Compliance Checking in the PolicyMaker Trust Management System

Matt Blaze   Joan Feigenbaum   Martin Strauss

AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932 USA
{mab,jf,mstrauss}@research.att.com

Abstract. Emerging electronic commerce services that use public-key cryptography on a mass-market scale require sophisticated mechanisms for managing trust. For example, any service that receives a signed request for action is forced to answer the central question "Is the key used to sign this request authorized to take this action?" In some services, this question reduces to "Does this key belong to this person?" In others, the authorization question is more complicated, and resolving it requires techniques for formulating security policies and security credentials, determining whether particular sets of credentials satisfy the relevant policies, and deferring trust to third parties. Blaze, Feigenbaum, and Lacy [1] identified this *trust management problem* as a distinct and important component of network services and described a general tool for addressing it, the *PolicyMaker trust management system.*

At the heart of a trust management system is an algorithm for *compliance checking.* The inputs to the compliance checker are a *request*, a *policy*, and a set of *credentials.* The compliance checker returns yes or no, depending on whether the credentials constitute a *proof* that the request complies with the policy. Thus a central challenge in trust management is to find an appropriate notion of "proof" and an efficient algorithm for checking proofs of compliance.

In this paper, we present the notion of proof that is used in the current version of the PolicyMaker trust management system. We show that this notion of proof leads to a compliance-checking problem that is undecidable in its most general form and is NP-hard even if restricted in several natural ways. We identify a special case of the problem that is solvable in polynomial time and is widely applicable. The algorithm that we give for this special case has been implemented and is used in the current version of the PolicyMaker system.

1 Introduction

Blaze, Feigenbaum, and Lacy [1] identified the *trust management problem* as a distinct and important component of security in network services. Aspects of the trust management problem include formulation of policies and credentials, deferral of trust to third parties, and a mechanism for "proving" that a request, supported by one or more credentials, complies with a policy. In [1], the authors describe a comprehensive approach to trust management that is independent of the needs of any particular product or service, and a *trust management system*, called *PolicyMaker*, that embodies the approach. They emphasize the following general principles.

- **Common language**: Policies, credentials, and trust relationships are expressed as programs (or parts of programs) in a "safe" programming language. A common language for policies, credentials, and relationships makes it possible for applications to handle security in a comprehensive, consistent, and largely transparent manner.
- **Flexibility**: PolicyMaker is expressively rich enough to support the complex trust relationships that can occur in the very large-scale network applications currently being developed. At the same time, simple and standard policies, credentials, and relationships can be expressed succinctly and comprehensibly.
- **Locality of contol**: Each party in the network can decide in each transaction whether to accept the credentials presented by a second party or, alternatively, which third party it should ask for additional credentials. Local control of trust relationships eliminates the need for the assumption of a globally known, monolithic hierarchy of "certifying authorities." Such hierarchies do not scale beyond single "communities of interest" in which trust can be defined unconditionally from the top down.
- **General compliance-checking mechanism**: The mechanism for checking that a set of credentials proves that a requested action complies with local policy does not depend on the semantics of the application-specific request, credentials, or policy. This allows many different applications with widely varying policy requirements to share a credential base and a trust management infrastructure.

The algorithmic core of trust management is the *compliance-checking problem.* The inputs to the compliance checker are a request, a policy, and a set of credentials. The compliance checker returns yes or no, depending on whether the credentials constitute a *proof* that the request complies with the policy. Thus a central challenge in building a trust management system is to find an appropriate notion of "proof" and an efficient algorithm for checking proofs of compliance. In this paper, we present the notion of proof that is used in the PolicyMaker compliance checker. We show that, in general, the PolicyMaker version of compliance checking is undecidable and that it is NP-hard even if restricted in a number of natural ways. However, we isolate a polynomial-time solvable case that is general enough to be useful in a wide variety of applications and is implemented and available in the current version of the PolicyMaker system.

1.1 Examples

We now give three examples of application-specific requests and local policies with which they may need to comply. Note that these are realistic examples of the types of transactions that users want to perform; individually, none of them is very complicated. Collectively, they demonstrate that an expressive, flexible notion of "proof of compliance" is needed. More examples can be found in [1–3, 9, 10].

Example 1: Signed Email. Consider an email system in which messages arrive with headers that include, among other things, the sender's name, the sender's public key, and a digital signature. When a recipient's email reader processes an incoming message, it uses the public key to verify that the message and the signature go together, *i.e.*, that an adversary has not spliced a signature from another message onto this message. The recipient should also be concerned about whether the name and the public key go together; for example, might an adversary have taken a legitimate message-signature pair that he produced with this own signing key and then attached to it his public key and someone else's name? The recipient needs a policy that determines which name-key pairs he considers trustworthy. Because signed messages will regularly arrive from senders whom he has never met, he cannot simply maintain a private database of name-key pairs. Here is a plausible policy.

(1) He maintains private copies of the name-key pairs (N_1, PK_1) and (N_2, PK_2). (A reasonable interpretation of this part of the policy is that he knows the people named N_1 and N_2 personally and can get reliable copies of their public keys directly from them.)

(2) He accepts "chains of trust" of length one or two. An arc in a chain of trust is a *certificate* of the form $(PK_i, (N_j, PK_j), S)$ and is interpreted to mean that (i) the owner N_i of PK_i vouches for the binding between name N_j and public key PK_j, and (ii) N_i attests that N_j is trusted to provide certificates of this form. The party N_i signs (N_j, PK_j) with his private key and the resulting signature is S.

(3) He insists that there be two disjoint chains of trust from the keys that he maintains privately to the name-key pair that arrives with a signed message.

Example 2: Banking. Consider a loan request submitted to an electronic banking system. Such a request might contain, among other things, the name of the requester and the amount requested. A plausible policy for approval of such loans might take the following form.

(1) Two approvals are needed for loans of less than \$5,000. Three approvals are needed for loans of between \$5,000 and \$10,000. Loans of more than \$10,000 are not handled by this automated loan-processing system.

(2) The head of the loan division must authorize approvers' public keys. The division head's public key is currently PK_3. This key will expire at 23:59:59 on December 31, 1998.

Example 3: Web Browsing. A typical request for action in a web-browsing system is "View URL `http://www.coolstuff.org/pictures.gif`." In setting a viewing policy, a user must decide what types of metadata or "labels" he wants documents to have before he views them, and he must decide whom he trusts to label documents. If he is concerned about Internet pornography, the user may insist that documents he views be rated according to the Recreational Software

Advisory Council (RSAC) rating system and that they be rated ($S \leq 2, L \leq 2, V = 0, N \leq 2$) on the Sex, Language, Violence, and Nudity scales, respectively. He may trust self-labeling by the Disney Corporation and any labels by a labeler that is approved by Good Housekeeping.

1.2 Related Work

While the concept of a "trust management system" *per se* originated in [1], there is previous work on "protection systems" that is loosely related. We briefly recall two examples of such work here; more recent work that is similarly related to ours can be found in, *e.g.*, [12].

The main thrust of the work we present in this paper is twofold: We define a general "proof-of-compliance problem" that is intractable, and we isolate a special case of the problem that is both tractable and useful. Protection systems, as described by Denning [4], address a similar (but not identical) problem to the one we address, and a similar type of result is sometimes obtained.

Harrison, Ruzzo, and Ullman [7] analyze a general protection system based on the *access matrix* model. In matrix A, indexed by subjects and objects, cell A_{so} records the rights of subject s over object o; a set of transition rules describes the rights needed as preconditions to modify A and the specific ways in which A can be modified, by creating subjects and objects or by entering or deleting rights at a single cell.

Harrison, Ruzzo, and Ullman showed that given

- an initial state A_0
- a set Δ of transition rules
- a right r

it is undecidable whether some sequence $\delta_{i_0} \cdots \delta_{i_t} \in \Delta$ transforms A_0 such that δ_{i_t} enters r into a cell not previously containing r, *i.e.*, whether it is possible for some subject, not having right r over some object, ever to gain that right. On the other hand, Harrison, Ruzzo, and Ullman identify several possible restrictions on Δ and give decision algorithms for input subject to one of these restrictions. One restriction they consider yields a PSPACE-complete problem.

Independently, Jones, Lipton, and Snyder [8] define and analyze *take-grant* directed-graph systems. Subjects and objects are nodes; an arc a from node n_1 to n_2 is labeled by the set of rights n_1 has over n_2. If subject n_1 has the *take* right over n_2, and n_2 has some right r over n_3, then a legal transition is for n_1 to "take" right r over n_3. Similarly, if subject n_1 has the *grant* right over n_2, and n_1 has some right r over n_3 then a legal transition is for n_1 to "grant" right r over n_3 to n_2. Besides these transitions, subjects can create new nodes and remove their own rights over their immediate successors. Although rights are constrained to flow only via take-grant paths, take-grant systems do model nontrivial applications [4].

Jones, Lipton, and Snyder asked whether a right r over a node x possessed by node n_1 but not possessed by n_2 could ever be acquired by n_2. They showed that this question can be decided in time linear in the original graph by depth-first search. Thus Denning [4] concludes that, although safety in protection systems is usually undecidable, the results in, *e.g.*, [7, 8] demonstrate that safety can be decided feasibly in systems with sets of transition rules from a restricted though nontrivial set. The related results on compliance checking that we present in Section 5 provide additional support for Denning's conclusion.

Having reviewed the basics of "protection systems," we can now explain why they address a similar but not identical problem to the one addressed by the PolicyMaker compliance-checking algorithm. In the protection-system world, there is a relatively small set of potentially dangerous actions that could ever be performed, and this set is agreed upon in advance by all parties involved. A data structure, *e.g.*, an access matrix, records which parties are allowed to take which actions. This data structure is *precomputed offline*, and, as requests for action arrive, their legitimacy is decided via a lookup operation in this data structure. "Transition rules" that change the data structure are applied infrequently, and they are implemented *by a different mechanism and in a separate system module* from the ones that handle individual requests for action.

In the "trust management system" world, the set of potentially dangerous actions is large, dynamic, and not known in advance. A system such as PolicyMaker provides a general notion of "proof of compliance" for use by diverse applications that require trust policies. The users of these applications and the semantics of their actions and policies are not even known to the PolicyMaker compliance-checking algorithm; hence it is not possible for all parties to agree in advance on a domain of discourse for all potentially dangerous actions. The compliance-checking question "is request r authorized by policy P and credential set C?" is analogous to the question "can subject s eventually obtain right r by transition rules Δ" in the protection-system world. Part of the novelty of the PolicyMaker system [1] and of its analysis as given here is the realization that *a single instance of request processing*, especially one that involves deferral of trust, can require a moderately complex computation and not just a lookup in a precomputed data structure. The work in this paper, for the first time to our knowledge, formalizes and analyzes the complexity of a general-purpose, working system for processing requests of this nature.

In summary, a general-purpose "trust management system" such as PolicyMaker is, very roughly speaking, a meta-system in the protection-system framework.

1.3 Outline of Paper

In the next section, we explain why an application-independent notion of compliance checking can be useful and can enhance security. Terminology and notation are given in Section 3, followed by a formal statement of the compliance-checking problem in Section 4. Negative and positive results are give in Sections 5.1 and 5.2, respectively. A brief discussion of our formulation of the problem and how it might be extended appears in Section 6.

2 Need for a General Compliance Checker

We now explain why we believe that a general, highly expressive, application-independent compliance checker is a good thing. Readers already familiar with these arguments as put forth in [1, 2, 9, 10] should skip to the next section.

Clearly, any product or service that requires some form of proof that requested transactions comply with policies could implement a special-purpose compliance checker from scratch. So what advantage does a developer gain by using a general-purpose compliance checker?

One important advantage is soundness and reliability of both the design and the implementation of the compliance checker. As will become clear in the following sections, formalizing the notion of "credentials' proving that a request complies with a policy" involves a lot of subtlety and detail. It is very easy to get wrong, and an application developer who sets out to implement something special-purpose and "simple" in order to avoid what he thinks is the overly "complicated" syntax and semantics of a general-purpose compliance checker is likely to find either that he has underestimated the complexity of his application's needs for expressiveness and proof or that his special-purpose compliance checker is not turning out to be as "simple" as he expected it to be. A general-purpose notion of "proof of compliance" can be explained, formalized, proven correct, and implemented in a standard package (such as PolicyMaker), thus freeing developers of individual applications from the need to reinvent the wheel. Applications that use a standard compliance checker can be assured that the answer returned for any given input (*i.e.*, a request, a policy, and a set of credentials) depends only on the input, and not on any implicit policy decisions (or bugs) in the design or implementation of the compliance checker. As policies and credentials become more diverse and complex, the issue of assuring correctness will become especially important, and modularity of function with a clean separation between the role of the application and the role of the compliance checker will make further development more manageable.

Two important sources of complexity that are often underestimated are delegation and cryptography. Products and services that need a notion of "credential" almost always have some notion of "delegation" of the authority to issue credentials. The simplest case, *i.e.*, unconditional delegation, is easily handled by a special-purpose mechanism. However, if the product or service grows in popularity and starts to be used in ways that were not foreseen when it was originally deployed, delegation can quickly become more complex, and a special-purpose language that restricts the types of conditional delegation that can be expressed becomes an impediment to widespread and imaginative use. The general framework that we develop for compliance checking avoids this by allowing delegation to be described by ordinary programs. Similarly, digital signatures and other cryptographic functions may not seem crucial when an application is first designed; for instance, web browsers may be designed to accommodate "safe surfing" policies configurable by Internet-aware parents but may not initially involve cryptographic functions. If the application is subsequently integrated into the wider world of electronic commerce, however (as web browsers have been), the need to accommodate increased use of cryptography will be pressing, and cryptographic credentials (such as public-key certificates) will need to be incorporated into the application's notion of proof of compliance. If the application is already using a general-purpose notion of proof of compliance, this can be done without having to rethink and recode the compliance-checker.

Finally, note that a general-purpose compliance checker facilitates interoperability. Requests, policies, and credentials, if originally written in the native language of a specific product or service, must be translated into a standard format understood by the compliance checker. Because a wide variety of applications will each have translators with the same target language, policies and credentials originally written for one application can be used by another. The fact that the compliance checker can serve as a locus of interoperability may prove particularly useful in e-commerce applications and, more generally, in all settings in which cryptographic credentials are needed.

3 Notation and Terminology

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, and it takes place *before* the POC instance is handed to the compliance checker; the rationale for this architectural decision is given in the original paper on the PolicyMaker trust management system [1].

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 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 assertions 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'.

We close this section by providing two concrete examples that show why we chose to allow assertions 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, we flesh out the "conditional approval" example given earlier. 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 our 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^i_0). 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^i_1), 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.

4 Problem Statement

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. In what follows, we consider various special cases of POC and ultimately obtain one that is both useful and computationally tractable.

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 $(r, \{(f_0, \text{POLICY}), (f_1, s_1),$

…, $(f_{n-1}, s_{n-1})\})$ 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."

In order to obtain a useful restricted version of POC, we consider adding various pieces of information to the problem instances. Specifically, we consider augmenting the instance (r, $\{(f_0$, POLICY), (f_1, s_1), …, $(f_{n-1}, s_{n-1})\}$) in one or more of the following ways:

Global runtime bound: An instance may contain an integer d such that a sequence of assertions $(f_{i_1},\ s_{i_1})$, …, $(f_{i_t},\ s_{i_t})$ 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, POLICY), …, (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 our 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.

5 Results

In stating and proving results about the complexity of POC, we use the notion of a *promise problem* [5]. 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, $\forall x[Q(x) \Rightarrow [M$ halts on x and $M(x)$ 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).

The class NPP consists of all promise problems with at least one solution in NP. A promise problem is NP-hard if it has at least one solution and all of its solutions are NP-hard. To prove that a promise problem (Q, R) is NP-hard, it suffices to start with an NP-hard language L and construct a reduction whose target instances all satisfy the promise Q and satisfy the property R if and only if they are images of strings in L.

5.1 NP-Hardness

The following are natural POC variants that we can show to be computationally intractable.

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. 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.

In proving that certain POC variants, while decidable, are computationally intractable, we use the fact that the Bounded Post Correspondence Problem is NP-complete [6, Problem SR11].

Bounded Post Correspondence (BPCP):
Input : A finite alphabet Σ, two sequences $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$ of strings from Σ^*, and a positive integer $K \leq n$.
Question : Is there a sequence $i_1, i_2, \ldots, i_k$ of $k \leq K$ (not necessarily distinct) positive integers, each between 1 and n, such that the two strings $a_{i_1} a_{i_2} \cdots a_{i_k}$ and $b_{i_1} b_{i_2} \cdots b_{i_k}$ are identical?

Theorem 1. The Locally Bounded Proof of Compliance promise problem is in NPP and is NP-hard.

Proof. LBPOC is clearly in NPP, because the obvious nondeterministic polynomial-time procedure works on all instances that satisfy the promise: Guess a sequence $i_1, \ldots, i_t$ of indices, simulate the assertions (f_{i_j}, s_{i_j}) in the order specified by the sequence, and accept if and only if the acceptance record $(0, \text{POLICY}, R)$ is in the final acceptance set.

We now give a reduction from BPCP to LBPOC all of whose target instances satisfy the promise; this shows that LBPOC is NP-hard. Let $(\Sigma, (a_1, \ldots, a_q), (b_1, \ldots, b_q), K)$ be a BPCP instance. In the corresponding LBPOC instance, the request r and equivalent action string R do not play a significant role and can be taken to be σ, for any fixed $\sigma \in \Sigma$. Similarly, the only value of s_i that matters is $s_0 = \text{POLICY}$; we can put $s_i = \sigma$, for all $i \neq 0$. The number of assertions is $n = q + 1$. For $1 \leq i \leq q$, assertion (f_i, σ) produces action strings of the form (a_i, b_i, e), where a_i and b_i are identical to the strings in the BPCP instance, and e is a positive integer. (The integer e is only needed because the inputs and outputs of assertions are unordered sets, rather than ordered lists.) Specifically, when fed acceptance set S as input, (f_i, σ) outputs $S \cup \{(i, \sigma, (a_i, b_i, |S|))\}$. Let $c = 2$, $l = K$, and $m = K$. The parameter s, which should be an upper bound on the size of an acceptance record produced by the assertions in the LBPOC instance, can be taken to be $2(\log_2(q+1) + \log_2 |\Sigma| + \max_{1 \leq i \leq q}(|a_i| + |b_i|) + \log_2 K)$.

The policy assertion (f_0, POLICY) behaves as follows on input S.

- If S is not of the form $\{(\Lambda, \Lambda, R), (j_1, \sigma, (a_{j_1}, b_{j_1}, e_1)), \ldots, (j_t, \sigma, (a_{j_t}, b_{j_t}, e_t))\}$, for some $1 \leq t \leq l$, output the empty set.
- Sort $\{(\Lambda, \Lambda, R), (j_1, \sigma, (a_{j_1}, b_{j_1}, e_1)), \ldots, (j_t, \sigma, (a_{j_t}, b_{j_t}, e_t))\}$ into increasing order with respect to the e_j's. Let $i_1, \ldots, i_t$ be the resulting sorted sequence of first coordinates of acceptance records.
- If $a_{i_1} \cdots a_{i_t} = b_{i_1} \cdots b_{i_t}$, then output $\{(0, \text{POLICY}, R)\}$. Else output the empty set.

This reduction shows that LBPOC is NP-hard, because $(\Sigma, (a_1, \ldots, a_q), (b_1, \ldots, b_q), K)$ is a yes-instance of BPCP if and only if $(r, (f_0, \text{POLICY}), \ldots, (f_{n-1}, \sigma), c, l, m, s)$ is a yes-instance of LBPOC. The parameter c is set to two so that (f_0, POLICY) has (more than enough) time to sort. The other assertions run in linear time. $\square$

Theorem 2. The Globally Bounded Proof of Compliance problem is NP-complete.

Proof. It is clear that GBPOC is in NP, because a nondeterministic machine can guess a sequence of assertions and then simulate them in polynomial time to verify that conditions 1 through 3 are met.

The reduction given in the proof of Theorem 1 can be modified to yield a reduction from BPCP to GBPOC. In an LBPOC instance, each assertion can output an acceptance set of size at most ms. The total number of assertions run is at most l, and thus the size of the input to any assertion is at most lms. The running time of any assertion is thus at most $\alpha \cdot (lms)^2$, for some constant $\alpha > 0$, and the total running time of the entire simulation is at most $\alpha \cdot l \cdot (lms)^2$. In the definition of GBPOC, N is defined to be the total length of the GBPOC instance. To get a reduction from BPCP to GBPOC, choose d so that $N^d \geq \alpha \cdot l \cdot (lms)^2$. □

Theorem 3. The Monotonic Proof of Compliance promise problem is NP-hard. It remains NP-hard if the requirement that t be bounded by l is omitted.

Proof. Consider the following assertion (f_a, s_a): If the input set contains action strings encoding i for all i, $1 \leq i \leq 2^{b^2}$, then output the union of the acceptance records in the input set and a set of acceptance records with action strings encoding $2^{b^2}+1, \ldots, 2^{(b+1)^2}$, assertion number a, and source-ID s_a. Otherwise, just return the input set. This assertion is monotonic and runs in time polynomial *in the size of its input.*

An instance of BPCP of size n can be mapped in polynomial time to an instance of MPOC consisting of n assertions that produce no output, the assertion (f_a, s_a) described above that counts to $2^{(b+1)^2}$, the parameter $l = n$, and a monotonic policy assertion (f_0, POLICY) that solves the BPCP problem from scratch. The compliance checker solves the target instance by running (f_a, s_a) $n = l$ times, thereby producing $\Omega(2^{n^2})$ acceptance records. Because its input has size $\Omega(2^{n^2})$, the assertion (f_0, POLICY) has time $2^{c(n^2)}$ at its disposal, and this is sufficient to solve BPCP from scratch for some choice of the parameter c in the MPOC instance. Note that the size *of the MPOC instance* is linear in n.

Because the reduction just sets the parameter l to n, the problem becomes no easier if the parameter l is omitted.

Note that this reduction produces a language that is NP-hard but is not known to be in NP. It is not known whether this promise problem is in NPP. □

Corollary 1. The agglomerative versions of LBPOC, GBPOC, and MPOC are also NP-hard.

Proof. Essentially the same reductions from BPCP that work for the composition versions of these problems work for the agglomerative versions, too. In the agglomerative cases of LBPOC and GBPOC, the credential assertion (f_i, σ) can map input S to $\{(i, \sigma, (a_i, b_i, |S|))\}$ instead of $S \cup \{(i, \sigma, (a_i, b_i, |S|))\}$, because the compliance checker maintains the agglomeration. A similar, straightforward modification works for the hardness proof for MPOC. □

5.2 Polynomial-time Algorithm

We now present the compliance-checking algorithm that is used in the current version of the PolicyMaker trust management system. We describe the special case of the POC problem that our algorithm is guaranteed to solve and, just as importantly, what the algorithm does when given a POC instance not of this special form. The promise that defines this special case includes some conditions that we have already discussed, namely monotonicity and bounds on the run-time of assertions and on the total size of acceptance sets that assertions can produce. For a working algorithm, however, we need to consider another condition, which we call "authenticity," that we could ignore 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 in this section 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 $(0, \text{POLICY}, R) \in (f_{i_t}, s_{i_t}) \star \cdots \star (f_{i_1}, s_{i_1})(\{(\Lambda, \Lambda, R)\})$.

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

We call (f_i, s_i) "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. We don't include pseudocode for these checks in the statement of CCA_1 displayed above, 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.

Like the nondeterministic algorithms presented in Section 5.1, CCA_1 accepts if and only if the acceptance record $(0, \text{POLICY}, R)$ is produced when it simulates the input assertions. Unlike the previous algorithms, however, it cannot nondeterministically guess an order in which to do the simulation; it must have an algorithmic method of finding an order. CCA$_1$ must also ensure that, if a proper subset F of the input assertions contains a proof that r complies with POLICY and every $(f_i, s_i) \in F$ satisfies the promise, then the remaining assertions do not destroy all or part of the acceptance records produced by F during the simulation and thus destroy the proof, *even if these remaining assertions do not satisfy the promise.* CCA$_1$ achieves this by maintaining one set of approved acceptance records, from which no records are ever deleted (*i.e.*, by agglomerating), and by discarding assertions that it discovers are ill-formed.

Fig. 1. Pseudocode for Algorithm CCA$_1$

```
CCA$_1$(r, {(f$_0$, POLICY), (f$_1$, s$_1$), ..., (f$_{n-1}$, s$_{n-1}$)}, c, m, s):
        {
            S $\leftarrow$ {($\Lambda$, $\Lambda$, R)}
            I $\leftarrow$ {}
            For j $\leftarrow$ 1 to mn
            {
                For i $\leftarrow$ n-1 to 0
                {
                    If (f$_i$, s$_i$)$\notin$ I, Then S' $\leftarrow$ (f$_i$,s$_i$)(S)
                    If IllFormed((f$_i$,s$_i$)), Then I $\leftarrow$ I $\cup$ {(f$_i$,s$_i$)},
                        Else S $\leftarrow$ S $\cup$ S'
                }
            }
            If (0, POLICY, R) $\in$ 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, POLICY), for a total of mn^2 assertion-simulations. Recall that a set $F = \{(f_{j_1}, s_{j_1}), \ldots, (f_{j_t}, s_{j_t})\} \subseteq \{(f_0, \text{POLICY}), \ldots, (f_{n-1}, s_{n-1})\}$ "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 $(0, \text{POLICY}, R) \in (f_{k_u}, s_{k_u}) \star \cdots \star (f_{k_1}, s_{k_1})(\{(\Lambda, \Lambda, R)\})$.

Theorem 4. Let $(r, \{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}, c, m, s)$ be an (agglomerative) LBMAPOC instance.

(1) Suppose that $F \subseteq \{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$ 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 $(r, \{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}, c, m, s)$.

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

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

Proof. The only nontrivial claim is (1). Let $F = \{(f_{j_1}, s_{j_1}), \ldots, (f_{j_t}, s_{j_t})\}$ be a set that satisfies the hypothesis of (1). Each assertion in F is monotonic, and, as CCA_1 simulates assertions agglomeratively, it never deletes acceptance records that have already been produced but rather just adds new ones. Therefore, we may assume without loss of generality that F contains all of the well-formed assertions in $\{(f_0, \text{POLICY}), (f_1, s_1), \ldots, (f_{n-1}, s_{n-1})\}$.

Let $k_1, \ldots, k_u$ be a sequence of indices, each in $\{j_1, \ldots, j_t\}$, but not necessarily distinct and not necessarily exhaustive of $\{j_1, \ldots, j_t\}$, such that $(0, \text{POLICY}, R) \in (f_{k_u}, s_{k_u}) \star \cdots \star (f_{k_1}, s_{k_1})(\{(\Lambda, \Lambda, R)\})$. Assume without loss of generality that no sequence of length less than u has this property. Let $A_1, \ldots, A_u$ be the acceptance sets produced by applying $(f_{k_1}, s_{k_1}), \ldots, (f_{k_u}, s_{k_u})$. Because $k_1, \ldots, k_u$ is a shortest sequence that proves compliance using assertions in F, each set A_p must contain at least one action string that is not present in any of $A_1, \ldots, A_{p-1}$. Thus u iterations of $(f_0, \text{POLICY}) \star (f_1, s_1) \star \cdots \star (f_{n-1}, s_{n-1})$ would suffice for CCA_1: At some point in the first iteration, (f_{k_1}, s_{k_1}) would be run, and, because CCA_1 adds but never deletes acceptance records, A_1 or some superset of A_1 would be produced. At some point during the second iteration, (f_{k_2}, s_{k_2}) would be run, and because A_1 would be contained in its input, A_2 or some superset of A_2 would be produced. And so forth.

Each $(f_{j_h}, s_{j_h}) \in F$ satisfies the local boundedness promise and thus produces at most m distinct action strings in any computation that begins with $\{(\Lambda, \Lambda, R)\}$, regardless of the behavior of other assertions, even ill-formed ones. Because $|F| = t \leq n$, at most mn distinct action strings could ever be produced by assertions in F, and at most mn sets A_p can be produced if each is to contain a record that is not contained in any set that comes earlier in the sequence. Thus, $u \leq mn$, and mn iterations of $(f_0, \text{POLICY}) \star (f_1, s_1) \star \cdots \star (f_{n-1}, s_{n-1})$ suffice. $\square$

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 $(0, \text{POLICY}, R)$ 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.

6 Discussion

6.1 The PolicyMaker Notion of Proof

We have shown in this paper that the PolicyMaker system uses a notion of "proof that a request complies with a policy" that is amenable to definition and analysis. However, the choice of this notion of proof is still a subjective one, and there is no way to show definitively that it is *the* right notion of proof. We now briefly discuss three nontrivial design decisions that went into this choice.

A policy and credential set that are input to the compliance checker can be regarded as a "distributed policy" with which the request may or may not comply – when the local policy asserts that it trusts a credential issuer to authorize certain types of requests, it delegates part of its policy-writing responsibility to that credential issuer. The job of the compliance checker is to have the executable assertions in this distributed policy cooperate to produce a proof, and this cooperation requires a mechanism for "inter-assertion communication" of intermediate results. For simplicity, we chose to have assertions communicate just by outputting acceptance records that can be input to other assertions. More sophisticated interactions, such as allowing assertions to call each other as subroutines, might be useful but would require a more complex execution environment than the one PolicyMaker provides. An open question for future work on trust management is the trade-off between the cost of building and analyzing such an execution environment and the potential power to be gained by using more sophisticated interactions to construct proofs of compliance. Preliminary work along those lines can be found in [3].

The choice of this simple communication mechanism implies that an important part of constructing a proof of compliance is choosing an order in which to execute assertions. PolicyMaker assigns the responsibility of choosing this order to the compliance checker and not, for example, to the calling application. Although the compliance checker's job could be made easier by requiring the calling application to give it the correct order as an input, such a requirement would not be consistent with PolicyMaker's overall goals, some of which are discussed in Section 2 above. Applications should be able to use credentials issued by diverse and far-flung sources, without having to make assumptions about the order in which these credentials expect to communicate via acceptance records. In an extreme case, the issuing sources will not be aware of each others' existence, and no such assumptions by the calling application would be valid; the compliance checker has to have a way to proceed even in this case.

Although the most general version of the POC problem allows assertions to be arbitrary functions, the computationally tractable version that is analyzed in Section 5.2 and implemented in PolicyMaker is guaranteed to be correct only when all assertions are monotonic. 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 desireable 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.

Thus, we believe that the notion of proof now implemented in PolicyMaker is quite widely applicable. However, the question of how to handle non-monotonic policies in a general-purpose trust management system is an important one for future research.

6.2 A Trade-off Between Expressiveness and Verifiability

We have formulated the POC problem in a way that allows assertions to be as expressive as possible. As a result, well-formedness promises such as monotonicity and boundedness, while formal and precise, cannot in general be verified. Each assertion that conditionally trusts an assertion source for application-specific expertise (such as suitability for a loan) must also trust that source only to write bounded and monotonic assertions *and* only to trust other similar sources of assertions. The resulting notion of soundness is that if there is no proof from a set of trusted, well-formed assertions, then CCA_1 will not accept the input.

Full expressiveness, however, is just one goal of a trust management system. Another goal is the clear separation of the trust relationships of assertions from programming details. To some extent, these goals are at odds — the compliance checker cannot be expected to perform verifications on fully general programs, and thus the assertion writers must worry about some programming details.

We note that one can require monotonic assertions actually to be written as AND-OR circuits and bounded assertions to "declare" the finite set from which they will produce output. A compliance-checking algorithm could then easily detect the ill-formed assertions and discard them. This would free assertion writers of the burden of deciding when another writer is trusted to write bounded and monotonic code, just as requiring assertions to be written in a safe (and therefore restricted) language frees the assertion writer from worrying about certain application-independent programming details. This verifiability comes at a price; listing a finite output set is relatively inexpensive, but there are monotonic functions that require exponentially bigger circuits to express over a basis of AND and OR than they require over a basis of AND, OR, and NOT [11]. In some applications it may be cheaper, on average, to write assertions that are verifiably bounded and monotonic than to determine the set of sources trusted (even indirectly) by a given assertion and to judge whether they are trusted to be monotonic and bounded.

We mention another possibility for the compliance checker that gives both expressiveness and verification tools, although only at a possible performance penalty. Another possible approach that we have not yet explored is for the compliance checker to make available to assertions reading acceptance records the original code of the assertions that produced those records. A conservative policy then, before trusting assertions (f_1, s_1) and (f_2, s_2), could require and check that f_1 and f_2 be *verifiably* monotonic and bounded *and* that f_1 and f_2 each include specific standard code to check all assertions whose acceptance records (f_1, s_1) and (f_2, s_2) wish to trust. A complex monotonic assertion that needs to be written compactly using NOT gates can, if desired, still be used with the modified compliance algorithm.

References

1. M. Blaze, J. Feigenbaum, and J. Lacy, *Decentralized Trust Management*, in Proceedings of the Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos, 1996, pp. 164–173.
2. M. Blaze, J. Feigenbaum, P. Resnick, and M. Strauss, *Managing Trust in an Information-Labeling System*, European Transactions on Telecommunications, 8 (1997), pp. 491–501. (Special issue of selected papers from the 1996 Amalfi Conference on Secure Communication in Networks.)
3. Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss, REFEREE: *Trust Management for Web Applications*, World Wide Web Journal, 2 (1997), pp. 127–139. (Reprinted from Proceedings of the 6th International World Wide Web Conference, World Wide Web Consortium, Cambridge, 1997, pp. 227–238.)
4. D. Denning, **Cryptography and Data Security**, Addison-Wesley, Reading, 1982.
5. S. Even, A. Selman, and Y. Yacobi, *The Complexity of Promise Problems with Applications to Public-Key Cryptography*, Information and Control, 61 (1984), pp. 159–174.
6. M. Garey and D. Johnson, **Computers and Intractability: A Guide to the Theory of NP-Completeness**, Freeman, San Fancisco, 1979.
7. M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, *Protection in Operating Systems*, Communications of the ACM, 19 (1976), pp. 461–471.
8. A. K. Jones, R. J. Lipton, and L. Snyder, *A Linear Time Algorithm for Deciding Security*, in Proceedings of the Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, 1976, pp. 33-41.
9. J. Lacy, D. P. Maher, and J. H. Snyder, *Music on the Internet and the Intellectual Property Protection Problem*, in Proceedings of the International Symposium on Industrial Electronics, IEEE Press, New York, 1997, pp. SS77–83.

10. R. Levien, L. McCarthy, and M. Blaze, *Transparent Internet E-mail Security*, `http://www.cs.umass.edu/~lmccarth/crypto/papers/email.ps`
11. E. Tardos, *The Gap Between Monotone and Non-monotone Circuit Complexity is Exponential*, Combinatorica, 8 (1988), pp. 141–142.
12. T. Y. C. Woo and S. S. Lam, *Authorization in Distributed Systems: A New Approach*, Journal of Computer Security, 2 (1993), pp. 107–36.