Experience with the KeyNote Trust Management System: Applications and Future Directions*

Matt Blaze[1], John Ioannidis[2], and Angelos D. Keromytis[3]

[1] AT&T Labs – Research, *mab@research.att.com*
[2] AT&T Labs – Research, *ji@research.att.com*
[3] CS Department, Columbia University, *angelos@cs.columbia.edu*

Abstract. Access control in distributed systems has been an area of intense research in recent years. One promising approach has been that of *trust management,* whereby authentication and authorization decisions are combined in a unified framework for evaluating security policies and credentials. In this paper, we report on our experience of the past seven years using the PolicyMaker and the KeyNote trust management systems in a variety of projects. We start with a brief overview of trust management in general, and KeyNote in particular; we describe several applications of trust management; we then discuss various features we found missing from our initial version of KeyNote, which would have been useful in the various applications it was used. We conclude the paper with our plans for future research.

1 Introduction

The problem of controlling access to protected data or services has been a central issue in computer and network security since the early days of the computing. At a high level of abstraction, access control systems mediate access to a protected resource by only allowing authorized users to perform an operation on said resource. A traditional "system-security approach" to the processing of a request for action treats the task as a combination of *authentication* and *authorization*. The receiving system first determines *who* the requester is, typically by using an authentication protocol through which the requester digitally "signs" the request, and then queries an internal database to decide *whether* the signer should be granted access to the resources needed to perform the requested action. It has been argued that this is the wrong approach for today's ever-changing networked world [1, 2]. 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.

* This work was partly supported by DARPA and the NSF under contracts F39502-99-1-0512-MOD P0001 and CCR-TC-0208972 respectively.

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

We give an overview of the trust management approach to authorization and access control (Section 2). We describe the KeyNote [3] trust-management system (Section 3), which has been used in a number of different projects, and give a brief description of these in Section 4. Section 5 discusses future improvements to KeyNote, as are a result of our experiences in building and using a trust management system for several years.

2 A Trust-Management Approach to Access Control

The *trust-management approach*, first introduced in [1], frames the question as follows: “Does the set C of *credentials* prove that the *request* r *complies* with the local security *policy* P?” The difference with access control using traditional public-key certificates is shown graphically in Figure 1.

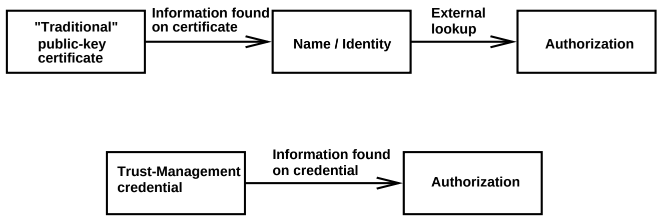
"Traditional" public-key certificate
Information found on certificate
Name / Identity
External lookup
Authorization
Trust-Management credential
Information found on credential
Authorization

Fig. 1. The difference between access control using traditional public-key certificates and trust management.

Each entity that receives requests must have a policy that serves as the ultimate source of authority in the local environment. The entity’s policy may directly authorize certain keys to take certain actions, but more typically it will *delegate* this responsibility to credential issuers that it trusts to have the required domain expertise as well

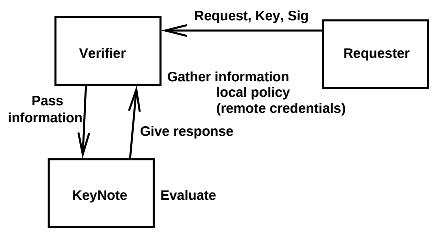

Fig. 2. **Interaction between an application and a trust-management system.**

as relationships with potential requesters. The *trust-management engine* is a separate system component that takes (r, C, P) as input, decides whether compliance with the policy has been proven, and may also output some additional information about how to proceed if the required proof has not been achieved. Figure 2 shows an example of the interactions between an application and a trust-management system.

An essential part of the trust-management approach is the use of a *general-purpose, application independent* algorithm for checking proofs of compliance. Why is this a good idea? The traditional approach that products or services have taken when they require some form of proof that requested transactions comply with policies, is to use a special-purpose algorithm or language implemented from scratch. Such algorithms/languages could be made more expressive and tuned to the particular intricacies of the application. Compared to this, the trust-management approach offers two main advantages. The first is simply one of engineering: it is preferable (in terms of simplicity and code reuse) to have a standard library or module, and a consistent API, which can be used in a variety of different applications. The second, and perhaps most important gain is in soundness and reliability of both the definition and the implementation of *proof of compliance.* Developers who set out to implement a hopefully simple, special-purpose compliance checker (in order to avoid what they think are the overly complicated syntax and semantics of a universal meta-policy) discover that they have underestimated their application's need for proof and expressiveness. As they discover the full extent of their requirements, they may ultimately wind up implementing a system that is as general and expressive as the complicated one they set out to avoid. A general-purpose compliance checker can be explained, formalized, proven correct, and implemented in a standard package, and applications that use it can be assured that the answer returned for any given input (r, C, P) depends only on the input and *not* on any implicit policy decisions (or bugs) in the design or implementation of the compliance checker.

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

– A language for describing *actions*, which are operations with security consequences that are to be controlled by the system.
– A mechanism for identifying *principals*, which are entities that can be authorized to perform actions.
– A language for specifying application *policies*, which govern the actions that principals are authorized to perform.

– A language for specifying *credentials*, which allow principals to delegate authorization to other principals.
– A *compliance checker*, which provides a service to applications for determining how an action requested by principals should be handled, given a policy and a set of credentials.

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

2.1 PolicyMaker

PolicyMaker was the first example of a *trust-management engine.* That is, it was the first tool for processing signed requests that embodied the *trust-management* principles articulated in Section 2. It addressed the authorization problem directly, rather than handling the problem indirectly via authentication and access control, and it provided an application-independent definition of *proof of compliance* for matching requests, credentials, and policies. PolicyMaker was introduced in the original trust-management paper by Blaze *et al.* [1], and its compliance-checking algorithm was later fleshed out in [4]. A full description of the system can be found in [1, 4], and experience using it in several applications is reported in [5–7].

PolicyMaker credentials and policies (collectively referred to as *assertions*) are fully programmable: they are represented as pairs (f, s), where s is the *source* of authority, and f is a program describing the nature of the authority being granted as well as the party or parties to whom it is being granted. In a policy assertion, the source is always the keyword **POLICY**. For the PolicyMaker 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,* which is the ultimate source of authority for the decision about this request, as shown in Figure 3. In a credential assertion, the source of authority is the public key of the issuing entity. Credentials must be signed by their issuers, and the signatures must be verified before the credentials can be used. PolicyMaker assertions can be written in any programming language that can be "safely" interpreted by a local environment that has to import credentials from diverse (and possibly untrusted) issuing authorities. A version of AWK without file I/O operations and with program execution time limits (to avoid denial of service attacks on the policy system) was developed for early experimental work on PolicyMaker, because AWK's pattern-matching constructs are a convenient way to express authorizations. For a credential assertion issued by a particular authority to be useful in a proof that a request complies with a policy, the recipient of the request must have an interpreter for the language in which the assertion is written (so that the program contained in the assertion can be executed). Thus, it would be desirable for assertion writers ultimately to converge on a small number of assertion languages so that receiving systems have to

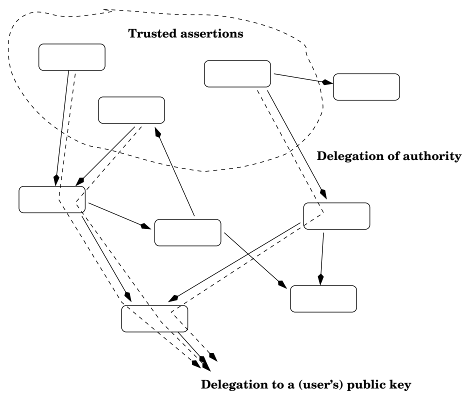

Fig. 3. Delegation in PolicyMaker, starting from a set of trusted assertions. The dotted lines indicate a delegation path from a trusted assertion (public key) to the user making a request. If all the assertions along that path authorize the request, it will be granted.

support only a small number of interpreters and so that carefully crafted credentials can be widely used. However, the question of which languages these would be was left open by the PolicyMaker project. A positive aspect of PolicyMaker's not insisting on a particular assertion language was that all the work that went into designing, analyzing, and implementing the PolicyMaker compliance-checking algorithm would not have to be redone every time the assertion language was changed or a new language was introduced. The *proof of compliance* and *assertion-language design* problems are orthogonal in PolicyMaker and can be worked on independently.

The main technical contribution of the PolicyMaker project was fully specifying and analyzing the notion of *proof of compliance*. We give an overview of PolicyMaker's approach to compliance checking here; a complete treatment of the compliance checker can be found in [4]. The PolicyMaker runtime system provides an environment in which the policy and credential assertions fed to it by the calling application can cooperate to try to produce a proof that the request complies with policy. Among the requirements for this cooperation are a method of inter-assertion communication and a method for determining that assertions have collectively succeeded or failed to produce a proof.

Inter-assertion communication in PolicyMaker is done via a simple, append-only data structure on which all participating assertions record intermediate results. Specifically, PolicyMaker initializes the proof process by creating a "blackboard" containing only the request string r and the fact that no assertions have thus far approved the request or anything else. Then PolicyMaker runs the various assertions, possibly multiple times each. When assertion (f_i, s_i) is run, it reads the contents of the blackboard and then adds to the blackboard one or more *acceptance records* (i, s_i, R_{ij}). 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. Some of the nontrivial decisions PolicyMaker must make include the order in which assertions should be run, the number of times each assertion should be run, and when to discard a non-cooperative assertion.

Although the most general version of the compliance-checking problem allows assertions to be arbitrary functions, the computationally tractable version that is analyzed in [4] 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 [4] for a formal definition. By evidence set we mean all the information an assertion uses to reach a decision; this information is typically related to the request r, but may contain additional information about the system's status *etc.*) In particular, correctness is guaranteed only for monotonic *policy* assertions, and this excludes certain types of policies that could 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 an 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.

3 The KeyNote Trust-Management System

The design of KeyNote [3] followed 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. KeyNote also requires that credentials and policies be written in a specific assertion language, designed to work smoothly with KeyNote's compliance checker. By using a specific assertion language that is flexible enough to handle the security policy needs of different applications, KeyNote goes further than PolicyMaker toward facilitating efficiency, interoperability, and widespread use of carefully written credentials and policies, at the cost of reduced expressibility and interaction between different policies, compared to PolicyMaker. A sample assertion is shown in Figure 4, with keys and signatures artificially shortened for readability.

In KeyNote, the authority to perform trusted actions is associated with one or more *principals*. A principal may be a physical entity, a process in an operating system, a public key, or any other convenient abstraction. KeyNote principals are identified by a string called a *Principal Identifier.* In some cases, a Principal Identifier will contain a cryptographic key interpreted by the KeyNote system (*e.g.,* for credential signature verification). In that case, the principal can digitally sign assertions and distribute them over untrusted networks for use by other KeyNote compliance checkers. These signed assertions are also called *credentials*, and serve a role similar to that of traditional public key certificates. Policies and credentials share the same syntax and are evaluated according to the same semantics. A principal can therefore convert its policy assertions into credentials simply by digitally signing them. In other cases, Principal Identifiers may have a structure that is opaque to KeyNote. Principals perform two functions of concern to KeyNote: they request *actions* and they issue *assertions.* Actions are any trusted operations that an application places under KeyNote control. Assertions delegate the authorization to perform actions to other principals.

A calling application passes to a KeyNote evaluator a list of credentials, policies, requester principals, and an *Action Attribute Set.* This last element consists of a list of attribute/value pairs, similar in some ways to the Unix shell environment. The action attribute set is constructed by the calling application and contains all information deemed relevant to the request and necessary for the trust decision. The action-environment attributes and the assignment of their values must reflect the security requirements of the application accurately. The semantics of the names and values are not interpreted by KeyNote itself; they vary from application to application and must be agreed upon by the writers of applications and the writers of the policies and credentials that will be used by them. Identifying the attributes to be included in the action attribute set is perhaps the most important task in integrating KeyNote into new applications. The result of the evaluation is an application-defined string that is passed back to the application. This policy compliance value returned from a KeyNote query advises the application how to process the requested action. In the simplest case, the compliance value is boolean (*e.g.,* "reject" or "approve"). Assertions can also be written to select from a range of possible compliance values, when appropriate for the application (*e.g.,* "no access", "restricted access", "full access"). Applications can configure the relative ordering (from weakest to strongest) of compliance values at query time.

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 need not be signed. 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 "programs" in KeyNote are encoded in the *Conditions* field and are essentially tests on action attributes. These tests are string comparisons, numerical opera-

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

Fig. 4. Sample KeyNote assertion authorizing either of the two keys appearing in the Licensees field to read the file "/etc/passwd".

tions and comparisons, and pattern-matching operations. We chose a simple language for KeyNote assertions for lightweight operation (compared to AWK, used by PolicyMaker), safety, and readability. As we shall see in Section 4, the simplicity of the language has not unduly impacted its usefulness in a variety of different applications, although there are several improvements we intend to make in a future release, as we discuss in Section 5.

In PolicyMaker, compliance proofs are constructed via repeated evaluation of assertions, along with an arbitrated "blackboard" for storage of intermediate results and inter-assertion communication. In contrast, KeyNote uses an algorithm that attempts (recursively) to satisfy at least one policy assertion. Referring again to Figure 3, KeyNote treats keys as vertices in the graph, with (directed) edges representing assertions delegating authority. In the prototype implementation, we used a Depth First Search algorithm, starting from the set of trusted ("POLICY") assertions and trying to construct a path to the key of the user making the request. An edge between two vertices in the graph exists only if *(a)* there exists an assertion where the *Authorizer* and the *Licensees* are the keys corresponding to the two vertices, and *(b)* the predicate encoded in the *Conditions* field of that KeyNote assertion authorizes the request.

Thus, satisfying an assertion entails satisfying both the *Conditions* field and the *Licensees* key expression. 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 research question.

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 [3], 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.)

It should be noted that PolicyMaker's restrictions regarding "negative credentials" also apply to KeyNote. Certificate revocation lists (CRLs) are not built into the KeyNote (or the PolicyMaker) system; these can be provided at a higher (or lower) level, perhaps even transparently to KeyNote. The problem of credential discovery is also not explicitly addressed in KeyNote, but we discuss possible solutions in Section 5.

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.

4 Applications of KeyNote

In this section we briefly describe the use of KeyNote is systems we and others have built. Although the ability to use KeyNote in such a wide range of applications validates its generality, we discovered several shortcomings to the system that we intend to fix in the next version. We discuss these future directions in the next section.

Network-layer Access Control One of the first applications of KeyNote was providing access control services for the IPsec [8] architecture. The IPsec protocol suite, which provides network-layer security for the Internet, has been standardized in the IETF and is beginning to make its way into commercial implementations of desktop, server, and router operating systems. IPsec does not itself address the problem of managing the *policies* governing the handling of traffic entering or leaving a node running the protocol. By itself, the IPsec protocol can protect packets from external tampering and eavesdropping, but does nothing to control which nodes are authorized for particular kinds of sessions or for exchanging particular kinds of traffic. In many configurations, especially when network-layer security is used to build firewalls and virtual private networks, such policies may necessarily be quite complex.

In [9, 10] we introduced a new policy management architecture for IPsec. A *compliance check* was added to the IPsec architecture that tests packet filters proposed when new security associations are created for conformance with the local security policy, based on credentials presented by the peer node. Security policies and credentials can be quite sophisticated (and specified in KeyNote), while still allowing very efficient packet-filtering for the actual IPsec traffic. The resulting implementation [11] has been in use in the OpenBSD [12] operating system for several years.

Distributed Firewalls Conventional firewalls rely on topology restrictions and controlled network entry points to enforce traffic filtering. The fundamental limitation of the firewall approach to network security is that a firewall cannot filter traffic it does not see; by implication, everyone on the protected side has to be considered trusted. While this model has worked well for small to medium size networks, networking trends such as increased connectivity, higher line speeds, extranets, and telecommuting threaten to make it obsolete. To address the shortcomings of traditional firewalls, the concept of a *distributed firewall* has been proposed [13]. In this scheme, security policy is still centrally defined, but enforcement is left up to the individual endpoints. Credentials distributed to every node express parts of the overall network policy. The use of KeyNote for access control at the network layer enabled us to develop a prototype distributed firewall [14]. Under certain circumstances, our prototype exhibited better performance than the traditional-firewall approach, as well as handle the increasing protocol complexity and the use of end-to-end encryption.

This functionality has been used in other projects where dynamic access control was necessary. In [15], the ability to effectively control a large number of firewalls, any of which can be contacted by any of a large number of potentially users was allowed to build a distributed denial of service (DDoS) resistant architecture for allowing authorized users to contact sites that are under attack.

The STRONGMAN Architecture The distributed firewall concept was later generalized in the STRONGMAN architecture, which allowed coordinated and decentralized management of a large number of nodes and services throughout the network stack [16, 17]. STRONGMAN offers three new approaches to scalability, applying the principle of local policy enforcement complying with global security policies. First is the use of a compliance checker to provide great local autonomy within the constraints of a global security policy. Second is a mechanism to compose policy rules into a coherent enforceable set, *e.g.,* at the boundaries of two locally autonomous application domains. Third is the lazy instantiation of policies to reduce the amount of state that enforcement points need to maintain. STRONGMAN is capable of managing such diverse resources and protocols as firewalls, web access control (discussed later), filesystem accesses, and process sandboxing. Work on STRONGMAN is continuing, focusing on the ease of management and correctness components of the system.

Web Access Control Another use of KeyNote has been in web access control, where it is used to mediate requests for pages or access to CGI scripts. To that end, we built a module for the Apache web server, *mod_keynote*, which performs the compliance checking functions on a per-request basis. This module has also been distributed with the OpenBSD operating system for several years, and the functionality has been folded into the STRONGMAN architecture.

Micropayments: Microchecks and Fileteller One of the more esoteric uses of KeyNote has been as a micropayment scheme that requires neither online transactions nor trusted hardware for either the payer or payee. Each payer is periodically issued certified credentials that encode the type of transactions and circumstances under which payment can be guaranteed. A risk management strategy, taking into account the payer's history, and other factors, can be used to generate these credentials in a way that limits the aggregated risk of uncollectible or fraudulent transactions to an acceptable level. [18] showed a practical architecture for such a system that used KeyNote to encode the credentials and policies, and described a prototype implementation of the system in which vending machine purchases were made using off-the-shelf consumer PDAs.

[19] uses this micropayment architecture to build a credential-based network file storage system with provisions for paying for file storage and getting paid when others access files. Users get access to arbitrary amounts of storage anywhere in the network, and use a micropayments system to pay for both the initial creation of the file and any subsequent accesses. Wide-scale information sharing requires that a number of issues

be addressed; these include distributed access, access control, payment, accounting, and delegation (so that information owners may allow others to access their stored content). Utilizing the same mechanism for both access control and payment results in an elegant and scalable architecture. Ongoing work in this area is examining distributed peer-to-peer filesystems and pay-per-use access to 802.11 networks [20].

Active Networking Finally, STRONGMAN has been used in the context of active networks [21] to provide access control services to programmable elements [22–24]. An active network is a network infrastructure that is programmable on a per-user or even per-packet basis. Increasing the flexibility of such network infrastructures invites new security risks. Coping with these security risks represents the most fundamental contribution of active network research. The security concerns can be divided into those which affect the network as a whole and those which affect individual elements. It is clear that the element problems must be solved first, as the integrity of network-level solutions will be based on trust of the network elements. In the SANE architecture, KeyNote was used to limit the privileges of network users and their mobile code, by specifying the operations such code was allowed to perform on any particular active node. KeyNote was used in a similar manner in the FLAME architecture [25–27], and to provide an economy for resources in an active network [28].

Grid Computing KeyNote is used to manage the authorization relationships in the Secure WebCom Metacomputer [29, 30]. WebCom [31] is a client/server based system that may be used to schedule mobile application components for execution across a network. In Secure WebCom, KeyNote credentials are used to determine the authorization of x509-authenticated SSL connections between WebCom masters and clients. Client credentials are used by WebCom masters to determine what operations the client is authorized to execute; WebCom master credentials are used by clients to determine if the master has the authorization to schedule the (trusted) mobile computation that the client is about to execute.

Transferable Micropayments WebCom is a network of systems that work together to solve large problems. Systems that provide access to their resources can be paid using hash-chain based micropayments [32]. KeyNote credentials are used to codify hash-chain micropayment contracts; determining whether a particular micropayment should be accepted amounts to a KeyNote compliance check that the micropayment is authorized. This scheme is generalized in [33] to support the efficient transfer of micropayment contracts whereby a transfer amounts to delegation of authorization for the contract. Characterizing a payment scheme as a trust management problem means that trust policies that are based on both monetary and conventional authorization concerns can be formulated.

Case-based Reasoning Systems It is considered in [34] how similarity techniques that are used by case-based reasoning systems might be adapted to support degrees of imprecision when delegating authority based on KeyNote credentials. A key is considered authorized for some action if it authorized for another similar action, within some degree of similarity. [34] demonstrates how to codify similarity measures within KeyNote credentials such that a test for authorization amounts to a compliance check.

5 Future Directions

We now discuss the various improvements we plan on making to the next version of KeyNote, based on our experiences with several applications as well as comments from the user community. These improvements include changes to the language and assertion format as well as enhancements to our distribution implementation. We focus on changes that will allow us to use KeyNote in new contexts and classes of applications.

Bit Operations A regrettable omission that became almost immediately obvious was KeyNote's lack of support for bit-level operations (bit-wise AND, OR, XOR, NOT, *etc.*). Lack of bit operations made it especially awkward to apply KeyNote to network-layer access control, as discussed in Section 4. In this application, we needed to test IP addresses against network subnets (*e.g.*, "does the address 128.59.19.32 belong to the 128.59.23.255/21 subnet ?"). In pseudo-code, this comparison would be expressed as "128.59.19.32 & 255.255.248.0 = = 128.59.23.255 & 255.255.248.0". Since KeyNote does not support bit-wise operations, we had to resort to tricks such as the string-wise comparisons "128.59.19.32 >= 128.59.16.0 && 128.59.19.32 <= 128.59.23.255". This is both visually unappealing and non-intuitive; IPv6, with its longer addresses, will only exacerbate this problem. These comparisons can be performed more concisely by breaking up the address into individual octets and performing numerical operations/comparisons, but the end-result is even less comprehensible to a human reader. We intend to extend KeyNote to natively support bit-wise operations, similar to string and numeric operations.

Sets and Arrays By modern programming language standards, KeyNote does not seem to support an especially rich collection of data types or data structures. This was a deliberate design decision, of course, in line with our minimalist philosophy, but, as with the lack of bit operations, experience has suggested some obvious potential enhancements. In particular, in access control systems and other security schemes, it is often natural to reason about sets, arrays, and lists. In the current KeyNote system, such operations must be simulated with simple strings and regular expression operations, but this often leads to rather opaque (and inefficient) constructions. A future version of the language may benefit from a richer collection of set data structures, along with operations for testing membership, *etc.*

Function Calls Currently, the action environment must be populated by the invoking application prior to performing the compliance check. In the prototype, it is possible to populate the action environment on-demand (*i.e.*, as each action attribute is accessed by an assertion). However, it is not possible to use *parameterized* action attributes, *i.e.*, populate an action attribute based on the content of some other variable or a parameter contained in the assertion itself. For example, consider a policy that allows any user access to a file as long as there is a particular entry for that user in a system database. In principle, the action environment could be populated with the complete database — however, the assertion writer would have to know *a priori* the names of all users (if they are used as the lookup key in the database), and these names would then be used as action attributes (containing that user's information), in order to access them from inside an assertion. This will result in considerable initialization costs as well as semantic confusion (since new variable will have to appear in the action environment)t as new users are added. To solve this problem, we intend to add application-specific function calls. These will take as argument a string, and return a string.

Naming and Scoping One of the earliest architectural goals of KeyNote (and, indeed, of the original PolicyMaker system) was to minimize the distinction between a local "policy" and a remote "credential." In our philosophy, the only difference between a policy and a credential is that the latter is signed; a corollary of this should be that one can convert a local policy into a remotely-distributable and usable credential simply by signing it. In practice, however, this is only true for the simplest of policies. Policies might refer to principal identifies in assertions that are meaningful only locally and that are not expected to be visible outside the context of the policy itself.

To make it truly possible to treat a policy as a single construct that can be signed and used remotely, KeyNote would need a scope system that would allow an entire group of assertions to share a single, private principal naming context and be signed and exported as a group. "Internal" principal names would not be visible outside their scope. This should be a straightforward change to the language, entailing the introduction of syntactic scope grouping operation and the obvious semantics for resolving names.

Exception Handling The algorithm currently used for determining the acceptance record of an assertion uses the highest value returned from all the clauses in the Conditions field. For example, the following pair of clauses will return "true" if the AES or 3DES is used as the encryption algorithm, regardless of the authentication algorithm.

```
Conditions:
  app_domain == "IPsec policy" -> {
    encryption_algorithm == "AES" ||
         encryption_algorithm == "3DES" -> "true";
    authentication_algorithm == "SHA1" -> "false";
   }
```

This approach has made rule specification very easy and flexible. One disadvantage, however, has been the difficulty of concisely disallowing one particular request from among a large set of acceptable requests. To address this, we intend to extend the KeyNote language to contain an "except" construct that can be used to wrap classes of clauses. Once all the classes in the wrapped class have been evaluated, another set of clauses associated with the construct are evaluated to handle any exceptions. If any of these clauses match, the acceptance record of the wrapped class is modified accordingly. Syntactically, the construct might look like:

```
Conditions:
  app_domain == "IPsec policy" && {
    encryption_algorithm == "3DES" ||
        encryption_algorithm == "AES" -> "true";
  } except {
      authentication_algorithm == "none" -> "false";
  }
```

Credential Attributes It is sometimes useful in the process of evaluating a KeyNote assertion to change the action attribute set that subsequent assertions will see. For example, if a KeyNote assertion would evaluate to true if a particular set of conditions is met, it may want to communicate this fact to subsequent assertions by adding an action attribute to the action attribute set that they will see. This tries to recapture some of PolicyMaker's blackboard functionality.

Syntactically, such a feature would be implemented as a new KeyNote field, the `Attributes:` field. The syntax is similar to the `Local-Constants:` field; just a list of name/value pairs. However, unlike the local-constants field which causes just lexical substitution in the current credential, the Attributes field affects the action attributes set of all subsequent evaluations.

Multiple Assertion Representations Some environments, *e.g.*, the WWW DAV system [35], use specific languages and representation medium in all aspects of the system (in the case of DAV, XML is used). When using KeyNote in such environments, it would be useful to have alternate, semantically equivalent representations for assertions. Our work in web access control (Section 4) has prompted us to investigate an XML schema for KeyNote assertions. One challenge is the integration of assertions, represented in different formats, in the same compliance check, *e.g.*, using policies in the currently-used format and credentials in an XML-based encoding.

Revocation Perhaps the most common request for a new feature is support for some form of revocation. As we discussed in Section 2, KeyNote (and monotonic trust-management systems in general) do not support the notion of "negative credentials", as these make it difficult to reason about the correctness of the system and the evaluation logic. Revocation can be built independent of the compliance checker, at a different level. However, this makes use of KeyNote less natural for applications that have an explicit notion of revocation; such applications must parse KeyNote credentials and perform the necessary revocation checks in the application code instead of leaving it to the trust management engine. The KeyNote language itself natively supports only time-based revocation (*i.e.*, credential expiration), by encoding the appropriate rules in the Conditions field of an assertion, *e.g.*, the following expression:

```
Authorizer: SOME_KEY
Licensees: SOME_OTHER_KEY
Conditions: app_domain == "IPsec" &&
  (encryption == "3DES" || encryption == "AES") &&
  current_time <= "20031215052500" -> "true";
Signature: ...
```

would cause the credential to return "false" after 5:25am, December 15, 2003.

Revocation is a difficult problem in general, and is especially so for monotonic trust management. One possible approach for a future version of KeyNote is to treat revocation as another principal in the Licensees field. To encode an revocation rule under a particular scheme, *e.g.*, the OCSP protocol, we would simply use a conjunctive expression in the Licensees field:

```
Authorizer: SOME_KEY
Licensees: SOME_OTHER_KEY &&
 "OCSP:revocation.cs.columbia.edu:3232:REVOCATION_KEY"
Conditions: app_domain == "IPsec" &&
  (encryption == "3DES" ||  encryption == "AES") &&
  current_time <= "20031215052500" -> "true";
Signature: ...
```

In this example, the assertion would allow a request to proceed if the conditions were fulfilled (the encryption algorithm was 3DES or AES), it had not expired, and the OCSP revocation server found at revocation.cs.columbia.edu, port 3232, did not indicate to us that the credential had been marked as invalid. The REVOCATION_KEY can be used to protect the communication between the compliance checker and the revocation server. Other revocation schemes such as Certificate Revocation Lists (CRLs), Delta-CRLs, refresher certificates, *etc.* can be used in the same manner.

Bytecode Interpreter The KeyNote prototype was built using the *lex* and *yacc* tools to parse the assertion format. As a result, the prototype contains a lot of code that depends on the standard *C* library, for doing string operations and memory allocation. Furthermore, because these tools are meant to support a large class of grammars, the generated parsers are fairly general and, consequently, more inefficient than a hand-crafted parser would be. While these constrains are not particularly restrictive, they make it practically impossible to use KeyNote inside an operating system kernel. To allow for easy integration of KeyNote inside an operating system kernel (or resource-limited embedded devices), we intend to create a bytecode interpreter. A front-end compiler will parse the credentials and convert them from the respective format (*e.g.*, the current format, or an XML-based one) to a bytecode program that can be uploaded to the kernel-resident compliance checker. In this manner, it will be similar to the BPF packet-filtering system [36]. Another advantage of this approach is the ability to use credentials of different formats in the context of the same policy evaluation. We are currently investigating the appropriate bytecode instruction format and semantics to use. Note that this improvement need not entail changes to the KeyNote language *per se*.

Debugging Tools Since we released KeyNote, we frequently receive requests for help in debugging some problem with KeyNote. In practically all cases, the users are not setting the correct values in the action environment (that is, values corresponding to what their policies check for). Similar problems arise from typos or using the wrong key as an Authorizer or Licensee in a credential. A good interactive debugging tool (*e.g.*, a GUI-based trial-and-error system) would save both KeyNote users and implementors considerable time and frustration, and, again, need not entail changes to the language.

6 Concluding Remarks

We reported our experience using the KeyNote trust-management system, on satisfying the needs of authorization and access control for a number of different applications. As a result, we have determined a set of features and improvements that we intend to integrate into a future version of KeyNote. Despite these omissions from the initial release, KeyNote has proven remarkably flexible and useful in a variety of contexts. We believe that we have thus validated our original hypothesis from 5 years ago, that a simple trust-management system can address the needs of most applications that have authorization and access control requirements. Our future work will focus on solving specific problems, easing the use of KeyNote in new environments, and exploring new directions for trust management, without diverting from our goal of simplicity and compactness.

Acknowledgments

We would like to thank Simon Foley for contributing some of the text in Section 4.

References

1. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized Trust Management. In: Proceedings of the 17th Symposium on Security and Privacy. (1996) 164–173
2. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The Role of Trust Management in Distributed Systems Security. In: Secure Internet Programming. Volume 1603 of Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA (1999) 185–210
3. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The KeyNote Trust Management System Version 2. Internet RFC 2704 (1999)
4. Blaze, M., Feigenbaum, J., Strauss, M.: Compliance Checking in the PolicyMaker Trust-Management System. In: Proceedings of the Financial Cryptography '98, Lecture Notes in Computer Science, vol. 1465. (1998) 254–274
5. Blaze, M., Feigenbaum, J., Resnick, P., Strauss, M.: Managing Trust in an Information Labeling System. In: European Transactions on Telecommunications, 8. (1997) 491–501
6. Lacy, J., Snyder, J., Maher, D.: Music on the Internet and the Intellectual Property Protection Problem. In: Proceedings of the International Symposium on Industrial Electronics, IEEE Press (1997) SS77–83
7. Levien, R., McCarthy, L., Blaze, M.: Transparent Internet E-mail Security. http://www.cs.-umass.edu/˜lmccarth/crypto/papers/email.ps (1996)
8. Kent, S., Atkinson, R.: Security Architecture for the Internet Protocol. RFC 2401 (1998)
9. Blaze, M., Ioannidis, J., Keromytis, A.: Trust Management for IPsec. In: Proceedings of Network and Distributed System Security Symposium (NDSS). (2001) 139–151
10. Blaze, M., Ioannidis, J., Keromytis, A.: Trust Management for IPsec. ACM Transactions on Information and System Security (TISSEC) **32** (2002) 1–24
11. Hallqvist, N., Keromytis, A.D.: Implementing Internet Key Exchange (IKE). In: Proceedings of the Annual USENIX Technical Conference, Freenix Track. (2000) 201–214
12. de Raadt, T., Hallqvist, N., Grabowski, A., Keromytis, A.D., Provos, N.: Cryptography in OpenBSD: An Overview. In: Proceedings of the 1999 USENIX Annual Technical Conference, Freenix Track. (1999) 93–101
13. Bellovin, S.M.: Distributed Firewalls. *;login:* magazine, issue on security (1999) 37–39
14. Ioannidis, S., Keromytis, A., Bellovin, S., Smith, J.: Implementing a Distributed Firewall. In: Proceedings of Computer and Communications Security (CCS). (2000) 190–199
15. Keromytis, A.D., Misra, V., Rubenstein, D.: SOS: Secure Overlay Services. In: Proceedings of ACM SIGCOMM. (2002) 61–72
16. Keromytis, A., Ioannidis, S., Greenwald, M., Smith, J.: The STRONGMAN Architecture. In: Proceedings of DISCEX III. (2003)

17. Keromytis, A.D.: STRONGMAN: A Scalable Solution To Trust Management In Networks. PhD thesis, University of Pennsylvania, Philadelphia (2001)
18. Blaze, M., Ioannidis, J., Keromytis, A.D.: Offline Micropayments without Trusted Hardware. In: Proceedings of the 5h International Conference on Financial Cr yptography. (2001) 21–40
19. Ioannidis, J., Ioannidis, S., Keromytis, A., Prevelakis, V.: Fileteller: Paying and Getting Paid for File Storage. In: Proceedings of the 6th International Conference on Financial Cryptography. (2002)
20. Miltchev, S., Prevelakis, V., Ioannidis, S., Ioannidis, J., Keromytis, A.D., Smith, J.M.: Secure and Flexible Global File Sharing. In: Proceedings of the USENIX Technical Annual Conference, Freenix Track. (2003)
21. Alexander, D.S., Arbaugh, W.A., Hicks, M., Kakkar, P., Keromytis, A.D., Moore, J.T., Gunter, C.A., Nettles, S.M., Smith, J.M.: The SwitchWare Active Network Architecture. IEEE Network, special issue on Active and Programmable Networks **12** (1998) 29–36
22. Alexander, D.S., Arbaugh, W.A., Keromytis, A.D., Smith, J.M.: A Secure Active Network Environment Architecture: Realization in SwitchWare. IEEE Network, special issue on Active and Programmable Networks **12** (1998) 37–45
23. Alexander, D.S., Arbaugh, W.A., Keromytis, A.D., Muir, S., Smith, J.M.: Secure Quality of Service Handling (SQoSH). IEEE Communications **38** (2000) 106–112
24. Alexander, D., Menage, P., Keromytis, A., Arbaugh, W., Anagnostakis, K., Smith, J.: The Price of Safety in an Active Network. Journal of Communications (JCN), special issue on programmable switches and routers **3** (2001) 4–18
25. Anagnostakis, K.G., Ioannidis, S., Miltchev, S., Smith, J.M.: Practical network applications on a lightweight active management environment. In: Proceedings of the 3rd International Working Conference on Active Networks (IWAN). (2001)
26. Anagnostakis, K.G., Ioannidis, S., Miltchev, S., Ioannidis, J., Greenwald, M.B., Smith, J.M.: Efficient packet monitoring for network management. In: Proceedings of IFIP/IEEE Network Operations and Management Symposium (NOMS) 2002. (2002)
27. Anagnostakis, K.G., Greenwald, M.B., Ioannidis, S., Miltchev, S.: Open Packet Monitoring on FLAME: Safety, Performance and Applications. In: Proceedings of the 4rd International Working Conference on Active Networks (IWAN). (2002)
28. Anagnostakis, K.G., Hicks, M.W., Ioannidis, S., Keromytis, A.D., Smith, J.M.: Scalable Resource Control in Active Networks. In: Proceedings of the Second International Working Conference on Active Networks (IWAN). (2000) 343–357
29. Foley, S., Quillinan, T., Morrison, J., Power, D., Kennedy, J.: Exploiting KeyNote in WebCom: Architecture Neutral Glue for Trust Management. In: Fifth Nordic Workshop on Secure IT Systems. (2001)
30. Foley, S., Quillinan, T., Morrison, J.: Secure Component Distribution Using WebCom. In: Proceedings of the 17th International Conference on Information Security (IFIP/SEC). (2002)
31. Morrison, J., Power, D., Kennedy, J.: WebCom: A Web Based Distributed Computation Platform. In: Proceedings of Distributed computing on the Web. (1999)
32. Foley, S., Quillinan, T.: Using Trust Management to Support MicroPayments. In: Proceedings of the Annual Conference on Information Technology and Telecommunications. (2002)
33. Foley, S.: Using Trust Management to Support Transferable Hash-Based Micropayments. In: Proceedings of the International Financial Cryptography Conference. (2003)
34. Foley, S.: Supporting Imprecise Delegation in KeyNote. In: Proceedings of 10th International Security Protocols Workshop. (2002)
35. Whitehead, E.: World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction. ACM StandardView **5** (1997) 3–8
36. McCanne, S., Jacobson, V.: A BSD Packet Filter: A New Architecture for User-level Packet Capture. In: Proceedings of USENIX Winter Technical Conference, Usenix (1993) 259–269