime bound and hence can be written in binary. Any subset of the parameters d, c, l, m, and s may be present in a POC instance, and each subset defines a POC variant, some of which are more natural and interesting than others. Including a global runtime bound d obviously makes the POC problem decidable, as does including parameters c and l.

In stating and proving results about the complexity of POC, we use the notion of a *promise problem* [ESY84]. In a standard decision problem, a language L is defined by a predicate R in that $x \in L \Leftrightarrow R(x)$. In a promise problem, there are two predicates, the *promise* Q and the *property* R. A machine M *solves* the promise problem (Q, R) if, for all inputs x for which the promise holds, the machine M halts and accepts x if and only if the property holds. Formally, $\forall x[Q(x) \Rightarrow [M \text{ halts on } x \text{ and } M(x) \text{ accepts} \Leftrightarrow R(x)]]$. Note that M's behavior is unconstrained on inputs that do not satisfy the promise, and each set of choices for the behavior of M on these inputs determines a different solution. Thus predicates Q and R define a family of languages, namely all L such that $L = L(M)$ for some M that solves (Q, R). A promise problem is NP-hard if it has at least one solution and all of its solutions are NP-hard.

The following natural variants of POC are NP-hard. Refer to Blaze *et al.* [BFS98] for the NP-hardness proofs.

Locally Bounded Proof of Compliance (LBPOC):
Input : A request r, a set $\{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ of assertions, and integers c, l, m, and s.

Promise : Each (f_i, s_i) runs in time $O(N^c)$. On any input set that contains (Λ, Λ, R), where R is the action string corresponding to request r, for each (f_i, s_i) there is a set O_i of at most m action strings such that (f_i, s_i) only produces output from O_i, and s is the maximum size of an acceptance record (i, s_i, R_{ij}), where $R_{ij} \in O_i$.
Question : Is there a sequence $i_1, \ldots, i_t$ of indices such that

1. Each i_j is in $\{0, 1, \ldots, n-1\}$, but the i_j need not be distinct or collectively exhaustive of $\{0, 1, \ldots, n-1\}$,
2. $t \leq l$, and
3. $(0, \text{POLICY}, R) \in (f_{i_t}, s_{i_t}) \circ \cdots \circ (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\})$?

Globally Bounded Proof of Compliance (GBPOC):
Input : A request r, a set $\{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ of assertions, and an integer d.
Question : Is there a sequence $i_1, \ldots, i_t$ of indices such that

1. Each i_j is in $\{0, 1, \ldots, n-1\}$, but the i_j need not be distinct or collectively exhaustive of $\{0, 1, \ldots, n-1\}$,
2. $(0, \text{POLICY}, R) \in (f_{i_t}, s_{i_t}) \circ \cdots \circ (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\})$, where R is the action string corresponding to request r, and
3. The computation of $(f_{i_t}, s_{i_t}) \circ \cdots \circ (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\})$ runs in (total) time $O(N^d)$?

Monotonic Proof of Compliance (MPOC):
Input : A request r, a set $\{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ of assertions, and integers l and c.
Promise : Each assertion (f_i, s_i) is monotonic and runs in time $O(N^c)$.
Question : Is there a sequence $i_1, \ldots, i_t$ of indices such that

1. Each i_j is in $\{0, 1, \ldots, n-1\}$, but the i_j need not be distinct or collectively exhaustive of $\{0, 1, \ldots, n-1\}$,
2. $t \leq l$, and
3. $(0, \text{POLICY}, R) \in (f_{i_t}, s_{i_t}) \circ \cdots \circ (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\})$, where R is the action string corresponding to request r?

Each version of POC can be defined using "agglomeration" $(f_2, s_2) \star (f_1, s_1)$ instead of composition $(f_2, s_2) \circ (f_1, s_1)$. The result of applying the sequence of assertions (f_{i_1}, s_{i_1}), ..., (f_{i_t}, s_{i_t}) agglomeratively to an acceptance set S_0 is defined inductively as follows: $S_1 \equiv (f_{i_1}, s_{i_1})(S_0) \cup S_0$ and, for $2 \leq j \leq t$, $S_j \equiv (f_{i_j}, s_{i_j})(S_{j-1}) \cup S_{j-1}$. Thus, for any acceptance set A, $A \subseteq (f_{i_t}, s_{i_t}) \star \cdots \star (f_{i_1}, s_{i_1})(A)$. The agglomerative versions of the decision problems are identical to the versions already given, except that the acceptance condition is "$(0, \text{POLICY}, R) \in (f_{i_t}, s_{i_t}) \star \cdots \star (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\})$?" We refer to "agglomerative POC," "agglomerative MPOC," etc., when we mean the version defined in terms of $\star$ instead of $\circ$.

A trust management system that defines "proof of compliance" in terms of agglomeration makes it impossible for an assertion to "undo" an approval that it or any other assertion has already given to an action string during the course of constructing a proof. Informally, it forces assertions to construct proofs by communicating on a "write-only blackboard." This definition of proof makes sense if it is important for the trust management system to guard against a rogue credential-issuer's ability to thwart legitimate proofs. Note that the question of whether the compliance checker combines assertions using agglomeration or composition is separate from the question of whether the assertions themselves are monotonic.

The agglomerative versions of GBPOC, LBPOC, and MPOC are also NP-hard; the NP-hardness proofs are given in [BFS98] and are simply minor variations on the NP-hardness proofs for the composition versions.

Finally, we present the compliance-checking algorithm that is used in the current version of the PolicyMaker trust management system. The promise that defines this special case includes some conditions that we have already discussed, namely monotonicity and bounds on the runtime of assertions and on the total size of acceptance sets that assertions can produce. It also includes "authenticity," something that can be ignored when proving hardness results. An authentic assertion (f_i, s_i) only produces acceptance records of the form (i, s_i, R_{ij}), *i.e.*, it does not "impersonate" another assertion by producing an acceptance record of the form $(i', s_{i'}, R_{i'j})$.

PolicyMaker constructs proofs in an agglomerative fashion, and hence we use $\star$ in the following problem statement. This variant of POC could be defined using $\circ$ as well, but the algorithm given below would *not* work for the $\circ$ version.

Locally Bounded, Monotonic, and Authentic Proof of Compliance (LBMAPOC):
Input : A request r, a set $\{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ of assertions, and integers c, m, and s.
Promise : Each assertion (f_i, s_i) is monotonic, authentic, and runs in time $O(N^c)$. On any input set that contains (Λ, Λ, R), where R is the action string corresponding to request r, for each (f_i, s_i) there is a set O_i of at most m action strings, such that (f_i, s_i) only produces output from O_i, and s is the maximum size of an acceptance record (i, s_i, R_{ij}), such that $R_{ij} \in O_i$.
Question : Is there a sequence $i_1, \ldots, i_t$ of indices such that each i_j is in $\{0, 1, \ldots, n-1\}$, but the i_j need not be distinct or collectively exhaustive of $\{0, 1, \ldots, n-1\}$, and $(0, \text{POLICY}, R) \in (f_{i_t}, s_{i_t}) \star \cdots \star (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\})$.

The current algorithm is called CCA_1, for "compliance-checking algorithm, version 1," to allow for the evolution of PolicyMaker, and for improved algorithms CCA_i, $i \geq 1$.

Assertion (f_i, s_i) is called "ill-formed" if it violates the promise. If CCA_1 discovers in the course of simulating it that (f_i, s_i) is ill-formed, CCA_1 ignores it for the remainder of the computation. Note that an assertion (f_i, s_i) may be undetectably ill-formed; for example, there may be sets $A \subseteq B$ such that $(f_i, s_i)(A) \not\subseteq (f_i, s_i)(B)$, but such that A and B do not arise in this run of the

compliance checker. The CCA_1 algorithm checks for violations of the promise every time it simulates an assertion. The pseudocode for these checks is omitted from the statement of CCA_1 given here, because it would not illustrate the basic structure of the algorithm; the predicate $IllFormed()$ is included in the main loop to indicate that the checks are done for each simulation.

Fig. 2. Pseudocode for Algorithm CCA_1

```
CCA1(r, {(f0, POLICY), (f1, s1), ..., (fn-1, sn-1)}, c, m, s):
        {
            S ← {(Λ, Λ, R)}
            I ← {}
            For j ← 1 to mn
            {
                For i ← n-1 to 0
                {
                    If (fi, si)∉ I, Then S' ← (fi,si)(S)
                    If IllFormed((fi,si)), Then I ← I ∪ {(fi,si)},
                        Else S ← S ∪ S'
                }
            }
            If (0, POLICY, R) ∈ S, Then Output(Accept),
                Else Output(Reject)
        }
```

Note that CCA_1 does mn iterations of the sequence $(f_{n-1}, s_{n-1}), \ldots, (f_1, s_1), (f_0, \mathrm{POLICY})$, for a total of mn^2 assertion-simulations. Recall that a set $F = \{(f_{j_1}, s_{j_1}), \ldots, (f_{j_t}, s_{j_t})\} \subseteq \{(f_0, \mathrm{POLICY}), \ldots, (f_{n-1}, s_{n-1})\}$ "contains a proof that r complies with POLICY" if there is some sequence $k_1, \ldots, k_u$ of the indices $j_1, \ldots, j_t$, not necessarily distinct and not necessarily exhaustive of $j_1, \ldots, j_t$, such that $(0, \mathrm{POLICY}, R) \in (f_{k_u}, s_{k_u}) \star \cdots \star (f_{k_1}, s_{k_1})(\{(\Lambda, \Lambda, R)\})$.

The following formal claim about this algorithm is proven in [BFS98].

Theorem 1. Let $(r, \{(f_0, \mathrm{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}, c, m, s)$ be an (agglomerative) LBMAPOC instance.

(1) Suppose that $F \subseteq \{(f_0, \mathrm{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ contains a proof that r complies with POLICY and that every $(f_i, s_i) \in F$ satisfies the promise of LBMAPOC. Then CCA_1 accepts $(r, \{(f_0, \mathrm{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}, c, m, s)$.

(2) If $\{(f_0, \mathrm{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ does not contain a proof that r complies with POLICY, then CCA_1 rejects $(r, \{(f_0, \mathrm{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}, c, m, s)$.

(3) CCA_1 runs in time $O(mn^2(nms)^c)$.

Note that cases (1) and (2) do not cover all possible inputs to CCA_1. There may be a subset F of the input assertions that does contain a proof that r complies with POLICY but that contains one or more ill-formed assertions. If CCA_1 does not detect that any of these assertions is ill-formed, because their ill-formedness is only exhibited on acceptance sets that do not occur in this computation, then CCA_1 will accept the input. If it does detect ill-formedness, then, as specified here, CCA_1 may or may not accept the input, perhaps depending on whether the record $(0, \mathrm{POLICY}, R)$ has already been produced at the time of detection. CCA_1 could be modified so that it restarts every time ill-formedness is detected, after discarding the ill-formed assertion so that it is not used in the new computation. It is not clear whether this modification would be worth the performance penalty. The point is simply that CCA_1 offers no guarantees about what it does when it is fed a policy that trusts, directly or indirectly, a source of ill-formed assertions, except that it will terminate in time $O(mn^2(nms)^c)$. It is the responsibility of the policy author to know which sources to trust and to modify the policy if some trusted sources are discovered to be issuing ill-formed assertions.

Finally, note that $O(mn^2(nms)^c)$ is a pessimistic upper bound on the running time of the compliance checker. It is straightforward to check (each time an assertion (f_i, s_i) is run, or at some other regular interval) whether the acceptance record $(0, \mathrm{POLICY}, R)$ has been produced and to "stop early" if it has. Thus, for many requests R that do comply with policy, the algorithm CCA_1 will find compliance proofs in time less than $O(mn^2(nms)^c)$.